

University of Oxford

College of Computer Science

Elzein Salah Elzein

22/12/2021

PHD Report Draft

Artificial Intelligent Agents and CASE STUDIES in Robotics

PHD Research

Artificial Intelligent Agents and CASE STUDIES in Robotics

Introduction

Artificial Intelligence and Agents

The history of AI is a history of fantasies, possibilities, demonstrations, and promise. Ever since Homer wrote of mechanical “tripods” waiting on the gods at dinner, imagined mechanical assistants have been a part of our culture. However, only in the last half century have we, the AI community, been able to build experimental machines that test hypotheses about the mechanisms of thought and intelligent behavior and thereby demonstrate mechanisms that formerly existed only as theoretical possibilities.

– Bruce Buchanan [2005]

This book is about artificial intelligence, a field built on centuries of thought, which has been a recognized discipline for over 50 years. As Buchanan points out in the quote above, we now have the tools to test hypotheses about the nature of thought itself, as well as solve practical problems. Deep scientific and engineering problems have already been solved and many more are waiting to be solved. Many practical applications are currently deployed and the potential exists for an almost unlimited number of future applications. In this book, we present the principles that underlie intelligent computational agents. Those principles can

help you understand current and future work in AI and equip you to contribute to the discipline yourself.

1.1 What Is Artificial Intelligence?

Artificial intelligence, or **AI**, is the field that studies *the synthesis and analysis of computational agents that act intelligently*. Let us examine each part of this definition.

An **agent** is something that acts in an environment – it does something. Agents include worms, dogs, thermostats, airplanes, robots, humans, companies, and countries.

We are interested in what an agent does; that is, how it **acts**. We judge an agent by its actions.

An agent acts **intelligently** when

- what it does is appropriate for its circumstances and its goals,
- it is flexible to changing environments and changing goals,
- it learns from experience, and
- it makes appropriate choices given its perceptual and computational limitations. An agent typically cannot observe the state of the world directly; it has only a finite memory and it does not have unlimited time to act.

A **computational** agent is an agent whose decisions about its actions can be explained in terms of computation. That is, the decision can be broken down into primitive operation that can be implemented in a physical device. This computation can take many forms. In humans this computation is carried out in “wetware”; in computers it is carried out in “hardware.” Although there are some agents that are arguably not computational, such as the wind and rain eroding a landscape, it is an open question whether all intelligent agents are computational.

The central **scientific goal** of AI is to understand the principles that make intelligent behavior possible in natural or artificial systems. This is done by

- the **analysis** of natural and artificial agents;
- formulating and testing hypotheses about what it takes to construct intelligent agents; and
- designing, building, and experimenting with computational systems that perform tasks commonly viewed as requiring intelligence.

As part of science, researchers build empirical systems to test hypotheses or to explore the space of possibilities. These are quite distinct from applications that are built to be useful for an application domain.

Note that the definition is not for intelligent *thought*. We are only interested in **thinking** intelligently insofar as it leads to better performance. The role of thought is to affect action.

The central **engineering goal** of AI is the **design** and **synthesis** of useful, intelligent artifacts. We actually want to build agents that act intelligently. Such

agents are useful in many applications.

1.1.1 Artificial and Natural Intelligence

Artificial intelligence (AI) is the established name for the field, but the term “artificial intelligence” is a source of much confusion because artificial intelligence may be interpreted as the opposite of real intelligence.

Interrogator: In the first line of your sonnet which reads “Shall I compare thee to a summer’s day,” would not “a spring day” do as well or better?

Witness: It wouldn’t scan.

Interrogator: How about “a winter’s day,” That would scan all right.

Witness: Yes, but nobody wants to be compared to a winter’s day.

Interrogator: Would you say Mr. Pickwick reminded you of Christmas?

Witness: In a way.

Interrogator: Yet Christmas is a winter’s day, and I do not think Mr. Pickwick would mind the comparison.

Witness: I don’t think you’re serious. By a winter’s day one means a typical winter’s day, rather than a special one like Christmas.

Figure 1.1: A possible dialog for the Turing test (from [Turing \[1950\]](#))

For any phenomenon, you can distinguish real versus fake, where the fake is non-real. You can also distinguish natural versus artificial. Natural means occurring in nature and artificial means made by people.

Example 1.1 A tsunami is a large wave in an ocean caused by an earthquake or a landslide. Natural tsunamis occur from time to time. You could imagine an artificial tsunami that was made by people, for example, by exploding a bomb in the ocean, yet which is still a real tsunami. One could also imagine fake tsunamis: either artificial, using computer graphics, or natural, for example, a mirage that looks like a tsunami but is not one.

It is arguable that intelligence is different: you cannot have *fake* intelligence. If an agent behaves intelligently, it is intelligent. It is only the external behavior that defines intelligence; acting intelligently is being intelligent. Thus, artificial intelligence, if and when it is achieved, will be real intelligence created artificially.

This idea of intelligence being defined by external behavior was the motivation for a test for intelligence designed by [Turing \[1950\]](#), which has become known as the **Turing test**. The Turing test consists of an imitation game where an interrogator can ask a witness, via a text interface, any question. If the interrogator cannot distinguish the witness from a human, the witness must be intelligent. Figure 1.1 shows a possible dialog that Turing suggested. An agent that is not really intelligent could not fake intelligence for arbitrary topics.

There has been much debate about the Turing test. Unfortunately, although it may provide a test for how to recognize intelligence, it does not provide a way to get there; trying each year to fake it does not seem like a useful avenue of research.

The obvious naturally intelligent agent is the human being. Some people might say that worms, insects, or bacteria are intelligent, but more people would say that dogs, whales, or monkeys are intelligent (see Exercise 1 (page 42)). One class of intelligent agents that may be more intelligent than humans is the class of *organizations*. Ant colonies are a prototypical example of organizations. Each individual ant may not be very intelligent, but an ant colony can act more intelligently than any individual ant. The colony can discover food and exploit it very effectively as well as adapt to changing circumstances. Similarly, companies can develop, manufacture, and distribute products where the sum of the skills required is much more than any individual could master. Modern computers, from low-level hardware to high-level software, are more complicated than any human can understand, yet they are manufactured daily by organizations of humans. Human *society* viewed as an agent is arguably the most intelligent agent known.

It is instructive to consider where human intelligence comes from. There are three main sources:

biology: Humans have evolved into adaptable animals that can survive in various habitats.

culture: Culture provides not only language, but also useful tools, useful concepts, and the wisdom that is passed from parents and teachers to children.

life-long learning: Humans learn throughout their life and accumulate knowledge and skills.

These sources interact in complex ways. Biological evolution has provided stages of growth that allow for different learning at different stages of life. We humans and our culture have evolved together so that humans are helpless at birth, presumably because of our culture of looking after infants. Culture interacts strongly with learning. A major part of lifelong learning is what people are taught by parents and teachers. Language, which is part of culture, provides distinctions in the world that should be noticed for learning.

1.2 A Brief History of AI

Throughout human history, people have used technology to model themselves. There is evidence of this from ancient China, Egypt, and Greece that bears witness to the universality of this activity. Each new technology has, in its turn, been exploited to build intelligent agents or models of mind. Clockwork, hydraulics, telephone switching systems, holograms, analog computers, and digital computers have all been proposed both as technological metaphors for intelligence and as mechanisms for modeling mind.

About 400 years ago people started to write about the nature of thought and reason. Hobbes (1588–1679), who has been described by Haugeland [1985,

p. 85] as the “Grandfather of AI,” espoused the position that thinking was symbolic reasoning like talking out loud or working out an answer with pen and paper. The idea of symbolic reasoning was further developed by Descartes (1596–1650), Pascal (1623–1662), Spinoza (1632–1677), Leibniz (1646–1716), and others who were pioneers in the philosophy of mind.

The idea of symbolic operations became more concrete with the development of computers. The first general-purpose computer designed (but not built until 1991, at the Science Museum of London) was the **Analytical Engine** by Babbage (1792–1871). In the early part of the 20th century, there was much work done on understanding computation. Several models of computation were proposed, including the Turing machine by Alan Turing (1912–1954), a theoretical machine that writes symbols on an infinitely long tape, and the lambda calculus of Church (1903–1995), which is a mathematical formalism for rewriting formulas. It can be shown that these very different formalisms are equivalent in that any function computable by one is computable by the others. This leads to the **Church-Turing thesis**:

Any effectively computable function can be carried out on a Turing machine (and so also in the lambda calculus or any of the other equivalent formalisms).

Here **effectively computable** means following well-defined operations; “computers” in Turing’s day were people who followed well-defined steps and computers as we know them today did not exist. This thesis says that all computation can be carried out on a Turing machine or one of the other equivalent computational machines. The Church-Turing thesis cannot be proved but it is a hypothesis that has stood the test of time. No one has built a machine that has carried out computation that cannot be computed by a Turing machine. There is no evidence that people can compute functions that are not Turing computable. An agent’s actions are a function of its abilities, its history, and its goals or preferences. This provides an argument that computation is more than just a metaphor for intelligence; reasoning *is* computation and computation can be carried out by a computer.

Once real computers were built, some of the first applications of computers were AI programs. For example, Samuel [1959] built a checkers program in 1952 and implemented a program that learns to play checkers in the late 1950s. Newell and Simon [1956] built a program, Logic Theorist, that discovers proofs in propositional logic.

In addition to that for high-level symbolic reasoning, there was also much work on low-level learning inspired by how neurons work. McCulloch and Pitts [1943] showed how a simple thresholding “formal neuron” could be the basis for a Turing-complete machine. The first learning for these neural networks was described by Minsky [1952]. One of the early significant works was the Perceptron of Rosenblatt [1958]. The work on neural networks went into decline for a number of years after the 1968 book by Minsky and Papert [1988],

Does Afghanistan border China?
 What is the capital of Upper Volta?
 Which country's capital is London?
 Which is the largest african country?
 How large is the smallest american country?
 What is the ocean that borders African countries and that borders Asian countries?
 What are the capitals of the countries bordering the Baltic?
 How many countries does the Danube flow through?
 What is the total area of countries south of the Equator and not in Australasia?
 What is the average area of the countries in each continent?
 Is there more than one country in each continent?
 What are the countries from which a river flows into the Black Sea?
 What are the continents no country in which contains more than two cities whose population exceeds 1 million?
 Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India?
 Which countries with a population exceeding 10 million border the Atlantic?

Figure 1.2: Some questions CHAT-80 could answer

which argued that the representations learned were inadequate for intelligent action.

These early programs concentrated on learning and search as the foundations of the field. It became apparent early that one of the main problems was how to represent the knowledge needed to solve a problem. Before learning, an agent must have an appropriate target language for the learned knowledge. There have been many proposals for representations from simple feature-based representations to complex logical representations of McCarthy and Hayes [1969] and many in between such as the frames of Minsky [1975].

During the 1960s and 1970s, success was had in building natural language understanding systems in limited domains. For example, the STUDENT program of Daniel Bobrow [1967] could solve high school algebra problems expressed in natural language. Winograd's [1972] SHRDLU system could, using restricted natural language, discuss and carry out tasks in a simulated blocks world. CHAT-80 [Warren and Pereira, 1982] could answer geographical questions placed to it in natural language. Figure 1.2 shows some questions that CHAT-80 answered based on a database of facts about countries, rivers, and so on. All of these systems could only reason in very limited domains using restricted vocabulary and sentence structure.

During the 1970s and 1980s, there was a large body of work on **expert systems**, where the aim was to capture the knowledge of an expert in some domain so that a computer could carry out expert tasks. For example, **DENDRAL** [Buchanan and Feigenbaum, 1978], developed from 1965 to 1983 in the field of organic chemistry, proposed plausible structures for new organic compounds. **MYCIN** [Buchanan and Shortliffe, 1984], developed from 1972 to 1980, diagnosed infectious diseases of the blood, prescribed antimicrobial therapy, and explained its reasoning. The 1970s and 1980s were also a period when AI reasoning became widespread in languages such as **Prolog** [Colmerauer and Roussel, 1996; Kowalski, 1988].

During the 1990s and the 2000s there was great growth in the subdisciplines of AI such as perception, probabilistic and decision-theoretic reasoning, planning, embodied systems, machine learning, and many other fields. There has also been much progress on the foundations of the field; these form the foundations of this book.

1.2.1 Relationship to Other Disciplines

AI is a very young discipline. Other disciplines as diverse as philosophy, neurobiology, evolutionary biology, psychology, economics, political science, sociology, anthropology, control engineering, and many more have been studying intelligence much longer.

The science of AI could be described as “synthetic psychology,” “experimental philosophy,” or “computational epistemology” – **epistemology** is the study of knowledge. AI can be seen as a way to study the old problem of the nature of knowledge and intelligence, but with a more powerful experimental tool than was previously available. Instead of being able to observe only the external behavior of intelligent systems, as philosophy, psychology, economics, and sociology have traditionally been able to do, AI researchers experiment with executable models of intelligent behavior. Most important, such models are open to inspection, redesign, and experiment in a complete and rigorous way. Modern computers provide a way to construct the models about which philosophers have only been able to theorize. AI researchers can experiment with these models as opposed to just discussing their abstract properties. AI theories can be empirically grounded in implementation. Moreover, we are often surprised when simple agents exhibit complex behavior. We would not have known this without implementing the agents.

It is instructive to consider an analogy between the development of flying machines over the past few centuries and the development of thinking machines over the past few decades. There are several ways to understand flying. One is to dissect known flying animals and hypothesize their common structural features as necessary fundamental characteristics of any flying agent. With this method, an examination of birds, bats, and insects would suggest that flying involves the flapping of wings made of some structure covered with feathers or a membrane. Furthermore, the hypothesis could be tested by

strapping feathers to one's arms, flapping, and jumping into the air, as Icarus did. An alternate methodology is to try to understand the principles of flying without restricting oneself to the natural occurrences of flying. This typically involves the construction of artifacts that embody the hypothesized principles, even if they do not behave like flying animals in any way except flying. This second method has provided both useful tools – airplanes – and a better understanding of the principles underlying flying, namely *aerodynamics*.

AI takes an approach analogous to that of aerodynamics. AI researchers are interested in testing general hypotheses about the nature of intelligence by building machines that are intelligent and that do not necessarily mimic humans or organizations. This also offers an approach to the question, “Can computers really think?” by considering the analogous question, “Can airplanes really fly?”

AI is intimately linked with the discipline of computer science. Although there are many non-computer scientists who are doing AI research, much, if not most, AI research is done within computer science departments. This is appropriate because the study of computation is central to AI. It is essential to understand algorithms, data structures, and combinatorial complexity to build intelligent machines. It is also surprising how much of computer science started as a spinoff from AI, from timesharing to computer algebra systems.

Finally, AI can be seen as coming under the umbrella of *cognitive science*. Cognitive science links various disciplines that study cognition and reasoning, from psychology to linguistics to anthropology to neuroscience. AI distinguishes itself within cognitive science by providing tools to build intelligence rather than just studying the external behavior of intelligent agents or dissecting the inner workings of intelligent systems.

1.3 Agents Situated in Environments

AI is about practical reasoning: reasoning in order to do something. A coupling of perception, reasoning, and acting comprises an **agent**. An agent acts in an **environment**. An agent's environment may well include other agents. An agent together with its environment is called a **world**.

An agent could be, for example, a coupling of a computational engine with physical sensors and actuators, called a **robot**, where the environment is a physical setting. It could be the coupling of an advice-giving computer – an **expert system** – with a human who provides perceptual information and carries out the task. An agent could be a program that acts in a purely computational environment – a **software agent**.

Figure 1.3 shows the inputs and outputs of an agent. At any time, what an agent does depends on its

- **prior knowledge** about the agent and the environment;
- **history** of interaction with the environment, which is composed of

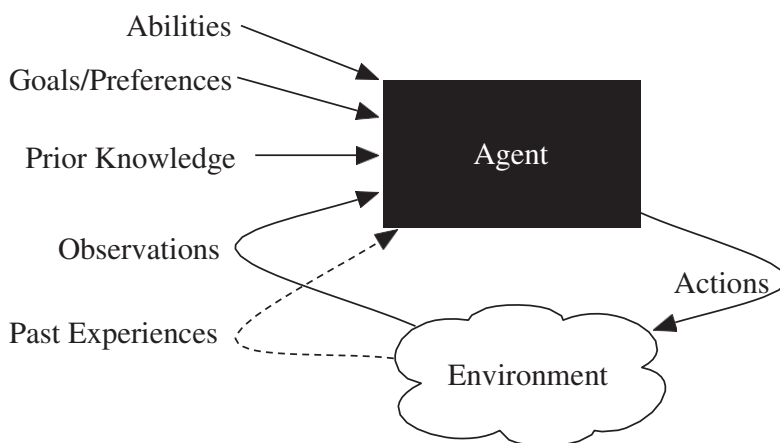


Figure 1.3: An agent interacting with an environment

- **observations** of the current environment and
- **past experiences** of previous actions and observations, or other data, from which it can learn;
- **goals** that it must try to achieve or preferences over states of the world; and
- **abilities**, which are the primitive actions it is capable of carrying out.

Two deterministic agents with the same prior knowledge, history, abilities, and goals should do the same thing. Changing any one of these can result in different actions.

Each agent has some internal state that can encode beliefs about its environment and itself. It may have goals to achieve, ways to act in the environment to achieve those goals, and various means to modify its beliefs by reasoning, perception, and learning. This is an all-encompassing view of intelligent agents varying in complexity from a simple thermostat, to a team of mobile robots, to a diagnostic advising system whose perceptions and actions are mediated by human beings, to society itself.

1.4 Knowledge Representation

Typically, a problem to solve or a task to carry out, as well as what constitutes a solution, is only given informally, such as “deliver parcels promptly when they arrive” or “fix whatever is wrong with the electrical system of the house.”

The general framework for solving problems by computer is given in Figure 1.4 (on the next page). To solve a problem, the designer of a system must

- flesh out the task and determine what constitutes a solution;
- represent the problem in a language with which a computer can reason;

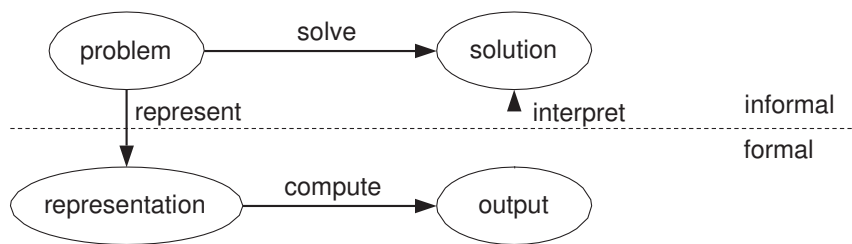


Figure 1.4: The role of representations in solving problems

- use the computer to compute an output, which is an answer presented to a user or a sequence of actions to be carried out in the environment; and
- interpret the output as a solution to the problem.

Knowledge is the information about a domain that can be used to solve problems in that domain. To solve many problems requires much knowledge, and this knowledge must be represented in the computer. As part of designing a program to solve problems, we must define how the knowledge will be represented. A **representation scheme** is the form of the knowledge that is used in an agent. A **representation** of some piece of knowledge is the internal representation of the knowledge. A representation scheme specifies the form of the knowledge. A **knowledge base** is the representation of all of the knowledge that is stored by an agent.

A good representation scheme is a compromise among many competing objectives. A representation should be

- rich enough to express the knowledge needed to solve the problem.
- as close to the problem as possible; it should be compact, natural, and maintainable. It should be easy to see the relationship between the representation and the domain being represented, so that it is easy to determine whether the knowledge represented is correct. A small change in the problem should result in a small change in the representation of the problem.
- amenable to efficient computation, which usually means that it is able to express features of the problem that can be exploited for computational gain and able to trade off accuracy and computation time.
- able to be acquired from people, data and past experiences.

Many different representation schemes have been designed. Many of these start with some of these objectives and are then expanded to include the other objectives. For example, some are designed for learning and then expanded to allow richer problem solving and inference abilities. Some representation schemes are designed with expressiveness in mind, and then inference and learning are added on. Some schemes start from tractable inference and then are made more natural, and more able to be acquired.

Some of the questions that must be considered when given a problem or a task are the following:

- What is a solution to the problem? How good must a solution be?
- How can the problem be represented? What distinctions in the world are needed to solve the problem? What specific knowledge about the world is required? How can an agent acquire the knowledge from experts or from experience? How can the knowledge be debugged, maintained, and improved?
- How can the agent compute an output that can be interpreted as a solution to the problem? Is worst-case performance or average-case performance the critical time to minimize? Is it important for a human to understand how the answer was derived?

These issues are discussed in the next sections and arise in many of the representation schemes presented later in the book.

1.4.1 Defining a Solution

Given an informal description of a problem, before even considering a computer, a knowledge base designer should determine what would constitute a solution. This question arises not only in AI but in any software design. Much of **software engineering** involves refining the specification of the problem.

Typically, problems are not well specified. Not only is there usually much left unspecified, but also the unspecified parts cannot be filled in arbitrarily. For example, if you ask a trading agent to find out all the information about resorts that may have health issues, you do not want the agent to return the information about all resorts, even though all of the information you requested is in the result. However, if the trading agent does not have complete knowledge about the resorts, returning all of the information may be the only way for it to guarantee that all of the requested information is there. Similarly, you do not want a delivery robot, when asked to take all of the trash to the garbage can, to take everything to the garbage can, even though this may be the only way to guarantee that all of the trash has been taken. Much work in AI is motivated by **commonsense reasoning**; we want the computer to be able to make commonsense conclusions about the unstated assumptions.

Given a well-defined problem, the next issue is whether it matters if the answer returned is incorrect or incomplete. For example, if the specification asks for all instances, does it matter if some are missing? Does it matter if there are some extra instances? Often a person does not want just any solution but the best solution according to some criteria. There are four common classes of solutions:

Optimal solution An **optimal solution** to a problem is one that is the best solution according to some measure of solution quality. This measure is typically specified as an **ordinal**, where only the order matters. However, in some

situations, such as when combining multiple criteria or when reasoning under uncertainty, you need a **cardinal** measure, where the relative magnitudes also matter. An example of an ordinal measure is for the robot to take out as much trash as possible; the more trash it can take out, the better. As an example of a cardinal measure, you may want the delivery robot to take as much of the trash as possible to the garbage can, minimizing the distance traveled, and explicitly specify a trade-off between the effort required and the proportion of the trash taken out. It may be better to miss some trash than to waste too much time. One general cardinal measure of desirability, known as **utility**, is used in decision theory (page 373).

Satisficing solution Often an agent does not need the best solution to a problem but just needs some solution. A **satisficing solution** is one that is good enough according to some description of which solutions are adequate. For example, a person may tell a robot that it must take all of trash out, or tell it to take out three items of trash.

Approximately optimal solution One of the advantages of a cardinal measure of success is that it allows for approximations. An **approximately optimal solution** is one whose measure of quality is close to the best that could theoretically be obtained. Typically agents do not need optimal solutions to problems; they only must get close enough. For example, the robot may not need to travel the optimal distance to take out the trash but may only need to be within, say, 10% of the optimal distance.

For some problems, it is much easier computationally to get an approximately optimal solution than to get an optimal solution. However, for other problems, it is (asymptotically) just as difficult to guarantee finding an approximately optimal solution as it is to guarantee finding an optimal solution. Some approximation algorithms guarantee that a solution is within some range of optimal, but for some algorithms no guarantees are available.

Probable solution A **probable solution** is one that, even though it may not actually be a solution to the problem, is likely to be a solution. This is one way to approximate, in a precise manner, a satisficing solution. For example, in the case where the delivery robot could drop the trash or fail to pick it up when it attempts to, you may need the robot to be 80% sure that it has picked up three items of trash. Often you want to distinguish the **false-positive error** rate (the proportion of the answers given by the computer that are not correct) from the **false-negative error** rate (which is the proportion of those answers not given by the computer that are indeed correct). Some applications are much more tolerant of one of these errors than of the other.

These categories are not exclusive. A form of learning known as probably approximately correct (PAC) learning considers probably learning an approximately correct concept (page 332).

1.4.2 Representations

Once you have some requirements on the nature of a solution, you must represent the problem so a computer can solve it.

Computers and human minds are examples of **physical symbol systems**. A **symbol** is a meaningful pattern that can be manipulated. Examples of symbols are written words, sentences, gestures, marks on paper, or sequences of bits. A **symbol system** creates, copies, modifies, and destroys symbols. Essentially, a symbol is one of the patterns manipulated as a unit by a symbol system.

The term physical is used, because symbols in a physical symbol system are physical objects that are part of the real world, even though they may be internal to computers and brains. They may also need to physically affect action or motor control.

Much of AI rests on the **physical symbol system hypothesis** of Newell and Simon [1976]:

A physical symbol system has the necessary and sufficient means for general intelligent action.

This is a strong hypothesis. It means that any intelligent agent is necessarily a physical symbol system. It also means that a physical symbol system is all that is needed for intelligent action; there is no magic or an as-yet-to-be-discovered quantum phenomenon required. It does not imply that a physical symbol system does not need a body to sense and act in the world. The physical symbol system hypothesis is an empirical hypothesis that, like other scientific hypotheses, is to be judged by how well it fits the evidence, and what alternative hypotheses exist. Indeed, it could be false.

An intelligent agent can be seen as manipulating symbols to produce action. Many of these symbols are used to refer to things in the world. Other symbols may be useful concepts that may or may not have external meaning. Yet other symbols may refer to internal states of the agent.

An agent can use physical symbol systems to model the world. A **model** of a world is a representation of the specifics of what is true in the world or of the dynamic of the world. The world does not have to be modeled at the most detailed level to be useful. All models are **abstractions**; they represent only part of the world and leave out many of the details. An agent can have a very simplistic model of the world, or it can have a very detailed model of the world. The **level of abstraction** provides a partial ordering of abstraction. A lower-level abstraction includes more details than a higher-level abstraction. An agent can have multiple, even contradictory, models of the world. The models are judged not by whether they are correct, but by whether they are useful.

Example 1.2 A delivery robot can model the environment at a high level of abstraction in terms of rooms, corridors, doors, and obstacles, ignoring distances, its size, the steering angles needed, the slippage of the wheels, the weight of parcels, the details of obstacles, the political situation in Canada, and

virtually everything else. The robot could model the environment at lower levels of abstraction by taking some of these details into account. Some of these details may be irrelevant for the successful implementation of the robot, but some may be crucial for the robot to succeed. For example, in some situations the size of the robot and the steering angles may be crucial for not getting stuck around a particular corner. In other situations, if the robot stays close to the center of the corridor, it may not need to model its width or the steering angles.

Choosing an appropriate level of abstraction is difficult because

- a high-level description is easier for a human to specify and understand.
- a low-level description can be more accurate and more predictive. Often high-level descriptions abstract away details that may be important for actually solving the problem.
- the lower the level, the more difficult it is to reason with. This is because a solution at a lower level of detail involves more steps and many more possible courses of action exist from which to choose.
- you may not know the information needed for a low-level description. For example, the delivery robot may not know what obstacles it will encounter or how slippery the floor will be at the time that it must decide what to do.

It is often a good idea to model an environment at multiple levels of abstraction. This issue is further discussed in Section 2.3 (page 50).

Biological systems, and computers, can be described at multiple levels of abstraction. At successively lower levels are the neural level, the biochemical level (what chemicals and what electrical potentials are being transmitted), the chemical level (what chemical reactions are being carried out), and the level of physics (in terms of forces on atoms and quantum phenomena). What levels above the neuron level are needed to account for intelligence is still an open question. Note that these levels of description are echoed in the hierarchical structure of science itself, where scientists are divided into physicists, chemists, biologists, psychologists, anthropologists, and so on. Although no level of description is more important than any other, we conjecture that you do not have to emulate every level of a human to build an AI agent but rather you can emulate the higher levels and build them on the foundation of modern computers. This conjecture is part of what AI studies.

The following are two levels that seem to be common to both biological and computational entities:

- The **knowledge level** is a level of abstraction that considers what an agent knows and believes and what its goals are. The knowledge level considers what an agent knows, but not how it reasons. For example, the delivery agent's behavior can be described in terms of whether it knows that a parcel has arrived or not and whether it knows where a particular person is or not. Both human and robotic agents can be described at the knowledge level. At this level, you do not specify how the solution will be computed or even which of the many possible strategies available to the agent will be used.

- The **symbol level** is a level of description of an agent in terms of the reasoning it does. To implement the knowledge level, an agent manipulates symbols to produce answers. Many cognitive science experiments are designed to determine what symbol manipulation occurs during reasoning. Note that whereas the knowledge level is about what the agent believes about the external world and what its goals are in terms of the outside world, the symbol level is about what goes on inside an agent to reason about the external world.

1.4.3 Reasoning and Acting

The manipulation of symbols to produce action is called **reasoning**.

One way that AI representations differ from computer programs in traditional languages is that an AI representation typically specifies **what** needs to be computed, not **how** it is to be computed. We might specify that the agent should find the most likely disease a patient has, or specify that a robot should get coffee, but not give detailed instructions on how to do these things. Much AI reasoning involves searching through the space of possibilities to determine how to complete a task.

In deciding what an agent will do, there are three aspects of computation that must be distinguished: (1) the computation that goes into the design of the agent, (2) the computation that the agent can do before it observes the world and needs to act, and (3) the computation that is done by the agent as it is acting.

- **Design time reasoning** is the reasoning that is carried out to design the agent. It is carried out by the designer of the agent, not the agent itself.
- **Offline computation** is the computation done by the agent before it has to act. It can include compilation and learning. Offline, the agent takes background knowledge and data and compiles them into a usable form called a **knowledge base**. **Background knowledge** can be given either at design time or offline.
- **Online computation** is the computation done by the agent between observing the environment and acting in the environment. A piece of information obtained online is called an **observation**. An agent typically must use both its knowledge base and its observations to determine what to do.

It is important to distinguish between the knowledge in the mind of the designer and the knowledge in the mind of the agent. Consider the extreme cases:

- At one extreme is a highly specialized agent that works well in the environment for which it was designed, but it is helpless outside of this niche. The designer may have done considerable work in building the agent, but the agent may not need to do very much to operate well. An example is a thermostat. It may be difficult to design a thermostat so that it turns on and off at exactly the right temperatures, but the thermostat itself does not have to do much computation. Another example is a painting robot that always

paints the same parts in an automobile factory. There may be much design time or offline computation to get it to work perfectly, but the painting robot can paint parts with little online computation; it senses that there is a part in position, but then it carries out its predefined actions. These very specialized agents do not adapt well to different environments or to changing goals. The painting robot would not notice if a different sort of part were present and, even if it did, it would not know what to do with it. It would have to be redesigned or reprogrammed to paint different parts or to change into a sanding machine or a dog washing machine.

- At the other extreme is a very flexible agent that can survive in arbitrary environments and accept new tasks at run time. Simple biological agents such as insects can adapt to complex changing environments, but they cannot carry out arbitrary tasks. Designing an agent that can adapt to complex environments and changing goals is a major challenge. The agent will know much more about the particulars of a situation than the designer. Even biology has not produced many such agents. Humans may be the only extant example, but even humans need time to adapt to new environments.

Even if the flexible agent is our ultimate dream, researchers have to reach this goal via more mundane goals. Rather than building a universal agent, which can adapt to any environment and solve any task, they have built particular agents for particular environmental niches. The designer can exploit the structure of the particular niche and the agent does not have to reason about other possibilities.

Two broad strategies have been pursued in building agents:

- The first is to simplify environments and build complex reasoning systems for these simple environments. For example, factory robots can do sophisticated tasks in the engineered environment of a factory, but they may be hopeless in a natural environment. Much of the complexity of the problem can be reduced by simplifying the environment. This is also important for building practical systems because many environments can be engineered to make them simpler for agents.
- The second strategy is to build simple agents in natural environments. This is inspired by seeing how **insects** can survive in complex environments even though they have very limited reasoning abilities. Researchers then make the agents have more reasoning abilities as their tasks become more complicated.

One of the advantages of simplifying environments is that it may enable us to prove properties of agents or to optimize agents for particular situations. Proving properties or optimization typically requires a model of the agent and its environment. The agent may do a little or a lot of reasoning, but an observer or designer of the agent may be able to reason about the agent and the environment. For example, the designer may be able to prove whether the agent can achieve a goal, whether it can avoid getting into situations that may be bad for the agent (**safety goals**), whether it will get stuck somewhere (**liveness**),

or whether it will eventually get around to each of the things it should do (**fairness**). Of course, the proof is only as good as the model.

The advantage of building agents for complex environments is that these are the types of environments in which humans live and want our agents to live.

Fortunately, research along both lines is being carried out. In the first case, researchers start with simple environments and make the environments more complex. In the second case, researchers increase the complexity of the behaviors that the agents can carry out.

1.5 Dimensions of Complexity

Agents acting in environments range in complexity from thermostats to companies with multiple goals acting in competitive environments. A number of dimensions of complexity exist in the design of intelligent agents. These dimensions may be considered separately but must be combined to build an intelligent agent. These dimensions define a **design space** of AI; different points in this space can be obtained by varying the values of the dimensions.

Here we present nine dimensions: modularity, representation scheme, planning horizon, sensing uncertainty, effect uncertainty, preference, number of agents, learning, and computational limits. These dimensions give a coarse division of the design space of intelligent agents. There are many other design choices that must be made to build an intelligent agent.

1.5.1 Modularity

The first dimension is the level of modularity.

Modularity is the extent to which a system can be decomposed into interacting modules that can be understood separately.

Modularity is important for reducing complexity. It is apparent in the structure of the brain, serves as a foundation of computer science, and is an important part of any large organization.

Modularity is typically expressed in terms of a hierarchical decomposition. For example, a human's visual cortex and eye constitute a module that takes in light and perhaps higher-level goals and outputs some simplified description of a scene. Modularity is hierarchical if the modules are organized into smaller modules, which, in turn, can be organized into even smaller modules, all the way down to primitive operations. This hierarchical organization is part of what biologists investigate. Large organizations have a hierarchical organization so that the top-level decision makers are not overwhelmed by details and do not have to micromanage all details of the organization. Procedural abstraction and object-oriented programming in computer science are designed to enable simplification of a system by exploiting modularity and abstraction.

In the **modularity dimension**, an agent's structure is one of the following:

- **flat**: there is no organizational structure;
- **modular**: the system is decomposed into interacting modules that can be understood on their own; or
- **hierarchical**: the system is modular, and the modules themselves are decomposed into interacting modules, each of which are hierarchical systems, and this recursion grounds out into simple components.

In a flat or modular structure the agent typically reasons at a single level of abstraction. In a hierarchical structure the agent reasons at multiple levels of abstraction. The lower levels of the hierarchy involve reasoning at a lower level of abstraction.

Example 1.3 In taking a trip from home to a holiday location overseas, an agent, such as yourself, must get from home to an airport, fly to an airport near the destination, then get from the airport to the destination. It also must make a sequence of specific leg or wheel movements to actually move. In a flat representation, the agent chooses one level of abstraction and reasons at that level. A modular representation would divide the task into a number of subtasks that can be solved separately (e.g., booking tickets, getting to the departure airport, getting to the destination airport, and getting to the holiday location). In a hierarchical representation, the agent will solve these subtasks in a hierarchical way, until the problem is reduced to simple problems such as sending an http request or taking a particular step.

A hierarchical decomposition is important for reducing the complexity of building an intelligent agent that acts in a complex environment. However, to explore the other dimensions, we initially ignore the hierarchical structure and assume a flat representation. Ignoring hierarchical decomposition is often fine for small or moderately sized problems, as it is for simple animals, small organizations, or small to moderately sized computer programs. When problems or systems become complex, some hierarchical organization is required.

How to build hierarchically organized agents is discussed in [Section 2.3](#) (page 50).

1.5.2 Representation Scheme

The **representation scheme dimension** concerns how the world is described.

The different ways the world could be to affect what an agent should do are called **states**. We can factor the state of the world into the agent's internal state (its belief state) and the environment state.

At the simplest level, an agent can reason explicitly in terms of individually identified states.

Example 1.4 A thermostat for a heater may have two belief states: *off* and *heating*. The environment may have three states: *cold*, *comfortable*, and *hot*. There are thus six states corresponding to the different combinations of belief and environment states. These states may not fully describe the world, but they are adequate to describe what a thermostat should do. The thermostat should move to, or stay in, *heating* if the environment is *cold* and move to, or stay in, *off* if the environment is *hot*. If the environment is *comfortable*, the thermostat should stay in its current state. The agent heats in the *heating* state and does not heat in the *off* state.

Instead of enumerating states, it is often easier to reason in terms of the state's features or propositions that are true or false of the state. A state may be described in terms of **features**, where a feature has a value in each state [see Section 4.1 (page 112)].

Example 1.5 An agent that has to look after a house may have to reason about whether light bulbs are broken. It may have features for the position of each switch, the status of each switch (whether it is working okay, whether it is shorted, or whether it is broken), and whether each light works. The feature *pos* *s2* may be a feature that has value *up* when switch *s2* is up and has value *down* when the switch is down. The state of the house's lighting may be described in terms of values for each of these features.

A **proposition** is a Boolean feature, which means that its value is either *true* or *false*. Thirty propositions can encode $2^{30} = 1,073,741,824$ states. It may be easier to specify and reason with the thirty propositions than with more than a billion states. Moreover, having a compact representation of the states indicates understanding, because it means that an agent has captured some regularities in the domain.

Example 1.6 Consider an agent that has to recognize letters of the alphabet. Suppose the agent observes a binary image, a 30×30 grid of pixels, where each of the 900 grid points is either on or off (i.e., it is not using any color or gray scale information). The action is to determine which of the letters *a*, ..., *z* is drawn in the image. There are 2^{900} different states of the image, and so $26^{2^{900}}$ different functions from the image state into the characters *a*, ..., *z*. We cannot even represent such functions in terms of the state space. Instead, we define features of the image, such as line segments, and define the function from images to characters in terms of these features.

When describing a complex world, the features can depend on relations and individuals. A relation on a single individual is a property. There is a feature for each possible relationship among the individuals.

Example 1.7 The agent that looks after a house in Example 1.5 could have the lights and switches as individuals, and relations *position* and *connected_to*. Instead of the feature *position* *s1* = *up*, it could use the relation *position*(*s1*, *up*).

This relation enables the agent to reason about all switches or for an agent to have knowledge about switches that can be used when the agent encounters a switch.

Example 1.8 If an agent is enrolling students in courses, there could be a feature that gives the grade of a student in a course, for every student–course pair where the student took the course. There would be a *passed* feature for every student–course pair, which depends on the *grade* feature for that pair. It may be easier to reason in terms of individual students, courses and grades, and the relations *grade* and *passed*. By defining how *passed* depends on *grade* once, the agent can apply the definition for each student and course. Moreover, this can be done before the agent knows of any of the individuals and so before it knows any of the features.

Thus, instead of dealing with features or propositions, it is often more convenient to have **relational descriptions** in terms of **individuals** and **relations** among them. For example, one binary relation and 100 individuals can represent $100^2 = 10,000$ propositions and 2^{10000} states. By reasoning in terms of relations and individuals, an agent can specify reason about whole classes of individuals without ever enumerating the features or propositions, let alone the states. An agent may have to reason about infinite sets of individuals, such as the set of all numbers or the set of all sentences. To reason about an unbounded or infinite number of individuals, an agent cannot reason in terms of states or features; it must reason at the relational level.

In the **representation scheme dimension**, the agent reasons in terms of

- states,
- features, or
- relational descriptions, in terms of individuals and relations.

Some of the frameworks will be developed in terms of states, some in terms of features and some relationally.

Reasoning in terms of states is introduced in Chapter 3. Reasoning in terms of features is introduced in Chapter 4. We consider relational reasoning starting in Chapter 12.

1.5.3 Planning Horizon

The next dimension is how far ahead in time the agent plans. For example, when a dog is called to come, it should turn around to start running in order to get a reward in the future. It does not act only to get an immediate reward. Plausibly, a dog does not act for goals arbitrarily far in the future (e.g., in a few months), whereas people do (e.g., working hard now to get a holiday next year).

How far the agent “looks into the future” when deciding what to do is called the **planning horizon**. That is, the planning horizon is how far ahead the

agent considers the consequences of its actions. For completeness, we include the non-planning case where the agent is not reasoning in time. The time points considered by an agent when planning are called **stages**.

In the **planning horizon dimension**, an agent is one of the following:

- A **non-planning** agent is an agent that does not consider the future when it decides what to do or when time is not involved.
- A **finite horizon** planner is an agent that looks for a fixed finite number of time steps ahead. For example, a doctor may have to treat a patient but may have time for some testing and so there may be two stages: a testing stage and a treatment stage to plan for. In the degenerate case where an agent only looks one time step ahead, it is said to be **greedy** or **myopic**.
- An **indefinite horizon** planner is an agent that looks ahead some finite, but not predetermined, number of steps ahead. For example, an agent that must get to some location may not know *a priori* how many steps it will take to get there.
- An **infinite horizon** planner is an agent that plans on going on forever. This is often called a **process**. For example, the stabilization module of a legged robot should go on forever; it cannot stop when it has achieved stability, because the robot has to keep from falling over.

1.5.4 Uncertainty

An agent could assume there is no uncertainty, or it could take uncertainty in the domain into consideration. Uncertainty is divided into two dimensions: one for uncertainty from sensing and one for uncertainty about the effect of actions.

Sensing Uncertainty

In some cases, an agent can observe the state of the world directly. For example, in some board games or on a factory floor, an agent may know exactly the state of the world. In many other cases, it may only have some noisy perception of the state and the best it can do is to have a probability distribution over the set of possible states based on what it perceives. For example, given a patient's symptoms, a medical doctor may not actually know which disease a patient may have and may have only a probability distribution over the diseases the patient may have.

The **sensing uncertainty dimension** concerns whether the agent can determine the state from the observations:

- **Fully observable** is when the agent knows the state of the world from the observations.
- **Partially observable** is when the agent does not directly observe the state of the world. This occurs when many possible states can result in the same observations or when observations are noisy.

Assuming the world is fully observable is often done as a simplifying assumption to keep reasoning tractable.

Effect Uncertainty

In some cases an agent knows the effect of an action. That is, given a state and an action, it can accurately predict the state resulting from carrying out that action in that state. For example, an agent interacting with a file system may be able to predict the effect of deleting a file given the state of the file system. In many cases, it is difficult to predict the effect of an action, and the best an agent can do is to have a probability distribution over the effects. For example, a person may not know the effect of calling his dog, even if he knew the state of the dog, but, based on experience, he has some idea of what it will do. The dog owner may even have some idea of what another dog, that he has never seen before, will do if he calls it.

The **effect uncertainty dimension** is that the dynamics can be

- **deterministic** – when the state resulting from an action is determined by an action and the prior state or
- **stochastic** – when there is only a probability distribution over the resulting states.

This dimension only makes sense when the world is fully observable. If the world is partially observable, a stochastic system can be modeled as a deterministic system where the effect of an action depends on some unobserved feature. It is a separate dimension because many of the frameworks developed are for the fully observable, stochastic action case.

Planning with deterministic actions is considered in Chapter 8. Planning with stochastic actions and with partially observable domains is considered in Chapter 9.

1.5.5 Preference

Agents act to have better outcomes for themselves. The only reason to choose one action over another is because the preferred action will lead to more desirable outcomes.

An agent may have a simple goal, which is a state to be reached or a proposition to be true such as getting its owner a cup of coffee (i.e., end up in a state where she has coffee). Other agents may have more complex preferences. For example, a medical doctor may be expected to take into account suffering, life expectancy, quality of life, monetary costs (for the patient, the doctor, and society), the ability to justify decisions in case of a lawsuit, and many other desiderata. The doctor must trade these considerations off when they conflict, as they invariably do.

The **preference dimension** is whether the agent has

- **goals**, either **achievement goals** to be achieved in some final state or **maintenance goals** that must be maintained in all visited states. For example, the goals for a robot may be to get two cups of coffee and a banana, and not to make a mess or hurt anyone.
- **complex preferences** involve trade-offs among the desirability of various outcomes, perhaps at different times. An **ordinal preference** is where only the ordering of the preferences is important. A **cardinal preference** is where the magnitude of the values matters. For example, an ordinal preference may be that Sam prefers cappuccino over black coffee and prefers black coffee over tea. A cardinal preference may give a trade-off between the wait time and the type of beverage, and a mess-taste trade-off, where Sam is prepared to put up with more mess in the preparation of the coffee if the taste of the coffee is exceptionally good.

Goals are considered in Chapter 8. Complex preferences are considered in Chapter 9.

1.5.6 Number of Agents

An agent reasoning about what it should do in an environment where it is the only agent is difficult enough. However, reasoning about what to do when there are other agents who are also reasoning is much more difficult. An agent in a multiagent setting should reason strategically about other agents; the other agents may act to trick or manipulate the agent or may be available to cooperate with the agent. With multiple agents, it is often optimal to act randomly because other agents can exploit deterministic strategies. Even when the agents are co-operating and have a common goal, the problem of coordination and communication makes multiagent reasoning more challenging. However, many domains contain multiple agents and ignoring other agents' strategic reasoning is not always the best way for an agent to reason.

Taking the point of view of a single agent, the **number of agents dimension** considers whether the agent does

- **single agent** reasoning, where the agent assumes that any other agents are just part of the environment. This is a reasonable assumption if there are no other agents or if the other agents are not going to change what they do based on the agent's action.
- **multiple agent** reasoning, where the agent takes the reasoning of other agents into account. This happens when there are other intelligent agents whose goals or preferences depend, in part, on what the agent does or if the agent must communicate with other agents.

Reasoning in the presence of other agents is much more difficult if the agents can act simultaneously or if the environment is only partially observable. Multiagent systems are considered in Chapter 10.

1.5.7 Learning

In some cases, a designer of an agent may have a good model of the agent and its environment. Often a designer does not have a good model, and an agent should use data from its past experiences and other sources to help it decide what to do.

The **learning dimension** determines whether

- **knowledge is given** or
- **knowledge is learned** (from data or past experience).

Learning typically means finding the best model that fits the data. Sometimes this is as simple as tuning a fixed set of parameters, but it can also mean choosing the best representation out of a class of representations. Learning is a huge field in itself but does not stand in isolation from the rest of AI. There are many issues beyond fitting data, including how to incorporate background knowledge, what data to collect, how to represent the data and the resulting representations, what learning biases are appropriate, and how the learned knowledge can be used to affect how the agent acts.

Learning is considered in Chapters 7, 11, and 14.

1.5.8 Computational Limits

Sometimes an agent can decide on its best action quickly enough for it to act. Often there are computational resource limits that prevent an agent from carrying out the best action. That is, the agent may not be able to find the best action quickly enough within its memory limitations to act while that action is still the best thing to do. For example, it may not be much use to take 10 minutes to derive what was the best thing to do 10 minutes ago, when the agent has to act *now*. Often, instead, an agent must trade off how long it takes to get a solution with how good the solution is; it may be better to find a reasonable solution quickly than to find a better solution later because the world will have changed during the computation.

The **computational limits dimension** determines whether an agent has

- **perfect rationality**, where an agent reasons about the best action without taking into account its limited computational resources; or
- **bounded rationality**, where an agent decides on the best action that it can find given its computational limitations.

Computational resource limits include computation time, memory, and numerical accuracy caused by computers not representing real numbers exactly.

An **anytime algorithm** is an algorithm whose solution quality improves with time. In particular, it is one that can produce its current best solution at any time, but given more time it could produce even better solutions. We can ensure that the quality doesn't decrease by allowing the agent to store the best solution found so far and return that when asked for a solution. However, waiting to act

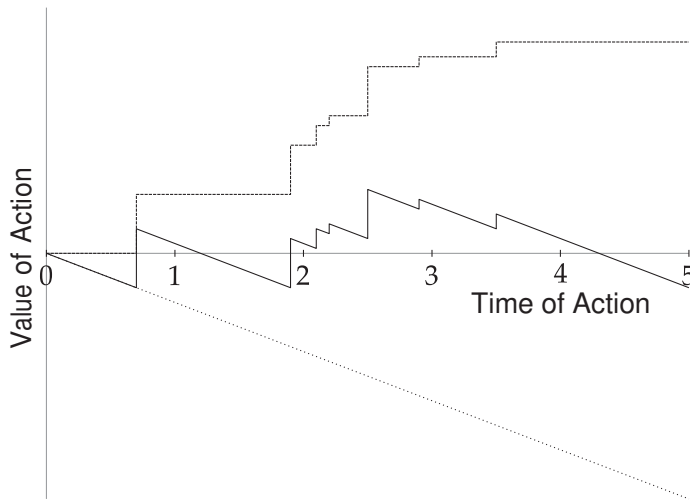


Figure 1.5: Solution quality as a function of time for an anytime algorithm. The agent has to choose an action. As time progresses, the agent can determine better actions. The value to the agent of the best action found so far, if it had been carried out initially, is given by the dashed line. The reduction in value to the agent by waiting to act is given by the dotted line. The net value to the agent, as a function of the time it acts, is given by the solid line.

has a cost; it may be better for an agent to act before it has found what would have been the best solution.

Example 1.9 Figure 1.5 shows how the computation time of an anytime algorithm can affect the solution quality. The agent has to carry out an action but can do some computation to decide what to do. The absolute solution quality, had the action been carried out at time zero, shown as the dashed line at the top, is improving as the agent takes time to reason. However, there is a penalty associated with taking time to act. In this figure, the penalty, shown as the dotted line at the bottom, is proportional to the time taken before the agent acts. These two values can be added to get the discounted quality, the time-dependent value of computation; this is the solid line in the middle of the graph. For the example of Figure 1.5, an agent should compute for about 2.5 time units, and then act, at which point the discounted quality achieves its maximum value. If the computation lasts for longer than 4.3 time units, the resulting discounted solution quality is worse than if the algorithm just outputs the initial guess it can produce with virtually no computation. It is typical that the solution quality improves in jumps; when the current best solution changes, there is a jump in the quality. However, the penalty associated with waiting is often not as simple as a straight line.

To take into account bounded rationality, an agent must decide whether it should act or think more. This is challenging because an agent typically does not know how much better off it would be if it only spent a little bit more time

Dimension	Values
Modularity	flat, modular, hierarchical
Representation scheme	states, features, relations
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Learning	knowledge is given, knowledge is learned
Number of agents	single agent, multiple agents
Computational limits	perfect rationality, bounded rationality

Figure 1.6: Dimensions of complexity

reasoning. Moreover, the time spent thinking about whether it should reason may detract from actually reasoning about the domain. However, bounded rationality can be the basis for approximate reasoning.

1.5.9 Interaction of the Dimensions

Figure 1.6 summarizes the dimensions of complexity. Unfortunately, we cannot study these dimensions independently because they interact in complex ways. Here we give some examples of the interactions.

The representation dimension interacts with the modularity dimension in that some modules in a hierarchy may be simple enough to reason in terms of a finite set of states, whereas other levels of abstraction may require reasoning about individuals and relations. For example, in a delivery robot, a module that maintains balance may only have a few states. A module that must prioritize the delivery of multiple parcels to multiple people may have to reason about multiple individuals (e.g., people, packages, and rooms) and the relations between them. At a higher level, a module that reasons about the activity over the day may only require a few states to cover the different phases of the day (e.g., there might be three states: busy time, available for requests, and recharge time).

The planning horizon interacts with the modularity dimension. For example, at a high level, a dog may be getting an immediate reward when it comes and gets a treat. At the level of deciding where to place its paws, there may be a long time until it gets the reward, and so at this level it may have to plan for an indefinite stage.

Sensing uncertainty probably has the greatest impact on the complexity of reasoning. It is much easier for an agent to reason when it knows the state of the world than when it doesn't. Although sensing uncertainty with states

is well understood, sensing uncertainty with individuals and relations is an active area of current research.

The effect uncertainty dimension interacts with the modularity dimension: at one level in a hierarchy, an action may be deterministic, whereas at another level, it may be stochastic. As an example, consider the result of flying to Paris with a companion you are trying to impress. At one level you may know where you are (in Paris); at a lower level, you may be quite lost and not know where you are on a map of the airport. At an even lower level responsible for maintaining balance, you may know where you are: you are standing on the ground. At the highest level, you may be very unsure whether you have impressed your companion.

Preference models interact with uncertainty because an agent must have a trade-off between satisfying a major goal with some probability or a less desirable goal with a higher probability. This issue is explored in Section 9.1 (page 373).

Multiple agents can also be used for modularity; one way to design a single agent is to build multiple interacting agents that share a common goal of making the higher-level agent act intelligently. Some researchers, such as [Minsky \[1986\]](#), argue that intelligence is an emergent feature from a “society” of unintelligent agents.

Learning is often cast in terms of learning with features – determining which feature values best predict the value of another feature. However, learning can also be carried out with individuals and relations. Much work has been done on learning hierarchies, learning in partially observable domains, and learning with multiple agents, although each of these is challenging in its own right without considering interactions with multiple dimensions.

Two of these dimensions, modularity and bounded rationality, promise to make reasoning more efficient. Although they make the formalism more complicated, breaking the system into smaller components, and making the approximations needed to act in a timely fashion and within memory limitations, should help build more complex systems.

1.6.1 A Diagnostic Assistant

A **diagnostic assistant** is intended to advise a human about some particular system such as a medical patient, the electrical system in a house, or an automobile. The diagnostic assistant should advise about potential underlying faults or diseases, what tests to carry out, and what treatment to prescribe. To give such advice, the assistant requires a model of the system, including knowledge of potential causes, available tests, and available treatments, and observations of the system (which are often called symptoms).

To be useful, the diagnostic assistant must provide added value, be easy for a human to use, and not be more trouble than it is worth. A diagnostic assistant connected to the Internet can draw on expertise from throughout the world, and its actions can be based on the most up-to-date research. However, it must be able to justify why the suggested diagnoses or actions are appropriate. Humans are, and should be, suspicious of computer systems that are opaque and

impenetrable. When humans are responsible for what they do, even if it is based on a computer system's advice, they should have reasonable justifications for the suggested actions.

In terms of the black box definition of an agent in Figure 1.3 (page 11), the diagnostic assistant has the following as inputs:

- prior knowledge, such as how switches and lights normally work, how diseases or malfunctions manifest themselves, what information tests provide, and the effects of repairs or treatments.
- past experience, in terms of data of previous cases that include the effects of repairs or treatments, the prevalence of faults or diseases, the prevalence of symptoms for these faults or diseases, and the accuracy of tests. These data are usually about similar artifacts or patients, rather than the actual one being diagnosed.
- goals of fixing the device and trade-offs, such as between fixing or replacing different components, or whether patients prefer to live longer if it means they will be in pain or be less coherent.
- observations of symptoms of a device or patient.

The output of the diagnostic assistant is in terms of recommendations of treatments and tests, along with a rationale for its recommendations.

Example 1.10 Figure 1.8 (on the next page) shows a depiction of an electrical distribution system in a house. In this house, power comes into the house through circuit breakers and then it goes to power outlets or to lights through light switches. For example, light l_1 is on if there is power coming into the house, if circuit breaker cb_1 is *on*, and if switches s_1 and s_2 are either both up or both down. This is the sort of model that normal householders may have of the electrical power in the house, and which they could use to determine what is

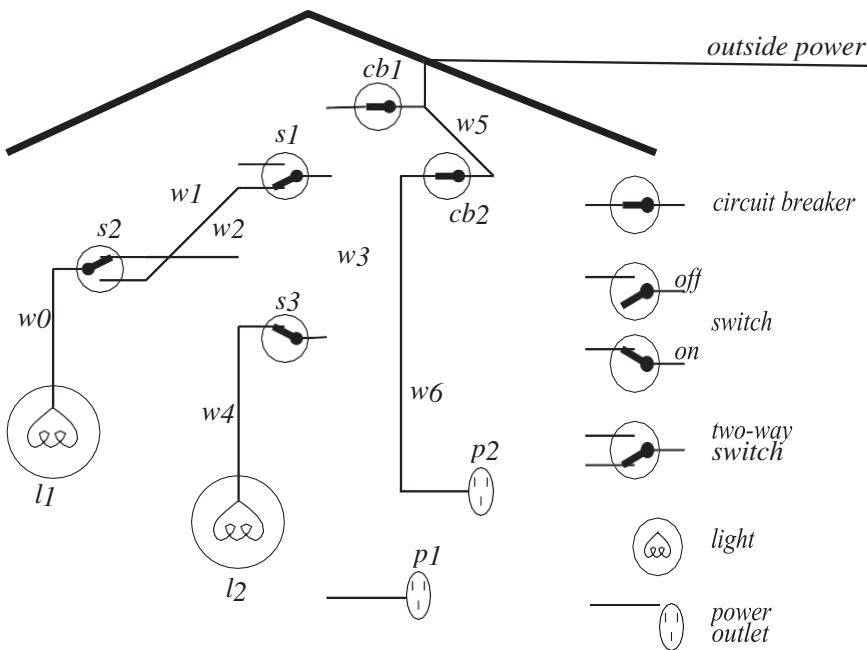


Figure 1.8: An electrical environment for the diagnostic assistant

wrong given evidence about the position of the switches and which lights are on and which are off. The diagnostic assistant is there to help a householder or an electrician troubleshoot electrical problems.

Each dimension is relevant to the diagnostic assistant:

- Hierarchical decomposition allows for very-high-level goals to be maintained while treating the lower-level causes and allows for detailed monitoring of the system. For example, in a medical domain, one module could take the output of a heart monitor and give higher-level observations such as notifying when there has been a change in the heart rate. Another module could take in this observation and other high-level observations and notice what other symptoms happen at the same time as a change in heart rate. In the electrical domain, Figure 1.8 is at one level of abstraction; a lower level could specify the voltages, how wires are spliced together, and the internals of switches.
- Most systems are too complicated to reason about in terms of the states, and so they are usually described in terms of the features or individual components and relations among them. For example, a human body may be described in terms of the values for features of its various components. Designers may want to model the dynamics without knowing the actual individuals. For example, designers of the electrical diagnosis system would model how lights and switches work before knowing which lights and switches exist in an actual house and, thus, before they know the features.

This can be achieved by modeling in terms of relations and their interaction and by adding the individual components when they become known.

- . It is possible to reason about a static system, such as reasoning about what could be wrong when a light is off given the position of switches. It is also possible to reason about a sequence of tests and treatments, where the agents keep testing and treating until the problem is fixed, or where the agent carries out ongoing monitoring of a system, continuously fixing whatever gets broken.
- . Sensing uncertainty is the fundamental problem that faces diagnosis. Diagnosis is required if an agent cannot directly observe the internals of the system.
- . Effect uncertainty also exists in that an agent may not know the outcome of a treatment and, often, treatments have unanticipated outcomes.
- . The goal may be as simple as “fix what is wrong,” but often there are complex trade-offs involving costs, pain, life expectancy, the probability that the diagnosis is correct, and the uncertainty as to efficacy and side effects of the treatment.
- . Although it is often a single-agent problem, diagnosis becomes more complicated when multiple experts are involved who perhaps have competing experience and models. There may be other patients with whom an agent must compete for resources (e.g., doctor’s time, surgery rooms).
- . Learning is fundamental to diagnosis. It is through learning that we understand the progression of diseases and how well treatments work or do not work. Diagnosis is a challenging domain for learning, because all patients are different, and each individual doctor’s experience is only with a few patients with any particular set of symptoms. Doctors also see a biased sample of the population; those who come to see them usually have unusual or painful symptoms.
- . Diagnosis often requires a quick response, which may not allow for the time to carry out exhaustive reasoning or perfect rationality.

1.6.2 An Intelligent Tutoring System

An **intelligent tutoring system** is a computer system that tutors students in some domain of study.

For example, in a tutoring system to teach elementary physics, such as mechanics, the system may present the theory and worked-out examples. The system can ask the student questions and it must be able to understand the student’s answers, as well as determine the student’s knowledge based on what answers were given. This should then affect what is presented and what other questions are asked of the student. The student can ask questions of the system, and so the system should be able to solve problems in the physics domain.

In terms of the black box definition of an agent in Figure 1.3 (page 11), an intelligent tutoring system has the following as inputs:

- prior knowledge, provided by the agent designer, about the subject matter being taught, teaching strategies, possible errors, and misconceptions of the students.
- past experience, which the tutoring system has acquired by interacting with students, about what errors students make, how many examples it takes to learn something, and what students forget. This can be information about students in general or about a particular student.
- preferences about the importance of each topic, the level of achievement of the student that is desired, and costs associated with usability. There are often complex trade-offs among these.
- observations of a student's test results and observations of the student's interaction (or non-interaction) with the system. Students can also ask questions or provide new examples with which they want help.

The output of the tutoring system is the information presented to the student, tests the students should take, answers to questions, and reports to parents and teachers.

Each dimension is relevant to the tutoring system:

- There should be both a hierarchical decomposition of the agent and a decomposition of the task of teaching. Students should be taught the basic skills before they can be taught higher-level concepts. The tutoring system has high-level teaching strategies, but, at a much lower level, it must design the details of concrete examples and specific questions for a test.
- A tutoring system may be able to reason in terms of the state of the student. However, it is more realistic to have multiple features for the student and the subject domain. A physics tutor may be able to reason in terms of features that are known at design time if the examples are fixed and it is only reasoning about one student. For more complicated cases, the tutoring system should refer to individuals and relations. If the tutoring system or the student can create examples with multiple individuals, the system may not know the features at design time and will have to reason in terms of individuals and relations.
- In terms of planning horizon, for the duration of a test, it may be reasonable to assume that the domain is static and that the student does not learn while taking a test. For some subtasks, a finite horizon may be appropriate. For example, there may be a teach, test, reteach sequence. For other cases, there may be an indefinite horizon where the system may not know at design time how many steps it will take until the student has mastered some concept. It may also be possible to model teaching as an ongoing process of learning and testing with appropriate breaks, with no expectation of the system finishing.
- Uncertainty will have to play a large role. The system cannot directly observe the knowledge of the student. All it has is some sensing input, based

on questions the student asks or does not ask, and test results. The system will not know for certain the effect of a particular teaching episode.

- Although it may be possible to have a simple goal such as to teach some particular concept, it is more likely that complex preferences must be taken into account. One reason is that, with uncertainty, there may be no way to guarantee that the student knows the concept being taught; any method that tries to maximize the probability that the student knows a concept will be very annoying, because it will continue to repeatedly teach and test if there is a slight chance that the student's errors are due to misunderstanding as opposed to fatigue or boredom. More complex preferences would enable a trade-off among fully teaching a concept, boring the student, the time taken, and the amount of retesting. The user may also have a preference for a teaching style that should be taken into account.
- It may be appropriate to treat this as a single-agent problem. However, the teacher, the student, and the parent may all have different preferences that must be taken into account. Each of these agents may act strategically by not telling the truth.
- We would expect the system to be able to learn about what teaching strategies work, how well some questions work at testing concepts, and what common mistakes students make. It could learn general knowledge, or knowledge particular to a topic (e.g., learning about what strategies work for teaching mechanics) or knowledge about a particular student, such as learning what works for Sam.
- One could imagine that choosing the most appropriate material to present would take a lot of computation time. However, the student must be responded to in a timely fashion. Bounded rationality would play a part in ensuring that the system does not compute for a long time while the student is waiting.

1.6.3 A Trading Agent

A **trading agent** is like a robot, but instead of interacting with a physical environment, it interacts with an information environment. Its task is to procure goods and services for a user. It must be able to be told the needs of a user, and it must interact with sellers (e.g., on the Web). The simplest trading agent involves proxy bidding for a user on an auction site, where the system will keep bidding until the user's price limit is reached. A more complicated trading agent will buy multiple complementary items, like booking a flight, a hotel, and a rental car that fit together, in addition to trading off competing preferences. Another example of a trading agent is one that monitors how much food and groceries are in a household, monitors the prices, and orders goods before they are needed, keeping costs to a minimum.

In terms of the black box definition of an agent in Figure 1.3 (page 11), the trading agent has the following as inputs:

- prior knowledge about types of goods and services, selling practices, and how auctions work;

- past experience about where is the best place to look for specials, how prices vary in time in an auction, and when specials tend to turn up;
- preferences in terms of what the user wants and how to trade off competing goals; and
- observations about what items are available, their price, and, perhaps, how long they are available.

The output of the trading agent is either a proposal to the user that they can accept or reject or an actual purchase.

The trading agent should take all of the dimensions into account:

- Hierarchical decomposition is essential because of the complexity of domains. Consider the problem of making all of the arrangements and purchases for a custom holiday for a traveler. It is simpler to have a module that can purchase a ticket and optimize connections and timing, rather than to do this at the same time as determining what doors to go through to get to the taxi stand.
- The state space of the trading agent is too large to reason in terms of individual states. There are also too many individuals to reason in terms of features. The trading agent will have to reason in terms of individuals such as customers, days, hotels, flights, and so on.
- A trading agent typically does not make just one purchase, but must make a sequence of purchases, often a large number of sequential decisions (e.g., purchasing one hotel room may require booking ground transportation, which may in turn require baggage storage), and often plans for ongoing purchasing, such as for an agent that makes sure a household has enough food on hand at all times.
- There is often sensing uncertainty in that a trading agent does not know all of the available options and their availability, but must find out information that can become old quickly (e.g., if some hotel becomes booked up). A travel agent does not know if a flight will be canceled or delayed or whether the passenger's luggage will be lost. This uncertainty means that the agent must plan for the unanticipated.
- There is also effect uncertainty in that the agent does not know if an attempted purchase will succeed.
- Complex preferences are at the core of the trading agent. The main problem is in allowing users to specify what they want. The preferences of users are typically in terms of functionality, not components. For example, typical computer buyers have no idea of what hardware to buy, but they know what functionality they want and they also want the flexibility to be able to use new features that might not yet exist. Similarly, in a travel domain, what activities a user may want may depend on the location. Users also may want the ability to participate in a local custom at their destination, even though they may not know what these customs are.
- A trading agent has to reason about other agents. In commerce, prices are governed by supply and demand; this means that it is important to reason about the other competing agents. This happens particularly in a world

1.6. Prototypical Applications	39
1.7. Overview of the Book	39

where many items are sold by auction. Such reasoning becomes particularly difficult when there are items that must complement each other, such as flights and hotel booking, and items that can substitute for each other, such as bus transport or taxis.

- A trading agent should learn about what items sell quickly, which of the suppliers are reliable, where to find good deals, and what unanticipated events may occur.
- A trading agent faces severe communication limitations. In the time between finding that some item is available and coordinating the item with other items, the item may have sold out. This can sometimes be alleviated by sellers agreeing to hold some items (not to sell them to someone else in the meantime), but sellers will not be prepared to hold an item indefinitely if others want to buy it.

Because of the personalized nature of the trading agent, it should be able to do better than a generic purchaser that, for example, only offers packaged tours.

1.7 References and Further Reading

The ideas in this chapter have been derived from many sources. Here, we will try to acknowledge those that are explicitly attributable to particular authors. Most of the other ideas are part of AI folklore; trying to attribute them to any- one would be impossible.

[Haugeland \[1997\]](#) contains a good collection of articles on the philosophy behind artificial intelligence, including that classic paper of [Turing \[1950\]](#) that proposes the Turing test. [Cohen \[2005\]](#) gives a recent discussion of the Turing test.

Nilsson [2009] gives a detailed description of the history of AI. Chrisley and Begeer [2000] present many classic papers on AI.

The physical symbol system hypothesis was posited by Newell and Simon [1976]. See also Simon [1996], who discusses the role of symbol systems in a multidisciplinary context. The distinctions between real, synthetic, and artificial intelligence are discussed by Haugeland [1985], who also provides useful introductory material on interpreted, automatic formal symbol systems and the Church–Turing thesis. For a critique of the symbol-system hypothesis see Brooks [1990] and Winograd [1990]. Nilsson [2007] evaluates the hypothesis in terms of recent criticisms.

The use of anytime algorithms is due to Horvitz [1989] and Boddy and Dean [1994]. See Dean and Wellman [1991, Chapter 8], Zilberstein [1996], and Russell [1997] for introductions to bounded rationality.

For discussions on the foundations of AI and the breadth of research in AI see Kirsh [1991a], Bobrow [1993], and the papers in the corresponding volumes, as well as Schank [1990] and Simon [1995]. The importance of knowledge in AI is discussed in Lenat and Feigenbaum [1991] and Smith [1991].

For overviews of cognitive science and the role that AI and other disciplines play in that field, see Gardner [1985], Posner [1989], and Stillings, Feinstein, Garfield, Rissland, Rosenbaum, Weisler, and Baker-Ward [1987].

Purchasing agents can become very complex. Sandholm [2007] describes how AI can be used for procurement of multiple goods with complex preferences.

A number of AI texts are valuable as reference books complementary to this book, providing a different perspective on AI. In particular, Russell and Norvig [2010] give a more encyclopedic overview of AI and provide a complementary source for many of the topics covered in this book. They provide an outstanding review of the scientific literature, which we do not try to duplicate. The *Encyclopedia of Artificial Intelligence* [Shapiro, 1992] is an encyclopedic reference on AI written by leaders in the field and still provides background on some of the classic topics. There are also a number of collections of classic research papers. The general collections of most interest to readers of this book include Webber and Nilsson [1981] and Brachman and Levesque [1985]. More specific collections are given in the appropriate chapters.

The Association for the Advancement of Artificial Intelligence (AAAI) provides introductory material and news at their *AI Topics* web site (<http://www.aaai.org/AITopics/html/welcome.html>). *AI Magazine*, published by AAAI, often has excellent overview articles and descriptions of particular applications. *IEEE Intelligent Systems* also provides accessible articles on AI research.

There are many journals that provide in-depth research contributions and conferences where the most up-to-date research is found. These include the journals *Artificial Intelligence*, the *Journal of Artificial Intelligence Research*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, and *Computational Intelligence*, as well as more specialized journals such as *Neural Computation*, *Computational Linguistics*, *Machine Learning*, the *Journal of Automated Reasoning*, the

Journal of Approximate Reasoning, *IEEE Transactions on Robotics and Automation*, and the *Theory and Practice of Logic Programming*. Most of the cutting-edge research is published first in conferences. Those of most interest to a general audience are the biennial International Joint Conference on Artificial Intelligence (IJCAI), the AAAI Annual Conference, the European Conference on AI (ECAI), the Pacific Rim International Conference on AI (PRICAI), various national conferences, and many specialized conferences and workshops.

Agent Architectures and Hierarchical Control

By a hierarchic system, or hierarchy, I mean a system that is composed of interrelated subsystems, each of the latter being in turn hierarchic in structure until we reach some lowest level of elementary subsystem. In most systems of nature it is somewhat arbitrary as to where we leave off the partitioning and what subsystems we take as elementary. Physics makes much use of the concept of “elementary particle,” although the particles have a disconcerting tendency not to remain elementary very long ...

Empirically a large proportion of the complex systems we observe in nature exhibit hierarchic structure. On theoretical grounds we would expect complex systems to be hierarchies in a world in which complexity had to evolve from simplicity.

– Herbert A. [Simon](#) [1996]

This chapter discusses how an intelligent agent can perceive, reason, and act over time in an environment. In particular, it considers the internal structure of an agent. As Simon points out in the quote above, hierarchical decomposition is an important part of the design of complex systems such as intelligent agents. This chapter presents ways to design agents in terms of hierarchical decompositions and ways that agents can be built, taking into account the knowledge that an agent needs to act intelligently.

2.1 Agents

An **agent** is something that acts in an environment. An agent can, for example, be a person, a robot, a dog, a worm, the wind, gravity, a lamp, or a computer program that buys and sells.

Purposive agents have preferences. They prefer some states of the world to other states, and they act to try to achieve the states they prefer most. The non-purposive agents are grouped together and called **nature**. Whether or not an agent is purposive is a modeling assumption that may, or may not, be appropriate. For example, for some applications it may be appropriate to model a dog as purposive, and for others it may suffice to model a dog as non-purposive.

If an agent does not have preferences, by definition it does not care what world state it ends up in, and so it does not matter what it does. The only reason to design an agent is to instill it with preferences – to make it prefer some world states and try to achieve them. An agent does not have to know its preferences. For example, a thermostat is an agent that senses the world and turns a heater either on or off. There are preferences embedded in the thermostat, such as to keep the occupants of a room at a pleasant temperature, even though the thermostat arguably does not know these are its preferences. The preferences of an agent are often the preferences of the designer of the agent, but sometimes an agent can be given goals and preferences at run time.

Agents interact with the environment with a **body**. An **embodied** agent has a physical body. A **robot** is an artificial purposive embodied agent. Sometimes agents that act only in an information space are called robots, but we just refer to those as agents.

This chapter considers how to build purposive agents. We use robots as a main motivating example, because much of the work has been carried out in the context of robotics and much of the terminology is from robotics. However, the discussion is intended to cover all agents.

Agents receive information through their **sensors**. An agent's actions depend on the information it receives from its sensors. These sensors may, or may not, reflect what is true in the world. Sensors can be noisy, unreliable, or broken, and even when sensors are reliable there is still ambiguity about the world based on sensor readings. An agent must act on the information it has available. Often this information is very weak, for example, "sensor s appears to be producing value v ."

Agents act in the world through their **actuators** (also called **effectors**). Actuators can also be noisy, unreliable, slow, or broken. What an agent controls is the message (command) it sends to its actuators. Agents often carry out actions to find more information about the world, such as opening a cupboard door to find out what is in the cupboard or giving students a test to determine their knowledge.

2.2 Agent Systems

Figure 2.1 depicts the general interaction between an agent and its environment. Together the whole system is known as an agent system.

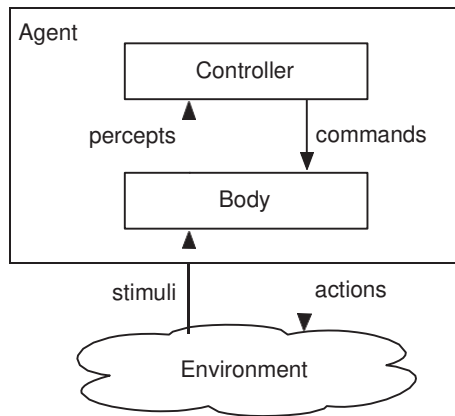


Figure 2.1: An agent system and its components

An **agent system** is made up of an agent and its environment. The agent receives **stimuli** from the environment and carries out **actions** in the environment.

An **agent** is made up of a **body** and a **controller**. The controller receives **percepts** from the body and sends **commands** to the body.

A body includes **sensors** that convert stimuli into percepts and **actuators** that convert commands into actions.

Stimuli include light, sound, words typed on a keyboard, mouse movements, and physical bumps. The stimuli can also include information obtained from a web page or from a database.

Common sensors include touch sensors, cameras, infrared sensors, sonar, microphones, keyboards, mice, and XML readers used to extract information from web pages. As a prototypical sensor, a camera senses light coming into its lens and converts it into a two-dimensional array of intensity values called **pixels**. Sometimes multiple pixel arrays exist for different colors or for multiple cameras. Such pixel arrays could be the percepts for our controller. More often, percepts consist of higher-level features such as lines, edges, and depth information. Often the percepts are more specialized – for example, the positions of bright orange dots, the part of the display a student is looking at, or the hand signals given by a human.

Actions include steering, accelerating wheels, moving links of arms, speaking, displaying information, or sending a post command to a web site. Commands include low-level commands such as to set the voltage of a motor to some particular value, and high-level specifications of the desired motion of a robot, such as “stop” or “travel at 1 meter per second due east” or “go to room 103.” Actuators, like sensors, are typically noisy. For example, stopping takes time; a robot is governed by the laws of physics and has momentum, and messages take time to travel. The robot may end up going only approximately

1 meter per second, approximately east, and both speed and direction may fluctuate. Even traveling to a particular room may fail for a number of reasons.

The controller is the brain of the agent. The rest of this chapter is about how to build controllers.

2.2.1 The Agent Function

Agents are situated in time: they receive sensory data in time and do actions in time. The action that an agent does at a particular time is a function of its inputs (page 10). We first consider the notion of time.

Let T be the set of **time** points. Assume that T is totally ordered and has some metric that can be used to measure the temporal distance between any two time points. Basically, we assume that T can be mapped to some subset of the real line.

T is **discrete** if there exist only a finite number of time points between any two time points; for example, there is a time point every hundredth of a second, or every day, or there may be time points whenever interesting events occur. T is **dense** if there is always another time point between any two time points; this implies there must be infinitely many time points between any two points. Discrete time has the property that, for all times, except perhaps a last time, there is always a next time. Dense time does not have a “next time.” Initially, we assume that time is discrete and goes on forever. Thus, for each time there is a next time. We write $t + 1$ to be the next time after time t ; it does not mean that the time points are equally spaced.

Assume that T has a starting point, which we arbitrarily call 0.

Suppose P is the set of all possible percepts. A **percept trace**, or **percept stream**, is a function from T into P . It specifies what is observed at each time.

Suppose C is the set of all commands. A **command trace** is a function from T into C . It specifies the command for each time point.

Example 2.1 Consider a household trading agent that monitors the price of some commodity (e.g., it checks online for special deals and for price increases for toilet paper) and how much the household has in stock. It must decide whether to buy more and how much to buy. The percepts are the price and the amount in stock. The command is the number of units the agent decides to buy (which is zero if the agent does not buy any). A percept trace specifies for each time point (e.g., each day) the price at that time and the amount in stock at that time. Percept traces are given in Figure 2.2. A command trace specifies how much the agent decides to buy at each time point. An example command trace is given in Figure 2.3.

The action of actually buying depends on the command but may be different. For example, the agent could issue a command to buy 12 rolls of toilet paper at a particular price. This does not mean that the agent actually buys 12 rolls because there could be communication problems, the store could have run out of toilet paper, or the price could change between deciding to buy and actually buying.

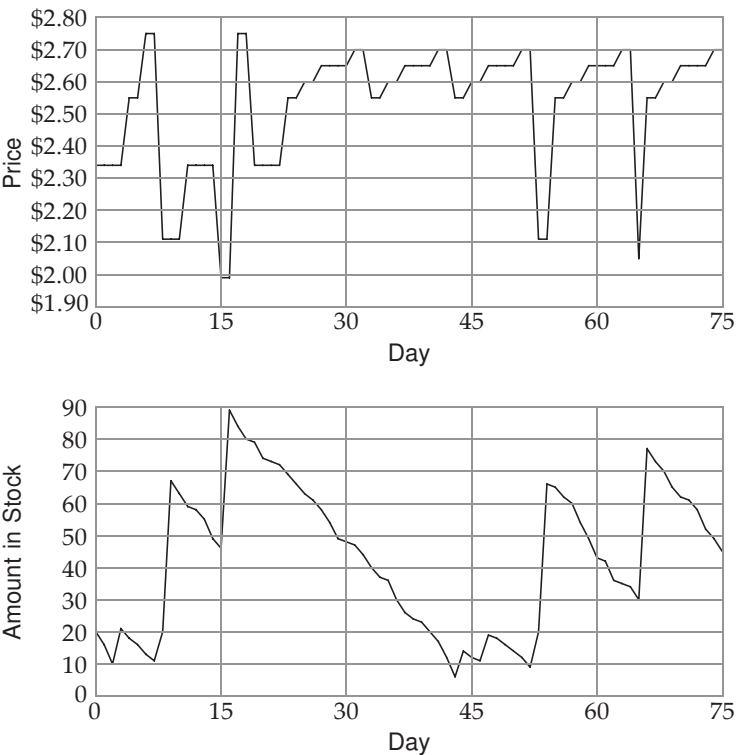


Figure 2.2: Percept traces for Example 2.1

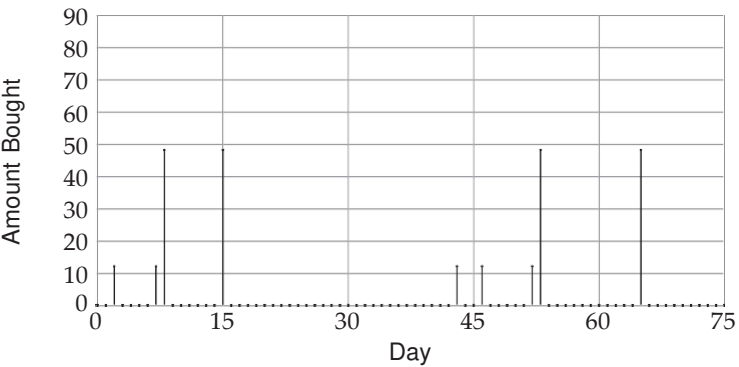


Figure 2.3: Command trace for Example 2.1

A percept trace for an agent is thus the sequence of all past, present, and future percepts received by the controller. A command trace is the sequence of all past, present, and future commands issued by the controller. The commands can be a function of the history of percepts. This gives rise to the concept of a **transduction**, a function that maps percept traces into command traces.

Because all agents are situated in time, an agent cannot actually observe full percept traces; at any time it has only experienced the part of the trace up to *now*. It can only observe the value of the trace at time t when it gets to time t . Its command can only depend on what it has experienced.

A transduction is **causal** if, for all times t , the command at time t depends only on percepts up to and including time t . The causality restriction is needed because agents are situated in time; their command at time t cannot depend on percepts after time t .

A **controller** is an implementation of a causal transduction.

The **history** of an agent at time t is the percept trace of the agent for all times before or at time t and the command trace of the agent before time t .

Thus, a **causal transduction** specifies a function from the agent's history at time t into the command at time t . It can be seen as the most general specification of an agent.

Example 2.2 Continuing Example 2.1 (page 46), a causal transduction specifies, for each time, how much of the commodity the agent should buy depending on the price history, the history of how much of the commodity is in stock (including the current price and amount in stock) and the past history of buying.

An example of a causal transduction is as follows: buy four dozen rolls if there are fewer than five dozen in stock and the price is less than 90% of the average price over the last 20 days; buy a dozen more rolls if there are fewer than a dozen in stock; otherwise, do not buy any.

Although a causal transduction is a function of an agent's history, it cannot be directly implemented because an agent does not have direct access to its entire history. It has access only to its current percepts and what it has remembered.

The **belief state** of an agent at time t is all of the information the agent has remembered from the previous times. An agent has access only to its history that it has encoded in its belief state. Thus, the belief state encapsulates all of the information about its history that the agent can use for current and future commands. At any time, an agent has access to its belief state and its percepts. The belief state can contain any information, subject to the agent's memory and processing limitations. This is a very general notion of belief; sometimes we use a more specific notion of belief, such as the agent's belief about what is true in the world, the agent's beliefs about the dynamics of the environment, or the agent's belief about what it will do in the future.

Some instances of belief state include the following:

- The belief state for an agent that is following a fixed sequence of instructions may be a program counter that records its current position in the sequence.

- The belief state can contain specific facts that are useful – for example, where the delivery robot left the parcel in order to go and get the key, or where it has already checked for the key. It may be useful for the agent to remember anything that is reasonably stable and that cannot be immediately observed.
- The belief state could encode a model or a partial model of the state of the world. An agent could maintain its best guess about the current state of the world or could have a probability distribution over possible world states; see Section 5.6 (page 199) and Chapter 6.
- The belief state could be a representation of the dynamics of the world and the meaning of its percepts, and the agent could use its perception to determine what is true in the world.
- The belief state could encode what the agent **desires**, the **goals** it still has to achieve, its **beliefs** about the state of the world, and its **intentions**, or the steps it intends to take to achieve its goals. These can be maintained as the agent acts and observes the world, for example, removing achieved goals and replacing intentions when more appropriate steps are found.

A controller must maintain the agent's belief state and determine what command to issue at each time. The information it has available when it must do this includes its belief state and its current percepts.

A **belief state transition function** for discrete time is a function

$$remember : S \times P \rightarrow S$$

where S is the set of belief states and P is the set of possible percepts; $s_{t+1} = remember(s_t, p_t)$ means that s_{t+1} is the belief state following belief state s_t when p_t is observed.

A **command function** is a function

$$do : S \times P \rightarrow C$$

where S is the set of belief states, P is the set of possible percepts, and C is the set of possible commands; $c_t = do(s_t, p_t)$ means that the controller issues command c_t when the belief state is s_t and when p_t is observed.

The belief-state transition function and the command function together specify a causal transduction for the agent. Note that a causal transduction is a function of the agent's history, which the agent doesn't necessarily have access to, but a command function is a function of the agent's belief state and percepts, which it does have access to.

Example 2.3 To implement the causal transduction of Example 2.2, a controller must keep track of the rolling history of the prices for the previous 20 days. By keeping track of the average (*ave*), it can update the average using

$$ave := ave + \frac{new - old}{20}$$

where *new* is the new price and *old* is the oldest price remembered. It can then discard *old*. It must do something special for the first 20 days.

A simpler controller could, instead of remembering a rolling history in order to maintain the average, remember just the average and use the average as a surrogate for the oldest item. The belief state can then contain one real number (*ave*). The state transition function to update the average could be

$$ave := ave + \frac{new - ave}{20}$$

This controller is much easier to implement and is not sensitive to what happened 20 time units ago. This way of maintaining estimates of averages is the basis for temporal differences in reinforcement learning (page 467).

If there exists a finite number of possible belief states, the controller is called a **finite state controller** or a **finite state machine**. A **factored representation** is one in which the belief states, percepts, or commands are defined by features (page 21). If there exists a finite number of features, and each feature can only have a finite number of possible values, the controller is a **factored finite state machine**. Richer controllers can be built using an unbounded number of values or an unbounded number of features. A controller that has countably many states can compute anything that is computable by a Turing machine.

2.3 Hierarchical Control

One way that you could imagine building an agent depicted in Figure 2.1 (page 45) is to split the body into the sensors and a complex perception system that feeds a description of the world into a reasoning engine implementing a controller that, in turn, outputs commands to actuators. This turns out to be a bad architecture for intelligent systems. It is too slow, and it is difficult to reconcile the slow reasoning about complex, high-level goals with the fast reaction that an agent needs, for example, to avoid obstacles. It also is not clear that there is a description of a world that is independent of what you do with it (see Exercise 1 (page 66)).

An alternative architecture is a hierarchy of controllers as depicted in Figure 2.4. Each layer sees the layers below it as a **virtual body** from which it gets percepts and to which it sends commands. The lower-level layers are able to run much faster, react to those aspects of the world that need to be reacted to quickly, and deliver a simpler view of the world to the higher layers, hiding inessential information.

In general, there can be multiple features passed from layer to layer and between states at different times.

There are three types of inputs to each layer at each time:

- the features that come from the belief state, which are referred to as the remembered or previous values of these features;
- the features representing the percepts from the layer below in the hierarchy; and
- the features representing the commands from the layer above in the hierarchy.

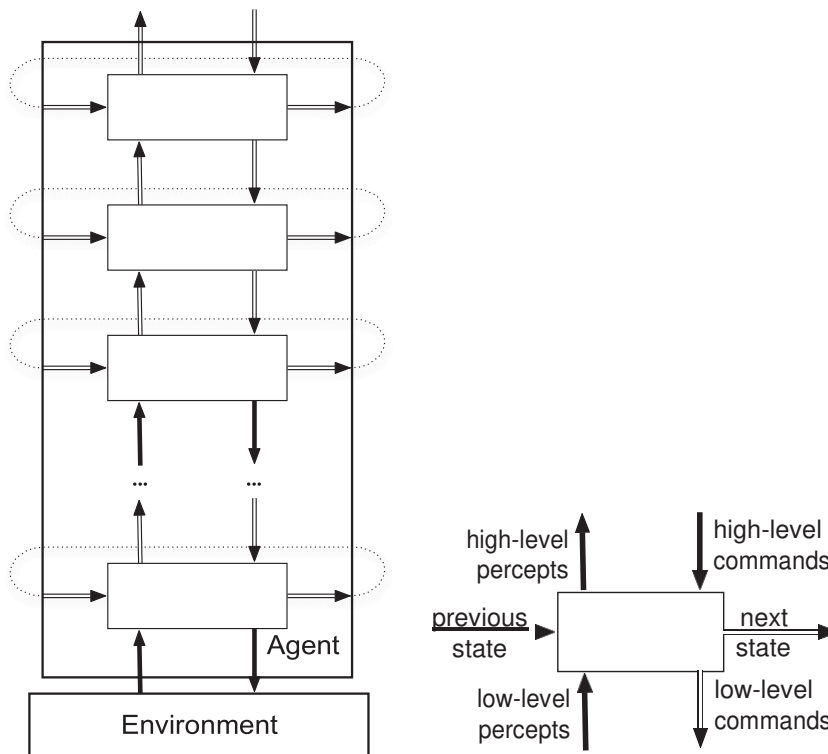


Figure 2.4: An idealized hierarchical agent system architecture. The unlabeled rectangles represent layers, and the double lines represent information flow. The dotted lines show how the output at one time is the input for the next time.

There are three types of outputs from each layer at each time:

- the higher-level percepts for the layer above,
- the lower-level commands for the layer below, and
- the next values for the belief-state features.

An implementation of a layer specifies how the outputs of a layer are a function of its inputs. Computing this function can involve arbitrary computation, but the goal is to keep each layer as simple as possible.

To implement a controller, each input to a layer must get its value from somewhere. Each percept or command input should be connected to an output of some other layer. Other inputs come from the remembered beliefs. The outputs of a layer do not have to be connected to anything, or they could be connected to multiple inputs.

High-level reasoning, as carried out in the higher layers, is often discrete and qualitative, whereas low-level reasoning, as carried out in the lower layers, is often continuous and quantitative (see box on page 52). A controller that reasons in terms of both discrete and continuous values is called a **hybrid system**.

Methodology

Qualitative Versus Quantitative Representations

Much of science and engineering considers **quantitative reasoning** with numerical quantities, using differential and integral calculus as the main tools. **Qualitative reasoning** is reasoning, often using logic, about qualitative distinctions rather than numerical values for given parameters.

Qualitative reasoning is important for a number of reasons:

- . An agent may not know what the exact values are. For example, for the delivery robot to pour coffee, it may not be able to compute the optimal angle that the coffee pot needs to be tilted, but a simple control rule may suffice to fill the cup to a suitable level.
- . The reasoning may be applicable regardless of the quantitative values. For example, you may want a strategy for a robot that works regardless of what loads are placed on the robot, how slippery the floors are, or what the actual charge is of the batteries, as long as they are within some normal operating ranges.
- . An agent needs to do qualitative reasoning to determine which quantitative laws are applicable. For example, if the delivery robot is filling a coffee cup, different quantitative formulas are appropriate to determine where the coffee goes when the coffee pot is not tilted enough for coffee to come out, when coffee comes out into a non-full cup, and when the coffee cup is full and the coffee is soaking into the carpet.

Qualitative reasoning uses discrete values, which can take a number of forms:

- . **Landmarks** are values that make qualitative distinctions in the individual being modeled. In the coffee example, some important qualitative distinctions include whether the coffee cup is empty, partially full, or full. These landmark values are all that is needed to predict what happens if the cup is tipped upside down or if coffee is poured into the cup.
- . **Orders-of-magnitude reasoning** involves approximate reasoning that ignores minor distinctions. For example, a partially full coffee cup may be full enough to deliver, half empty, or nearly empty. These **fuzzy terms** have ill-defined borders. Some relationship exists between the actual amount of coffee in the cup and the qualitative description, but there may not be strict numerical divisors.
- . **Qualitative derivatives** indicate whether some value is increasing, decreasing, or staying the same.

A flexible agent needs to do qualitative reasoning before it does quantitative reasoning. Sometimes qualitative reasoning is all that is needed. Thus, an agent does not always need to do quantitative reasoning, but sometimes it needs to do both qualitative and quantitative reasoning.

Example 2.4 Consider a delivery robot (page 32) able to carry out high-level navigation tasks while avoiding obstacles. Suppose the delivery robot is required to visit a sequence of named locations in the environment of Figure 1.7 (page 32), avoiding obstacles it may encounter.

Assume the delivery robot has wheels like a car, and at each time can either go straight, turn right, or turn left. It cannot stop. The velocity is constant and the only command is to set the steering angle. Turning the wheels is instantaneous, but adjusting to a certain direction takes time. Thus, the robot can only travel straight ahead or go around in circular arcs with a fixed radius.

The robot has a position sensor that gives its current coordinates and orientation. It has a single whisker sensor that sticks out in front and slightly to the right and detects when it has hit an obstacle. In the example below, the whisker points 30° to the right of the direction the robot is facing. The robot does not have a map, and the environment can change (e.g., obstacles can move).

A layered controller for such a delivery robot is depicted in Figure 2.5 (on the next page). The robot is given a high-level plan to execute. The plan is a sequence of named locations to visit in order. The robot needs to sense the world and to move in the world in order to carry out the plan. The details of the lower layer are not shown in this figure.

The top layer, called *follow plan*, is described in Example 2.6 (page 56). That layer takes in a plan to execute. The plan is a list of named locations to visit in order. The locations are selected in order. Each selected location becomes the current target. This layer determines the x - y coordinates of the target. These coordinates are the target position for the lower level. The upper level knows about the names of locations, but the lower levels only know about coordinates.

The top layer maintains a belief state consisting of a list of names of locations that the robot still needs to visit and the coordinates of the current target. It issues commands to the middle layer in terms of the coordinates of the current target.

The middle layer, which could be called *go to target and avoid obstacles*, tries to keep traveling toward the current target position, avoiding obstacles. The middle layer is described in Example 2.5 (page 55). The target position, *target pos*, is obtained from the top layer. When the middle layer has arrived at the target position, it signals to the top layer that it has achieved the target by setting *arrived* to be true. This signal can be implemented either as the middle layer issuing an interrupt to the top layer, which was waiting, or as the top layer continually monitoring the middle layer to determine when *arrived* becomes true. When *arrived* becomes true, the top layer then changes the target position to the coordinates of the next location on the plan. Because the top layer changes the current target position, the middle layer must use the *previous* target position to determine whether it has arrived. Thus, the middle layer must get both the current and the previous target positions from the top layer: the previous target position to determine whether it has arrived, and the current target position to travel to.

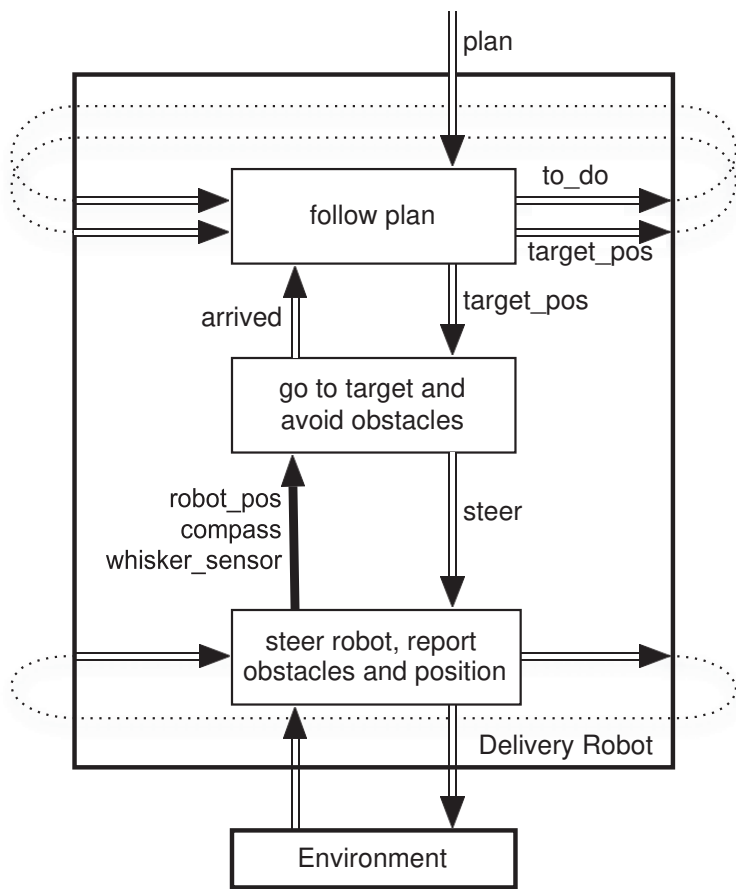


Figure 2.5: A hierarchical decomposition of the delivery robot

The middle layer can access the robot’s current position and direction and can determine whether its single whisker sensor is on or off. It can use a simple strategy of trying to head toward the target unless it is blocked, in which case it turns left.

The middle layer is built on a lower layer that provides a simple view of the robot. This lower layer could be called *steer robot and report obstacles and position*. It takes in steering commands and reports the robot’s position, orientation, and whether the sensor is on or off.

Inside a layer are features that can be functions of other features and of the inputs to the layers. There is an arc into a feature from the features or inputs on which it is dependent. The graph of how features depend on each other must be acyclic. The acyclicity of the graph allows the controller to be implemented by running a program that assigns the values in order. The features that make up the belief state can be written to and read from memory.

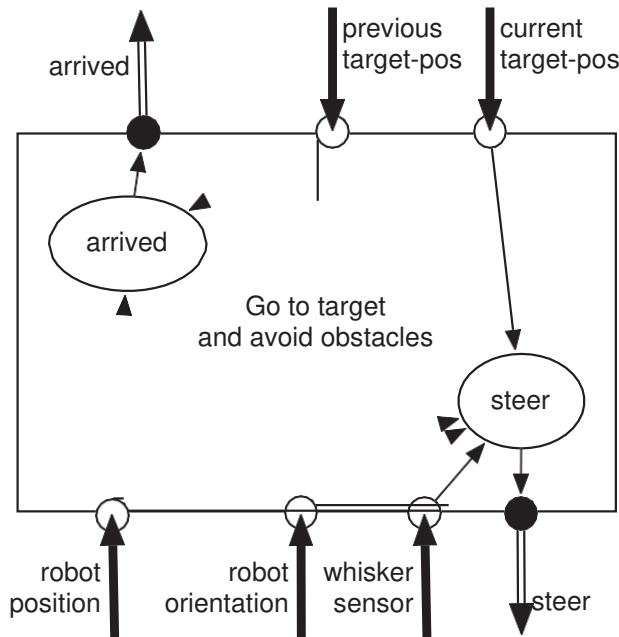


Figure 2.6: The middle layer of the delivery robot

Example 2.5 The middle *go to location and avoid obstacles* layer steers the robot to avoid obstacles. The inputs and outputs of this layer are given in Figure 2.6.

The robot has a single whisker sensor that detects obstacles touching the whisker. The one bit value that specifies whether the whisker sensor has hit an obstacle is provided by the lower layer. The lower layer also provides the robot position and orientation. All the robot can do is steer left by a fixed angle, steer right, or go straight. The aim of this layer is to make the robot head toward its current target position, avoiding obstacles in the process, and to report when it has arrived.

This layer of the controller maintains no internal belief state, so the belief state transition function is vacuous. The command function specifies the robot's steering direction as a function of its inputs and whether the robot has arrived.

The robot has arrived if its current position is close to the previous target position. Thus, *arrived* is assigned a value that is a function of the robot position and previous target position, and a threshold constant:

$$arrived := distance(previous\ target\ pos, robot\ pos) < threshold$$

where $:=$ means assignment, *distance* is the Euclidean distance, and *threshold* is a distance in the appropriate units.

The robot steers left if the whisker sensor is on; otherwise it heads toward the target position. This can be achieved by assigning the appropriate value to

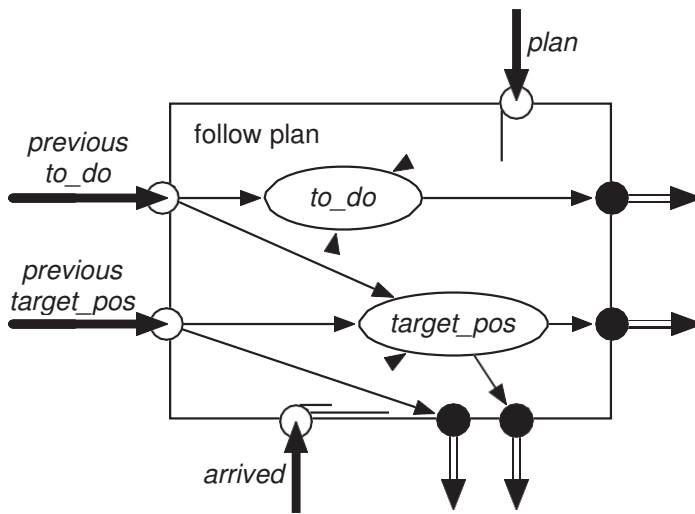


Figure 2.7: The top layer of the delivery robot controller

the *steer* variable:

```

if whisker_sensor = on
  then steer := left
else if straight_ahead(robot_pos, robot_dir, current_target_pos)
  then steer := straight
else if left_of(robot_position, robot_dir, current_target_pos)
  then steer := left
else steer := right
end if

```

where *straight_ahead*(*robot_pos*, *robot_dir*, *current_target_pos*) is true when the robot is at *robot_pos*, facing the direction *robot_dir*, and when the current target position, *current_target_pos*, is straight ahead of the robot with some threshold (for later examples, this threshold is 11° of straight ahead). The function *left_of* tests if the target is to the left of the robot.

This layer is purely quantitative. It reasons in terms of numerical quantities rather than discrete values.

Example 2.6 The top layer, *follow plan*, is given a plan – a list of named locations to visit in order. These are the kinds of targets that could be produced by a planner, such as those developed in Chapter 8. The top layer is also told when the robot has arrived at the previous target. It must output target coordinates to the middle layer, and remember what it needs to carry out the plan. The layer is shown in Figure 2.7.

This layer maintains an internal belief state. It remembers the current target position and what locations it still has to visit. The *to do* feature has as its value a

list of all pending locations to visit. The *target_pos* feature maintains the position for the current target.

Once the robot has arrived at its previous target, the next target position is the coordinate of the next location to visit. The top-level plan given to the robot is in terms of named locations, so these must be translated into coordinates for the middle layer to use. The following code shows how the target position and the *to do* list are changed when the robot has arrived at its previous target position:

```

if arrived and not empty(to_do)
  then
    target_post := coordinates(head(to_do))
    to_dot := tail(to_do)
  end if

```

where to_do^t is the next value for the *to do* feature, and $target\_pos^t$ is the next target position. Here $head(to\_do)$ is the first element of the *to do* list, $tail(to\_do)$ is the rest of the *to do* list, and $empty(to\_do)$ is true when the *to do* list is empty.

In this layer, if the *to do* list becomes empty, the robot does not change its target position. It keeps going around in circles. See Exercise 2.3 (page 67).

This layer determines the coordinates of the named locations. This could be done by simply having a database that specifies the coordinates of the locations. Using such a database is sensible if the locations do not move and are known a priori. However, if the locations can move, the lower layer must be able to tell the upper layer the current position of a location. The top layer would have to ask the lower layer the coordinates of a given location. See Exercise 2.8 (page 68).

To complete the controller, the belief state variables must be initialized, and the top-level plan must be input. This can be done by initializing the *to do* list with the tail of the plan and the *target_pos* with the location of the first location.

A simulation of the plan `[goto(o109), goto(storage), goto(o109), goto(o103)]` with one obstacle is given in Figure 2.8 (on the next page). The robot starts at position (0, 5) facing 90° (north), and there is a rectangular obstacle between the positions (20, 20) and (35, -5).

2.3.1 Agents Modeling the World

The definition of a belief state is very general and does not constrain what should be remembered by the agent. Often it is useful for the agent to maintain some model of the world, even if its model is incomplete and inaccurate. A **model** of a world is a representation of the state of the world at a particular time and/or the dynamics of the world.

One method is for the agent to maintain its belief about the world and to update these beliefs based on its commands. This approach requires a model of both the state of the world and the dynamics of the world. Given the state at one time, and the dynamics, the state at the next time can be predicted. This

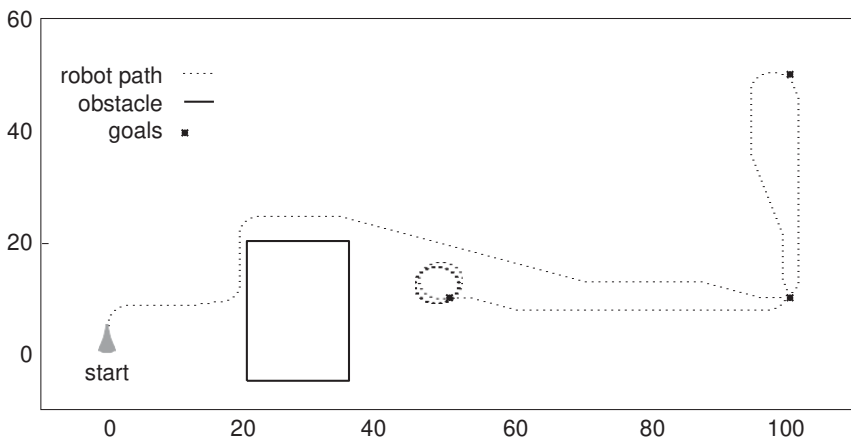


Figure 2.8: A simulation of the robot carrying out the plan of Example 2.6

process is known as **dead reckoning**. For example, a robot could maintain its estimate of its position and update it based on its actions. When the world is dynamic or when there are noisy actuators (e.g., a wheel slips, it is not of exactly the right diameter, or acceleration is not instantaneous), the noise accumulates, so that the estimates of position soon become so inaccurate that they are useless. However, if the model is accurate at some level of abstraction, this may be an appropriate model of that level of abstraction.

An alternative is to use **perception** to build a model of the relevant part of the world. Perception is the use of sensing information to understand the world. This could, for example, involve using vision to detect features of the world and use these features to determine the position of a robot and obstacles or packages to be picked up. Perception tends to be ambiguous and noisy. It is difficult to build a model of a three-dimensional world based on a single image of the world.

A more promising approach is to combine the agent's prediction of the world state with sensing information. This can take a number of forms:

- If both the noise of forward prediction and sensor noise are modeled, the next belief state can be estimated using Bayes' rule (page 227). This is known as **filtering** (page 267).
- With more complicated sensors such as vision, a model can be used to predict where visual features can be found, and then vision can be used to look for these features close to the predicted location. This makes the vision task much simpler and vision can greatly reduce the errors in position arising from forward prediction alone.

A control problem is **separable** if the best action can be obtained by first finding the best model of the world and then using that model to determine the best action. Unfortunately, most control problems are not separable. This means that the agent should consider multiple models to determine what to

do, and what information it gets from the world depends on what it will do with that information. Usually, there is no best model of the world that is independent of what the agent will do with the model.

2.4 Embedded and Simulated Agents

There are a number of ways an agent's controller can be used:

- An **embedded agent** is one that is run in the real world, where the actions are carried out in a real domain and where the sensing comes from a domain.
- A **simulated agent** is one that is run with a simulated body and environment; that is, where a program takes in the commands and returns appropriate percepts. This is often used to debug a controller before it is deployed.
- A **agent system model** is where there are models of the controller (which may or may not be the actual code), the body, and the environment that can answer questions about how the agent will behave. Such a model can be used to prove properties of agents before they are built, or it can be used to answer hypothetical questions about an agent that may be difficult or dangerous to answer with the real agent.

Each of these is appropriate for different purposes.

- Embedded mode is how the agent must run to be useful.
- A simulated agent is useful to test and debug the controller when many design options must be explored and building the body is expensive or when the environment is dangerous or inaccessible. It also allows us to test the agent under unusual combinations of conditions that may be difficult to arrange in the actual world.

How good the simulation is depends on how good the model of the environment is. Models always have to abstract some aspect of the world. Appropriate abstraction is important for simulations to be able to tell us whether the agent will work in a real environment.

- A model of the agent, a model of the set of possible environments, and a specification of correct behavior allow us to prove theorems about how the agent will work in such environments. For example, we may want to prove that a robot running a particular controller will always get within a certain distance of the target, that it will never get stuck in mazes, or that it will never crash. Of course, whether what is proved turns out to be true depends on how accurate the models are.
- Given a model of the agent and the environment, some aspects of the agent can be left unspecified and can be adjusted to produce the desired or optimal behavior. This is the general idea behind optimization and planning.
- In reinforcement learning (page 463), the agent improves its performance while interacting with the real world.

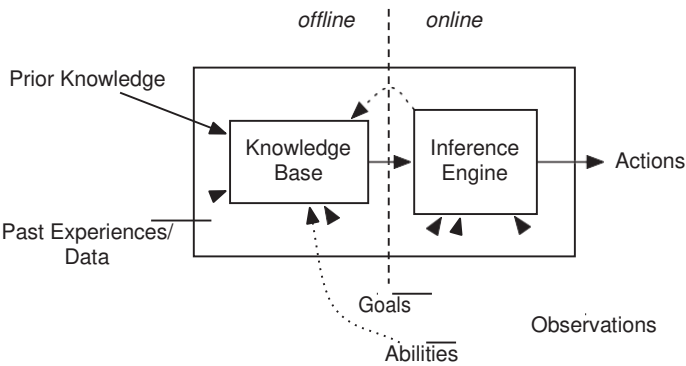


Figure 2.9: Offline and online decomposition of an agent

2.5 Acting with Reasoning

The previous sections assumed that an agent has some belief state that it maintains through time. For an intelligent agent, the belief state can be very complex, even for a single layer.

Experience in studying and building intelligent agents has shown that an intelligent agent requires some internal representation of its belief state. **Knowledge** is the information about a domain that is used for solving problems in that domain. Knowledge can include general knowledge that can be applied to particular situations. Thus, it is more general than the beliefs about a specific state. A **knowledge-based system** is a system that uses knowledge about a domain to act or to solve problems.

Philosophers have defined knowledge as true, justified belief. AI researchers tend to use the terms knowledge and belief more interchangeably. Knowledge tends to mean general information that is taken to be true. Belief tends to mean information that can be revised based on new information. Often beliefs come with measures of how much they should be believed and models of how the beliefs interact. In an AI system, knowledge is typically not necessarily true and is justified only as being useful. This distinction often becomes blurry when one module of an agent may treat some information as true but another module may be able to revise that information.

Figure 2.9 shows a refinement of Figure 1.3 (page 11) for a knowledge-based agent. A **knowledge base** is built offline and is used online to produce actions. This decomposition of an agent is orthogonal to the layered view of an agent; an intelligent agent requires both hierarchical organization and knowledge bases.

Online (page 17), when the agent is acting, the agent uses its knowledge base, its observations of the world, and its goals and abilities to choose what to do and to update its knowledge base. The **knowledge base** is its **long-term**

memory, where it keeps the knowledge that is needed to act in the future. This knowledge comes from prior knowledge and is combined with what is learned from data and past experiences. The **belief state** (page 48) is the **short-term memory** of the agent, which maintains the model of current environment needed between time steps. A clear distinction does not always exist between general knowledge and specific knowledge; for example, an outside delivery robot could learn general knowledge about a particular city. There is feedback from the inference engine to the knowledge base, because observing and acting in the world provide more data from which to learn.

Offline, before the agent has to act, it can build the knowledge base that is useful for it to act online. The role of the offline computation is to make the online computation more efficient or effective. The knowledge base is built from prior knowledge and from data of past experiences (either its own past experiences or data it has been given). Researchers have traditionally considered the case involving lots of data and little prior knowledge in the field of **machine learning**. The case of lots of prior knowledge and little or no data from which to learn has been studied under the umbrella of **expert systems**. However, for most non-trivial domains, the agent must use whatever information is available, and so it requires both rich prior knowledge and lots of data.

The goals and abilities are given offline, online, or both, depending on the agent. For example, a delivery robot could have general goals of keeping the lab clean and not damaging itself or other objects, but it could get other delivery goals at runtime. The online computation can be made more efficient if the knowledge base is tuned for the particular goals and abilities. However, this is often not possible when the goals and abilities are only available at runtime.

Figure 2.10 (on the next page) shows more detail of the interface between the agents and the world.

2.5.1 Design Time and Offline Computation

The knowledge base required for online computation can be built initially at design time and then augmented offline by the agent.

An **ontology** is a specification of the meaning of the symbols used in an information system. It specifies what is being modeled and the vocabulary used in the system. In the simplest case, if the agent is using explicit state-based representation with full observability, the ontology specifies the mapping between the world and the state. Without this mapping, the agent may know it is in, say, state 57, but, without the ontology, this information is just a meaningless number to another agent or person. In other cases, the ontology defines the features or the individuals and relationships. It is what is needed to convert raw sense data into something meaningful for the agent or to get meaningful input from a person or another knowledge source.

Ontologies are built by communities, often independently of a particular knowledge base or specific application. It is this shared vocabulary that

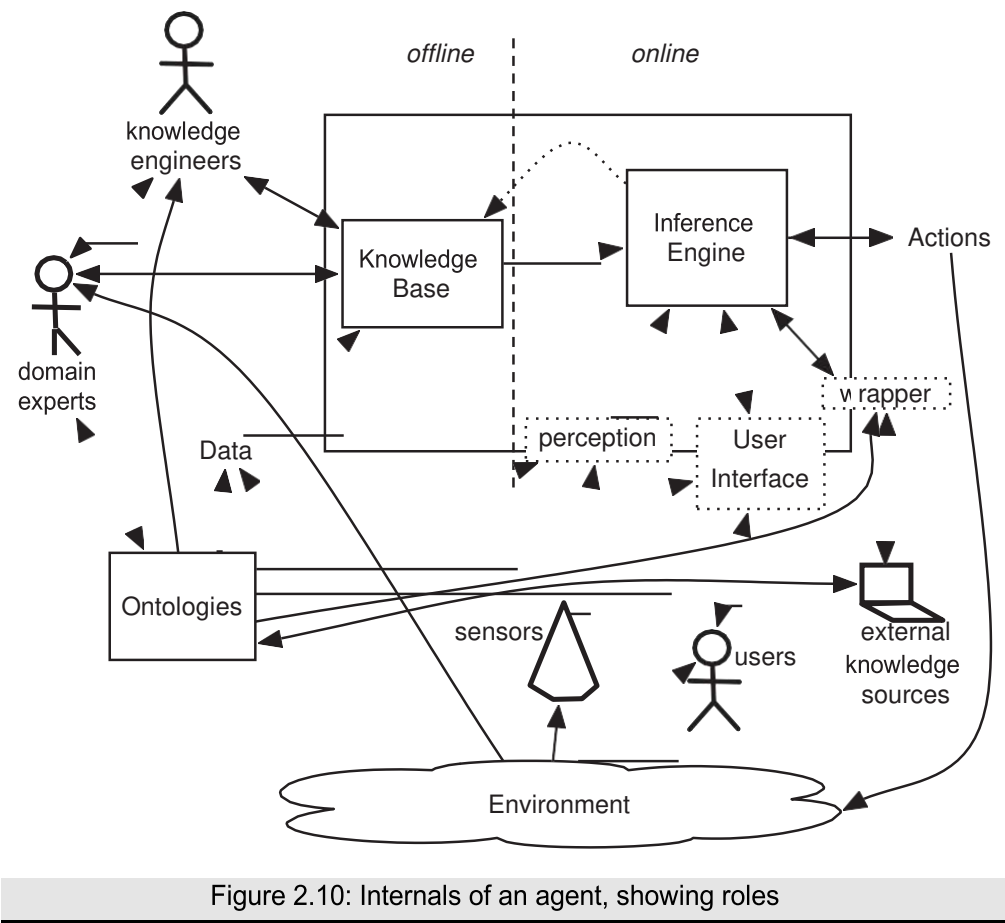


Figure 2.10: Internals of an agent, showing roles

allows for effective communication and interoperation of the data from multiple sources (sensors, humans, and databases). Ontologies for the case of individuals and relationships are discussed in Section 13.3 (page 563).

The ontology logically comes before the data and the prior knowledge: we require an ontology to have data or to have knowledge. Without an ontology, data are just sequences of bits. Without an ontology, a human does not know what to input; it is the ontology that gives the data meaning. Often the ontology evolves as the system is being developed.

The ontology specifies a level or levels of abstraction. If the ontology changes, the data must change. For example, a robot may have an ontology of obstacles (e.g., every physical object is an obstacle to be avoided). If the ontology is expanded to differentiate people, chairs, tables, coffee mugs, and the like, different data about the world are required.

The knowledge base is typically built offline from a combination of expert knowledge and data. It is usually built before the agent knows the particulars of the environment in which it must act. Maintaining and tuning the knowledge base is often part of the online computation.

Offline, there are three major **roles** involved with a knowledge-based system:

- **Software engineers** build the inference engine and user interface. They typically know nothing about the contents of the knowledge base. They need not be experts in the use of the system they implement; however, they must be experts in the use of a programming language like Java, Lisp, or Prolog rather than in the knowledge representation language of the system they are designing.
- **Domain experts** are the people who have the appropriate prior knowledge about the domain. They know about the domain, but typically they know nothing about the particular case that may be under consideration. For example, a medical domain expert would know about diseases, symptoms, and how they interact but would not know the symptoms or the diseases of the particular patient. A delivery robot domain expert may know the sort of individuals that must be recognized, what the battery meter measures, and the costs associated with various actions. Domain experts typically do not know the particulars of the environment the agent would encounter – for example, the details of the patient for the diagnostic assistant or the details of the room a robot is in.

Domain experts typically do not know about the internal workings of the AI system. Often they have only a semantic view of the knowledge (page 161) and have no notion of the algorithms used by the inference engine. The system should interact with them in terms of the domain, not in terms of the steps of the computation. For example, it is unreasonable to expect that domain experts could debug a knowledge base if they were presented with traces of how an answer was produced. Thus, it is not appropriate to have debugging tools for domain experts that merely trace the execution of a program.

- **Knowledge engineers** design, build, and debug the knowledge base in consultation with domain experts. They know about the details of the system and about the domain through the domain expert. They know nothing about any particular case. They should know about useful inference techniques and how the complete system works.

The same people may fill multiple roles: A domain expert who knows about AI may act as a knowledge engineer; a knowledge engineer may be the same person who writes the system. A large system may have many different software engineers, knowledge engineers, and experts, each of whom may specialize in part of the system. These people may not even know they are part of the system; they may publish information for anyone to use.

Offline, the agent can combine the expert knowledge and the data. At this stage, the system can be tested and debugged. The agent is able to do computation that is not particular to the specific instance. For example, it can compile parts of the knowledge base to allow more efficient inference.

2.5.2 Online Computation

Online, the information about the particular case becomes available, and the agent has to act. The information includes the observations of the domain and often information about the available actions and the preferences or goals. The agent can get observations from sensors, users, and other information sources (such as web sites), but we assume it does not have access to the domain experts or knowledge engineer.

An agent typically has much more time for offline computation than for online computation. However, during online computation it can take advantage of particular goals and particular observations.

For example, a medical diagnosis system only has the details of a particular patient online. Offline, it can acquire knowledge about how diseases and symptoms interact and do some debugging and compilation. It can only do the computation about a particular patient online.

Online the following roles are involved:

- A **user** is a person who has a need for expertise or has information about individual cases. Users typically are not experts in the domain of the knowledge base. They often do not know what information is needed by the system. Thus, it is unreasonable to expect them to volunteer the information about a particular case. A simple and natural interface must be provided because users do not typically understand the internal structure of the system. They often, however, must make an informed decision based on the recommendation of the system; thus, they require an explanation of why the recommendation is appropriate.
- **Sensors** provide information about the environment. For example, a thermometer is a sensor that can provide the current temperature at the location of the thermometer. Sensors may be more sophisticated, such as a vision sensor. At the lowest level, a vision sensor may simply provide an array of 720 × 80 pixels at 30 frames per second. At a higher level, a vision system may be able to answer specific questions about the location of particular features, whether some type of individual is in the environment, or whether some particular individual is in the scene. An array of microphones can be used at a low level of abstraction to provide detailed vibration information. It can also be used as a component of a higher-level sensor to detect an explosion and to provide the type and the location of the explosion.

Sensors come in two main varieties. A **passive sensor** continuously feeds information to the agent. Passive sensors include thermometers, cameras, and microphones. The designer can typically choose where the sensors are or where they are pointing, but they just feed the agent information. In contrast, an **active sensor** is controlled or asked for information. Examples of an active sensor include a medical probe able to answer specific questions about a patient or a test given to a student in an intelligent tutoring system. Often sensors that are passive sensors at lower levels of abstraction can be seen as active sensors at higher levels of abstraction. For example, a camera could be asked whether a particular person is in the room. To do this it may need

to zoom in on the faces in the room, looking for distinguishing features of the person.

- An **external knowledge source**, such as a web site or a database, can typically be asked questions and can provide the answer for a limited domain. An agent can ask a weather web site for the temperature at a particular location or an airline web site for the arrival time of a particular flight. The knowledge sources have various protocols and efficiency trade-offs. The interface between an agent and an external knowledge source is called a **wrapper**. A wrapper translates between the representation the agent uses and the queries the external knowledge source is prepared to handle. Often wrappers are designed so that the agent can ask the same query of multiple knowledge sources. For example, an agent may want to know about airplane arrivals, but different airlines or airports may require very different protocols to access that information. When web sites and databases adhere to a common ontology, they can be used together because the same symbols have the same meaning. Having the same symbols mean the same thing is called **semantic interoperability**. When they use different ontologies, there must be mappings between the ontologies to allow them to interoperate.

Again, these roles are separate, even though the people in these roles may overlap. The domain expert, for example, may act as a user to test or debug the system. Each of the roles has different requirements for the tools they need. The tools that explain to a user how the system reached a result can be the same tools that the domain experts use to debug the knowledge.

2.6 References and Further Reading

The model of agent systems is based on the constraint nets of [Zhang and Mackworth \[1995\]](#), also on [Rosenschein and Kaelbling \[1995\]](#). The hierarchical control is based on [Albus \[1981\]](#) and the subsumption architecture of [Brooks \[1986\]](#). *Turtle Geometry*, by [Abelson and DiSessa \[1981\]](#), investigates mathematics from the viewpoint of modeling simple reactive agents. [Luenberger \[1979\]](#) is a readable introduction to the classical theory of agents interacting with environments. [Simon \[1996\]](#) argues for the importance of hierarchical control.

For more detail on agent control see [Dean and Wellman \[1991\]](#), [Latombe \[1991\]](#), and [Agre \[1995\]](#).

The methodology for building intelligent agents is discussed by [Haugeland \[1985\]](#), [Brooks \[1991\]](#), [Kirsh \[1991b\]](#), and [Mackworth \[1993\]](#).

Qualitative reasoning is described by [Forbus \[1996\]](#) and [Kuipers \[2001\]](#). [Weld and de Kleer \[1990\]](#) contains many seminal papers on qualitative reasoning. See also [Weld \[1992\]](#) and related discussion in the same issue. For a recent review see [Price, Travé-Massuyàs, Milne, Ironi, Forbus, Bredeweg, Lee, Struss, Snooke, Lucas, Cavazza, and Coghill \[2006\]](#).

Part II

Representing and Reasoning

Methodology

States and Searching

Have you ever watched a crab on the shore crawling backward in search of the Atlantic Ocean, and missing? That's the way the mind of man operates.

– H. L. Mencken (1880–1956)

The previous chapter discussed how an agent perceives and acts, but not how its goals affect its actions. An agent could be programmed to act in the world to achieve a fixed set of goals, but then it may not adapt to changing goals and so would not be intelligent. Alternatively, an agent could reason about its abilities and its goals to determine what to do. This chapter shows how the problem of an agent deciding what to do can be cast as the problem of searching to find a path in a graph, and it presents a number of ways that such problems can be solved on a computer. As Mencken suggests in the quote above, the mind uses search to solve problems, although not always successfully.

3.1 Problem Solving as Search

In the simplest case of an agent reasoning about what it should do, the agent has a state-based model of the world, with no uncertainty and with goals to achieve. This is either a flat (non-hierarchical) representation or a single level of a hierarchy. The agent can determine how to achieve its goals by searching in its representation of the world state space for a way to get from its current state to a goal state. It can find a sequence of actions that will achieve its goal before it has to act in the world.

This problem can be abstracted to the mathematical problem of finding a path from a start node to a goal node in a directed graph. Many other problems can also be mapped to this abstraction, so it is worthwhile to consider

this level of abstraction. Most of this chapter explores various algorithms for finding such paths.

This notion of search is computation inside the agent. It is different from searching in the world, when it may have to act in the world, for example, an agent searching for its keys, lifting up cushions, and so on. It is also different from searching the web, which involves searching for information. Searching in this chapter means searching in an internal representation for a path to a goal.

The idea of search is straightforward: the agent constructs a set of potential partial solutions to a problem that can be checked to see if they truly are solutions or if they could lead to solutions. Search proceeds by repeatedly selecting a partial solution, stopping if it is a path to a goal, and otherwise extending it by one more arc in all possible ways.

Search underlies much of artificial intelligence. When an agent is given a problem, it is usually given only a description that lets it recognize a solution, not an algorithm to solve it. It has to search for a solution. The existence of NP-complete problems (page 170), with efficient means to recognize answers but no efficient methods for finding them, indicates that searching is, in many cases, a necessary part of solving problems.

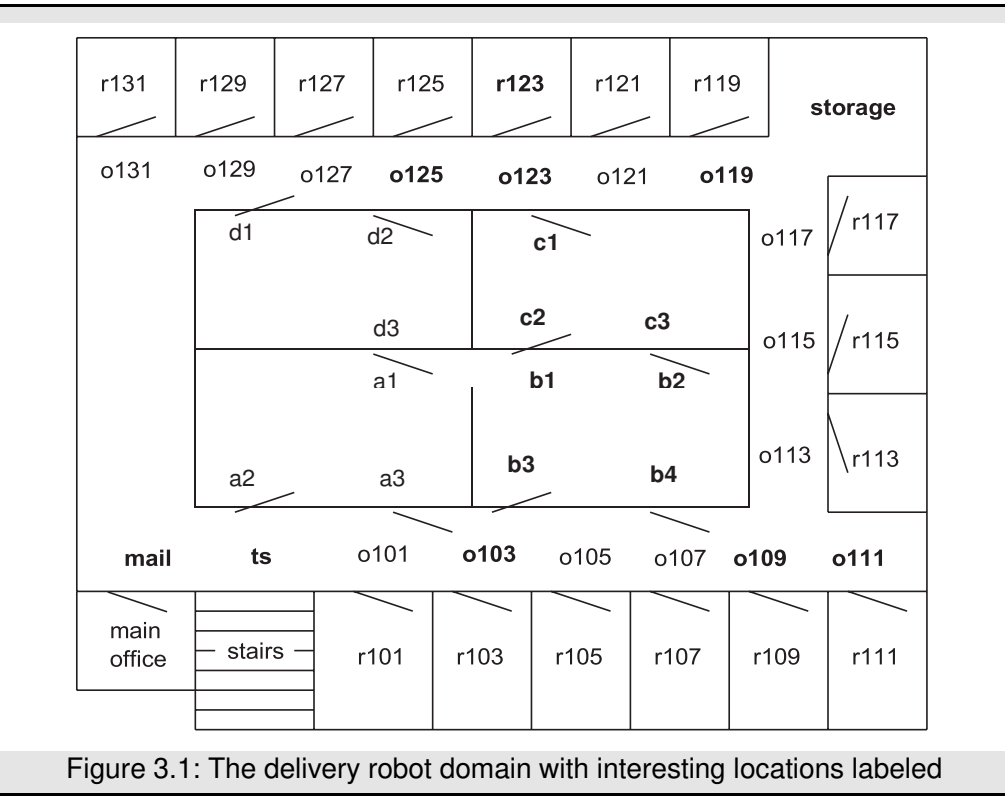
It is often believed that humans are able to use intuition to jump to solutions to difficult problems. However, humans do not tend to solve general problems; instead they solve specific instances about which they may know much more than the underlying search space. Problems in which little structure exists or in which the structure cannot be related to the physical world are very difficult for humans to solve. The existence of public key encryption codes, where the search space is clear and the test for a solution is given – for which humans nevertheless have no hope of solving and computers cannot solve in a realistic time frame – demonstrates the difficulty of search.

The difficulty of search and the fact that humans are able to solve some search problems efficiently suggests that computer agents should exploit knowledge about special cases to guide them to a solution. This extra knowledge beyond the search space is **heuristic knowledge**. This chapter considers one kind of heuristic knowledge in the form of an estimate of the cost from a node to a goal.

3.2 State Spaces

One general formulation of intelligent action is in terms of **state space**. A **state** contains all of the information necessary to predict the effects of an action and to determine if it is a goal state. State-space searching assumes that

- the agent has perfect knowledge of the state space and can observe what state it is in (i.e., there is full observability);
- the agent has a set of actions that have known deterministic effects;



- some states are goal states, the agent wants to reach one of these goal states, and the agent can recognize a goal state; and
- a **solution** is a sequence of actions that will get the agent from its current state to a goal state.

Example 3.1 Consider the robot delivery domain and the task of finding a path from one location to another in Figure 3.1. This can be modeled as a state- space search problem, where the states are locations. Assume that the agent can use a lower-level controller to carry out the high-level action of getting from one location to a neighboring location. Thus, at this level of abstraction, the actions can involve deterministic traveling between neighboring locations.

An example problem is where the robot is outside room *r103*, at position *o103*, and the goal is to get to room *r123*. A solution is a sequence of actions that will get the robot to room *r123*.

Example 3.2 In a more complicated example, the delivery robot may have a number of parcels to deliver to various locations. In this case, the state may consist of the location of the robot, the parcels the robot is carrying, and the locations of the other parcels. The possible actions may be for the robot to move, to pick up parcels that are at the same location as the robot, or to put down whatever parcels it is carrying. A goal state may be one in which some specified

parcels are at their desired locations. There may be many goal states because we may not care where the robot is or where some of the other parcels are.

Notice that this representation has ignored many details, for example, how the robot is carrying the parcels (which may affect whether it can carry other parcels), the battery level of the robot, whether the parcels are fragile or damaged, and the color of the floor. By not having these as part of the state space, we assume that these details are not relevant to the problem at hand.

Example 3.3 In a tutoring system, a state may consist of the set of topics that the student knows. The action may be teaching a particular lesson, and the result of a teaching action may be that the student knows the topic of the lesson as long as the student knows the topics that are prerequisites for the lesson being taught. The aim is for the student to know some particular set of topics.

If the effect of teaching also depends on the aptitude of the student, this detail must be part of the state space, too. We do not have to model what the student is carrying if that does not affect the result of actions or whether the goal is achieved.

A **state-space problem** consists of

- a set of states;
- a distinguished set of states called the **start states**;
- a set of actions available to the agent in each state;
- an **action function** that, given a state and an action, returns a new state;
- a set of goal states, often specified as a Boolean function, $goal(s)$, that is true when s is a goal state; and
- a criterion that specifies the quality of an acceptable solution. For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost. This is called an **optimal** solution. Alternatively, it may be satisfied with any solution that is within 10% of optimal.

This framework is extended in subsequent chapters to include cases where an agent can exploit the internal features of the states, where the state is not fully observable (e.g., the robot does not know where the parcels are, or the teacher does not know the aptitude of the student), where the actions are stochastic (e.g., the robot may overshoot, or the student perhaps does not learn a topic that is taught), and where complex preferences exist in terms of rewards and punishments, not just goal states.

3.3 Graph Searching

In this chapter, we abstract the general mechanism of searching and present it in terms of searching for paths in directed graphs. To solve a problem, first define the underlying search space and then apply a search algorithm to that

search space. Many problem-solving tasks can be transformed into the problem of finding a path in a graph. Searching in graphs provides an appropriate level of abstraction within which to study simple problem solving independent of a particular domain.

A (directed) graph consists of a set of nodes and a set of directed arcs between nodes. The idea is to find a path along these arcs from a start node to a goal node.

The abstraction is necessary because there may be more than one way to represent a problem as a graph. Whereas the examples in this chapter are in terms of state-space searching, where nodes represent states and arcs represent actions, future chapters consider different ways to represent problems as graphs to search.

3.3.1 Formalizing Graph Searching

A directed **graph** consists of

- a set N of **nodes** and
- a set A of ordered pairs of nodes called **arcs**.

In this definition, a node can be anything. All this definition does is constrain arcs to be ordered pairs of nodes. There can be infinitely many nodes and arcs. We do not assume that the graph is represented explicitly; we require only a procedure that can generate nodes and arcs as needed.

The arc (n_1, n_2) is an **outgoing arc** from n_1 and an **incoming arc** to n_2 .

A node n_2 is a **neighbor** of n_1 if there is an arc from n_1 to n_2 ; that is, if $(n_1, n_2) \in A$. Note that being a neighbor does not imply symmetry; just because n_2 is a neighbor of n_1 does not mean that n_1 is necessarily a neighbor of n_2 . Arcs may be **labeled**, for example, with the action that will take the agent from one state to another.

A **path** from node s to node g is a sequence of nodes $(n_0, n_1, \dots, n_k)$ such that $s = n_0$, $g = n_k$, and $(n_{i-1}, n_i) \in A$ for each i . Sometimes it is useful to view a path as the sequence of arcs, $(n_0, n_1), (n_1, n_2), \dots, (n_{k-1}, n_k)$, or a sequence of labels of these arcs.

A **cycle** is a nonempty path such that the end node is the same as the start node— that is, a cycle is a path $(n_0, n_1, \dots, n_k)$ such that $n_0 = n_k$ and $k > 0$. A directed graph without any cycles is called a **directed acyclic graph (DAG)**. This should probably be an **acyclic directed graph**, because it is a directed graph that happens to be acyclic, not an acyclic graph that happens to be directed, but DAG sounds better than ADG!

A **tree** is a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc. The node with no incoming arcs is called the **root** of the tree and nodes with no outgoing arcs are called **leaves**.

To encode problems as graphs, one set of nodes is referred to as the **start nodes** and another set is called the **goal nodes**. A **solution** is a path from a start node to a goal node.

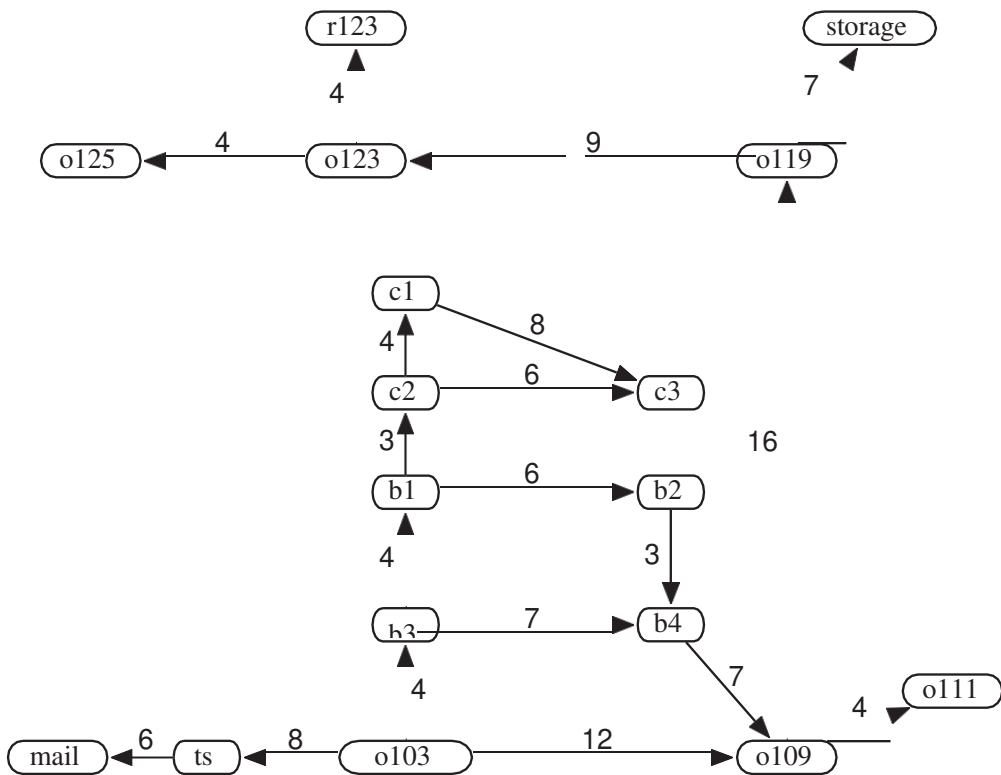


Figure 3.2: A graph with arc costs for the delivery robot domain

Sometimes there is a **cost** – a positive number – associated with arcs. We write the cost of arc n_i, n_j as $cost(n_i, n_j)$. The costs of arcs induces a cost of paths.

Given a path $p = (n_0, n_1, \dots, n_k)$, the cost of path p is the sum of the costs of the arcs in the path:

$$cost(p) = cost((n_0, n_1)) + \dots + cost((n_{k-1}, n_k))$$

An **optimal solution** is one of the least-cost solutions; that is, it is a path p from a start node to a goal node such that there is no path p^t from a start node to a goal node where $cost(p^t) < cost(p)$.

Example 3.4 Consider the problem of the delivery robot finding a path from location **o103** to location **r123** in the domain depicted in Figure 3.1 (page 73). In this figure, the interesting locations are named. For simplicity, we consider only the locations written in bold and we initially limit the directions that the robot can travel. Figure 3.2 shows the resulting graph where the nodes represent locations and the arcs represent possible single steps between locations. In this figure, each arc is shown with the associated cost of getting from one location to the next.

In this graph, the nodes are $N = \text{mail}, ts, o103, b3, o109, \dots$ and the arcs are $A = ts, \text{mail}, (o103, ts), (o103, b3), (o103, o109), \dots$. Node $o125$ has no neighbors. Node ts has one neighbor, namely mail . Node $o103$ has three neighbors, namely $ts, b3$, and $o109$.

There are three paths from $o103$ to $r123$:

($o103, o109, o119, o123, r123$)
 ($o103, b3, b4, o109, o119, o123, r123$)
 ($o103, b3, b1, b2, b4, o109, o119, o123, r123$)

If $o103$ were a start node and $r123$ were a goal node, each of these three paths would be a solution to the graph-searching problem.

In many problems the search graph is not given explicitly; it is dynamically constructed as needed. All that is required for the search algorithms that follow is a way to generate the neighbors of a node and to determine if a node is a goal node.

The **forward branching factor** of a node is the number of arcs leaving the node. The **backward branching factor** of a node is the number of arcs entering the node. These factors provide measures of the complexity of graphs. When we discuss the time and space complexity of the search algorithms, we assume that the branching factors are bounded from above by a constant.

Example 3.5 In the graph of Figure 3.2, the forward branching factor of node $o103$ is three; there are three arcs coming out of node $o103$. The backward branching factor of node $o103$ is zero; there are no arcs coming into node $o103$. The forward branching factor of mail is zero and the backward branching factor of mail is one. The forward branching factor of node $b3$ is two and the backward branching factor of $b3$ is one.

The branching factor is important because it is a key component in the size of the graph. If the forward branching factor for each node is b , and the graph is a tree, there are b^n nodes that are n arcs away from any node.

3.4 A Generic Searching Algorithm

This section describes a generic algorithm to search for a solution path in a graph. The algorithm is independent of any particular search strategy and any particular graph.

The intuitive idea behind the generic search algorithm, given a graph, a set of start nodes, and a set of goal nodes, is to incrementally explore paths from the start nodes. This is done by maintaining a **frontier** (or **fringe**) of paths from the start node that have been explored. The frontier contains all of the paths that could form initial segments of paths from a start node to a goal node. (See Figure 3.3 (on the next page), where the frontier is the set of paths to the gray shaded nodes.) Initially, the frontier contains trivial paths containing no

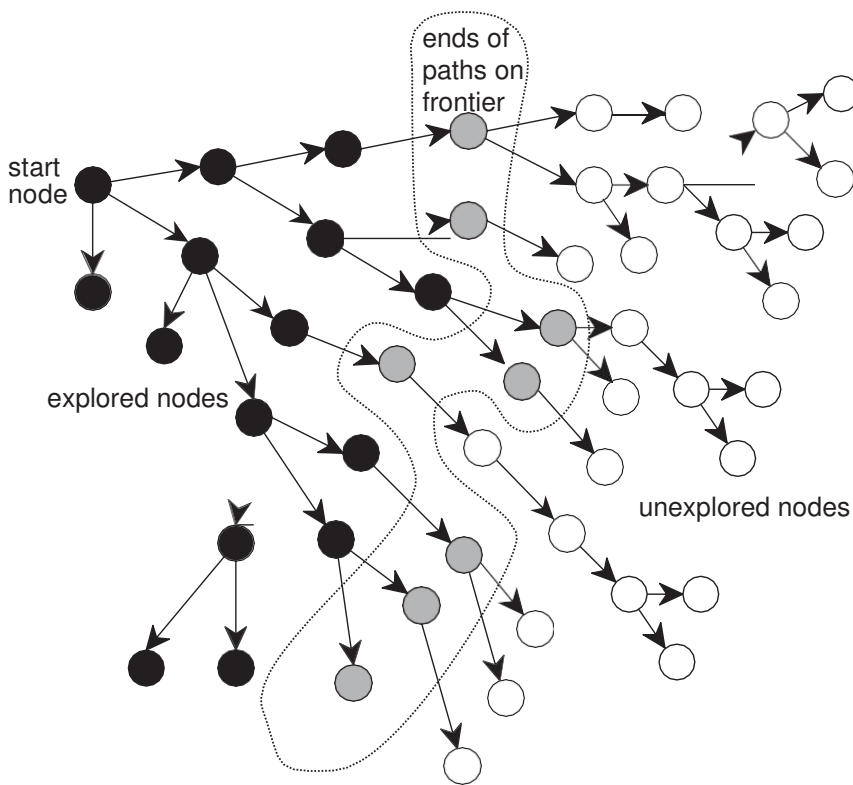


Figure 3.3: Problem solving by graph searching

arcs from the start nodes. As the search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered. To expand the frontier, the searcher selects and removes a path from the frontier, extends the path with each arc leaving the last node, and adds these new paths to the frontier. A search strategy defines which element of the frontier is selected at each step.

The generic search algorithm is shown in Figure 3.4. Initially, the frontier is the set of empty paths from start nodes. At each step, the algorithm advances the frontier by removing a path $s_0, \dots, s_k$ from the frontier. If $\text{goal}(s_k)$ is true (i.e., s_k is a goal node), it has found a solution and returns the path that was found, namely $s_0, \dots, s_k$. Otherwise, the path is extended by one more arc by finding the neighbors of s_k . For every neighbor s of s_k , the path $s_0, \dots, s_k, s$ is added to the frontier. This step is known as **expanding** the node s_k .

This algorithm has a few features that should be noted:

- The selection of a path at line 13 is non-deterministic. The choice of path that is selected can affect the efficiency; see the box on page 170 for more details on our use of “select”. A particular search strategy will determine which path is selected.
- It is useful to think of the *return* at line 15 as a temporary return; another path to a goal can be searched for by continuing to line 16.

```

1: procedure Search(G, S, goal)
2:   Inputs
3:     G: graph with nodes N and arcs A
4:     S: set of start nodes
5:     goal: Boolean function of states
6:   Output
7:     path from a member of S to a node for which goal is true
8:     or  $\perp$  if there are no solution paths
9:   Local
10:    Frontier: set of paths
11:    Frontier  $\leftarrow \{(s) : s \in S\}$ 
12:    while Frontier  $\neq \emptyset$  do
13:      select and remove  $(s_0, \dots, s_k)$  from Frontier
14:      if goal( $s_k$ ) then
15:        return  $(s_0, \dots, s_k)$ 
16:      Frontier  $\leftarrow \text{Frontier} \cup \{(s_0, \dots, s_k, s) : (s_k, s) \in A\}$ 
17:    return  $\perp$ 

```

Figure 3.4: Generic graph searching algorithm

- If the procedure returns $\perp$ no solutions exist (or there are no remaining solutions if the proof has been retried).
- The algorithm only tests if a path ends in a goal node *after* the path has been selected from the frontier, not when it is added to the frontier. There are two main reasons for this. Sometimes a very costly arc exists from a node on the frontier to a goal node. The search should not always return the path with this arc, because a lower-cost solution may exist. This is crucial when the least-cost path is required. The second reason is that it may be expensive to determine whether a node is a goal node.

If the path chosen does not end at a goal node and the node at the end has no neighbors, extending the path means removing the path. This outcome is reasonable because this path could not be part of a path from a start node to a goal node.

3.5 Uninformed Search Strategies

A problem determines the graph and the goal but not which path to select from the frontier. This is the job of a search strategy. A search strategy specifies which paths are selected from the frontier. Different strategies are obtained by modifying how the selection of paths in the frontier is implemented.

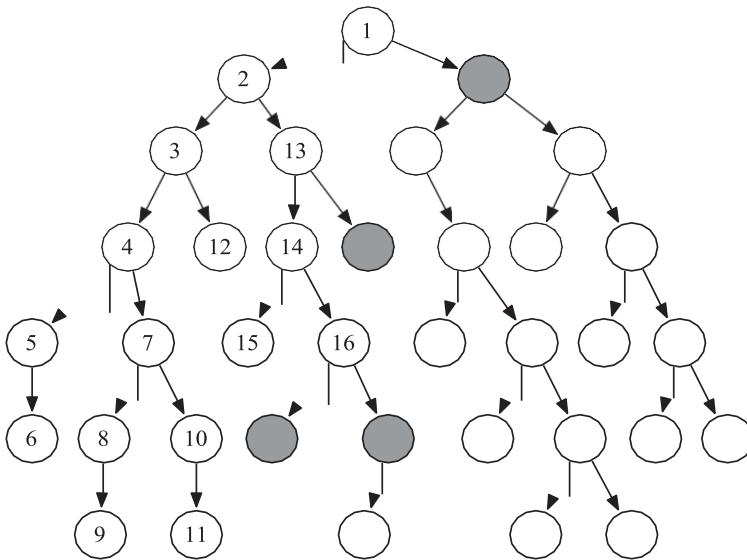


Figure 3.5: The order nodes are expanded in depth-first search

This section presents three **uninformed search strategies** that do not take into account the location of the goal. Intuitively, these algorithms ignore where they are going until they find a goal and report success.

3.5.1 Depth-First Search

The first strategy is **depth-first search**. In depth-first search, the frontier acts like a last-in first-out **stack**. The elements are added to the stack one at a time. The one selected and taken off the frontier at any time is the last element that was added.

Example 3.6 Consider the tree-shaped graph in Figure 3.5. Suppose the start node is the root of the tree (the node at the top) and the nodes are ordered from left to right so that the leftmost neighbor is added to the stack last. In depth-first search, the order in which the nodes are expanded does not depend on the location of the goals. The first sixteen nodes expanded are numbered in order of expansion in Figure 3.5. The shaded nodes are the nodes at the ends of the paths on the frontier after the first sixteen steps.

Notice how the first six nodes expanded are all in a single path. The sixth node has no neighbors. Thus, the next node that is expanded is a child of the lowest ancestor of this node that has unexpanded children.

Implementing the frontier as a stack results in paths being pursued in a depth-first manner – searching one path to its completion before trying an alternative path. This method is said to involve **backtracking**: The algorithm selects a first alternative at each node, and it *backtracks* to the next alternative

when it has pursued all of the paths from the first selection. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.

This algorithm does not specify the order in which the neighbors are added to the stack that represents the frontier. The efficiency of the algorithm is sensitive to this ordering.

Example 3.7 Consider depth-first search from $o103$ in the graph given in Figure 3.2. The only goal node is $r123$. In this example, the frontier is shown as a list of paths with the top of the stack at the beginning of the list.

Initially, the frontier contains the trivial path $(o103)$.

At the next stage, the frontier contains the following paths:

$[(o103, ts), (o103, b3), (o103, o109)]$.

Next, the path $(o103, ts)$ is selected because it is at the top of the stack. It is removed from the frontier and replaced by extending it by one arc, resulting in the frontier

$[(o103, ts, mail), (o103, b3), (o103, o109)]$.

Next, the first path $(o103, ts, mail)$ is removed from the frontier and is replaced by the set of paths that extend it by one arc, which is the empty set because *mail* has no neighbors. Thus, the resulting frontier is

$[(o103, b3), (o103, o109)]$.

At this stage, the path $(o103, b3)$ is the top of the stack. Notice what has happened: depth-first search has pursued all paths from *ts* and, when all of these paths were exhausted (there was only one), it backtracked to the next element of the stack. Next, $(o103, b3)$ is selected and is replaced in the frontier by the paths that extend it by one arc, resulting in the frontier

$[(o103, b3, b1), (o103, b3, b4), (o103, o109)]$.

Then $(o103, b3, b1)$ is selected from the frontier and is replaced by all one-arc extensions, resulting in the frontier

$[(o103, b3, b1, c2), (o103, b3, b1, b2), (o103, b3, b4), (o103, o109)]$.

Now the first path is selected from the frontier and is extended by one arc, resulting in the frontier

$[(o103, b3, b1, c2, c3), (o103, b3, b1, c2, c1), (o103, b3, b1, b2), (o103, b3, b4), (o103, o109)]$.

Node $c3$ has no neighbors, and thus the search “backtracks” to the last alternative that has not been pursued, namely to the path to $c1$.

Suppose $n_0, \dots, n_k$ is the selected path in the frontier. Then every other element of the frontier is of the form $n_0, \dots, n_i, m$, for some index $i < k$ and some node m that is a neighbor of n_i ; that is, it follows the selected path for a number of arcs and then has exactly one extra node.

To understand the complexity (see the box on page 83) of depth-first search, consider an analogy using family trees, where the neighbors of a node correspond to its children in the tree. At the root of the tree is a start node. A branch down this tree corresponds to a path from a start node. Consider the node at the end of path at the top of the frontier. The other elements of the frontier correspond to children of ancestors of that node – the “uncles,” “great uncles,” and so on. If the branching factor is b and the first element of the list has length n , there can be at most $n(b-1)$ other elements of the frontier. These elements correspond to the $b-1$ alternative paths from each node. Thus, for depth-first search, the space used is linear in the depth of the path length from the start to a node.

If there is a solution on the first branch searched, then the time complexity is linear in the length of the path; it considers only those elements on the path, along with their siblings. The worst-case complexity is infinite. Depth-first search can get trapped on infinite branches and never find a solution, even if one exists, for infinite graphs or for graphs with loops. If the graph is a finite tree, with the forward branching factor bounded by b and depth n , the worst-case complexity is $O(b^n)$.

Example 3.8 Consider a modification of the delivery graph, in which the agent has much more freedom in moving between locations. The new graph is presented in Figure 3.6 (page 84). An infinite path leads from *ts* to *mail*, back to *ts*, back to *mail*, and so forth. As presented, depth-first search follows this path forever, never considering alternative paths from *b3* or *o109*. The frontiers for the first five iterations of the path-finding search algorithm using depth-first search are

[(*o103*)]
 [(*o103, ts*) , (*o103, b3*) , (*o103, o109*)]
 [(*o103, ts, mail*) , (*o103, ts, o103*) , (*o103, b3*) , (*o103, o109*)]
 [(*o103, ts, mail, ts*) , (*o103, ts, o103*) , (*o103, b3*) , (*o103, o109*)]
 [(*o103, ts, mail, ts, mail*) , (*o103, ts, mail, ts, o103*) , (*o103, ts, o103*) ,
 (*o103, b3*) , (*o103, o109*)]

Depth-first search can be improved by not considering paths with cycles (page 93).

Because depth-first search is sensitive to the order in which the neighbors are added to the frontier, care must be taken to do it sensibly. This ordering can be done statically (so that the order of the neighbors is fixed) or dynamically (where the ordering of the neighbors depends on the goal).

Depth-first search is appropriate when either

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution; or
- the order of the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

Comparing Algorithms

Algorithms (including search algorithms) can be compared on

- the time taken,
- the space used, and
- the quality or accuracy of the results.

The time taken, space used, and accuracy of an algorithm are a function of the inputs to the algorithm. Computer scientists talk about the **asymptotic complexity** of algorithms, which specifies how the time or space grows with the input size of the algorithm. An algorithm has time (or space) complexity $O(f(n))$ – read “big-oh of $f(n)$ ” – for input size n , where $f(n)$ is some function of n , if there exist constants n_0 and k such that the time, or space, of the algorithm is less than $k f(n)$ for all $n > n_0$. The most common types of functions are exponential functions such as 2^n , 3^n , or 1.015^n ; polynomial functions such as n^5 , n^2 , n , or $n^{1/2}$; and logarithmic functions, $\log n$. In general, exponential algorithms get worse more quickly than polynomial algorithms which, in turn, are worse than logarithmic algorithms.

An algorithm has time or space complexity $\Omega(f(n))$ for input size n if there exist constants n_0 and k such that the time or space of the algorithm is greater than $k f(n)$ for all $n > n_0$. An algorithm has time or space complexity $\Theta(n)$ if it has complexity $O(n)$ and $\Omega(n)$. Typically, you cannot give an $\Theta(f(n))$ complexity on an algorithm, because most algorithms take different times for different inputs. Thus, when comparing algorithms, one has to specify the class of problems that will be considered.

Algorithm A is better than B , using a measure of either time, space, or accuracy, could mean:

- the worst case of A is better than the worst case of B ; or
- A works better in practice, or the average case of A is better than the average case of B , where you average over typical problems; or
- you characterize the class of problems for which A is better than B , so that which algorithm is better depends on the problem; or
- for every problem, A is better than B .

The worst-case asymptotic complexity is often the easiest to show, but it is usually the least useful. Characterizing the class of problems for which one algorithm is better than another is usually the most useful, if it is easy to determine which class a given problem is in. Unfortunately, this characterization is usually very difficult.

Characterizing when one algorithm is better than the other can be done either theoretically using mathematics or empirically by building implementations. Theorems are only as valid as the assumptions on which they are based. Similarly, empirical investigations are only as good as the suite of test cases and the actual implementations of the algorithms. It is easy to disprove a conjecture that one algorithm is better than another for some class of problems by showing a counterexample, but it is much more difficult to prove such a conjecture.

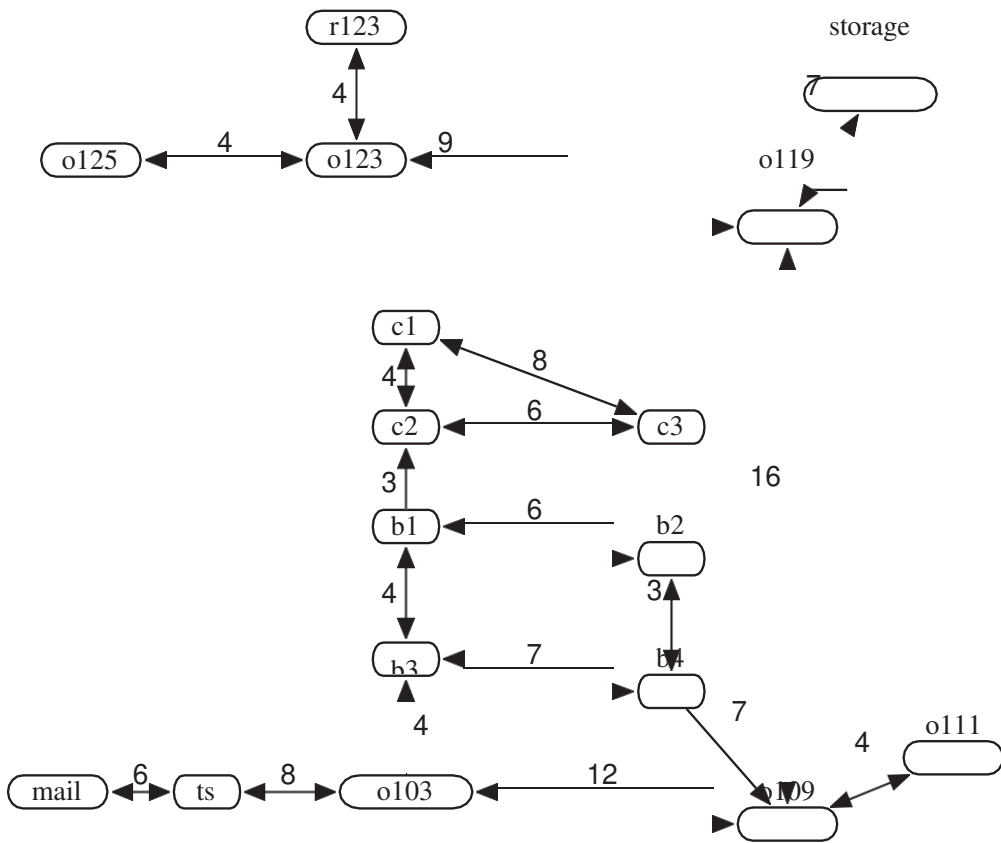


Figure 3.6: A graph, with cycles, for the delivery robot domain. Edges of the form $X \leftrightarrow Y$ means there is an arc from X to Y and an arc from Y to X . That is, $(X, Y) \in A$ and $(Y, X) \in A$.

It is a poor method when

- it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or
- solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions.

Depth-first search is the basis for a number of other algorithms, such as iterative deepening (page 95).

3.5.2 Breadth-First Search

In **breadth-first search** the frontier is implemented as a FIFO (first-in, first-out) queue. Thus, the path that is selected from the frontier is the one that was added earliest.

This approach implies that the paths from the start node are generated in order of the number of arcs in the path. One of the paths with the fewest arcs is selected

Suppose the branching factor of the search is b . If the first path of the frontier contains n arcs, there are at least b^{n-1} elements of the frontier. All of these paths contain n or $n + 1$ arcs. Thus, both space and time complexities are exponential in the number of arcs of the path to a goal with the fewest arcs. This method is guaranteed, however, to find a solution if one exists and will find a solution with the fewest arcs.

Breadth-first search is useful when

- space is not a problem;
- you want to find the solution containing the fewest arcs;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

3.5.3 Lowest-Cost-First Search

When a non-unit cost is associated with arcs, we often want to find the solution that minimizes the total cost of the path. For example, for a delivery robot, costs may be distances and we may want a solution that gives the minimum total distance. Costs for a delivery robot may be resources required by the robot to carry out the action represented by the arc. The cost for a tutoring system may be the time and effort required by the students. In each of these cases, the searcher should try to minimize the total cost of the path found to reach the goal.

The search algorithms considered thus far are not guaranteed to find the minimum-cost paths; they have not used the arc cost information at all. Breadth-first search finds a solution with the fewest arcs first, but the distribution of arc costs may be such that a path of fewest arcs is not one of minimal cost.

The simplest search method that is guaranteed to find a minimum cost path is similar to breadth-first search; however, instead of expanding a path with the fewest number of arcs, it selects a path with the minimum cost. This is implemented by treating the frontier as a priority queue ordered by the *cost* function (page 76).

Example 3.11 Consider a lowest-cost-first search from $o103$ in the graph given in Figure 3.2 (page 76). The only goal node is $r123$. In this example, paths are denoted by the end node of the path. A subscript shows the cost of the path.

Initially, the frontier is $[o103_0]$. At the next stage it is $[b3_4, ts_8, o109_{12}]$. The path to $b3$ is selected, with the resulting frontier

$$[b1_8, ts_8, b4_{11}, o109_{12}].$$

The path to $b1$ is then selected, resulting in frontier

$[ts_8, c2_{11}, b4_{11}, o109_{12}, b2_{14}]$.

Then the path to ts is selected, and the resulting frontier is

$[c2_{11}, b4_{11}, o109_{12}, mail_{14}, b2_{14}]$.

Then $c2$ is selected, and so forth. Note how the lowest-cost-first search grows many paths incrementally, always expanding the path with lowest cost.

If the costs of the arcs are bounded below by a positive constant and the branching factor is finite, the lowest-cost-first search is guaranteed to find an optimal solution – a solution with lowest path cost – if a solution exists. Moreover, the first path to a goal that is found is a path with least cost. Such a solution is optimal, because the algorithm generates paths from the start in order of path cost. If a better path existed than the first solution found, it would have been selected from the frontier earlier.

The bounded arc cost is used to guarantee the lowest-cost search will find an optimal solution. Without such a bound there can be infinite paths with a finite cost. For example, there could be nodes $n_0, n_1, n_2, \dots$ with an arc n_{i-1}, n_i for each $i > 0$ with cost $1/2^i$. Infinitely many paths of the form $n_0, n_1, n_2, \dots, n_k$ exist, all of which have a cost of less than 1. If there is an arc from n_0 to a goal node with a cost greater than or equal to 1, it will never be selected. This is the basis of Zeno's paradoxes that Aristotle wrote about more than 2,300 years ago.

Like breadth-first search, lowest-cost-first search is typically exponential in both space and time. It generates *all* paths from the start that have a cost less than the cost of the solution.

3.6 Heuristic Search

All of the search methods in the preceding section are uninformed in that they did not take into account the goal. They do not use any information about where they are trying to get to unless they happen to stumble on a goal. One form of heuristic information about which nodes seem the most promising is a heuristic function $h(n)$, which takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node. The function $h(n)$ is an *underestimate* if $h(n)$ is less than or equal to the actual cost of a lowest-cost path from node n to a goal.

The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal.

There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node. Typically a trade-off exists between the amount of work it takes to derive a heuristic value for a node and

how accurately the heuristic value of a node measures the actual path cost from the node to a goal.

A standard way to derive a heuristic function is to solve a simpler problem and to use the actual cost in the simplified problem as the heuristic function of the original problem.

Example 3.12 For the graph of Figure 3.2 (page 76), the straight-line distance in the world between the node and the goal position can be used as the heuristic function.

The examples that follow assume the following heuristic function:

$h(mail)$	$= 26$	$h(ts)$	$= 23$	$h(o103)$	$= 21$
$h(o109)$	$= 24$	$h(o111)$	$= 27$	$h(o119)$	$= 11$
$h(o123)$	$= 4$	$h(o125)$	$= 6$	$h(r123)$	$= 0$
$h(b1)$	$= 13$	$h(b2)$	$= 15$	$h(b3)$	$= 17$
$h(b4)$	$= 18$	$h(c1)$	$= 6$	$h(c2)$	$= 10$
$h(c3)$	$= 12$	$h(storage)$	$= 12$		

This h function is an underestimate because the h value is less than or equal to the exact cost of a lowest-cost path from the node to a goal. It is the exact cost for node $o123$. It is very much an underestimate for node $b1$, which seems to be close, but there is only a long route to the goal. It is very misleading for $c1$, which also seems close to the goal, but no path exists from that node to the goal.

Example 3.13 Consider the delivery robot of Example 3.2 (page 73), where the state space includes the parcels to be delivered. Suppose the cost function is the total distance traveled by the robot to deliver all of the parcels. One possible heuristic function is the largest distance of a parcel from its destination. If the robot could only carry one parcel, a possible heuristic function is the sum of the distances that the parcels must be carried. If the robot could carry multiple parcels at once, this may not be an underestimate of the actual cost.

The h function can be extended to be applicable to (non-empty) paths. The heuristic value of a path is the heuristic value of the node at the end of the path. That is:

$$h((n_0, \dots, n_k)) = h(n_k)$$

A simple use of a heuristic function is to order the neighbors that are added to the stack representing the frontier in depth-first search. The neighbors can be added to the frontier so that the best neighbor is selected first. This is known as **heuristic depth-first search**. This search chooses the locally best path, but it explores all paths from the selected path before it selects another path. Although it is often used, it suffers from the problems of depth-first search.

Another way to use a heuristic function is to always select a path on the frontier with the lowest heuristic value. This is called **best-first search**. It

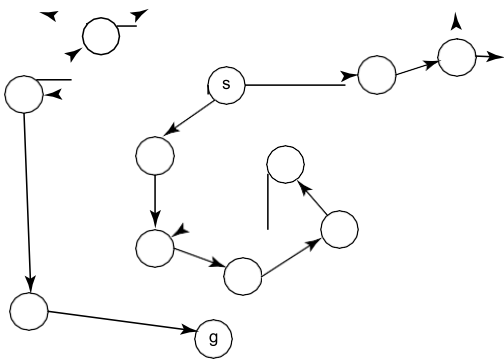


Figure 3.8: A graph that is bad for best-first search

usually does not work very well; it can follow paths that look promising because they are close to the goal, but the costs of the paths may keep increasing.

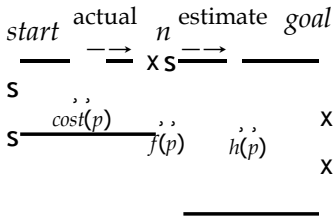
Example 3.14 Consider the graph shown in Figure 3.8, where the cost of an arc is its length. The aim is to find the shortest path from s to g . Suppose the Euclidean distance to the goal g is used as the heuristic function. A heuristic depth-first search will select the node below s and will never terminate. Similarly, because all of the nodes below s look good, a best-first search will cycle between them, never trying an alternate route from s .

3.6.1 A^* Search

A^* **search** is a combination of lowest-cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand. For each path on the frontier, A^* uses an estimate of the total path cost from a start node to a goal node constrained to start along that path. It uses $cost(p)$, the cost of the path found, as well as the heuristic function $h(p)$, the estimated path cost from the end of p to the goal.

For any path p on the frontier, define $f(p) = cost(p) + h(p)$. This is an estimate of the total path cost to follow path p then go to a goal node.

If n is the node at the end of path p , this can be depicted as follows:



If $h(n)$ is an underestimate of the path costs from node n to a goal node, then $f(p)$ is an underestimate of a path cost of going from a start node to a goal node

A^* is implemented by treating the frontier as a priority queue ordered by $f(p)$.

Example 3.15 Consider using A^* search in Example 3.4 (page 76) using the heuristic function of Figure 3.12 (page 88). In this example, the paths on the frontier are shown using the final node of the path, subscripted with the f -value of the path. The frontier is initially $[o103_{21}]$, because $h(o103) = 21$ and the cost of the path is zero. It is replaced by its neighbors, forming the frontier

$$[b3_{21}, ts_{31}, o109_{36}].$$

The first element represents the path $o103, b3$; its f -value is $f(o103, b3) = \text{cost}(o103, b3) + h(b3) = 4 + 17 = 21$. Next $b3$ is selected and replaced by its neighbors, forming the frontier

$$[b1_{21}, b4_{29}, ts_{31}, o109_{36}].$$

Then the path to $b1$ is selected and replaced by its neighbors, forming the frontier

$$[c2_{21}, b4_{29}, b2_{29}, ts_{31}, o109_{36}].$$

Then the path to $c2$ is selected and replaced by its neighbors, forming

$$[c1_{21}, b4_{29}, b2_{29}, c3_{29}, ts_{31}, o109_{36}].$$

Up to this stage, the search has been continually exploring what seems to be the direct path to the goal. Next the path to $c1$ is selected and is replaced by its neighbors, forming the frontier

$$[b4_{29}, b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}].$$

At this stage, there are two different paths to the node $c3$ on the queue. The path to $c3$ that does not go through $c1$ has a lower f -value than the one that does. Later (page 93), we consider the situation when one of these paths can be pruned.

There are two paths with the same f -value. The algorithm does not specify which is selected. Suppose the path to $b4$ is selected next and is replaced by its neighbors, forming

$$[b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}, o109_{42}].$$

Then the path to $b2$ is selected and replaced by its neighbors, which is the empty set, forming

$$[c3_{29}, ts_{31}, c3_{35}, b4_{35}, o109_{36}, o109_{42}].$$

Then the path to $c3$ is removed and has no neighbors; thus, the new frontier is

$$[ts_{31}, c3_{35}, b4_{35}, o109_{36}, o109_{42}].$$

Note how A^* pursues many different paths from the start.

A lowest-cost path is eventually found. The algorithm is forced to try many different paths, because several of them temporarily seemed to have the lowest cost. It still does better than either lowest-cost-first search or best-first search.

Example 3.16 Consider Figure 3.8 (page 89), which was a problematic graph for the other heuristic methods. Although it initially searches down from s because of the heuristic function, eventually the cost of the path becomes so large that it picks the node on an actual optimal path.

The property that A^* always finds an optimal path, if one exists, and that the first path found to a goal is optimal is called the **admissibility** of A^* . Admissibility means that, even when the search space is infinite, if solutions exist, a solution will be found and the first path found will be an optimal solution – a lowest-cost path from a start node to a goal node.

Proposition 3.1. (A^* admissibility): *If there is a solution, A^* always finds a solution, and the first solution found is an optimal solution, if*

- *the branching factor is finite (each node has only a finite number of neighbors),*
- *arc costs are greater than some $c > 0$, and*
- *$h(n)$ is a lower bound on the actual minimum cost of the lowest-cost path from n to a goal node.*

Proof. Part A: A solution will be found. If the arc costs are all greater than some $c > 0$, eventually, for all paths p in the frontier, $cost(p)$ will exceed any finite number and, thus, will exceed a solution cost if one exists (at depth in the search tree no greater than m/c , where m is the solution cost). Because the branching factor is finite, only a finite number of nodes must be expanded before the search tree could get to this size, but the A^* search would have found a solution by then.

Part B: The first path to a goal selected is an optimal path. The f -value for any node on an optimal solution path is less than or equal to the f -value of an optimal solution. This is because h is an underestimate of the actual cost from a node to a goal. Thus, the f -value of a node on an optimal solution path is less than the f -value for any non-optimal solution. Thus, a non-optimal solution can never be chosen while a node exists on the frontier that leads to an optimal solution (because an element with minimum f -value is chosen at each step). So, before it can select a non-optimal solution, it will have to pick all of the nodes on an optimal path, including each of the optimal solutions. $\square$

It should be noted that the admissibility of A^* does not ensure that every intermediate node selected from the frontier is on an optimal path from the start node to a goal node. Admissibility relieves the algorithm from worrying about cycles and ensures that the first solution found will be optimal. It does not ensure that the algorithm will not change its mind about which partial path is the best while it is searching.

To see how the heuristic function improves the efficiency of A^* , suppose c is the cost of a shortest path from a start node to a goal node. A^* , with an admissible heuristic, expands every path from a start node in the set

$$\{p : cost(p) + h(p) < c\}$$

Strategy	Selection from Frontier	Halts?	Space
Depth-first	Last node added	No	Linear
Breadth-first	First node added	Yes	Exponential
Best-first	Globally minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal $cost(p)$	Yes	Exponential
A^*	Minimal $cost(p)+ h(p)$	Yes	Exponential

“Halts?” means “Is the method guaranteed to halt if there is a path to a goal on a (possibly infinite) graph with a finite number of neighbors for each node and where the arc costs have a positive lower bound?” Those search strategies where the answer is “Yes” have worst-case time complexity which increases exponentially with the size of the path length. Those algorithms that are not guaranteed to halt have infinite worst-case time complexity.

Space refers to the space complexity, which is either “Linear” in the path length or “Exponential” in the path length.

Figure 3.9: Summary of search strategies

and some of the paths in the set

$$\{p : cost(p)+ h(p)= c\}.$$

Improving h affects the efficiency of A^* if it reduces the size of the first of these sets.

3.6.2 Summary of Search Strategies

The table in Figure 3.9 gives a summary of the searching strategies presented so far.

The depth-first methods are linear in space with respect to the path lengths explored but are not guaranteed to find a solution if one exists. Breadth-first, lowest-cost-first, and A^* may be exponential in both space and time, but they are guaranteed to find a solution if one exists, even if the graph is infinite (as long as there are finite branching factors and positive non-trivial arc costs).

Lowest-cost-first and A^* searches are guaranteed to find the least-cost solution as the first solution found.

3.7 More Sophisticated Search

A number of refinements can be made to the preceding strategies. First, we present two methods that are applicable when there are cycles in the graph; one checks explicitly for cycles, whereas the other method checks for multiple paths to a node. Next, we present iterative deepening and depth-first branch- and-bound searches, which are general methods that are guaranteed to find

a solution (even an optimal solution), like breadth-first search or A^* search, but using the space advantages of depth-first search. We present problem-reduction methods to break down a search problem into a number of smaller search problems, each of which may be much easier to solve. Finally, we show how dynamic programming can be used for path finding and for constructing heuristic functions.

3.7.1 Cycle Checking

It is possible for a graph representing a search space to include cycles. For example, in the robot delivery domain of Figure 3.6 (page 84), the robot can go back and forth between nodes $o103$ and $o109$. Some of the aforementioned search methods can get trapped in cycles, continuously repeating the cycle and never finding an answer even in finite graphs. The other methods can loop through cycles, but eventually they still find a solution.

The simplest method of pruning the search tree, while guaranteeing that a solution will be found in a finite graph, is to ensure that the algorithm does not consider neighbors that are already on the path from the start. A **cycle check** or **loop check** checks for paths where the last node already appears on the path from the start node to that node. With a cycle check, only the paths $s_0, \dots, s_k$, where $s_k / s_0, \dots, s_{k-1} \notin \{s_0, \dots, s_{k-1}\}$ are added to the frontier at line 16 of Figure 3.4 (page 79). Alternatively, the check can be made after a node is selected; paths with a cycle can be thrown away.

The computational complexity of a cycle check depends on the search method being used. For depth-first methods, where the graph is explicitly stored, the overhead can be as low as a constant factor; a bit can be added to each node in the graph that is assigned a value of 1 when the node is expanded, and assigned a value of 0 on backtracking. A search algorithm can avoid cycles by never expanding a node with its bit set to 1. This approach works because depth-first search maintains a single current path. The elements on the frontier are alternative branches from this path. Even if the graph is generated dynamically, as long as an efficient indexing structure is used for the nodes on the current path, a cycle check can be done efficiently.

For the search strategies that maintain multiple paths – namely, all of those with exponential space in Figure 3.9 – a cycle check takes time linear in the length of the path being searched. These algorithms cannot do better than searching up the partial path being considered, checking to ensure they do not add a node that already appears in the path.

3.7.2 Multiple-Path Pruning

There is often more than one path to a node. If only one path is required, a search algorithm can prune from the frontier any path that leads to a node to which it has already found a path.

Multiple-path pruning can be implemented by keeping a **closed list** of nodes that have been expanded. When a path is selected at line 13 of Figure 3.4 (page 79), if its last node is in the closed list, the path can be discarded. Otherwise, its last node is added to the closed list, and the algorithm proceeds as before.

This approach does not necessarily guarantee that the shortest path is not discarded. Something more sophisticated may have to be done to guarantee that an optimal solution is found. To ensure that the search algorithm can still find a lowest-cost path to a goal, one of the following can be done:

- Make sure that the first path found to any node is a lowest-cost path to that node, then prune all subsequent paths found to that node, as discussed earlier.
- If the search algorithm finds a lower-cost path to a node than one already found, it can remove all paths that used the higher-cost path to the node (because these cannot be on an optimal solution). That is, if there is a path p on the frontier $s, (\dots, n, \dots, m)$ and a path p^t to n is found that is shorter than the portion of the path from s to n in p , then p can be removed from the frontier.
- Whenever the search finds a lower-cost path to a node than a path to that already found, it can incorporate a new initial section on the paths that have extended the initial path. Thus, if there is a path $p = (s, \dots, n, \dots, m)$ on the frontier, and a path p^t to n is found that is shorter than the portion of p from s to n , then p^t can replace the initial part of p to n .

The first of these alternatives allows the use of the closed list without losing the ability to find an optimal path. The others require more sophisticated algorithms.

In lowest-cost-first search, the first path found to a node (i.e., when the node is selected from the frontier) is the least-cost path to the node. Pruning subsequent paths to that node cannot remove a lower-cost path to that node, and thus pruning subsequent paths to each node still enables an optimal solution to be found.

As described earlier, A^* (page 89) does not guarantee that when a path to a node is selected for the first time it is the lowest cost path to that node. Note that the admissibility theorem (page 91) guarantees this for every path to a *goal* node but not for every path. To see when pruning subsequent paths to a node can remove the optimal solution, suppose the algorithm has selected a path p to node n for expansion, but there exists a lower-cost path to node n , which it has not found yet. Then there must be a path p^t on the frontier that is the initial part of the lower-cost path. Suppose path p^t ends at node n^t . It must be that $f(p) \leq f(p^t)$, because p was selected before p^t . This means that

$$\text{cost}(p) + h(n) \leq \text{cost}(p^t) + h(n^t).$$

If the path to n via p^t has a lower cost than the path p ,

$$\text{cost}(p^t) + d(n^t, n) < \text{cost}(p),$$

where $d(n^t, n)$ is the actual cost of the shortest path from node n^t to n . From these two equations, we can derive

$$d(n^t, n) < \text{cost}(p) - \text{cost}(p^t) \leq h(p^t) - h(p) = h(n^t) - h(n).$$

Thus, we can ensure that the first path found to any node is the lowest-cost path if $h(n^t) \leq h(n) + d(n^t, n)$ for any two nodes n and n^t . The **monotone restriction** on h is that $h(n^t) \leq h(n) + d(n^t, n)$ for any two nodes n and n^t . That is, the difference in the heuristic values for two nodes must be less than or equal to the actual cost of the lowest-cost path between the nodes. It is applicable to, for example, the heuristic function of Euclidean distance (the straight-line distance in an n -dimensional Euclidean space) between two points when the cost function is distance. It is also typically applicable when the heuristic function is a solution to a simplified problem that has shorter solutions.

With the monotone restriction, the f -values on the frontier are monotonically non-decreasing. That is, when the frontier is expanded, the f -values do not get smaller. Thus, with the monotone restriction, subsequent paths to any node can be pruned in A^* search.

Multiple-path pruning subsumes a cycle check, because a cycle is another path to a node and is therefore pruned. Multiple-path pruning can be done in constant time, if the graph is explicitly stored, by setting a bit on each node to which a path has been found. It can be done in logarithmic time (in the number of nodes expanded, as long as it is indexed appropriately), if the graph is dynamically generated, by storing the closed list of all of the nodes that have been expanded. Multiple-path pruning is preferred over cycle checking for breadth-first methods where virtually all of the nodes considered have to be stored anyway. For depth-first search strategies, however, the algorithm does not otherwise have to store all of the nodes already considered. Storing them makes the method exponential in space. Therefore, cycle checking is preferred over multiple-path checking for depth-first methods.

3.7.3 Iterative Deepening

So far, none of the methods discussed have been ideal; the only ones that guarantee that a path will be found require exponential space (see Figure 3.9 (page 92)). One way to combine the space efficiency of depth-first search with the optimality of breadth-first methods is to use **iterative deepening**. The idea is to recompute the elements of the frontier rather than storing them. Each recomputation can be a depth-first search, which thus uses less space.

Consider making a breadth-first search into an iterative deepening search. This is carried out by having a depth-first searcher, which searches only to a limited depth. It can first do a depth-first search to depth 1 by building paths of length 1 in a depth-first manner. Then it can build paths to depth 2, then depth 3, and so on. It can throw away all of the previous computation each time and start again. Eventually it will find a solution if one exists, and, as it is

enumerating paths in order, the path with the fewest arcs will always be found first.

When implementing an iterative deepening search, you have to distinguish between

- failure because the depth bound was reached and
- failure that does not involve reaching the depth bound.

In the first case, the search must be retried with a larger depth bound. In the second case, it is a waste of time to try again with a larger depth bound, because no path exists no matter what the depth. We say that failure due to reaching the depth bound is **failing unnaturally**, and failure without reaching the depth bound is **failing naturally**.

An implementation of iterative-deepening search, *IdSearch*, is presented in Figure 3.10. The local procedure *dbsearch* implements a depth-bounded depth-first search (using recursion to keep the stack) that places a limit on the length of the paths for which it is searching. It uses a depth-first search to find all paths of length $k + b$, where k is the path length of the given path from the start and b is a non-negative integer. The iterative-deepening searcher calls this for increasing depth bounds. This program finds the paths to goal nodes in the same order as does the breadth-first search. As in the generic graph searching algorithm, to find more solutions after the *return* on line 22, the search can continue from line 23.

The iterative-deepening search fails whenever the breadth-first search would fail. When asked for multiple answers, it only returns each successful path once, even though it may be rediscovered in subsequent iterations. Halting is achieved by keeping track of when increasing the bound could help find an answer:

- The depth bound is increased if the depth bound search was truncated by reaching the depth bound. In this case, the search failed *unnaturally*. The search failed *naturally* if the search did not prune any paths due to the depth bound. In this case, the program can stop and report no (more) paths.
- The search only reports a solution path if that path would not have been reported in the previous iteration. Thus, it only reports paths whose length is the depth bound.

The obvious problem with iterative deepening is the wasted computation that occurs at each step. This, however, may not be as bad as one might think, particularly if the branching factor is high. Consider the running time of the algorithm. Assume a constant branching factor of $b > 1$. Consider the search where the bound is k . At depth k , there are b^k nodes; each of these has been generated once. The nodes at depth $k - 1$ have been generated twice, those at depth $k - 2$ have been generated three times, and so on, and the nodes at depth 1 have been generated k times. Thus, the total number of nodes

```

1: procedure IdSearch(G, s, goal)
2:   Inputs
3:     G: graph with nodes N and arcs A
4:     s: set of start nodes
5:     goal: Boolean function on states
6:   Output
7:     path from s to a node for which goal is true
8:     or  $\perp$  if there is no such path
9:   Local
10:    natural failure: Boolean
11:    bound: integer
12:    procedure dbsearch( $\langle n_0, \dots, n_k \rangle, b$ )
13:      Inputs
14:         $\langle n_0, \dots, n_k \rangle$ : path
15:        b: integer,  $b \geq 0$ 
16:      Output
17:        path to goal of length  $k + b$ 
18:      if  $b > 0$  then
19:        for each arc  $(n_k, n) \in A$  do
20:          dbsearch( $\langle n_0, \dots, n_k, n \rangle, b - 1$ )
21:        else if goal( $n_k$ ) then
22:          return  $\langle n_0, \dots, n_k \rangle$ 
23:        else if  $n_k$  has any neighbors then
24:          natural failure := false
25:      bound := 0
26:      repeat
27:        natural failure := true
28:        dbsearch( $\{ \langle s \rangle : s \in S \}, \text{bound}$ )
29:        bound := bound + 1
30:      until natural failure
31:      return  $\perp$ 

```

Figure 3.10: Iterative deepening search

generated is

$$\begin{aligned}
 & b^k + 2b^{k-1} + 3b^{k-2} + \dots + kb \\
 &= b^k \sum_{i=0}^{\infty} (1+i) b^{-i} \\
 &\leq b \sum_{i=1}^{\infty} i b^{i-1} \\
 &= b_k \cdot \frac{b}{b-1} \cdot \Sigma_2.
 \end{aligned}$$

There is a constant overhead $(b/(b-1))^2$ times the cost of generating the nodes at depth n . When $b = 2$ there is an overhead factor of 4, and when $b = 3$ there is an overhead of 2.25 over generating the frontier. This algorithm is $O(b^k)$ and there cannot be an asymptotically better uninformed search strategy. Note that, if the branching factor is close to 1, this analysis does not work (because then the denominator would be close to zero); see Exercise 3.9 (page 109).

Iterative deepening can also be applied to an A^* search. **Iterative deepening A^*** (IDA*) performs repeated depth-bounded depth-first searches. Instead of the bound being on the number of arcs in the path, it is a bound on the value of $f(n)$. The threshold starts at the value of $f(s)$, where s is the starting node with minimal h -value. IDA* then carries out a depth-first depth-bounded search but never expands a node with a higher f -value than the current bound. If the depth-bounded search fails *unnaturally*, the next bound is the minimum of the f -values that exceeded the previous bound. IDA* thus checks the same nodes as A^* but recomputes them with a depth-first search instead of storing them.

3.7.4 Branch and Bound

Depth-first **branch-and-bound** search is a way to combine the space saving of depth-first search with heuristic information. It is particularly applicable when many paths to a goal exist and we want an optimal path. As in A^* search, we assume that $h(n)$ is less than or equal to the cost of a lowest-cost path from n to a goal node.

The idea of a branch-and-bound search is to maintain the lowest-cost path to a goal found so far, and its cost. Suppose this cost is *bound*. If the search encounters a path p such that $\text{cost}(p) + h(p) \geq \text{bound}$, path p can be pruned. If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to the cost of this new solution. It then keeps searching for a better solution.

Branch-and-bound search generates a sequence of ever-improving solutions. Once it has found a solution, it can keep improving it.

Branch-and-bound search is typically used with depth-first search, where the space saving of the depth-first search can be achieved. It can be implemented similarly to depth-bounded search, but where the bound is in terms of path cost and reduces as shorter paths are found. The algorithm remembers the lowest-cost path found and returns this path when the search finishes.

The algorithm is shown in Figure 3.11. The internal procedure *cbsearch*, for cost-bounded search, uses the global variables to provide information to the main procedure.

Initially, *bound* can be set to infinity, but it is often useful to set it to an overestimate, *bound₀*, of the path cost of an optimal solution. This algorithm

```

1: procedure DFBranchAndBound(G, s, goal, h, bound0)
2:   Inputs
3:     G: graph with nodes N and arcs A
4:     s: start node
5:     goal: Boolean function on nodes
6:     h: heuristic function on nodes
7:     bound0: initial depth bound (can be  $\infty$  if not specified)
8:   Output
9:   a least-cost path from s to a goal node if there is a solution with cost
    less than bound0
10:  or  $\perp$  if there is no solution with cost less than bound0
11:  Local
12:    best_path: path or  $\perp$ 
13:    bound: non-negative real
14:    procedure cbsearch(n0, ..., nk)
15:      if cost(n0, ..., nk) + h(nk) < bound then
16:        if goal(nk) then
17:          best_path := n0, ..., nk
18:          bound := cost(n0, ..., nk)
19:        else
20:          for each arc (nk, n) ∈ A do
21:            cbsearch(n0, ..., nk, n)
22:    best_path :=  $\perp$ 
23:    bound := bound0
24:    cbsearch(s)
25:  return best_path

```

Figure 3.11: Depth-first branch-and-bound search

will return an optimal solution – a least-cost path from the start node to a goal node – if there is a solution with cost less than the initial bound *bound*₀.

If the initial bound is slightly above the cost of a lowest-cost path, this algorithm can find an optimal path expanding no more arcs than *A** search. This happens when the initial bound is such that the algorithm prunes any path that has a higher cost than a lowest-cost path; once it has found a path to the goal, it only explores paths whose the *f*-value is lower than the path found. These are exactly the paths that *A** explores when it finds one solution.

If it returns $\perp$ when *bound*₀ = ∞ , there are no solutions. If it returns $\perp$ when *bound*₀ is some finite value, it means no solution exists with cost less than *bound*₀. This algorithm can be combined with iterative deepening to increase the bound until either a solution is found or it can be shown there is no solution. See Exercise 3.13 (page 109).

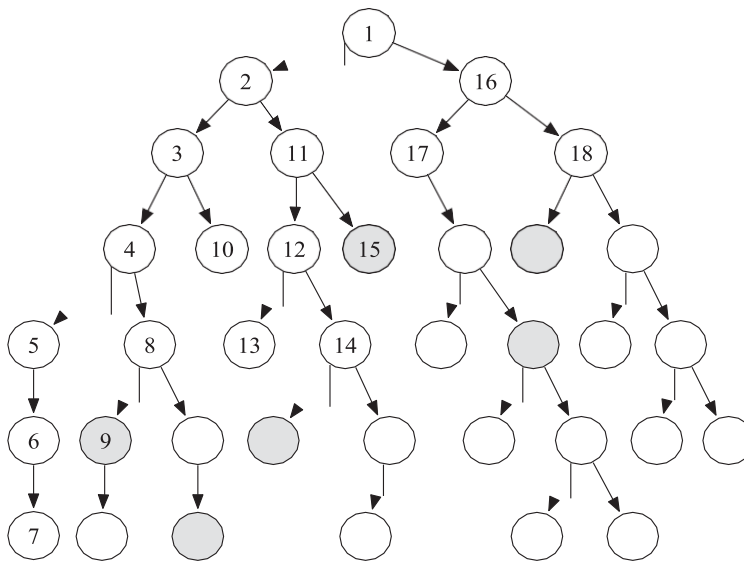


Figure 3.12: The nodes expanded in depth-first branch-and-bound search.

Example 3.17 Consider the tree-shaped graph in Figure 3.12. The goal nodes are shaded. Suppose that each arc has length 1, and there is no heuristic information (i.e., $h(n) = 0$ for each node n). In the algorithm, suppose $depth_0 = \infty$ and the depth-first search always chooses the leftmost child first. This figure shows the order in which the nodes are checked to determine if they are a goal node. The nodes that are not numbered are not checked for being a goal node.

The subtree under the node numbered “5” does not have a goal and is explored fully (or up to depth $depth_0$ if it had a finite value). The ninth node checked is a goal node. It has a path cost of 5, and so the bound is set to 5. From then on, only paths with a length of less than 5 are checked for being a solution. The fifteenth node checked is also a goal. It has a path cost of 3, and so the bound is reduced to 3. There are no other goal nodes found, and so the path to the node labeled 15 is returned. It is an optimal path. There is another optimal path that is pruned; the algorithm never checks the children of the node labeled with 18.

If there was heuristic information, it could be used to prune parts of the search space, as in A^* search.

3.7.5 Direction of Search

The size of the search space of the generic search algorithm on finite uniform graphs is b^k , where b is the branching factor and k is the path length. Anything that can be done to reduce k or b can potentially give great savings.

The abstract definition of the graph-searching method of problem solving is symmetric in the sense that the algorithm can either begin with a start node and

search forward for a goal node or begin with a goal node and search backward for a start node in the inverse graph. Note that in many applications the goal is determined implicitly by a Boolean function that returns *true* when a goal is found, and not explicitly as a set of nodes, so backward search may not be possible.

For those cases where the goal nodes are explicit, it may be more efficient to search in one direction than in the other. The size of the search space is exponential in the branching factor. It is typically the case that forward and backward searches have different branching factors. A general principle is to search forward or backward, depending on which has the smaller branching factor.

The following sections consider some ways in which search efficiency can be improved beyond this for many search spaces.

Bidirectional Search

The idea of a bidirectional search is to reduce the search time by searching forward from the start and backward from the goal simultaneously. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start state through the frontier intersection to the goal.

A new problem arises during a bidirectional search, namely ensuring that the two search frontiers actually meet. For example, a depth-first search in both directions is not likely to work well because its small search frontiers are likely to pass each other by. Breadth-first search in both directions would be guaranteed to meet.

A combination of depth-first search in one direction and breadth-first search in the other would guarantee the required intersection of the search frontiers, but the choice of which to apply in which direction may be difficult. The decision depends on the cost of saving the breadth-first frontier and searching it to check when the depth-first method will intersect one of its elements.

There are situations where a bidirectional search can result in substantial savings. For example, if the forward and backward branching factors of the search space are both b , and the goal is at depth k , then breadth-first search will take time proportional to b^k , whereas a symmetric bidirectional search will take time proportional to $2b^{k/2}$. This is an exponential savings in time, even though the time complexity is still exponential. Note that this complexity analysis assumes that finding the intersection of frontiers is free, which may not be a valid assumption for many applications (see Exercise 3.10 (page 109)).

Island-Driven Search

One of the ways that search may be made more efficient is to identify a limited number of places where the forward search and backward search can meet. For example, in searching for a path from two rooms on different floors, it may be appropriate to constrain the search to first go to the elevator on one level,

then to the elevator on the goal level. Intuitively, these designated positions are **islands** in the search graph, which are constrained to be on a solution path from the start node to a goal node.

When islands are specified, an agent can decompose the search problem into several search problems, for example, one from the initial room to the elevator, one from the elevator on one level to the elevator on the other level, and one from the elevator to the destination room. This reduces the search space by having three simpler problems to solve. Having smaller problems helps to reduce the combinatorial explosion of large searches and is an example of how extra knowledge about a problem can be used to improve efficiency of search.

To find a path between s and g using islands:

1. Identify a set of islands $i_0, \dots, i_k$;
2. Find paths from s to i_0 , from i_{j-1} to i_j for each j from 1 to k , and from i_k to g .

Each of these searching problems should be correspondingly simpler than the general problem and therefore easier to solve.

The identification of islands is extra knowledge which may be beyond that which is in the graph. The use of inappropriate islands may make the problem more difficult (or even impossible to solve). It may also be possible to identify an alternate decomposition of the problem by choosing a different set of islands and search through the space of possible islands. Whether this works in practice depends on the details of the problem.

Searching in a Hierarchy of Abstractions

The notion of islands can be used to define problem-solving strategies that work at multiple levels of detail or multiple levels of abstraction.

The idea of searching in a hierarchy of abstractions first involves abstracting the problem, leaving out as many details as possible. A partial solution to a problem may be found – one that requires further details to be worked out. For example, the problem of getting from one room to another requires the use of many instances of turning, but an agent would like to reason about the problem at a level of abstraction where the details of the actual steering are omitted. It is expected that an appropriate abstraction solves the problem in broad strokes, leaving only minor problems to be solved. The route planning problem for the delivery robot is too difficult to solve by searching without leaving out details until it must consider them.

One way this can be implemented is to generalize island-driven search to search over possible islands. Once a solution is found at the island level, sub-problems can be solved recursively in the same manner. Information that is found at the lower level can inform higher levels that some potential solution does not work as well as expected. The higher level can then use that information to replan. This process typically does not result in a guaranteed optimal solution because it only considers some of the high-level decompositions.

Searching in a hierarchy of abstractions depends very heavily on *how* one decomposes and *abstracts* the problem to be solved. Once the problems are abstracted and decomposed, any of the search methods can be used to solve them. It is not easy, however, to recognize useful abstractions and problem decompositions.

3.7.6 Dynamic Programming

Dynamic programming is a general method for optimization that involves storing partial solutions to problems, so that a solution that has already been found can be retrieved rather than being recomputed. Dynamic programming algorithms are used throughout AI.

Dynamic programming can be used for finding paths in graphs. Intuitively, **dynamic programming** for graph searching can be seen as constructing the *perfect* heuristic function so that A^* , even if it keeps only one element of the frontier, is guaranteed to find a solution. This cost-to-goal function represents the exact cost of a minimal-cost path from each node to the goal.

A **policy** is a specification of which arc to take from each node. The cost-to-goal function can be computed offline and can be used to build an optimal policy. Online, an agent can use this policy to determine what to do at each point.

Let $cost\ to\ goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\ to\ goal(n)$ can be defined as

$$cost\_to\_goal(n) = \begin{cases} 0 & \text{if } is\_goal(n), \\ \min_{(n,m) \in A} (cost((n,m)) + cost\_to\_goal(m)) & \text{otherwise.} \end{cases}$$

The general idea is to start at the goal and build a table of the $cost\ to\ goal(n)$ value for each node. This can be done by carrying out a lowest-cost-first search, with multiple-path pruning, from the goal nodes in the *inverse graph*, which is the graph with all arcs reversed. Rather than having a goal to search for, the dynamic programming algorithm records the $cost\ to\ goal$ values for each node found. It uses the inverse graph to compute the costs from each node to the goal and not the costs from the goal to each node. In essence, dynamic programming works backward from the goal by trying to build the lowest-cost paths to the goal from each node in the graph.

For a particular goal, once the $cost\ to\ goal$ value for each node has been recorded, an agent can use the $cost\ to\ goal$ value to determine the next arc on an optimal path. From node n it should go to a neighbor m that minimizes $cost((n,m)) + cost\_to\_goal(m)$. Following this policy will take the agent from any node to a goal along a lowest-cost path. Given $cost\ to\ goal$, determining which arc is optimal takes constant time with respect to the size of the graph, assuming a bounded number of neighbors for each node. Dynamic programming takes time and space linear in the size of the graph to build the $cost\ to\ goal$ table.

Dynamic programming is useful when

- the goal nodes are explicit (the previous methods only assumed a function that recognizes goal nodes);
- a lowest-cost path is needed;
- the graph is finite and small enough to be able to store the *cost to goal* value for each node;
- the goal does not change very often; and
- the policy is used a number of times for each goal, so that the cost of generating the *cost to goal* values can be amortized over many instances of the problem.

The main problems with dynamic programming are that

- it only works when the graph is finite and the table can be made small enough to fit into memory,
- an agent must recompute a policy for each different goal, and
- the time and space required is linear in the size of the graph, where the graph size for finite graphs is typically exponential in the path length.

Example 3.18 For the graph given in Figure 3.2 (page 76), the cost from *r123* to the goal is 0; thus,

$$\text{cost\_to\_goal}(r123) = 0.$$

Continuing with a lowest-cost-first search from *r123*:

$$\begin{aligned}\text{cost\_to\_goal}(o123) &= 4 \\ \text{cost\_to\_goal}(o119) &= 13 \\ \text{cost\_to\_goal}(o109) &= 29 \\ \text{cost\_to\_goal}(b4) &= 36 \\ \text{cost\_to\_goal}(b2) &= 39 \\ \text{cost\_to\_goal}(o103) &= 41 \\ \text{cost\_to\_goal}(b3) &= 43 \\ \text{cost\_to\_goal}(b1) &= 45\end{aligned}$$

At this stage the backward search halts. Two things can be noticed here. First, if a node does not have a *cost to goal* value, then no path to the goal exists from that node. Second, an agent can quickly determine the next arc on a lowest-cost path to the goal for any node. For example, if the agent is at *o103*, to determine a lowest-cost path to *r123* it compares $4 + 43$ (the cost of going via *b3*) with $12 + 29$ (the cost of going straight to *o109*) and can quickly determine to go to *o109*.

When building the *cost to goal* function, the searcher has implicitly determined which neighbor leads to the goal. Instead of determining at run time which neighbor is on an optimal path, it can store this information.

Optimality of the A^* algorithm

A search algorithm is **optimal** if no other search algorithm uses less time or space or expands fewer nodes, both with a guarantee of solution quality. The optimal search algorithm would be one that picks the correct node at each choice. However, this specification is not effective because we cannot directly implement it. Whether such an algorithm is possible is an open question (as to whether $P = NP$). There does, however, seem to be a statement that can be proved.

Optimality of A^* : Among search algorithms that only use arc costs and a heuristic estimate of the cost from a node to a goal, no algorithm expands fewer nodes than A^* and guarantees to find a lowest-cost path.

Proof sketch: Given only the information about the arc costs and the heuristic information, unless the algorithm has expanded each path p , where $f(p)$ is less than the cost of an optimal path, it does not know whether p leads to a lower-cost path. More formally, suppose an algorithm A^t found a path for a problem P where some path p was not expanded such that $f(p)$ was less than the solution found. Suppose there was another problem P^t , which was the same as P , except that there really was a path via p with cost $f(p)$. The algorithm A^t cannot tell P^t from P , because it did not expand the path p , so it would report the same solution for P^t as for P , but the solution found for P would not be optimal for P^t because the solution found has a higher cost than the path via p . Therefore, an algorithm is not guaranteed to find a lowest-cost path unless it explores all paths with f -values less than the value of an optimal path; that is, it must explore all the paths that A^* explores.

Counterexample: Although this proof seems reasonable, there are algorithms that explore fewer nodes. Consider an algorithm that does a forward A^* -like search and a backward dynamic programming search, where the steps are interleaved in some way (e.g., by alternating between the forward steps and the backward steps). The backward search builds a table of *cost to goal*(n) values of the actual discovered cost from n to a goal, and it maintains a bound b , where it has explored all paths of cost less than b to a goal. The forward search uses a priority queue on $\text{cost}(p) + c(n)$, where n is the node at the end of the path p , and $c(n)$ is *cost to goal*(n) if it has been computed; otherwise, $c(n)$ is $\max(h(n), b)$. The intuition is that, if a path exists from the end of path p to a goal node, either it uses a path that has been discovered by the backward search or it uses a path that costs at least b . This algorithm is guaranteed to find a lowest-cost path and often expands fewer nodes than A^* (see Exercise 3.11 (page 109)).

Conclusion: Having a counterexample would seem to mean that the optimality of A^* is false. However, the proof does seem to have some appeal and perhaps it should not be dismissed outright. A^* is not optimal out of the class of all algorithms, but the proof seems right for the class of algorithms that only do forward search. See Exercise 3.12 (page 109).

Dynamic programming can be used to construct heuristics for A^* and branch-and-bound searches. One way to build a heuristic function is to simplify the problem (e.g., by leaving out some details) until the simplified problem has a small enough state space. Dynamic programming can be used to find the optimal path length to a goal in the simplified problem. This information can then be used as a heuristic for the original problem.

3.8 References and Further Reading

There is a lot of information on search techniques in the literature of operations research, computer science, and AI. Search was seen early on as one of the foundations of AI. The AI literature emphasizes heuristic search.

Basic search algorithms are discussed in Nilsson [1971]. For a detailed analysis of heuristic search see Pearl [1984]. The A^* algorithm was developed by Hart, Nilsson, and Raphael [1968].

Depth-first iterative deepening is described in Korf [1985].

Branch-and-bound search was developed in the operations research community and is described in Lawler and Wood [1966].

Dynamic programming is a general algorithm that will be used as a dual to search algorithms in other parts of the book. The specific algorithm presented here was invented by Dijkstra [1959]. See Cormen, Leiserson, Rivest, and Stein [2001] for more details on the general class of dynamic programming algorithms.

The idea of using dynamic programming as a source of heuristics for A^* search was proposed by Culberson and Schaeffer [1998] and further developed by Felner, Korf, and Hanan [2004].

Minsky [1961] discussed islands and problem reduction.

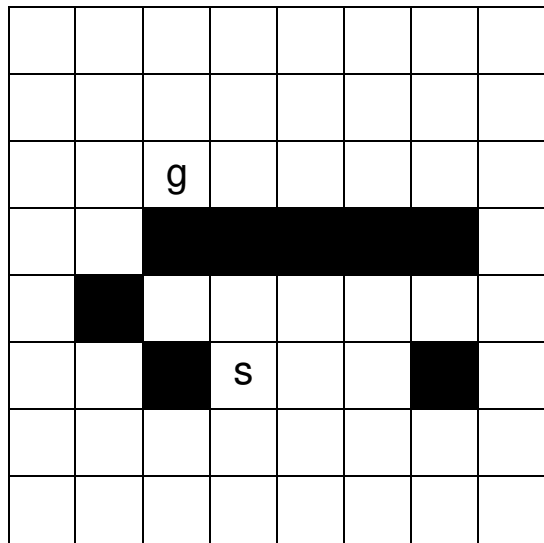


Figure 3.13: A grid-searching problem

Chapter 4

Features and Constraints

*Every task involves constraint,
Solve the thing without complaint;
There are magic links and chains
Forged to loose our rigid brains.
Structures, strictures, though they bind,
Strangely liberate the mind.*

– James Falen

Instead of reasoning explicitly in terms of states, it is often better to describe states in terms of features and then to reason in terms of these features. Often these features are not independent and there are **hard constraints** that specify legal combinations of assignments of values to variables. As Falen’s elegant poem emphasizes, the mind discovers and exploits constraints to solve tasks. Common examples occur in planning and scheduling, where an agent must assign a time for each action that must be carried out; typically, there are constraints on when actions can be carried out and constraints specifying that the actions must actually achieve a goal. There are also often preferences over values that can be specified in terms of **soft constraints**. This chapter shows how to generate assignments that satisfy a set of hard constraints and how to optimize a collection of soft constraints.

4.1 Features and States

For any practical problem, an agent cannot reason in terms of states; there are simply too many of them. Moreover, most problems do not come with an explicit list of states; the states are typically described implicitly in terms of

features. When describing a real state space, it is usually more natural to describe the features that make up the state rather than explicitly enumerating the states.

The definitions of states and features are intertwined; we can describe either in terms of the other.

- States can be defined in terms of features: features can be primitive and a state corresponds to an assignment of a value to each feature.
- Features can be defined in terms of states: the states can be primitive and a feature is a function of the states. Given a state, the function returns the value of the feature on that state.

Each feature has a **domain** that is the set of values that it can take on. The domain of the feature is the range of the function on the states.

Example 4.1 In the electrical environment of Figure 1.8 (page 34), there may be a feature for the position of each switch that specifies whether the switch is up or down. There may be a feature for each light that specifies whether the light is lit or not. There may be a feature for each component specifying whether it is working properly or if it is broken. A state consists of the position of every switch, the status of every device, and so on.

If the features are primitive, a state is an assignment of a value to each feature. For example, a state may be described as switch 1 is up, switch 2 is down, fuse1 is okay, wire3 is broken, and soon.

If the states are primitive, the functions may be, for example, the position of switch 1. The position is a function of the state, and it may be *up* in some states and *down* in other states.

One main advantage of reasoning in terms of features is the computational savings. For a **binary feature** the domain has two values. Many states can be described by a few features:

- 10 binary features can describe $2^{10} = 1,024$ states.
- 20 binary features can describe $2^{20} = 1,048,576$ states.
- 30 binary features can describe $2^{30} = 1,073,741,824$ states.
- 100 binary features can describe $2^{100} = 1,267,650,600,228,229,401,496,703,205,376$ states.

Reasoning in terms of thirty features may be easier than reasoning in terms of more than a billion states. One hundred features is not that many, but reasoning in terms of more than 2^{100} states explicitly is not possible. Many problems have thousands if not millions of features.

Typically the features are not independent, in that there may be constraints on the values of different features. One problem is to determine what states are possible given the features and the constraints.

4.2 Possible Worlds, Variables, and Constraints

To keep the formalism simple and general, we develop the notion of features without considering time explicitly. Constraint satisfaction problems will be described in terms of possible worlds.

When we are not modeling change, there is a direct one-to-one correspondence between features and variables, and between states and possible worlds. A **possible world** is a possible way the world (the real world or some imaginary world) could be. For example, when representing a crossword puzzle, the possible worlds correspond to the ways the crossword could be filled out. In the electrical environment, a possible world specifies the position of every switch and the status of every component.

Possible worlds are described by algebraic variables. An **algebraic variable** is a symbol used to denote features of possible worlds. Algebraic variables will be written starting with an upper-case letter. Each algebraic variable V has an associated **domain**, $dom(V)$, which is the set of values the variable can take on.

For this chapter, we refer to an algebraic variable simply as a **variable**. These algebraic variables are different from the variables used in logic, which are discussed in Chapter 12. Algebraic variables are the same as the random variables used in probability theory, which are discussed in Chapter 6.

A **discrete variable** is one whose domain is finite or countably infinite. One particular case of a discrete variable is a **Boolean variable**, which is a variable with domain $true, false$. If X is a Boolean variable, we write $X = true$ as its lower-case equivalent, x , and write $X = false$ as x . We can also have variables that are not discrete; for example, a variable whose domain corresponds to a subset of the real line is a **continuous variable**.

Example 4.2 The variable *Class time* may denote the starting time for a particular class. The domain of *Class time* may be the following set of possible times:

$$dom(Class\ time) = \{8, 9, 10, 11, 12, 1, 2, 3, 4, 5\}.$$

The variable *Height joe* may refer to the height of a particular person at a particular time and have as its domain the set of real numbers, in some range, that represent the height in centimeters. *Raining* may be a Boolean random variable with value *true* if it is raining at a particular time.

Example 4.3 Consider the electrical domain depicted in Figure 1.8 (page 34).

- $S_1.pos$ may be a discrete binary variable denoting the position of switch s_1 with domain $up, down$, where $S_1.pos = up$ means switch s_1 is up, and $S_1.pos = down$ means switch s_1 is down.
- $S_1.st$ may be a variable denoting the status of switch s_1 with domain $ok, upside\ down, short, intermittent, broken$, where $S_1.st = ok$ means switch s_1 is working normally, $S_1.st = upside\ down$ means switch s_1 is installed upside down, $S_1.st = short$ means switch s_1 is shorted and acting as a wire, $S_1.st = intermittent$ means switch s_1 is working intermittently, and $S_1.st = broken$ means switch s_1 is broken and does not allow electricity to flow.

- *Number_of_broken_switches* may be an integer-valued variable denoting the number of switches that are broken.
- *Current.w₁* may be a real-valued variable denoting the current, in amps, flowing through wire *w₁*. *Current.w₁ = 1.3* means there are 1.3 amps flowing through wire *w₁*. We also allow inequalities between variables and constants as Boolean features; for example, *Current w₁ ≥ 1.3* is true when there are at least 1.3 amps flowing through wire *w₁*.

Symbols and

Algebraic variables are symbols.

Internal to a computer, a **symbol** is just a sequence of bits that can be distinguished from other symbols. Some symbols have a fixed interpretation, for example, symbols that represent numbers and symbols that represent characters. Symbols that do not have fixed meaning appear in many programming languages. In Java, starting from Java 1.5, they are called *enumeration types*. Lisp refers to them as *atoms*. Usually, they are implemented as indexes into a symbol table that gives the name to print out. The only operation performed on these symbols is equality to determine if two symbols are the same or not. To a **user** of a computer, symbols have meanings. A person who inputs constraints or interprets the output associates meanings with the symbols that make up the constraints or the outputs. He or she associates a symbol with some concept or object in the world. For example, the variable *HarrysHeight*, to the computer, is just a sequence of bits. It has no relationship to *HarrysWeight* or *SuesHeight*. To a person, this variable may mean the height, in particular units, of a particular person at a particular time.

The meaning associated with a variable-value pair must satisfy the **clarity principle**: an **omniscient agent** – a fictitious agent who knows the truth and the meanings associated with all of the symbols – should be able to determine the value of each variable. For example, the *height of Harry* only satisfies the clarity principle if the particular person being referred to and the particular time are specified as well as the units. For example, we may want to reason about the height, in centimeters, of Harry Potter at the start of the second movie of J. K. Rowling's book. This is different from the height, in inches, of Harry Potter at the end of the same movie (although they are, of course, related). If you want to refer to Harry's height at two different times, you must have two different variables.

You should have a consistent meaning. When stating constraints, you must have the same meaning for the same variable and the same values, and you can use this meaning to interpret the output.

The bottom line is that symbols can have meanings because we give them meanings. For this chapter, assume that the computer does not know what the symbols mean. A computer can only know what a symbol means if it can perceive and manipulate the environment.

Example 4.4 A classic example of a constraint satisfaction problem is a crossword puzzle. There are two different representations of crossword puzzles in terms of variables:

1. In one representation, the variables are the numbered squares with the direction of the word (down or across), and the domains are the set of possible words that can be put in. A possible world corresponds to an assignment of a word for each of the variables.
2. In another representation of a crossword, the variables are the individual squares and the domain of each variable is the set of letters in the alphabet. A possible world corresponds to an assignment of a letter to each square.

Possible worlds can be defined in terms of variables or variables can be defined in terms of possible worlds:

- Variables can be primitive and a **possible world** corresponds to a **total assignment** of a value to each variable.
- Worlds can be primitive and a variable is a function from possible worlds into the domain of the variable; given a possible world, the function returns the value of that variable in that possible world.

Example 4.5 If there are two variables, A with domain $\{0, 1, 2\}$ and B with domain $\{true, false\}$ there are six possible worlds, which you can name $w_0, \dots, w_5$. One possible arrangement of variables and possible worlds is

- $w_0: A = 0$ and $B = true$
- $w_1: A = 0$ and $B = false$
- $w_2: A = 1$ and $B = true$
- $w_3: A = 1$ and $B = false$
- $w_4: A = 2$ and $B = true$
- $w_5: A = 2$ and $B = false$

Example 4.6 The trading agent, in planning a trip for a group of tourists, may be required to schedule a given set of activities. There can be two variables for each activity: one for the date, for which the domain is the set of possible days for the activity, and one for the location, for which the domain is the set of possible towns where it may occur. A possible world corresponds to an assignment of a date and a town for each activity.

4.2.1 Constraints

In many domains, not all possible assignments of values to variables are permissible. A **hard constraint**, or simply constraint, specifies legal combinations of assignments of values to the variables.

A **scope** or **scheme** is a set of variables. A **tuple** on scope S is an assignment of a value to each variable in S . A **constraint** c on a scope S is a set of tuples on S . A constraint is said to **involve** each of the variables in its scope.

If S^t is a set of variables such that $S \subseteq S^t$, and t is a tuple on S^t , constraint c is said to **satisfy** t if t , restricted to S , is in c .

Constraints can be defined using the terminology of relational databases (page 635). The main difference between constraints and database relations is that a constraint specifies legal values, whereas a database relation specifies what happens to be true in some situation. Constraints are also often defined **intensionally**, in terms of predicates (Boolean functions), to recognize legal assignments rather than **extensionally** by representing each assignment explicitly in a table. Extensional definitions can be implemented either by representing the legal assignments or by representing the illegal assignments.

Example 4.7 Consider a constraint on the possible dates for three activities. Let A , B , and C be variables that represent the date of each activity. Suppose the domain of each variable is $\{1, 2, 3, 4\}$.
The constraint could be written as a table that gives the legal combinations:

A	B	C
2	2	4
1	1	4
1	2	3
1	2	4

which has scope $\{A, B, C\}$. Each row is a tuple that specifies a legal assignment of a value to each variable in the scope of the constraint. The first tuple is

$$\{A = 2, B = 2, C = 4\}.$$

This tuple, which assigns A the value of 2, B the value of 2, and C the value of 4, specifies one of the four legal assignments of the variables.

This constraint satisfies the tuple $A = 1, B = 2, C = 3, D = 3, E = 1$, because that tuple assigns legal values to the variables in the scope.

This constraint could also be described intensionally by using a predicate – a logical formula – to specify the legal assignments. The preceding constraint could be specified by

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C = 3),$$

where $\wedge$ means *and*, and $\neg$ means *not*. This formula says that A is on the same date or before B , and B is before 3, and B is before C , and it cannot be that A and B are on the same date and C is on day 3.

A **unary constraint** is a constraint on a single variable (e.g., $X = 4$). A **binary constraint** is a constraint over a pair of variables (e.g., $X = Y$). In general, a k -ary constraint has a scope of size k .

A possible world w **satisfies** a set of constraints if, for every constraint, the values assigned in w to the variables in the scope of the constraint satisfy the constraint. In this case, we say that the possible world is a **model** of the constraints. That is, a **model** is a possible world that satisfies all of the constraints.

Example 4.8 Suppose the delivery robot must carry out a number of delivery activities, a, b, c, d , and e . Suppose that each activity happens at any of times 1, 2, 3, or 4. Let A be the variable representing the time that activity a will occur, and similarly for the other activities. The variable domains, which represent possible times for each of the deliveries, are

$$\begin{aligned} \text{dom}(A) &= \{1, 2, 3, 4\}, \text{dom}(B) = \{1, 2, 3, 4\}, \text{dom}(C) = \{1, 2, 3, 4\}, \\ \text{dom}(D) &= \{1, 2, 3, 4\}, \text{dom}(E) = \{1, 2, 3, 4\}. \end{aligned}$$

Suppose the following constraints must be satisfied:

$$\{(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D), \\ (E < A), (E < B), (E < C), (E < D), (B \neq D)\}$$

The aim is to find a model, an assignment of a value to each variable, such that all the constraints are satisfied.

Example 4.9 Consider the constraints for the two representations of crossword puzzles of Example 4.4.

1. For the case in which the domains are words, the constraint is that the letters where a pair of words intersect must be the same.
2. For the representation in which the domains are the letters, the constraint is that contiguous sequences of letters must form legal words.

Example 4.10 In Example 4.6 (page 115), consider some typical constraints. It may be that certain activities have to be on different days or that other activities have to be in the same town on the same day. There may also be constraints that some activities must occur before other activities, or that there must be a certain number of days between two activities, or that there cannot be three activities on three consecutive days.

4.2.2 Constraint Satisfaction Problems

A **constraint satisfaction problem** (CSP) consists of

- a set of variables,
- a domain for each variable, and
- a set of constraints.

The aim is to choose a value for each variable so that the resulting possible world satisfies the constraints; we want a model of the constraints.

A finite CSP has a finite set of variables and a finite domain for each variable. Many of the methods considered in this chapter only work for finite CSPs, although some are designed for infinite, even continuous, domains.

The multidimensional aspect of these problems, where each variable can be seen as a separate dimension, makes them difficult to solve but also provides structure that can be exploited.

Given a CSP, there are a number of tasks that can be performed:

- Determine whether or not there is a model.
- Find a model.
- Find all of the models or enumerate the models.
- Count the number of models.
- Find the best model, given a measure of how good models are; see Section 4.10 (page 144).
- Determine whether some statement holds in all models.

This chapter mostly considers the problem of finding a model. Some of the methods can also determine if there is no solution. What may be more surprising is that some of the methods can find a model if one exists, but they cannot tell us that there is no model if none exists.

CSPs are very common, so it is worth trying to find relatively efficient ways to solve them. Determining whether there is a model for a CSP with finite domains is NP-hard (see box on page 170) and no known algorithms exist to solve such problems that do not use exponential time in the worst case. However, just because a problem is NP-hard does not mean that all instances are difficult to solve. Many instances have structure that can be exploited.

4.3 Generate-and-Test Algorithms

Any finite CSP can be solved by an exhaustive generate-and-test algorithm. The **assignment space**, D , is the set of assignments of values to all of the variables; it corresponds to the set of all possible worlds. Each element of D is a total assignment of a value to each variable. The algorithm returns those assignments that satisfy all of the constraints.

Thus, the **generate-and-test** algorithm is as follows: check each total assignment in turn; if an assignment is found that satisfies all of the constraints, return that assignment.

Example 4.11 In Example 4.8 the assignment space is

$$D = \{ \{A = 1, B = 1, C = 1, D = 1, E = 1\}, \\ \{A = 1, B = 1, C = 1, D = 1, E = 2\}, \dots, \\ \{A = 4, B = 4, C = 4, D = 4, E = 4\} \}.$$

In this case there are $|D| = 4^5 = 1,024$ different assignments to be tested. In the crossword example of Exercise 1 (page 152) there are $40^6 = 4,096,000,000$ possible assignments.

If each of the n variable domains has size d , then D has d^n elements. If there are e constraints, the total number of constraints tested is $O(ed^n)$. As n becomes large, this very quickly becomes intractable, and so we must find alternative solution methods.

4.4 Solving CSPs Using Search

Generate-and-test algorithms assign values to all variables before checking the constraints. Because individual constraints only involve a subset of the variables, some constraints can be tested before all of the variables have been assigned values. If a partial assignment is inconsistent with a constraint, any complete assignment that extends the partial assignment will also be inconsistent.

Example 4.12 In the delivery scheduling problem of Example 4.8 (page 117), the assignments $A = 1$ and $B = 1$ are inconsistent with the constraint $A = B$ regardless of the values of the other variables. If the variables A and B are assigned values first, this inconsistency can be discovered before any values are assigned to C , D , or E , thus saving a large amount of work.

An alternative to generate-and-test algorithms is to construct a search space from which the search strategies of the previous chapter can be used. The search problem can be defined as follows:

- The nodes are assignments of values to some subset of the variables.
- The neighbors of a node N are obtained by selecting a variable V that is not assigned in node N and by having a neighbor for each assignment of a value to V that does not violate any constraint.

Suppose that node N represents the assignment $X_1 = v_1, \dots, X_k = v_k$. To find the neighbors of N , select a variable Y that is not in the set $\{X_1, \dots, X_k\}$. For each value $y_i \in \text{dom}(Y)$, such that $X_1 = v_1, \dots, X_k = v_k, Y = y_i$ is consistent with the constraints, $X_1 = v_1, \dots, X_k = v_k, Y = y_i$ is a neighbor of N .

- The start node is the empty assignment that does not assign a value to any variables.
- A goal node is a node that assigns a value to every variable. Note that this only exists if the assignment is consistent with the constraints.

In this case, it is not the path that is of interest, but the goal nodes.

Example 4.13 Suppose you have a CSP with the variables A , B , and C , each with domain $\{1, 2, 3, 4\}$. Suppose the constraints are $A < B$ and $B < C$. A possible search tree is shown in Figure 4.1 (on the next page). In this figure, a node corresponds to all of the assignments from the root to that node. The potential nodes that are pruned because they violate constraints are labeled with **X**.

The leftmost **X** corresponds to the assignment $A = 1, B = 1$. This violates the $A < B$ constraint, and so it is pruned.

This CSP has four solutions. The leftmost one is $A = 1, B = 2, C = 3$. The size of the search tree, and thus the efficiency of the algorithm, depends on which variable is selected at each time. A static ordering, such as always splitting on A then B then C , is less efficient than the dynamic ordering used here. The set of answers is the same regardless of the variable ordering.

In the preceding example, there would be $4^3 = 64$ assignments tested in a generate-and-test algorithm. For the search method, there are 22 assignments generated.

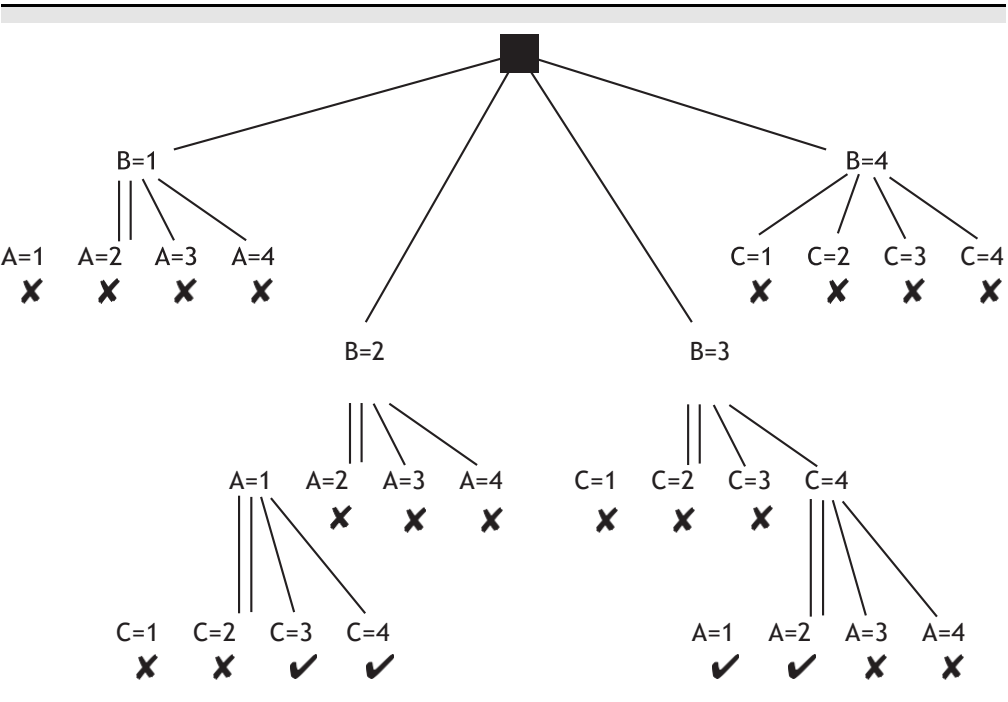


Figure 4.1: Search tree for the CSP of Example 4.13

Searching with a depth-first search, typically called **backtracking**, can be much more efficient than generate and test. Generate and test is equivalent to not checking constraints until reaching the leaves. Checking constraints higher in the tree can prune large subtrees that do not have to be searched.

4.5 Consistency Algorithms

Although depth-first search over the search space of assignments is usually a substantial improvement over generate and test, it still has various inefficiencies that can be overcome.

Example 4.14 In Example 4.13, the variables A and B are related by the constraint $A < B$. The assignment $A = 4$ is inconsistent with each of the possible assignments to B because $\text{dom}(B) = \{1, 2, 3, 4\}$. In the course of the backtrack search (see Figure 4.1), this fact is rediscovered for different assignments to B and C . This inefficiency can be avoided by the simple expedient of deleting 4 from $\text{dom}(A)$, once and for all. This idea is the basis for the consistency algorithms.

The consistency algorithms are best thought of as operating over the network of constraints formed by the CSP:

- There is a node for each variable. These nodes are drawn as ovals.
- There is a node for each constraint. These nodes are drawn as rectangles.
- Associated with each variable, X , is a set D_X of possible values. This set of

values is initially the domain of the variable.

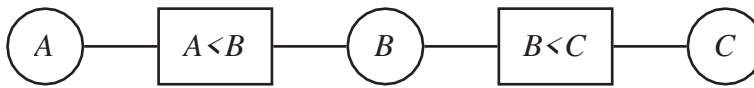


Figure 4.2: Constraint network for the CSP of Example 4.15

- For every constraint c , and for every variable X in the scope of c , there is an arc (X, c) .

Such a network is called a **constraint network**.

Example 4.15 Consider Example 4.13 (page 119). There are three variables A , B , and C , each with domain $\{1, 2, 3, 4\}$. The constraints are $A < B$ and $B < C$. In the constraint network, shown in Figure 4.2, there are four arcs:

$(A, A < B)$
 $(B, A < B)$
 $(B, B < C)$
 $(C, B < C)$

Example 4.16 The constraint $X \neq 4$ has one arc:

$(X, X \neq 4)$

The constraint $X + Y = Z$ has three arcs:

$(X, X + Y = Z)$
 $(Y, X + Y = Z)$
 $(Z, X + Y = Z)$

The simplest case is when the constraint has just one variable in its scope. In this case, the arc is **domain consistent** if every value of the variable satisfies the constraint.

Example 4.17 The constraint $B = 3$ has scope B . With this constraint, and with $D_B = \{1, 2, 3, 4\}$, the arc $B, B = 3$ is not domain consistent because $B = 3$ violates the constraint. If the value 3 were removed from the domain of B , then it would be domain consistent.

Suppose constraint c has scope $\{X, Y_1, \dots, Y_k\}$. Arc (X, c) is **arc consistent** if, for each value $x \in D_X$, there are values $y_1, \dots, y_k$ where $y_i \in D_{Y_i}$ such that $c(X = x, Y_1 = y_1, \dots, Y_k = y_k)$ is satisfied. A network is arc consistent if all its arcs are arc consistent.

Example 4.18 Consider the network of Example 4.15. None of the arcs are arc consistent. The first arc is not arc consistent because for $A = 4$ there is no corresponding value for B for which $A < B$. If 4 were removed from the domain of A , then it would be arc consistent. The second arc is not arc consistent because there is no corresponding value for A when $B = 1$.

```

1: procedure GAC( $V, dom, C$ )
2:   Inputs
3:      $V$ : a set of variables
4:      $dom$ : a function such that  $dom(X)$  is the domain of variable  $X$ 
5:      $C$ : set of constraints to be satisfied
6:   Output
7:     arc-consistent domains for each variable
8:   Local
9:      $D_X$  is a set of values for each variable  $X$ 
10:     $TDA$  is a set of arcs
11:    for each variable  $X$  do
12:       $D_X \leftarrow dom(X)$ 
13:       $TDA \leftarrow \{(X, c) \mid c \in C \text{ and } X \in scope(c)\}$ 
14:      while  $TDA \neq \{\}$  do
15:        select  $(X, c) \in TDA$ ;
16:         $TDA \leftarrow TDA \setminus \{(X, c)\}$ ;
17:         $ND_X \leftarrow \{x \mid x \in D_X \text{ and some } \{X = x, Y_1 = y_1, \dots, Y_k = y_k\} \in c$ 
           where  $y_i \in D_{Y_i}$  for all  $i\}$ 
18:        if  $ND_X \neq D_X$  then
19:           $TDA \leftarrow TDA \cup \{(Z, c^t) \mid X \in scope(c^t), c^t \text{ is not } c, Z \in scope(c^t) \setminus$ 
            $\{X\}\}$ 
20:           $D_X \leftarrow ND_X$ 
21:    return  $\{D_X \mid X \text{ is a variable}\}$ 

```

Figure 4.3: Generalized arc consistency algorithm

If an arc X, c is not arc consistent, there are some values of X for which there are no values for $Y_1, \dots, Y_k$ for which the constraint holds. In this case, all values of X in D_X for which there are no corresponding values for the other variables can be deleted from D_X to make the arc X, c consistent()

The **generalized arc consistency algorithm** is given in Figure 4.3. It makes the entire network arc consistent by considering a set of potentially inconsistent arcs, the *to-do* arcs, in the set TDA . TDA initially consists of all the arcs in the graph. Although the set is not empty, an arc X, c is removed from the set and considered. If the arc is not consistent, it is made consistent by pruning the domain of variable X . All of the previously consistent arcs that could, as a result of pruning X , have become inconsistent are placed back into the set TDA . These are the arcs Z, c^t , where c^t is a constraint different from c that involves X , and Z is a variable involved in c^t other than X .

Example 4.19 Consider the algorithm GAC operating on the network from Example 4.15. Initially, all of the arcs are in the TDA set. Here is one possible sequence of selections of arcs:

- Suppose the algorithm first selects the arc $A, A < B$. For $A = 4$, there is no value of B that satisfies the constraint. Thus, 4 is pruned from the

domain of A . Nothing is added to TDA because there is no other arc currently outside TDA .

- Suppose that $(B, A < B)$ is selected next. The value 1 can be pruned from the domain of B . Again no element is added to TDA .
- Suppose that $(B, B < C)$ is selected next. The value 4 can be removed from the domain of B . Because the domain of B has been reduced, the arc $(A, A < B)$ must be added back into the TDA set because the domain of A could potentially be reduced further now that the domain of B is smaller.
- If the arc $(A, A < B)$ is selected next, the value $A = 3$ can be pruned from the domain of A .
- The remaining arc on TDA is $(C, B < C)$. The values 1 and 2 can be removed from the domain of C . No arcs are added to TDA and TDA becomes empty.

The algorithm then terminates with $D_A = \{1, 2\}$, $D_B = \{2, 3\}$, $D_C = \{3, 4\}$. Although this has not fully solved the problem, it has greatly simplified it.

Example 4.20 Consider applying GAC to the scheduling problem of Example 4.8 (page 117). The network shown in Figure 4.4 (on the next page) has already been made domain consistent (the value 3 has been removed from the domain of B and 2 has been removed from the domain of C). Suppose arc $(D, C < D)$ is considered first. The arc is not arc consistent because $D = 1$ is not consistent with any value in D_C , so 1 is deleted from D_D . D_D becomes $\{2, 3, 4\}$ and arcs $(A, A = D)$, $(B, B \neq D)$ and $(E, E < D)$ could be added to TDA but they are on it already.

Suppose arc $(C, E < C)$ is considered next; then D_C is reduced to $\{3, 4\}$ and arc $(D, C < D)$ goes back into the TDA set to be reconsidered.

Suppose arc $(D, C < D)$ is next; then D_D is further reduced to the singleton $\{4\}$. Processing arc $(C, C < D)$ prunes D_C to 3. Making arc $(A, A \neq D)$ consistent reduces D_A to 4. Processing $(B, B = D)$ reduces D_B to 1, 2. Then arc $(B, E < B)$ reduces D_B to 2. Finally, arc $(E, E < B)$ reduces D_E to 1. All arcs remaining in the queue are consistent, and so the algorithm terminates with the

TDA set empty. The set of reduced variable domains is returned. In this case, the domains all have size 1 and there is a unique solution: $A = 4$, $B = 2$, $C = 3$, $D = 4$, $E = 1$.

Regardless of the order in which the arcs are considered, the algorithm will terminate with the same result, namely, an arc-consistent network and the same set of reduced domains. Three cases are possible, depending on the state of the network upon termination:

- In the first case, one domain is empty, indicating there is no solution to the CSP. Note that, as soon as any one domain becomes empty, all the domains of connected nodes will become empty before the algorithm terminates.
- In the second case, each domain has a singleton value, indicating that there is a unique solution to the CSP, as in Example 4.20.
- In the third case, every domain is non-empty and at least one has multiple values left in it. In this case, we do not know whether there is a solution or

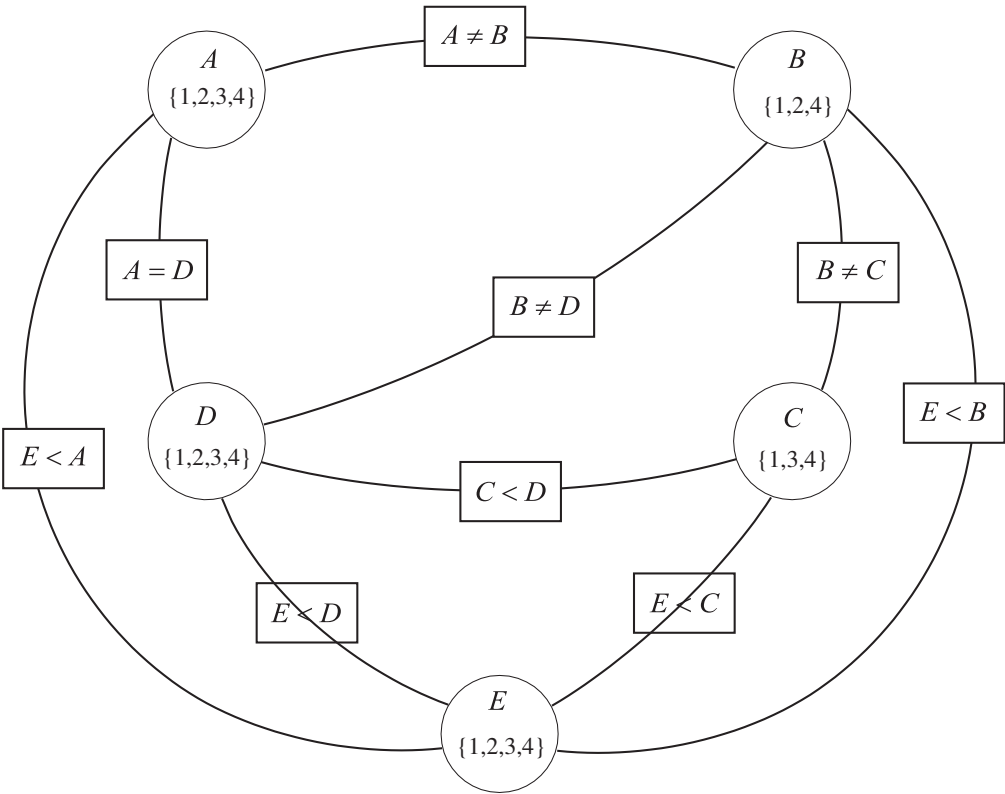


Figure 4.4: Domain-consistent constraint network. The variables are depicted as circles or ovals with their corresponding domain. The constraints are represented as rectangles. There is an arc between each variable and each constraint that involves that variable.

what the solutions look like. We require some other methods to solve the problem; some such methods are explored in the following sections.

The following example shows that it is possible for a network to be arc consistent even though there is no solution.

Example 4.21 Suppose there are three variables, A , B and C , each with the domain $\{1, 2, 3\}$. Consider the constraints $A = B$, $B = C$, and $A = C$. This is arc consistent: no domain can be pruned using any single constraint. However, there are no solutions. There is no assignment to the three variables that satisfies the constraints.

If each variable domain is of size d and there are e constraints to be tested, then the algorithm GAC does $O(ed^3)$ consistency checks. For some CSPs, for example, if the constraint graph is a tree, GAC alone solves the CSP and does it in time linear in the number of variables.

Various extensions to the arc-consistency technique are also possible. The domains need not be finite; they may be specified using descriptions, not just

lists of their values. It is also possible to prune the constraints if the constraints are represented extensionally: if a value has been pruned for a variable X , this value can be pruned from all constraints that involve X . Higher-order consistency techniques, such as **path consistency**, consider k -tuples of variables at a time, not just pairs of variables that are connected by a constraint. For example, by considering all three variables, you can recognize that there is no solution in Example 4.21. These higher-order methods are often less efficient for solving a problem than using arc consistency augmented with the methods described below.

4.6 Domain Splitting

Another method for simplifying the network is **domain splitting** or **case analysis**. The idea is to split a problem into a number of disjoint cases and solve each case separately. The set of all solutions to the initial problem is the union of the solutions to each case.

In the simplest case, suppose there is a binary variable X with domain $\{t, f\}$. All of the solutions either have $X = t$ or $X = f$. One way to find them is to set $X = t$, find all of the solutions with this assignment, and then assign $X = f$ and find all solutions with this assignment. Assigning a value to a variable gives a smaller *reduced* problem to solve.

If the domain of a variable has more than two elements, for example if the domain of A is $\{1, 2, 3, 4\}$, there are a number of ways to split it:

- Split the domain into a case for each value. For example, split A into the four cases $A = 1$, $A = 2$, $A = 3$, and $A = 4$.
- Always split the domain into two disjoint subsets. For example, split A into the two cases $A \in \{1, 2\}$ and the case $A \in \{3, 4\}$.

The first approach makes more progress with one split, but the second may allow for more pruning with fewer steps. For example, if the same values for B can be pruned whether A is 1 or 2, the second case allows this fact to be discovered once and not have to be rediscovered for each element of A . This saving depends on how the domains are split.

Recursively solving the cases using domain splitting, recognizing when there is no solution based on the assignments, is equivalent to the search algorithm of Section 4.4 (page 119). We can be more efficient by interleaving arc consistency with the search.

One effective way to solve a CSP is to use arc consistency to simplify the network before each step of domain splitting. That is, to solve a problem,

- simplify the problem using arc consistency; and,
- if the problem is not solved, select a variable whose domain has more than one element, split it, and recursively solve each case.

One thing to notice about this algorithm is that it does not require a restart of the arc consistency from scratch after domain splitting. If the variable X has its

domain split, *TDA* can start with just the arcs that are possibly no longer arc consistent as a result of the split; these are only the arcs of the form (Y, r) , where X appears in r and Y is a variable, other than X , that appears in constraint r .

Example 4.22 In Example 4.19 (page 122), arc consistency simplified the network, but did not solve the problem. After arc consistency had completed, there were multiple elements in the domains. Suppose B is split. There are two cases:

- $B = 2$. In this case $A = 2$ is pruned. Splitting on C produces two of the answers.
- $B = 3$. In this case $C = 3$ is pruned. Splitting on A produces the other two answers.

This search tree should be contrasted with the search tree of Figure 4.1 (page 120). The search space with arc consistency is much smaller, and it was not as sensitive to the selection of variable orderings. [Figure 4.1 (page 120) would be much bigger with different variable orderings.]

One other enhancement can make domain splitting much more efficient. If assigning values to the variables disconnects the graph, each disconnected component can be solved separately. The solution to the complete problem is the cross product of the solutions to each component. This can save much computation when one is counting the number of solutions or finding all solutions. For example, if one component has 100 solutions and the other component has 20 solutions, there are 2,000 solutions. This is a more efficient way to count than finding each of the 2,000 solutions separately.

4.6.1 Exploiting Propositional Structure

A fundamental idea in AI is to exploit structure in a domain. One form of structure for CSPs arises from the exploitation of aspects of restricted classes of variables and constraints. One such class is the class of **propositional satisfiability** problems. These problems are characterized by

- **Boolean variables:** a Boolean variable (page 113) is a variable with domain $\{true, false\}$. Given a Boolean variable *Happy*, the proposition *happy* means $Happy = true$, and *happy* means $Happy = false$.
- **clausal constraints:** a **clause** is an expression of the form $l_1 \vee l_2 \vee \dots \vee l_k$, where each l_i is a literal. A **literal** is an assignment of a value to a Boolean variable. A clause is satisfied, or true, in a possible world if and only if at least one of the literals that makes up the clause is true in that possible world.

For example, the clause *happy sad living* is a constraint among the variables *Happy*, *Sad*, and *Living*, which is true if *Happy* has value *true*, *Sad* has value *true*, or *Living* has value *false*.

A clause is a constraint on a set of Boolean variables that removes one of the assignments from consideration – the assignment that makes all literals false. Thus, the clause *happy sad living* specifies that the assignment *happy, sad, living* is not allowed.

When there are only two values, pruning a value from the domain is equivalent to assigning the opposite value. Thus, if X has domain $\{true, false\}$, pruning $true$ from the domain of X is the same as assigning X to have value $false$.

Arc consistency can be used to prune the set of values and the set of constraints. Assigning a value to a Boolean variable can simplify the set of constraints:

- If X is assigned $true$, all of the clauses with $X = true$ become redundant; they are automatically satisfied. These clauses can be removed. Similarly, assigning the X the value of $false$ can remove the clauses containing $X = false$.
- If X is assigned the value of $true$, any clause with $X = false$ can be simplified by removing $X = false$ from the clause. Similarly, if X is assigned the value of $false$, then $X = true$ can be removed from any clause it appears in. This step is called **unit resolution**.

Following some steps of pruning the clauses, clauses may exist that contain just one assignment, $Y = v$. In this case, the other value can be removed from the domain of Y . The aforementioned procedure can be repeated for the assignment of $Y = v$. If all of the assignments are removed from a clause, the constraints are unsatisfiable.

Example 4.23 Consider the clause $\bar{x} \vee y \vee \bar{z}$. When X is assigned to $true$, the clause can be simplified to $y \vee \bar{z}$. If Y is then assigned to $false$, the clause can be simplified to $\bar{z}$. Thus, $true$ can be pruned from the domain of Z .

If, instead, X is assigned to $false$, the preceding clause can be removed.

If a variable has the same value in all remaining clauses, and the algorithm must only find one model, it can assign that value to that variable. For example, if variable Y only appears as $Y = true$ (i.e., y is not in any clause), then Y can be assigned the value $true$. That assignment does not remove all of the models; it only simplifies the problem because the set of clauses remaining after setting $Y = true$ is a subset of the clauses that would remain if Y were assigned the value $false$. A variable that only has one value in all of the clauses is called a **pure symbol**.

It turns out that pruning the domains and constraints, domain splitting, assigning pure symbols, and efficiently managing the constraints (i.e., determining when all but one of the disjuncts are false) turns out to be a very efficient algorithm. It is called the **DPLL** algorithm, after its authors.

4.7 Variable Elimination

Arc consistency simplifies the network by removing values of variables. A complementary method is **variable elimination (VE)**, which simplifies the network by removing variables.

The idea of VE is to remove the variables one by one. When removing a variable X , VE constructs a new constraint on some of the remaining variables

reflecting the effects of X on all of the other variables. This new constraint replaces all of the constraints that involve X , forming a reduced network that does not involve X . The new constraint is constructed so that any solution to the reduced CSP can be extended to a solution of the larger CSP that contains X . In addition to creating the new constraint, VE provides a way to construct a solution to the CSP that contains X from a solution to the reduced CSP.

The following algorithm is described using the relational algebra calculations of join and project (page 635).

When eliminating X , the influence of X on the remaining variables is through the constraint relations that involve X . First, the algorithm collects all of the constraints that involve X . Let the join of all of these relations be the relation $r_X(X, Y)$, where Y is the set of other variables in the scope of r_X . It then projects r_X onto Y ; this relation can replace all of the relations that involve X . It now has a reduced CSP that involves one less variable, which it can solve recursively. Once it has a solution for the reduced CSP, it can extend that solution to a solution for the original CSP by joining the solution with r_X .

When only one variable is left, it returns those domain elements that are consistent with the constraints on this variable.

Example 4.24 Consider a CSP that contains the variables A, B , and C , each with domain $\{1, 2, 3, 4\}$. Suppose the constraints that involve B are $A < B$ and $B < C$. There may be many other variables, but if B does not have any constraints in common with these variables, eliminating B will not impose any new constraints on these other variables. To remove B , first join on the relations that involve B :

A	B		B	C		A	B	C
1	2		1	2	=	1	2	3
1	3		1	3		1	2	4
1	4	∞	1	4		1	3	4
2	3		2	3		1	4	4
2	4		2	4		2	3	4
3	4		3	4				

To get the relation on A and C induced by B , project this join onto A and C , which gives

A	C
1	3
1	4
2	4

This relation on A and C contains all of the information about the constraints on B that affect the solutions of the rest of the network.

The original constraints on B are replaced with the new constraint on A and C . It then solves the rest of the network, which is now simpler because it does not involve the variable B . It must remember the joined relation to construct a solution that involves B from a solution to the reduced network.

```

1: procedure VE CSP( $V, C$ )
2:   Inputs
3:      $V$ : a set of variables
4:      $C$ : a set of constraints on  $V$ 
5:   Output
6:     a relation containing all of the consistent variable assignments
7:   if  $V$  contains just one element then
8:     return the join of all the relations in  $C$ 
9:   else
10:    select variable  $X$  to eliminate
11:     $V^t := V \setminus \{X\}$ 
12:     $C_X := \{T \text{ in } C : T \text{ involves } X\}$ 
13:    let  $R$  be the join of all of the constraints in  $C_X$ 
14:    let  $R^t$  be  $R$  projected onto the variables other than  $X$ 
15:     $S := \text{VE\_CSP}(V^t, (C \setminus C_X) \cup \{R^t\})$ 
16:    return  $R \bowtie S$ 

```

Figure 4.5: Variable elimination for CSPs

Figure 4.5 gives a recursive algorithm for variable elimination, *VE CSP*, to find all solutions for a CSP.

The base case of the recursion occurs when only one variable is left. In this case (line 8), a solution exists if and only if there are rows in the join of the final relations. Note that these relations are all relations on a single variable, and so they are the sets of legal values for this variable. The join of these relations is the intersection of these sets.

In the non-base case, a variable X is selected for elimination (line 10). Which variable is selected does not affect the correctness of the algorithm, but it may affect the efficiency. To eliminate variable X , the algorithm propagates the effect of X onto those variables that X is directly related to. This is achieved by joining all of the relations in which X is involved (line 13) and then projecting X out of the resulting relation (line 14). Thus, a simplified problem (with one less variable) has been created that can be solved recursively. To get the possible values for X , the algorithm joins the solution of the simplified problem with the relation R that defines the effect of X .

If you only wanted to find one solution, instead of returning $R \bowtie S$, the algorithm can return one element of the join. No matter which element it returns, that element is guaranteed to be part of a solution. If any value of R in this algorithm contains no tuples, there are no solutions.

The efficiency of the VE algorithm depends on the order in which variables are selected. The intermediate structure – which variables the intermediate relations are over – depends not on the actual content of relations but only on the graph structure of the constraint network. The efficiency of this algorithm can be determined by considering the graphical structure. In general, VE is efficient

when the constraint network is sparse. The number of variables in the largest relation returned for a particular variable ordering is called the **treewidth** of the graph for that variable ordering. The treewidth of a graph is the minimum treewidth for any ordering. The complexity of VE is exponential in treewidth and linear in the number of variables. This can be compared to searching [see Section 4.4 (page 119)], which is exponential in the number of variables.

Finding an elimination ordering that results in the smallest treewidth is NP-hard. However, some good heuristics exist. The two most common are

- **min-factor**: at each stage, select the variable that results in the smallest relation.
- **minimum deficiency** or **minimum fill**: at each stage, select the variable that adds the smallest number of arcs to the remaining constraint network. The deficiency of a variable X is the number of pairs of variables that are in a relationship with X that are not in a relationship with each other. The intuition is that it is okay to remove a variable that results in a large relation as long as it does not make the network more complicated.

The minimum deficiency has usually been found empirically to give a smaller treewidth than min-factor, but it is more difficult to compute.

VE can also be combined with arc consistency; whenever VE removes a variable, arc consistency can be used to further simplify the problem. This approach can result in smaller intermediate tables.

4.8 Local Search

The previous sections have considered algorithms that systematically search the space. If the space is finite, they will either find a solution or report that no solution exists. Unfortunately, many search spaces are too big for systematic search and are possibly even infinite. In any reasonable time, systematic search will have failed to consider enough of the search space to give any meaningful results. This section and the next consider methods intended to work in these very large spaces. The methods do not systematically search the whole search space but they are designed to find solutions quickly on average. They do not guarantee that a solution will be found even if one exists, and so they are not able to prove that no solution exists. They are often the method of choice for applications where solutions are known to exist or are very likely to exist.

Local search methods start with a complete assignment of a value to each variable and try to iteratively improve this assignment by improving steps, by taking random steps, or by restarting with another complete assignment. A wide variety of local search techniques has been proposed. Understanding when these techniques work for different problems forms the focus of a number of research communities, including those from both operations research and AI.

The generic local search algorithm for CSPs is given in Figure 4.6. A specifies an assignment of a value to each variable. The first *for each* loop assigns a

```

1: procedure Local-Search( $V, dom, C$ )
2:   Inputs
3:      $V$ : a set of variables
4:      $dom$ : a function such that  $dom(X)$  is the domain of variable  $X$ 
5:      $C$ : set of constraints to be satisfied
6:   Output
7:     complete assignment that satisfies the constraints
8:   Local
9:      $A[V]$  an array of values indexed by  $V$ 
10:  repeat
11:    for each variable  $X$  do
12:       $A[X] \leftarrow$  a random value in  $dom(X)$ ;
13:    while stopping criterion not met &  $A$  is not a satisfying assignment
14:  do
15:    Select a variable  $Y$  and a value  $V \in dom(Y)$ 
16:    Set  $A[Y] \leftarrow V$ 
17:    if  $A$  is a satisfying assignment then
18:      return  $A$ 
19:  until termination

```

Figure 4.6: Local search for finding a solution to a CSP

random value to each variable. The first time it is executed is called a **random initialization**. Each iteration of the outer loop is called a **try**. A common way to implement a new try is to do a **random restart**. An alternative to random initialization is to use a construction heuristic that guesses a solution, which is then iteratively improved.

The while loop does a **local search**, or a **walk**, through the assignment space. It maintains a current assignment S , considers a set of **neighbors** of the current assignment, and selects one to be the next current assignment. In Figure 4.6, the neighbors of a total assignment are those assignments that differ in the assignment of a single variable. Alternate sets of neighbors can also be used and will result in different search algorithms.

This walk through assignments continues until either a satisfying assignment is found and returned or some stopping criterion is satisfied. The stopping criterion is used to decide when to stop the current local search and do a random restart, starting again with a new assignment. A stopping criterion could be as simple as stopping after a certain number of steps.

This algorithm is not guaranteed to halt. In particular, it goes on forever if there is no solution, and it is possible to get trapped in some region of the search space. An algorithm is **complete** if it finds an answer whenever there is one. This algorithm is incomplete.

One instance of this algorithm is **random sampling**. In this algorithm, the stopping criterion is always true so that the while loop is never executed. Random sampling keeps picking random assignments until it finds one that satisfies the constraints, and otherwise it does not halt. Random sampling is complete in the sense that, given enough time, it guarantees that a solution will be found if one exists, but there is no upper bound on the time it may take. It is very slow. The efficiency depends only on the product of the domain sizes and how many solutions exist.

Another instance is a **random walk**. In this algorithm the while loop is only exited when it has found a satisfying assignment (i.e., the stopping criterion is always false and there are no random restarts). In the while loop it selects a variable and a value at random. Random walk is also complete in the same sense as random sampling. Each step takes less time than resampling all variables, but it can take more steps than random sampling, depending on how the solutions are distributed. Variants of this algorithm are applicable when the domain sizes of the variables differ; a random walk algorithm can either select a variable at random and then a value at random, or select a variable–value pair at random. The latter is more likely to select a variable when it has a larger domain.

4.8.1 Iterative BestImprovement

In **iterative best improvement**, the neighbor of the current selected node is one that optimizes some **evaluation function**. In **greedy descent**, a neighbor is chosen to minimize an evaluation function. This is also called **hill climbing** or **greedy ascent** when the aim is to maximize. We only consider minimization; if you want to maximize a quantity, you can minimize its negation.

Iterative best improvement requires a way to evaluate each total assignment. For constraint satisfaction problems, a common evaluation function is the number of constraints that are violated by the total assignment that is to be minimized. A violated constraint is called a **conflict**. With the evaluation function being the number of conflicts, a solution is a total assignment with an evaluation of zero. Sometimes this evaluation function is refined by weighting some constraints more than others.

A **local optimum** is an assignment such that no neighbor improves the evaluation function. This is also called a local minimum in greedy descent, or a local maximum in greedy ascent.

Example 4.25 Consider the delivery scheduling in Example 4.8 (page 117). Suppose gradient descent starts with the assignment $A = 2, B = 2, C = 3, D = 2, E = 1$. This assignment has an evaluation of 3, because it does not satisfy $A = B, B \neq D$, and $C < D$. Its neighbor with the minimal evaluation has $B = 4$ with an evaluation of 1 because only $C < D$ is unsatisfied. This is now a local minimum. A random walk can then change D to 4, which has an

evaluation of 2. It can change A to 4, with an evaluation of 2, and then change B to 2 with an evaluation of zero, and a solution is found.

The following gives a trace of the assignments through the walk:

A	B	C	D	E	evaluation
2	2	3	2	1	3
2	4	3	2	1	1
2	4	3	4	1	2
4	4	3	4	1	2
4	2	3	4	1	0

Different initializations, or different choices when multiple assignments have the same evaluation, give different results.

If the variables have small finite domains, a local search algorithm can consider all other values of the variable when considering a neighbor. If the domains are large, the cost of considering all other values may be too high. An alternative is to consider only a few other values, often the close values, for one of the variables. Sometimes quite sophisticated methods are used to select an alternative value.

Local search typically considers the best neighboring assignment even if it is equal to or even worse than the current assignment. It is often better to make a quick choice than to spend a lot of time making the best choice. There are many possible variants of which neighbor to choose:

- Select the value and variable together. Out of all of the different assignments to any of the variables, select one of them that minimizes the evaluation function. If more than one has the minimum value; pick one of them at random.
- Select a variable, then select its value. To select a variable, there are a number of possibilities:
 - Maintain how many violated constraints each variable is involved in, and pick one of the variables involved in the most violated constraints.
 - Select randomly a variable involved in any violated constraint.
 - Select a variable at random.

Once the variable has been selected, it can either select one of the values that has the best evaluation or just select a value at random.

- Select a variable and/or value at random and accept the change if it improves the evaluation.

Optimizing over all variable–value pairs makes bigger improvements than the other methods but it is more computationally expensive. Although some theoretical results exist, deciding which method works better in practice is an empirical question.

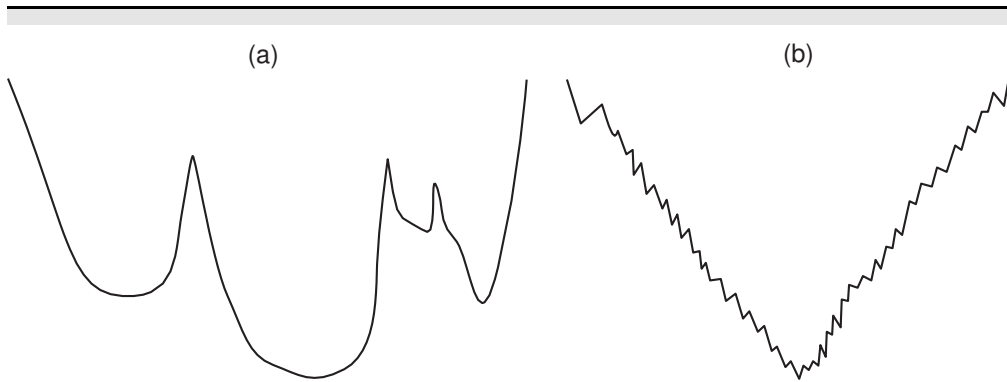


Figure 4.7: Two search spaces; find the minimum

4.8.2 Randomized Algorithms

Iterative best improvement randomly picks one of the best neighbors of the current assignment. Randomness can also be used to escape local minima that are not global minima in two main ways:

- random restart, in which values for all variables are chosen at random. This lets the search start from a completely different part of the search space.
- random walk, in which some random steps are taken interleaved with the optimizing steps. With greedy descent, this process allows for upward steps that may enable random walk to escape a local minimum that is not a global minimum.

A random walk is a local random move, whereas a random restart is a global random move. For problems involving a large number of variables, a random restart can be quite expensive.

A mix of greedy descent with random moves is an instance of a class of algorithms known as **stochastic local search**.

Unfortunately, it is very difficult to visualize the search space to understand what algorithms you expect to work because there are often thousands of dimensions, each with a discrete set of values. Some of the intuitions can be gleaned from lower-dimensional problems. Consider the two one-dimensional search spaces in Figure 4.7, where we try to find the minimum value. Suppose that a neighbor is obtained by a small step, either left or right of the current position. To find the global minimum in the search space (a), we would expect the greedy descent with random restart to find the optimal value quickly. Once a random choice has found a point in the deepest valley, greedy descent quickly leads to the global minimum. One would not expect a random walk to work well in this example, because many random steps are required to exit one of the local, but not global, minima. However, for search space (b), a random restart quickly gets stuck on one of the jagged peaks and does not work very well. However, a random walk combined with greedy descent enables it to escape

these local minima. One or two random steps may be enough to escape a local minimum. Thus, you would expect that a random walk would work better in this search space.

As presented, the local search has no memory. It does not remember anything about the search as it proceeds. A simple way to use memory to improve a local search is a **tabu list**, which is a list of the most recently changed variable-value assignments. The idea is, when selecting a new assignment, not to select a variable-value pair that is on the tabu list. This prevents cycling among a few assignments. A tabu list is typically of a small fixed size; the size of the tabu list is one of the parameters that can be optimized. A tabu list of size 1 is equivalent to not allowing the same assignment to be immediately revisited.

Algorithms can differ in how much work they require to guarantee the best improvement step. At one extreme, an algorithm can guarantee to select a new assignment that gives the best improvement out of all of the neighbors. At the other extreme, an algorithm can select a new assignment at random and reject the assignment if it makes the situation worse. The rest of this section gives some typical algorithms that differ in how much computational effort they put in to ensure that they make the best improvement. Which of these methods works best is, typically, an empirical question.

Most Improving Step

The first method is to always select the variable-value pair that makes the best improvement. The naive way of doing this is to linearly scan the variables; for each value of each variable, determine how many fewer constraints would be violated with this assignment compared to the current assignment to all variables, then select one of the variable-value pairs that results in the best improvement, even if that improvement is negative. The complexity of one step is $O(n dr)$, where n is the number of variables, d is the average domain size, and r is the number of neighbors of an assignment.

A more sophisticated alternative is to have a priority queue of variable-value pairs. For any variable X , and value v in the domain of X such that X is not assigned to v in the current assignment, the pair X, v would be in the priority queue; the weight of the pair is the evaluation of the current total assignment minus the evaluation of the total assignment if X , instead, had value v . That is, it maintains the change in the evaluation for each alternate value. At each stage, the algorithm selects a value with minimum weight, which is the neighbor that gives the biggest improvement.

Once a variable X has been given a new value, the weights of all variables that participate in a constraint that has been made satisfied or unsatisfied by the new assignment to X must have their weights changed and must be reinserted into the priority queue.

The complexity of one step of this algorithm is $O(\log nd + rd \log nd)$, where n is the number of variables, d is the average domain size, and r is the number of neighbors of an assignment.

This algorithm spends much time maintaining the data structures to ensure that the most improving step is taken at each time.

Two-Stage Choice

An alternative is to split the choice of a variable–value pair into first choosing a variable to change and then choosing a value.

This algorithm maintains a priority queue of variables, where the weight of a variable is the number of conflicts in which it participates. At each time, the algorithm selects a variable with maximum weight. Once a variable has been chosen, it can be assigned a value that minimizes the number of conflicts. For each conflict that has its value changed as a result of this new assignment, the other variables participating in the conflict must have their weight changed.

The complexity of one step of this algorithm is $O(\log n + r \log n)$. Compared to selecting the best variable–value pair, this does less work for each step and so more steps are achievable for any fixed time period. However, the steps tend to give less improvement, and so a trade-off exists between the number of steps and the complexity per step that must be evaluated empirically.

Any Conflict

An even simpler alternative, instead of choosing the best step, is to select any variable participating in a conflict and change its value. At each step, one of the variables involved in a violated constraint is selected at random. The algorithm assigns to that variable one of the values that minimizes the number of violated constraints.

To implement this alternative, we require a data structure to represent the set C of variables involved in a conflict. This data structure should be designed for quick selection of a random member of C . When a variable X has its value changed, each constraint in which X is involved is checked. For each such constraint that has become violated, all of the variables involved in the constraint are added to C . For each such constraint that is no longer violated, those variables involved in that constraint, but not involved in another violated constraint, are removed from C . One way to check whether a variable is involved in another conflict is to maintain a count of the number of conflicts for each variable, which is incremented when a constraint becomes violated and decremented when a constraint is no longer violated.

Each of the preceding algorithms can be combined with random steps, random restarts, and a tabu mechanism.

Simulated Annealing

The last method maintains no data structure of conflicts; instead it picks a neighbor at random and either rejects or accepts the new assignment.

Annealing is a process in metallurgy where metals are slowly cooled to make them reach a state of low energy where they are very strong. Simulated

Temperature	Probability of acceptance		
	1-worse	2-worse	3-worse
10	0.9	0.82	0.74
1	0.37	0.14	0.05
0.25	0.018	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

Figure 4.8: Probability of simulated annealing accepting worsening steps

annealing is an analogous method for optimization. It is typically described in terms of thermodynamics. The random movement corresponds to high temperature; at low temperature, there is little randomness. Simulated annealing is a process where the temperature is reduced slowly, starting from a random search at high temperature eventually becoming pure greedy descent as it approaches zero temperature. The randomness should tend to jump out of local minima and find regions that have a low heuristic value; greedy descent will lead to local minima. At high temperatures, worsening steps are more likely than at lower temperatures.

Simulated annealing maintains a current assignment of values to variables. At each step, it picks a variable at random, then picks a value at random. If assigning that value to the variable is an improvement or does not increase the number of conflicts, the algorithm accepts the assignment and there is a new current assignment. Otherwise, it accepts the assignment with some probability, depending on the temperature and how much worse it is than the current assignment. If the change is not accepted, the current assignment is unchanged. To control how many worsening steps are accepted, there is a positive real-valued temperature T . Suppose A is the current assignment of a value to each variable. Suppose that $h(A)$ is the evaluation of assignment A to be minimized. For solving constraints, h is typically the number of conflicts. Simulated annealing selects a neighbor at random, which gives a new assignment A^t . If $h(A^t) \leq h(A)$, it accepts the assignment and A^t becomes the new assignment. Otherwise, the assignment is only accepted randomly with probability

$$e^{(h(A)-h(A^t))/T}.$$

Thus, if $h(A^t)$ is close to $h(A)$, the assignment is more likely to be accepted. If the temperature is high, the exponent will be close to zero, and so the probability will be close to 1. As the temperature approaches zero, the exponent approaches ∞ , and the probability approaches zero.

Figure 4.8 shows the probability of accepting worsening steps at different temperatures. In this figure, k -worse means that $h(A^t) - h(A) = k$. For example, if the temperature is 10 (i.e., $T = 10$), a change that is one worse (i.e., if $h(A^t) = h(A) + 1$) will be accepted with probability $e^{-0.1} \approx 0.9$; a change that is two worse will be accepted with probability $e^{-0.2} \approx 0.82$. If the temperature T is 1, accepting a change that is one worse will happen with probability

$e^{-1} \approx 0.37$. If the temperature is 0.1, a change that is one worse will be accepted with probability $e^{-10} \approx 0.00005$. At this temperature, it is essentially only performing steps that improve the value or leave it unchanged.

If the temperature is high, as in the $T = 10$ case, the algorithm tends to accept steps that only worsen a small amount; it does not tend to accept very large worsening steps. There is a slight preference for improving steps. As the temperature is reduced (e.g., when $T = 1$), worsening steps, although still possible, become much less likely. When the temperature is low (e.g., 0.1), it is very rare that it chooses a worsening step.

Simulated annealing requires an **annealing schedule**, which specifies how the temperature is reduced as the search progresses. Geometric cooling is one of the most widely used schedules. An example of a geometric cooling schedule is to start with a temperature of 10 and multiply by 0.97 after each step; this will have a temperature of 0.48 after 100 steps. Finding a good annealing schedule is an art.

4.8.3 Evaluating Randomized Algorithms

It is difficult to compare randomized algorithms when they give a different result and a different run time each time they are run, even for the same problem. It is especially difficult when the algorithms sometimes do not find an answer; they either run forever or must be stopped at an arbitrary point.

Unfortunately, summary statistics, such as the mean or median run time, are not very useful. For example, if you were to compare algorithms on the mean run time, you must consider how to average in unsuccessful runs (where no solution was found). If you were to ignore these unsuccessful runs in computing the average, then an algorithm that picks a random assignment and then stops would be the best algorithm; it does not succeed very often, but when it does, it is very fast. If you were to treat the non-terminating runs as having infinite time, then all algorithms that do not find a solution will have infinite averages. If you use the stopping time as the time for the non-terminating runs, the average is more of a function of the stopping time than of the algorithm itself, although this does allow for a crude trade-off between finding some solutions fast versus finding more solutions.

If you were to compare algorithms using the median run time, you would prefer an algorithm that solves the problem 51% of the time but very slowly over one that solves the problem 49% of the time but very quickly, even though the latter is more useful. The problem is that the median (the 50th percentile) is just an arbitrary value; you could just as well consider the 47th percentile or the 87th percentile.

One way to visualize the run time of an algorithm for a particular problem is to use a **run-time distribution**, which shows the variability of the run time of a randomized algorithm on a single problem instance. On the x -axis is either the number of steps or the run time. The y -axis shows, for each value of x , the number of runs or proportion of the runs solved within that run time or number of steps. Thus, it provides a cumulative distribution of how often the

problem was solved within some number of steps or run time. For example, you can find the run time of the 30th percentile of the runs by finding the x -value that maps to 30% on the y -scale. The run-time distribution can be plotted (or approximated) by running the algorithm for a large number of times (say, 100 times for a rough approximation or 1,000 times for a reasonably accurate plot) and then by sorting the runs by run time.

Example 4.26 Four empirically generated run-time distributions for a single problem are shown in Figure 4.9 (on the next page). On the x -axis is the number of steps, using a logarithmic scale. On the y -axis is the number of instances that were successfully solved out of 1,000 runs. This shows four run-time distributions on the same problem instance. Algorithms 1 and 2 solved the problem 40% of the time in 10 or fewer steps. Algorithm 3 solved the problem in about 50% of the runs in 10 or fewer steps. Algorithm 4 found a solution in 10 or fewer steps in about 12% of the runs. Algorithms 1 and 2 found a solution in about 58% of the runs. Algorithm 3 could find a solution about 80% of the time. Algorithm 4 always found a solution. This only compares the number of steps; the time taken would be a better evaluation but is more difficult to measure for small problems and depends on the details of the implementation.

One algorithm strictly dominates another for this problem if its run-time distribution is to the left (and above) the run-time distribution of the second algorithm. Often two algorithms are incomparable under this measure. Which algorithm is better depends on how much time you have or how important it is to actually find a solution.

A run-time distribution allows us to predict how the algorithm will work with random restart after a certain number of steps. Intuitively, a random restart will repeat the lower left corner of the run-time distribution, suitably scaled down, at the stage where the restart occurs. A random restart after a certain number of greedy descent steps will make any algorithm that sometimes finds a solution into an algorithm that always finds a solution, given that one exists, if it is run for long enough.

Example 4.27 In the run-time distributions of Figure 4.9 (on the next page), Algorithm 3 dominates algorithms 1 and 2. Algorithms 1 and 2 are actually different sets of runs of the same algorithm with the same settings. This shows the errors that are typical of multiple runs of the same stochastic algorithm on the same problem instance. Algorithm 3 is better than Algorithm 4 up to 60 steps, after which Algorithm 4 is better.

By looking at the graph, you can see that Algorithm 3 can often solve the problem in its first four or five steps, after which it is not as effective. This may lead you to try to suggest using Algorithm 3 with a random restart after five steps and this does, indeed, dominate all of the algorithms for this problem instance, at least in the number of steps (counting a restart as a single step). However, because the random restart is an expensive operation, this algorithm may not be the most efficient. This also does not necessarily predict how well the algorithm will work in other problem instances.

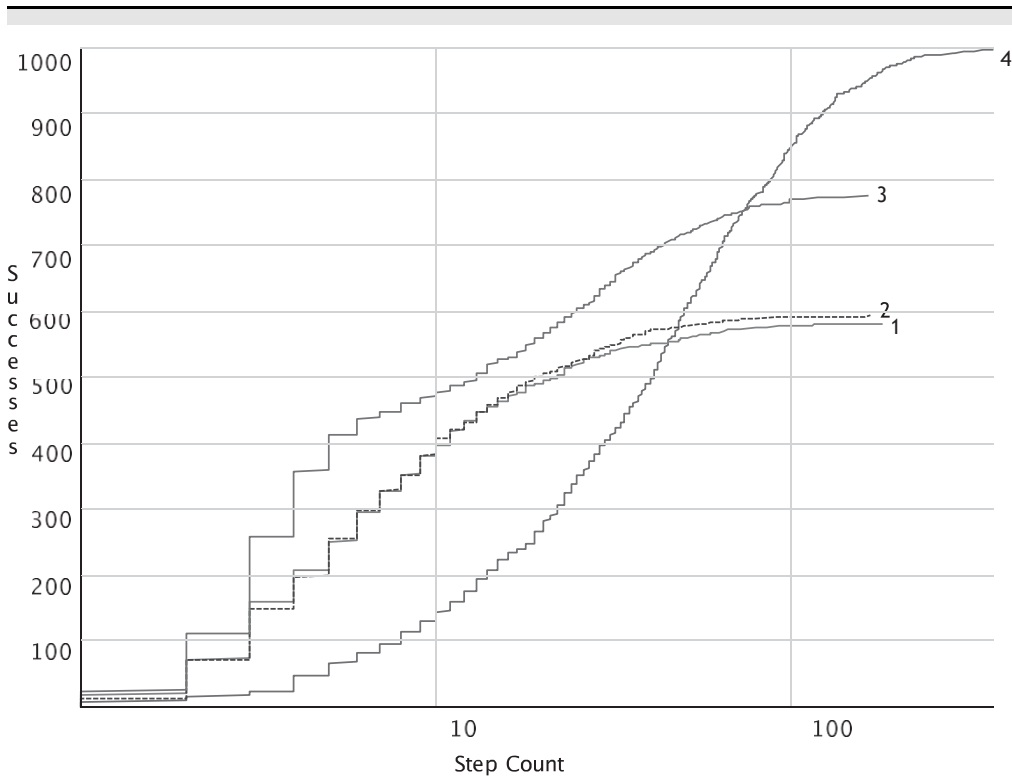


Figure 4.9: Run-time distributions. These are empirical run-time distributions of 1,000 runs, with each run having a limit of 1,000 steps. On the x -axis is the number of steps (using a logarithmic scale), and on the y -axis is the number of successes out of 1,000 runs. This is for the sample CSP, “Scheduling Problem 1,” of [AIspace.org](https://aispace.org). Distributions 1 and 2 are two separate runs for the two-stage greedy descent. Distribution 3 is for the one-stage greedy descent. Distribution 4 is a greedy descent with random walk, where first a random node that participates in a conflict (a red node in [AIspace.org](https://aispace.org)) is chosen, then the best value for this variable is chosen with a 50% chance and a random value is chosen otherwise.

4.8.4 Exploiting Propositional Structure in Local Search

Stochastic local search is simpler for CSPs that are in the form of propositional satisfiability problems (page 126), with Boolean variables and constraints that are clauses. Each local search step can be made more efficient for three main reasons:

- Because only one alternative value exists for each assignment to a variable, the algorithm does not have to search through the alternative values.
- Changing any value in an unsatisfied clause makes the clause satisfied. As a result, it is easy to satisfy a clause, but this may make other clauses unsatisfied.

- If a variable is changed to be true, only those clauses where it appears negatively can be made unsatisfied and similarly for variables becoming false. This enables fast indexing of clauses.

It is possible to convert any finite CSP into a propositional satisfiable problem. A variable Y with domain $v_1, \dots, v_k$ can be converted into k Boolean variables $Y_1, \dots, Y_k$ where Y_i is true when Y has value v_i and is false otherwise. Each Y_i is called an **indicator variable**. There is a clause for each false tuple in each constraint, which specifies which assignments to the Y_i are not allowed by the constraint. There are also constraints that specify that Y_i and Y_j cannot both be true when $i = j$. There is also a constraint that one of the variables Y_i must have value *true*. Converting a CSP into a propositional satisfiability problem has three potential advantages:

- Each local step can be simpler and therefore implemented more efficiently.
- The search space is expanded. In particular, before a solution has been found, more than one Y_i could be true (which corresponds to Y having multiple values) or all of the Y_i variables could be false (which corresponds to Y having no values). This can mean that some assignments that were local minima in the original problem may not be local minima in the new representation.
- Satisfiability has been studied much more extensively than most other types of CSPs and more efficient solvers are currently known because more of the space of potential algorithms has been explored by researchers.

Whether the conversion makes search performance better for a particular example is, again, an empirical question.

4.9 Population-Based Methods

The preceding local search algorithms maintain a single current assignment. This section considers algorithms that maintain multiple assignments. The first method, beam search, maintains the best k assignments. The next algorithm, stochastic beam search, selects which assignments to propagate stochastically. In genetic algorithms, which are inspired by biological evolution, the k assignments forming a population interact in various ways to produce the new population. In these algorithms, a total assignment of a value to each variable is called an **individual** and the set of current individuals is a **population**.

Beam search is a method similar to iterative best improvement, but it maintains up to k assignments instead of just one. It reports success when it finds a satisfying assignment. At each stage of the algorithm, it selects the k best neighbors of the current individuals (or all of them if there are less than k) and picks randomly in the case of ties. It repeats with this new set of k total assignments.

Beam search considers multiple assignments at the same time. Beam search is useful for memory-bounded cases, where k can be selected depending on the memory available.

Stochastic beam search is an alternative to beam search, which, instead of choosing the best k individuals, selects k of the individuals at random; the individuals with a better evaluation are more likely to be chosen. This is done by making the probability of being chosen a function of the evaluation function. A standard way to do this is to use a **Gibbs distribution** or **Boltzmann distribution** and to select an assignment A with probability proportional to

$$e^{-h(A)/T},$$

where $h(A)$ is the evaluation function and T is a temperature.

Stochastic beam search tends to allow more diversity in the k individuals than does plain beam search. In terms of evolution in biology, the evaluation function reflects the fitness of the individual; the fitter the individual, the more likely it is to pass that part of its variable assignment that is good onto the next generation. Stochastic beam search is like asexual reproduction; each individual gives slightly mutated children and then stochastic beam search proceeds with survival of the fittest. Note that under stochastic beam search it is possible for an individual to be selected multiple times at random.

Genetic algorithms further pursue the evolution analogy. Genetic algorithms are like stochastic beam searches, but each new element of the population is a combination of a pair of individuals – its parents. In particular, genetic algorithms select pairs of individuals and then create new offspring by taking some of the values for the offspring's variables from one of the parents and the rest from the other parent, loosely analogous to how DNA is spliced in sexual reproduction.

The new operation that occurs in genetic algorithms is called **crossover**. Uniform crossover selects two individuals (the parents) and creates two new individuals, called the **children**. The value for each variable in a child comes from one of the parents. A common method is **one-point crossover**, which assumes a total ordering of the variables. An index i is selected at random. One of the children is constructed by selecting the values for the variables before i from one of the parents, and the values for variables after i from the other parent. The other child gets the other values. The effectiveness of the crossover depends on the total ordering of the variables. The ordering of the variables is part of the design of the genetic algorithm.

Assume that you have a population of k individuals (where k is even). A basic genetic algorithm proceeds by maintaining k individuals as a generation and then using these individuals to generate a new generation via the following steps:

- Randomly select pairs of individuals where the fitter individuals are more likely to be chosen. How much more likely a fit individual is to be chosen than a less fit individual depends on the difference in fitness levels and a temperature parameter.

```

1: procedure GeneticAlgorithm( $V, dom, C, S, k$ )
2:   Inputs
3:      $V$ : a set of variables
4:      $dom$ : a function;  $dom(X)$  is the domain of variable  $X$ 
5:      $C$ : set of constraints to be satisfied
6:      $S$ : a cooling schedule for the temperature
7:      $k$ : population size – an even integer
8:   Output
9:     complete assignment that satisfies the constraints
10:  Local
11:     $Pop$ : a set of assignments
12:     $T$ : real
13:     $Pop \leftarrow k$  complete random assignments
14:     $T$  is assigned a value according to  $S$ 
15:  repeat
16:    if some  $A \in Pop$  satisfies all constraints in  $C$  then
17:      return  $A$ 
18:     $Npop \leftarrow \{\}$ 
19:    repeat  $k/2$  times
20:       $A_1 \leftarrow RandomSelection(Pop, T)$ 
21:       $A_2 \leftarrow RandomSelection(Pop, T)$ 
22:       $N_1, N_2 \leftarrow combine(A_1, A_2)$ 
23:       $Npop \leftarrow Npop \cup \{mutate(N_1), mutate(N_2)\}$ 
24:     $Pop \leftarrow Npop$ 
25:     $T$  is updated according to  $S$ 
26:  until termination
27: procedure RandomSelection( $Pop, T$ )
28:   select  $A$  from  $Pop$  with probability proportional to  $e^{-h(A)/T}$ 
29:   return  $A$ 
30: procedure Combine( $A_1, A_2$ )
31:   select integer  $i, 1 \leq i \leq \Psi$  at random
32:   Let  $N_1 \leftarrow (X_j = v_j) \mid A_1 \text{ for } j = 1 \dots i \cup (X_j = v_j) \mid A_2 \text{ for } j > i$  33:}
   Let  $N_2 \leftarrow (X_j = v_j) \mid A_2 \text{ for } j = 1 \dots i \cup (X_j = v_j) \mid A_1 \text{ for } j > i$  34:}
return  $N_1, N_2$ 

```

Figure 4.10: Genetic algorithm for finding a solution to a CSP

- For each pair, perform a crossover.
- Randomly mutate some (very few) values by choosing other values for some randomly chosen variables. This is a random walk step.

It proceeds in this way until it has created k individuals, and then the operation proceeds to the next generation. The algorithm is shown in Figure 4.10.

Example 4.28 Consider Example 4.8 (page 117). Suppose we use the same evaluation function as in Example 4.25 (page 132), namely the number of unsatisfied constraints. The individual $A = 2, B = 2, C = 3, D = 1, E = 1$ has an evaluation of 4. It has a low value mainly because $E = 1$. Its offspring that preserve this property will tend to have a lower evaluation than those that do not and, thus, will be more likely to survive. Other individuals may have low values for different reasons; for example, the individual $A = 4, B = 2, C = 3, D = 4, E = 4$ also has an evaluation of 4. It is low mainly because of the assignment of values to the first four variables. Again, offspring that preserve this property will be fitter and more likely to survive than those that do not. If these two were to mate, some of the offspring would inherit the bad properties of both and would die off. Some, by chance, would inherit the good properties of both. These would then have a better chance of survival.

As in other stochastic local search algorithms, it is difficult to design features and the evaluation function so that the algorithm does not get trapped in local minima. Efficiency is very sensitive to the variables used to describe the problem and the ordering of the variables. Getting this to work is an art. As with many other heuristic algorithms, evolutionary algorithms have many degrees of freedom and, therefore, are difficult to configure or tune for good performance. Also, analogies to natural evolution can be very misleading, because what is at work in nature is not always the best strategy for solving combinatorial decision or optimization problems.

A large community of researchers are working on genetic algorithms to make them practical for real problems and there have been some impressive results. What we have described here is only one of the possible genetic algorithms.

4.10 Optimization

Instead of just having possible worlds satisfy constraints or not, we often have a **preference** relation over possible worlds, and we want a best possible world according to the preference. The preference is often to minimize some error.

An **optimization problem** is given

- a set of variables, each with an associated domain;
- an **objective function** that maps total assignments to real numbers; and
- an **optimality criterion**, which is typically to find a total assignment that minimizes or maximizes the objective function.

The aim is to find a total assignment that is optimal according to the optimality criterion. For concreteness, we assume that the optimality criterion is to minimize the objective function.

A **constrained optimization problem** is an optimization problem that also has hard constraints specifying which variable assignments are possible. The aim is to find a best assignment that satisfies the hard constraints.

A huge literature exists on optimization. There are many techniques for particular forms of constrained optimization problems. For example, linear programming is the class of constrained optimization where the variables are real valued, the objective function is a linear function of the variables, and the hard constraints are linear inequalities. We do not cover these specific techniques. Although they have many applications, they have limited applicability in the space of all optimization problems. We do cover some general techniques that allow more general objective functions. However, if the problem you are interested in solving falls into one of the classes for which there are more specific algorithms, or can be transformed into one, it is generally better to use those techniques than the general algorithms presented here.

In a **constraint optimization problem**, the objective function is factored into a set of functions of subsets of the variables called soft constraints. A **soft constraint** assigns a cost for each assignment of values to some subset of the variables. The value of the objective function on a total assignment is the sum of the costs given by the soft constraints on that total assignment. A typical optimality criterion is to minimize the objective function.

Like a hard constraint (page 115), a soft constraint has a **scope** that is a set of variables. A soft constraint is a function from the domains of the variables in its scope into a real number, called its **evaluation**. Thus, given an assignment of a value to each variable in its scope, this function returns a real number.

Example 4.29 Suppose a number of delivery activities must be scheduled, similar to Example 4.8 (page 117), but, instead of hard constraints, there are preferences on times for the activities. The soft constraints are costs associated with combinations of times. The aim is to find a schedule with the minimum total sum of the costs.

Suppose variables A , C , D , and E have domain $\{1, 2\}$, and variable B has domain $\{1, 2, 3\}$. The soft constraints are

c_1 :	<table><tr><th>A</th><th>B</th><th>Cost</th></tr><tr><td>1</td><td>1</td><td>5</td></tr><tr><td>1</td><td>2</td><td>2</td></tr><tr><td>1</td><td>3</td><td>2</td></tr><tr><td>2</td><td>1</td><td>0</td></tr><tr><td>2</td><td>2</td><td>4</td></tr><tr><td>2</td><td>3</td><td>3</td></tr></table>	A	B	Cost	1	1	5	1	2	2	1	3	2	2	1	0	2	2	4	2	3	3	c_2 :	<table><tr><th>B</th><th>C</th><th>Cost</th></tr><tr><td>1</td><td>1</td><td>5</td></tr><tr><td>1</td><td>2</td><td>2</td></tr><tr><td>2</td><td>1</td><td>0</td></tr><tr><td>2</td><td>2</td><td>4</td></tr><tr><td>3</td><td>1</td><td>2</td></tr><tr><td>3</td><td>2</td><td>0</td></tr></table>	B	C	Cost	1	1	5	1	2	2	2	1	0	2	2	4	3	1	2	3	2	0	c_3 :	<table><tr><th>B</th><th>D</th><th>Cost</th></tr><tr><td>1</td><td>1</td><td>3</td></tr><tr><td>1</td><td>2</td><td>0</td></tr><tr><td>2</td><td>1</td><td>2</td></tr><tr><td>2</td><td>2</td><td>2</td></tr><tr><td>3</td><td>1</td><td>2</td></tr><tr><td>3</td><td>2</td><td>4</td></tr></table>	B	D	Cost	1	1	3	1	2	0	2	1	2	2	2	2	3	1	2	3	2	4
A	B	Cost																																																																		
1	1	5																																																																		
1	2	2																																																																		
1	3	2																																																																		
2	1	0																																																																		
2	2	4																																																																		
2	3	3																																																																		
B	C	Cost																																																																		
1	1	5																																																																		
1	2	2																																																																		
2	1	0																																																																		
2	2	4																																																																		
3	1	2																																																																		
3	2	0																																																																		
B	D	Cost																																																																		
1	1	3																																																																		
1	2	0																																																																		
2	1	2																																																																		
2	2	2																																																																		
3	1	2																																																																		
3	2	4																																																																		

Thus, the scope of c_1 is $\{A, B\}$, the scope of c_2 is $\{B, C\}$, and the scope of c_3 is $\{B, D\}$. Suppose there are also constraints $c_4(C, E)$ and $c_5(D, E)$.

An assignment to some of the variables in a soft constraint results in a soft constraint that is a function of the other variables. In the preceding example, $c_1(A=1)$ is a function of B , which when applied to $B=2$ evaluates to 2.

Given a total assignment, the evaluation of the total assignment is the sum of the evaluations of the soft constraints applied to the total assignment.

One way to formalize this is to define operations on the soft constraints. Soft constraints can be added pointwise. The sum of two soft constraints is a soft constraint with scope that is the union of their scopes. The value of any assignment to the scope is the sum of the two functions on that assignment.

Example 4.30 Consider functions c_1 and c_2 in the previous example. $c_1 + c_2$ is a function with scope $\{A, B, C\}$, given by

$c_1 + c_2$:	A	B	C	Cost
	1	1	1	10
	1	1	2	7
	1	2	1	2
	...	...	...	...

The second value is computed as follows:

$$\begin{aligned}(c_1 + c_2)(A=1, B=1, C=2) &= c_1(A=1, B=1) + c_2(B=1, C=2) \\ &= 5 + 2 \\ &= 7\end{aligned}$$

One way to find the optimal assignment, corresponding to **generate and test** (page 118), is to compute the sum of the soft constraints and to choose an assignment with minimum value. Later we consider other, more efficient, algorithms for optimization.

Hard constraints can be modeled as having a cost of infinity for violating a constraint. As long as the cost of an assignment is finite, it does not violate a hard constraint. An alternative is to use a large number – larger than the sum of the soft constraints could be – as the cost of violating a hard constraint. Then optimization can be used to find a solution with the fewest number of violated hard constraints and, among those, one with the lowest cost.

Optimization problems have one difficulty that goes beyond constraint satisfaction problems. It is difficult to know whether an assignment is optimal. Whereas, for a CSP, an algorithm can check whether an assignment is a solution by just considering the assignment and the constraints, in optimization problems an algorithm can only determine if an assignment is optimal by comparing it to other assignments.

Many of the methods for solving hard constraints can be extended to optimization problems, as outlined in the following sections.

4.10.1 Systematic Methods for Optimization

Arc consistency (page 121) can be generalized to optimization problems by allowing pruning of dominated assignments. Suppose $c_1, \dots, c_k$ are the soft constraints that involve X . Let soft constraint $c = c_1 + \dots + c_k$. Suppose Y are the variables, other than X , that are involved in c . A value v for variable X is

strictly dominated if, for all values y of Y , some value v^t of X exists such that $c(X = v^t, Y = y) < c(X = v, Y = y)$. Pruning strictly dominated values does not remove an optimal solution. The pruning of domains can be done repeatedly, as in the GAC algorithm (page 122).

Weakly dominated has the same definition as strictly dominated, but with “less than” replaced by “less than or equal to.” If only one solution is required, weakly dominated values can be pruned sequentially. Which weakly dominated values are removed may affect which optimal solution is found, but removing a weakly dominated value does not remove all optimal solutions. As with arc consistency for hard constraints, pruning (strictly or weakly) dominated values can greatly simplify the problem but does not, by itself, always solve the problem.

Domain splitting (page 125) can be used to build a search tree. Domain splitting picks some variable X and considers each value of X . Assigning a value to X allows the constraints that involve X to be simplified and values for other variables to be pruned. In particular, pruning weakly dominated values means that, when there is only one variable left, a best value for that variable can be computed. Repeated domain splitting can build a search tree, as in Figure 4.1 (page 120), but with values at the leaves. By assigning costs when they can be determined, search algorithms such as A^* or branch-and-bound can be used to find an optimal solution.

Domain splitting can be improved via two techniques. First, if, under a split on X , an assignment to another variable does not depend on the value of X , the computation for that variable can be shared among the subtrees for the values of X ; the value can be computed once and cached. Second, if removing a set of variables would disconnect the constraint graph, then when those variables have been assigned values, the disconnected components can be solved independently.

Variable elimination is the dynamic programming variant of domain splitting. The variables are eliminated one at a time. A variable X is eliminated as follows. Let R be the set of constraints that involve X . T is a new constraint whose scope is the union of the scopes of the constraints in R and whose value is the sum of the values of R . Let $V = \text{scope}(T) \setminus X$. For each value of the variables in V , select a value of X that minimizes T , resulting in a new soft constraint, N , with scope V . The constraint N replaces the constraints in R . This results in a new problem, with fewer variables and a new set of constraints, which can be solved recursively. A solution, S , to the reduced problem is an assignment to the variables in V . Thus, $T(S)$, the constraint T under the assignment S , is a function of X . An optimal value for X is obtained by choosing a value that results in the minimum value of $T(S)$.

Figure 4.11 (on the next page) gives pseudocode for the VE algorithm. The elimination ordering can be given a priori or can be computed on the fly, for example, using the elimination ordering heuristics discussed for CSP VE (page 130). It is possible to implement this without storing T and only by constructing an extensional representation of N .

```

1: procedure VE SC(Vs, Fs)
2:   Inputs
3:     Vs: set of variables
4:     Fs: set of constraints
5:   Output
6:     an optimal assignment to Vs.
7:   if Vs contains a single element or Fs contains a single constraint then
8:     let F be the sum of the constraints in Fs
9:     return assignment with minimum value in F
10:  else
11:    select  $X \in Vs$  according to some elimination ordering
12:     $R = \{F \in Fs : F \text{ involves } X\}$ 
13:    let T be the sum of the constraints in R
14:     $N := \min_X T$ 
15:     $S := \text{VE\_SC}(Vs \setminus \{X\}, Fs \setminus R \cup \{N\})$ 
16:     $X_{opt} := \arg \min_X T(S)$ 
17:    return  $S \cup \{X = X_{opt}\}$ 

```

Figure 4.11: Variable elimination for optimizing with soft constraints

Example 4.31 Consider Example 4.29 (page 145). First consider eliminating *A*. It appears in only one constraint, $c_1(A, B)$. Eliminating *A* gives

$$c_6(B) = \arg \min_A c_1(A, B):$$

<i>B</i>	Cost
1	0
2	2
3	2

The constraint $c_1(A, B)$ is replaced by $c_6(B)$.

Suppose *B* is eliminated next. *B* appears in three constraints: $c_2(B, C)$, $c_3(B, D)$, and $c_6(B)$. These three constraints are added, giving

$$c_2(B, C) + c_3(B, D) + c_6(B):$$

<i>B</i>	<i>C</i>	<i>D</i>	Cost
1	1	1	8
1	1	2	5
...			
2	1	1	4
2	1	2	4
...			
3	1	1	6
3	1	2	8

The constraints c_2 , c_3 , and c_6 are replaced by

$$c_7(C, D) = \min_B (c_2(B, C) + c_3(B, D) + c_6(B)):$$

<i>C</i>	<i>D</i>	Cost
1	1	4
1	2	4
...		

There are now three remaining constraints: $c_4(C, E)$, $c_5(D, E)$, and $c_7(C, D)$. These can be optimized recursively.

Suppose the recursive call returns the solution $C = 1, D = 2, E = 2$. An optimal value for B is the value that gives the minimum in $c_2(B, C = 1) + c_3(B, D = 2) + c_6(B)$, which is $B = 2$.

From $c_1(A, B)$, the value of A that minimizes $c_1(A, B = 2)$ is $A = 1$. Thus, an optimal solution is $A = 1, B = 2, C = 1, D = 2, E = 2$.

The complexity of VE depends on the structure of the constraint graph, as it does with hard constraints (page 129). Sparse graphs may result in small intermediate constraints in VE algorithms, including *VE-SC*. Densely connected graphs result in large intermediate constraints.

4.10.2 Local Search for Optimization

Local search is directly applicable to optimization problems, using the objective function of the optimization problem as the evaluation function of the local search. The algorithm runs for a certain amount of time (perhaps including random restarts to explore other parts of the search space), always keeping the best assignment found thus far, and returning this as its answer.

Local search for optimization has one extra complication that does not arise with only hard constraints: it is difficult to determine whether a total assignment is the best possible solution. A **local minimum** is a total assignment that is at least as good, according to the optimality criterion, as any of its neighbors. A **global minimum** is a total assignment that is at least as good as all of the other total assignments. Without systematically searching the other assignments, the algorithm may not know whether the best assignment found so far is a global optimum or whether a better solution exists in a different part of the search space.

When solving constrained optimization problems, with both hard and soft constraints, there could be a trade-off between violating hard constraints and making the evaluation function worse. We typically do not treat a violated constraint as having a cost of infinity, because then the algorithms would not distinguish violating one hard constraint from violating many. It is sometimes good to allow hard constraints to be temporarily violated in the search.

Continuous Domains

For optimization where the domains are continuous, a local search becomes more complicated because it is not obvious what the neighbors of a total assignment are. This problem does not arise with hard constraint satisfaction problems because the constraints implicitly discretize the space.

For optimization, **gradient descent** can be used to find a minimum value, and **gradient ascent** can be used to find a maximum value. Gradient descent is like walking downhill and always taking a step in the direction that goes down the most. The general idea is that the neighbor of a total assignment is to step downhill in proportion to the slope of the evaluation function h . Thus, gradient

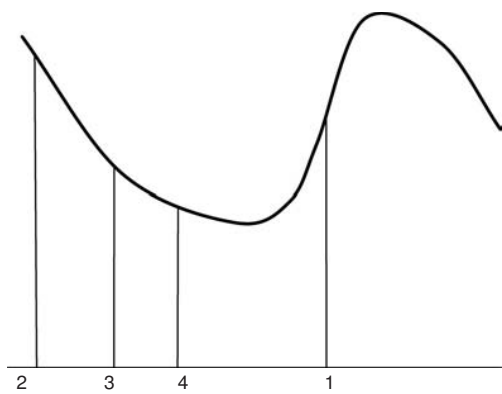


Figure 4.12: Gradient descent

descent takes steps in each direction proportional to the negative of the partial derivative in that direction.

In one dimension, if X is a real-valued variable with the current value of v , the next value should be

$$v - \eta \times \frac{dh}{dX} (v),$$

where

- η , the **step size**, is the constant of proportionality that determines how fast gradient descent approaches the minimum. If η is too large, the algorithm can overshoot the minimum; if η is too small, progress becomes very slow.
- $\frac{dh}{dX}$, the derivative of h with respect to X , is a function of X and is evaluated for $X = v$.

Example 4.32 Figure 4.12 shows a typical one-dimensional example for finding a local minimum of a one-dimensional function. It starts at a position marked as 1. The derivative is a big positive value, so it takes a step to the left to position 2. Here the derivative is negative, and closer to zero, so it takes a smaller step to the right to position 3. At position 3, the derivative is negative and closer to zero, so it takes a smaller step to the right. As it approaches the local minimum value, the slope becomes closer to zero and it takes smaller steps.

For multidimensional optimization, when there are many variables, gradient descent takes a step in each dimension proportional to the partial derivative of that dimension. If $X_1, \dots, X_n$ are the variables that have to be assigned values, a total assignment corresponds to a tuple of values $v_1, \dots, v_n$. Assume that the evaluation function, h , is differentiable. The next neighbor of the total assignment $v_1, \dots, v_n$ is obtained by moving in each direction in proportion to the slope of h in that direction. The new value for X_i is

$$v_i - \eta \times \frac{\partial h}{\partial X_i} (v_1, \dots, v_n),$$

where η is the **step size**. The partial derivative, $\frac{\partial h}{\partial X_i}$ is a function of $X_1, \dots, X_n$. Applying it to the point $(v_1, \dots, v_n)$ gives

$$\frac{\partial h}{\partial X_i}(v_1, \dots, v_n) = \lim_{c \rightarrow 0} \frac{h(v_1, \dots, v_i + c, \dots, v_n) - h(v_1, \dots, v_i, \dots, v_n)}{c}.$$

If the partial derivative of h can be computed analytically, it is usually good to do so. If not, it can be estimated using a small value of c .

Gradient descent is used for parameter learning (page 304), in which there may be thousands of real-valued parameters to be optimized. There are many variants of this algorithm that we do not discuss. For example, instead of using a constant step size, the algorithm could do a binary search to determine a locally optimal step size.

4.11 References and Further Reading

Constraint satisfaction techniques are described in [Dechter \[2003\]](#) and [Freuder and Mackworth \[2006\]](#). The GAC algorithm was invented by [Mackworth \[1977\]](#).

The DPLL algorithm (page 127) was invented by [Davis, Logemann, and Loveland \[1962\]](#).

VE for propositional satisfiability was proposed by [Davis and Putnam \[1960\]](#). VE for optimization has been called **non-serial dynamic programming** and was invented by [Bertelè and Brioschi \[1972\]](#).

A1,D1	D2	D3
A2		
A3		

Word list:
add, ado, age, ago, aid,
ail, aim, air, and, any,
ape, apt, arc, are, ark,
arm, art, ash, ask, auk,
awe, awl, aye, bad, bag,
ban, bat, bee, boa, ear,
eel, eft, far, fat, fit,
lee, oaf, rat, tar, tie.

Figure 4.13: A crossword puzzle to be solved with six words

Stochastic local search is described by [Spall \[2003\]](#) and [Hoos and Stützle \[2004\]](#). The any-conflict heuristic is based on the min-conflict heuristic of [Minton, Johnston, Philips, and Laird \[1992\]](#). Simulated annealing was invented by [Kirkpatrick, Gelatt, and Vecchi \[1983\]](#).

Genetic algorithms were pioneered by [Holland \[1975\]](#). A huge literature exists on genetic algorithms; for overviews see [Goldberg \[1989\]](#), [Koza \[1992\]](#), [Mitchell \[1996\]](#), [Bäck \[1996\]](#), [Whitley \[2001\]](#), and [Goldberg \[2002\]](#).

CASE STUDIES

5.6 Prototypical Applications

AI applications are widespread and diverse and include medical diagnosis, scheduling factory processes, robots for hazardous environments, game play- ing, autonomous vehicles in space, natural language translation systems, and tutoring systems. Rather than treating each application separately, we abstract the essential features of such applications to allow us to study the principles behind intelligent reasoning and action.

This section outlines four application domains that will be developed in ex- amples throughout the book. Although the particular examples presented are simple – otherwise they would not fit into the book – the application domains

are representative of the range of domains in which AI techniques can be, and are being, used.

The four application domains are as follows:

- An **autonomous delivery robot** roams around a building delivering packages and coffee to people in the building. This delivery agent should be able to find paths, allocate resources, receive requests from people, make decisions about priorities, and deliver packages without injuring people or itself.
- A **diagnostic assistant** helps a human troubleshoot problems and suggests repairs or treatments to rectify the problems. One example is an electrician's assistant that suggests what may be wrong in a house, such as a fuse blown, a light switch broken, or a light burned out, given some symptoms of electrical problems. Another example is a medical diagnostician that finds potential diseases, useful tests, and appropriate treatments based on knowledge of a particular medical domain and a patient's symptoms and history. This assistant should be able to explain its reasoning to the person who is carrying out the tests and repairs and who is ultimately responsible for their actions.
- A **tutoring system** interacts with a student, presenting information about some domain and giving tests of the student's knowledge or performance. This entails more than presenting information to students. Doing what a good teacher does, namely tailoring the information presented to each student based on his or her knowledge, learning preferences, and misunderstandings, is more challenging. The system must understand both the subject matter and how students learn.
- A **trading agent** knows what a person wants and can buy goods and services on her behalf. It should know her requirements and preferences and how to trade off competing objectives. For example, for a family holiday a travel agent must book hotels, airline flights, rental cars, and entertainment, all of which must fit together. It should determine a customer's trade-offs. If the most suitable hotel cannot accommodate the family for all of the days, it should determine whether they would prefer to stay in the better hotel for part of the stay or if they prefer not to move hotels. It may even be able to shop around for specials or to wait until good deals come up.

These four domains will be used for the motivation for the examples in the book. In the next sections, we discuss each application domain in detail.

5.1.6 An Autonomous Delivery Robot

Imagine a robot that has wheels and can pick up objects and put them down. It has sensing capabilities so that it can recognize the objects that it must manipulate and can avoid obstacles. It can be given orders in natural language and obey them, making reasonable choices about what to do when its goals conflict. Such a robot could be used in an office environment to deliver packages, mail, and/or coffee, or it could be embedded in a wheelchair to help disabled people. It should be useful as well as safe.

In terms of the black box characterization of an agent in Figure 1.3 (page 11), the autonomous delivery robot has the following as inputs:

- prior knowledge, provided by the agent designer, about its own capabilities, what objects it may encounter and have to differentiate, what requests mean, and perhaps about its environment, such as a map;
- past experience obtained while acting, for instance, about the effect of its actions, what objects are common in the world, and what requests to expect at different times of the day;
- goals in terms of what it should deliver and when, as well as preferences that specify trade-offs, such as when it must forgo one goal to pursue another, or the trade-off between acting quickly and acting safely; and
- observations about its environment from such input devices as cameras, sonar, touch, sound, laser range finders, or keyboards.

The robot's outputs are motor controls that specify how its wheels should turn, where its limbs should move, and what it should do with its grippers. Other outputs may include speech and a video display.

In terms of the dimensions of complexity, the simplest case for the robot is a flat system, represented in terms of states, with no uncertainty, with achievement goals, with no other agents, with given knowledge, and with perfect rationality. In this case, with an indefinite stage planning horizon, the problem of deciding what to do is reduced to the problem of finding a path in a graph of states. This is explored in Chapter 3.

Each dimension can add conceptual complexity to the task of reasoning:

- A hierarchical decomposition can allow the complexity of the overall system to be increased while allowing each module to be simple and able to be understood by itself. This is explored in Chapter 2.
- Modeling in terms of features allows for a much more comprehensible system than modeling explicit states. For example, there may be features for the robot's location, the amount of fuel it has, what it is carrying, and so forth. Reasoning in terms of the states, where a state is an assignment of a value to each feature, loses the structure that is provided by the features. Reasoning in terms of the feature representation can be exploited for computational gain. Planning in terms of features is discussed in Chapter 8. When dealing with multiple individuals (e.g., multiple people or objects to deliver), it may be easier to reason in terms of individuals and relations. Planning in terms of individuals and relations is explored in Section 14.1 (page 598).
- The planning horizon can be finite if the agent only looks ahead a few steps. The planning horizon can be indefinite if there is a fixed set of goals to achieve. It can be infinite if the agent has to survive for the long term, with ongoing requests and actions, such as delivering mail whenever it arrives and recharging its battery when its battery is low.
- There could be goals, such as "deliver coffee to Chris and make sure you always have power." A more complex goal may be to "clean up the lab, and put everything where it belongs." There can be complex preferences, such as

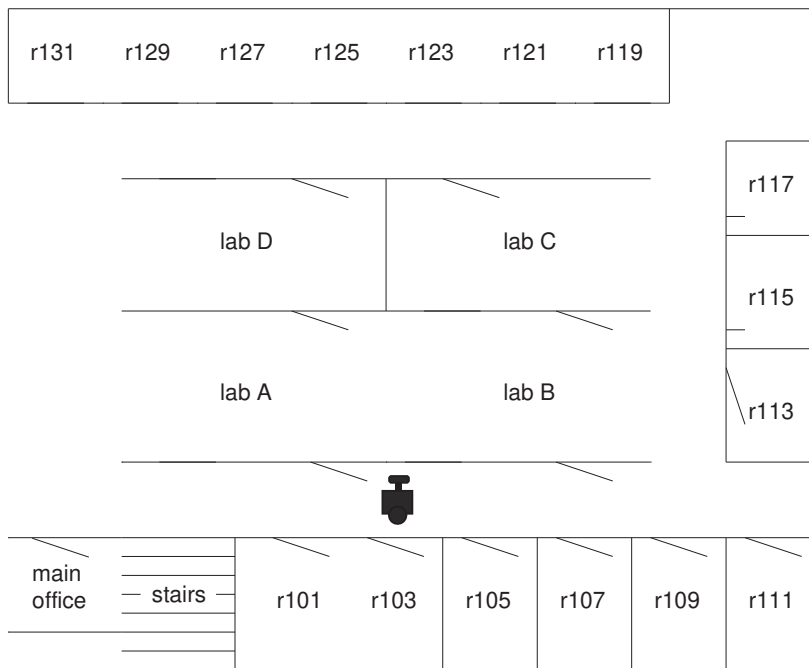


Figure 1.7: An environment for the delivery robot, which shows a typical laboratory environment. This also shows the locations of the doors and which way they open.

“deliver mail when it arrives and service coffee requests as soon as possible, but it is more important to deliver messages marked as important, and Chris really needs her coffee quickly when she asks for it.”

- There can be sensing uncertainty because the robot does not know what is in the world based on its limited sensors.
- There can be uncertainty about the effects of an action, both at the low level, such as due to slippage of the wheels, or at the high level in that the agent might not know if putting the coffee on Chris’s desk succeeded in delivering coffee to her.
- There can be multiple robots, which can coordinate to deliver coffee and parcels and compete for power outlets. There may also be children out to trick the robot.
- A robot has lots to learn, such as how slippery floors are as a function of their shininess, where Chris hangs out at different parts of the day and when she will ask for coffee, and which actions result in the highest rewards.

Figure 1.7 depicts a typical laboratory environment for a delivery robot. This environment consists of four laboratories and many offices. The robot can only push doors, and the directions of the doors in the diagram reflect the directions in which the robot can travel. Rooms require keys, and those keys can be obtained from various sources. The robot must deliver parcels,

beverages, and dishes from room to room. The environment also contains a stairway that is potentially hazardous to the robot.

Chapter 6

Reasoning Under Uncertainty

It is remarkable that a science which began with the consideration of games of chance should become the most important object of human knowledge ... The most important questions of life are, for the most part, really only problems of probability . . .

The theory of probabilities is at bottom nothing but common sense reduced to calculus.

– Pierre Simon de [Laplace](#) [1812]

All of the time, agents are forced to make decisions based on incomplete information. Even when an agent senses the world to find out more information, it rarely finds out the exact state of the world. A robot does not know exactly where an object is. A doctor does not know exactly what is wrong with a patient. A teacher does not know exactly what a student understands. When intelligent agents must make decisions, they have to use whatever information they have. This chapter considers reasoning under uncertainty: determining what is true in the world based on observations of the world. This is used in Chapter 9 as a basis for acting under uncertainty, where the agent must make decisions about what action to take even though it cannot precisely predict the outcomes of its actions. This chapter starts with probability, shows how to represent the world by making appropriate independence assumptions, and shows how to reason with such representations.

6.1 Probability

To make a good decision, an agent cannot simply assume what the world is like and act according to those assumptions. It must consider multiple possible contingencies and their likelihood. Consider the following example.

219

Example 6.1 Many people consider it sensible to wear a seat belt when traveling in a car because, in an accident, wearing a seat belt reduces the risk of serious injury. However, consider an agent that commits to assumptions and bases its decision on these assumptions. If the agent assumes it will not have an accident, it will not bother with the inconvenience of wearing a seat belt. If it assumes it will have an accident, it will not go out. In neither case would it wear a seat belt! A more intelligent agent may wear a seat belt because the inconvenience of wearing a seat belt is far outweighed by the increased risk of injury or death if it has an accident. It does not stay at home too worried about an accident to go out; the benefits of being mobile, even with the risk of an accident, outweigh the benefits of the extremely cautious approach of never going out. The decisions of whether to go out and whether to wear a seat belt depend on the likelihood of having an accident, how much a seat belt helps in an accident, the inconvenience of wearing a seat belt, and how important it is to go out. The various trade-offs may be different for different agents. Some people do not wear seat belts, and some people do not go out because of the risk of accident.

Reasoning under uncertainty has been studied in the fields of probability theory and decision theory. Probability is the calculus of **gambling**. When an agent makes decisions and uncertainties are involved about the outcomes of its action, it is gambling on the outcome. However, unlike a gambler at the casino, the agent cannot opt out and decide not to gamble; whatever it does – including doing nothing – involves uncertainty and risk. If it does not take the probabilities into account, it will eventually lose at gambling to an agent that does. This does not mean, however, that making the best decision guarantees a win.

Many of us learn probability as the theory of tossing coins and rolling dice. Although this may be a good way to present probability theory, probability is applicable to a much richer set of applications than coins and dice. In general, we want a calculus for belief that can be used for making decisions.

The view of probability as a measure of belief, as opposed to being a frequency, is known as **Bayesian probability** or **subjective probability**. The term *subjective* does not mean *arbitrary*, but rather it means “belonging to the subject.” For example, suppose there are three agents, Alice, Bob, and Chris, and one die that has been tossed. Suppose Alice observes that the outcome is a “6” and tells Bob that the outcome is even, but Chris knows nothing about the outcome. In this case, Alice has a probability of 1 that the outcome is a “6,” Bob has a probability of $\frac{1}{3}$ that it is a “6” (assuming Bob believes Alice and treats all of the even outcomes with equal probability), and Chris may have probability of $\frac{1}{6}$ that the outcome is a “6.” They all have different probabilities because they all have different knowledge. The probability is about the outcome of this particular toss of the die, not of some generic event of tossing dice. These agents may have the same or different probabilities for the outcome of other coin tosses.

The alternative is the **frequentist** view, where the probabilities are long-run frequencies of repeatable events. The Bayesian view of probability is appropriate for intelligent agents because a measure of belief in particular situations

is what is needed to make decisions. Agents do not encounter generic events but have to make a decision based on uncertainty about the particular circumstances they face.

Probability theory can be defined as the study of *how knowledge affects belief*. Belief in some proposition, a , can be measured in terms of a number between 0 and 1. The probability a is 0 means that a is believed to be definitely false (no new evidence will shift that belief), and a probability of 1 means that a is believed to be definitely true. Using 0 and 1 is purely a convention.

Adopting the belief view of probabilities does not mean that statistics are ignored. Statistics of what has happened in the past is knowledge that can be conditioned on and used to update belief. (See Chapter 7 for how to learn probabilities.)

We are assuming that the uncertainty is **epistemological** – pertaining to an agent’s knowledge of the world – rather than **ontological** – how the world is. We are assuming that an agent’s knowledge of the truth of propositions is uncertain, not that there are degrees of truth. For example, if you are told that someone is very tall, you know they have some height; you only have vague knowledge about the actual value of their height.

If an agent’s probability of some a is greater than zero and less than one, this does not mean that a is true to some degree but rather that the agent is ignorant of whether a is true or false. The probability reflects the agent’s ignorance.

For the rest of this chapter, we ignore the agent whose beliefs we are modeling and only talk about the probability.

6.1.1 Semantics of Probability

Probability theory is built on the same foundation of worlds and variables as constraint satisfaction [see Section 4.2 (page 113)]. Instead of having constraints that eliminate some worlds and treat every other world as possible, probabilities put a measure over the possible worlds. The variables in probability theory are referred to as **random variables**. The term *random variable* is somewhat of a misnomer because it is neither random nor variable. As discussed in Section 4.2 (page 113), worlds can be described in terms of variables; a world corresponds to an assignment of a value to each variable. Alternatively, variables can be described in terms of worlds; a variable is a function that returns a value on each world.

First we define probability as a measure on sets of worlds, then define probabilities on propositions, then on variables.

A **probability measure** over the worlds is a function μ from sets of worlds into the non-negative real numbers such that

- if Ω_1 and Ω_2 are disjoint sets of worlds (i.e., if $\Omega_1 \cap \Omega_2 = \{\}$), then $\mu(\Omega_1 \cup \Omega_2) = \mu(\Omega_1) + \mu(\Omega_2)$;
- if Ω is the set of all worlds, $\mu(\Omega) = 1$.

Note that the use of 1 as the probability of the set of all of the worlds is just by convention. You could just as well use 100.

It is possible to have infinitely many worlds when some variables have infinite domains or when infinitely many variables exist. When there are infinitely many worlds, we do not require a measure for all subsets of Ω – just for those sets that can be described using some language that we assume allows us to describe the intersection, union, and complement of sets. The set of subsets describable by these operations has the structure of what mathematicians call an **algebra**.

The measure μ can be extended to worlds by defining $\mu(\omega) = \mu(\{\omega\})$ for world ω . When finitely many worlds exist, the measure over individual worlds is adequate to define μ . When infinitely many worlds exist, it is possible that the measure of individual worlds is not enough information to define μ , or that it may not make sense to have a measure over individual worlds.

Example 6.2 ~~Suppose the worlds correspond~~ to the possible real-valued heights, in centimeters, of a particular person. In this example, there are infinitely many possible worlds. The measure of the set of heights in the range $[175, 180)$ could be 0.2 and the measure of the range $[180, 190)$ could be 0.3. Then the measure of the range $[175, 190)$ is 0.5. However, the measure of any particular height could be zero.

As described in Section 5.1 (page 157), a primitive proposition is an assignment of a value to a variable. Propositions are built from primitive propositions using logical connectives. We then use this property to define probability distributions over variables.

The **probability** of proposition a , written $P(a)$, is the measure of the set of possible worlds in which a is true. That is,

$$P(a) = \mu(\{\omega : \omega \models a\}),$$

where $\omega \models a$ means a is true in world ω . Thus, $P(a)$ is the measure of the set of worlds in which a is true.

This use of the symbol $\models$ differs from its use in the previous chapter [see Section 5.1.2 (page 160)]. There, the left-hand side was a knowledge base; here, the left-hand side is a world. Which meaning is intended should be clear from the context.

A **probability distribution**, $P(X)$, over a random variable X is a function from the domain of X into the real numbers such that, given a value $x \in \text{dom}(X)$, $P(x)$ is the probability of the proposition $X = x$. We can also define a probability distribution over a set of variables analogously. For example, $P(X, Y)$ is a probability distribution over X and Y such that $P(X = x, Y = y)$, where $x \in \text{dom}(X)$ and $y \in \text{dom}(Y)$, has the value $P(X = x \wedge Y = y)$, where $X = x \wedge Y = y$ is a proposition and P is the function on propositions. We will use probability distributions when we want to treat a set of probabilities as a unit.

6.1.2 Axioms for Probability

The preceding section gave a semantic definition of probability. We can also give an axiomatic definition of probability that specifies axioms of what properties we may want in a calculus of belief.

Probability Density Functions

The definition of probability is sometimes specified in terms of probability density functions when the domain is continuous (e.g., a subset of the real numbers). A probability density function provides a way to give a measure over sets of possible worlds. The measure is defined in terms of an integral of a probability density function. The formal definition of an integral is the limit of the discretizations as the discretizations become finer.

The only non-discrete probability distribution we will use in this book is where the domain is the real line. In this case, there is a possible world for each real number. A **probability density function**, which we write as p , is a function from reals into non-negative reals that integrates to 1. The probability that a real-valued random variable X has value between a and b is given by

$$P(a \leq X \leq b) = \int_a^b p(X) dX.$$

A **parametric distribution** is one where the density function can be described by a formula. Although not all distributions can be described by formulas, all of the ones that we can represent are. Sometimes statisticians use the term parametric to mean the distribution can be described using a fixed, finite number of parameters. A **non-parametric distribution** is one where the number of parameters is not fixed. (Oddly, non-parametric typically means “many parameters”).

A common parametric distribution is the **normal or Gaussian distribution** with mean μ and variance σ^2 defined by

$$p(X) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}((X-\mu)/\sigma)^2},$$

where σ is the standard deviation. The normal distribution is used for measurement errors, where there is an average value, given by μ , and a variation in values specified by σ . The **central limit theorem**, proved by [Laplace \[1812\]](#), specifies that a sum of independent errors will approach the Gaussian distribution. This and its nice mathematical properties account for the widespread use of the normal distribution.

Other distributions, including the beta and Dirichlet distributions, are discussed in the Learning Under Uncertainty section (page [334](#)).

Suppose P is a function from propositions into real numbers that satisfies the following three **axioms of probability**:

Axiom 1 $0 \leq P(a)$ for any proposition a . That is, the belief in any proposition cannot be negative.

Axiom 2 $P(\tau) = 1$ if τ is a tautology. That is, if τ is true in all possible worlds, its probability is 1.

Axiom 3 $P(a \vee b) = P(a) + P(b)$ if a and b are contradictory propositions; that is, if $(a \wedge b)$ is a contradiction. In other words, if two propositions cannot both be true (they are mutually exclusive), the probability of their disjunction is the sum of their probabilities.

These axioms are meant to be intuitive properties that we would like to have of any reasonable measure of belief. If a measure of belief follows these intuitive axioms, it is covered by probability theory, whether or not the measure is derived from actual frequency counts. These axioms form a sound and complete axiomatization of the meaning of probability. Soundness means that probability, as defined by the possible-worlds semantics, follows these axioms. Completeness means that any system of beliefs that obeys these axioms has a probabilistic semantics.

Proposition 6.1. *If there are a finite number of finite discrete random variables, Axioms 1, 2, and 3 are sound and complete with respect to the semantics.*

It is easy to check that these axioms are true of the semantics. In the other way around, you can use the axioms to compute any probability from the probability of worlds, because the descriptions of two worlds are mutually exclusive. The full proof is left as an exercise.

Proposition 6.2. *The following hold for all propositions a and b :*

(a) *Negation of a proposition:*

$$P(\neg a) = 1 - P(a).$$

(b) *If $a \equiv b$, then $P(a) = P(b)$. That is, logically equivalent propositions have the same probability.*

(c) *Reasoning by cases:*

$$P(a) = P(a \wedge b) + P(a \wedge \neg b).$$

(d) *If V is a random variable with domain D , then, for all propositions a ,*

$$P(a) = \sum_{d \in D} P(a \wedge V = d).$$

(e) *Disjunction for non-exclusive propositions:*

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b).$$

Proof.

- (a) The propositions $a \vee \neg a$ and $\neg(a \wedge \neg a)$ are tautologies. Therefore, $1 = P(a \vee \neg a) = P(a) + P(\neg a)$. Rearranging gives the desired result.
- (b) If $a \wedge \neg b$ is a tautology, so $P(a \wedge \neg b) = 1$, and a and $\neg b$ are contradictory statements, so we can use Axiom 3 to give $P(a \wedge \neg b) = P(a) + P(\neg b)$. Using part (a), $P(\neg b) = 1 - P(b)$. Thus, $P(a) + 1 - P(b) = 1$, and so $P(a) = P(b)$.
- (c) The proposition $a \leftrightarrow ((a \wedge b) \vee (a \wedge \neg b))$ and $\neg((a \wedge b) \wedge (a \wedge \neg b))$ are tautologies. Thus, $P(a) = P((a \wedge b) \vee (a \wedge \neg b)) = P(a \wedge b) + P(a \wedge \neg b)$.
- (d) The proof is analogous to the proof of part (c).
- (e) $(a \vee b) \leftrightarrow ((a \wedge \neg b) \vee b)$ is a tautology. Thus,

$$\begin{aligned} P(a \vee b) &= P((a \wedge \neg b) \vee b) \\ &= P(a \wedge \neg b) + P(b). \end{aligned}$$

Part (c) shows $P(a \wedge \neg b) = P(a) - P(a \wedge b)$. Thus,

$$P(a \vee b) = P(a) - P(a \wedge b) + P(b).$$

—

6.1.3 Conditional Probability

Typically, we do not only want to know the prior probability of some proposition, but we want to know how this belief is updated when an agent observes new evidence.

The measure of belief in proposition h based on proposition e is called the **conditional probability of h given e** , written $P(h|e)$.

A formula e representing the conjunction of *all* of the agent's **observations** of the world is called **evidence**. Given evidence e , the conditional probability $P(h|e)$ is the agent's **posterior probability** of h . The probability $P(h)$ is the **prior probability** of h and is the same as $P(h|true)$ because it is the probability before the agent has observed anything.

The posterior probability involves conditioning on *everything* the agent knows about a particular situation. All evidence must be conditioned on to obtain the correct posterior probability.

Example 6.3 For the diagnostic assistant, the patient's symptoms will be the evidence. The prior probability distribution over possible diseases is used before the diagnostic agent finds out about the particular patient. The posterior probability is the probability the agent will use after it has gained some evidence. When the agent acquires new evidence through discussions with the patient, observing symptoms, or the results of lab tests, it must update its posterior probability to reflect the new evidence. The new evidence is the conjunction of the old evidence and the new observations.

Example 6.4 The information that the delivery robot receives from its sensors is its evidence. When sensors are noisy, the evidence is what is known, such as the particular pattern received by the sensor, not that there is a person in front of the robot. The robot could be mistaken about what is in the world but it knows what information it received.

Semantics of Conditional Probability

Evidence e will rule out all possible worlds that are incompatible with e . Like the definition of logical consequence, the given formula e selects the possible worlds in which e is true. Evidence e induces a new measure, μ_e , over possible worlds where all worlds in which e is false have measure 0, and the remaining worlds are normalized so that the sum of the measures of the worlds is 1.

Here we go back to basic principles to define conditional probability. This basic definition is often useful when faced with unusual cases.

The definition of the measure follows from two intuitive properties:

- If S is a set of possible worlds, all of which have e true, define $\mu_e(S) = c \times \mu(S)$ for some constant c (which we derive below).
- If S is a set of worlds, all of which have e false, define $\mu_e(S) = 0$.

Other Possible Measures of

Justifying other measures of belief is problematic. Consider, for example, the proposal that the belief in $a \rightarrow b$ is some function of the belief in a and the belief in b . Such a measure of belief is called **compositional**. To see why this is not sensible, consider the single toss of a fair coin. Compare the case where a_1 is “the coin will land heads” and b_1 is “the coin will land tails” with the case where a_2 is “the coin will land heads” and b_2 is “the coin will land heads.” For these two cases, the belief in a_1 would seem to be the same as the belief in a_2 , and the belief in b_1 would be the same as the belief in b_2 . But the belief in $a_1 \rightarrow b_1$, which is impossible, is very different from the belief in $a_2 \rightarrow b_2$, which is the same as a_2 .

The conditional probability $P(f|e)$ is very different from the probability of the implication $P(e \rightarrow f)$. The latter is the same as $P(\neg e \rightarrow f)$, which is the measure of the interpretations for which f is true or e is false. For example, suppose you have a domain where birds are relatively rare, and non-flying birds are a small proportion of the birds. Here $P(\neg \text{flies} | \text{bird})$ would be the proportion of birds that do not fly, which would be low. $P(\text{bird} \rightarrow \text{flies})$ is the same as $P(\neg \text{bird} \vee \text{flies})$, which would be dominated by non-birds and so would be high. Similarly, $P(\text{bird} \rightarrow \text{flies})$ would also be high, the probability also being dominated by the non-birds. It is difficult to imagine a situation where the probability of an implication is the kind of knowledge that is appropriate or useful.

We want μ_e to be a probability measure, so if Ω is the set of all possible worlds, $\mu_e(\Omega) = 1$. Thus, $1 = \mu_e(\Omega) = \mu_e(\{\omega : \omega \models e\}) + \mu_e(\{\omega : \omega \models \neg e\}) = c \times \mu(\{\omega : \omega \models e\}) + 0 = c \times P(e)$. Therefore, $c = 1/P(e)$.

The conditional probability of formula h given evidence e is the measure, using μ_e , of the possible worlds in which h is true. That is,

$$\begin{aligned} P(h|e) &= \mu_e(\{\omega : \omega \models h\}) \\ &= \mu_e(\{\omega : \omega \models h \wedge e\}) + \mu_e(\{\omega : \omega \models h \wedge \neg e\}) \\ &= \frac{\mu(\{\omega : \omega \models h \wedge e\})}{P(e)} + 0 \\ &= \frac{P(h \wedge e)}{P(e)} \end{aligned}$$

The last form above is usually given as the definition of conditional probability.

For the rest of this chapter, assume that, if e is the evidence, $P(e) > 0$. We do not consider the problem of conditioning on propositions with zero probability (i.e., on sets of worlds with measure zero).

A **conditional probability distribution**, written $P(X|Y)$ where X and Y are variables or sets of variables, is a function of the variables: given a value $x \in \text{dom}(X)$ for X and a value $y \in \text{dom}(Y)$ for Y , it gives the value $P(X = x | Y = y)$, where the latter is the conditional probability of the propositions.

The definition of conditional probability lets us decompose a conjunction into a product of conditional probabilities:

Proposition 6.3. (Chain rule) *Conditional probabilities can be used to decompose conjunctions. For any propositions $a_1, \dots, a_n$:*

$$\begin{aligned} P(a_1 \wedge a_2 \wedge \dots \wedge a_n) &= P(a_1) \times \\ &\quad P(a_2|a_1) \times \\ &\quad P(a_3|a_1 \wedge a_2) \times \\ &\quad \vdots \\ &\quad P(a_n|a_1 \wedge \dots \wedge a_{n-1}) \\ &= \prod_{i=1}^n P(a_i|a_1 \wedge \dots \wedge a_{i-1}), \end{aligned}$$

where the right-hand side is assumed to be zero if any of the products are zero (even if some of them are undefined).

Note that any theorem about unconditional probabilities is a theorem about conditional probabilities if you add the same evidence to each probability. This is because the conditional probability measure is another probability measure.

Bayes' Rule

An agent must update its belief when it observes new evidence. A new piece of evidence is conjoined to the old evidence to form the complete set of evidence.

Background Knowledge and Observation

The difference between background knowledge and observation was described in Section 5.3.1 (page 174). When we use reasoning under uncertainty, the background model is described in terms of a probabilistic model, and the observations form evidence that must be conditioned on.

Within probability, there are two ways to state that a is true:

- The first is to state that the probability of a is 1 by writing $P(a) = 1$.
- The second is to condition on a , which involves using a on the right-hand side of the conditional bar, as in $P(\cdot | a)$.

The first method states that a is true in all possible worlds. The second says that the agent is only interested in worlds where a happens to be true.

Suppose an agent was told about a particular animal:

$$P(\text{flies}|\text{bird}) = 0.8,$$

$$P(\text{bird}|\text{emu}) = 1.0,$$

$$P(\text{flies}|\text{emu}) = 0.001.$$

If it determines the animal is an emu, it cannot add the statement $P(\text{emu}) = 1$. No probability distribution satisfies these four assertions. If emu were true in all possible worlds, it would not be the case that in 0.8 of the possible worlds, the individual flies. The agent, instead, must condition on the fact that the individual is an emu.

To build a probability model, a knowledge base designer must take some knowledge into consideration and build a probability model based on this knowledge. All subsequent knowledge acquired must be treated as observations that are conditioned on.

Suppose the agent's observations at some time are given by the proposition k . The agent's subsequent belief states can be modeled by either of the following:

- construct a probability theory, based on a measure μ , for the agent's belief before it had observed k and then condition on the evidence k conjoined with the subsequent evidence e , or
- construct a probability theory, based on a measure μ_k , which models the agent's beliefs after observing k , and then condition on subsequent evidence e .

All subsequent probabilities will be identical no matter which construction was used. Building μ_k directly is sometimes easier because the model does not have to cover the cases of when k is false. Sometimes, however, it is easier to build μ and condition on k .

What is important is that there is a coherent stage where the probability model is reasonable and where every subsequent observation is conditioned on.

Bayes' rule specifies how an agent should update its belief in a proposition based on a new piece of evidence.

Suppose an agent has a current belief in proposition h based on evidence k already observed, given by $P(h|k)$, and subsequently observes e . Its new belief in h is $P(h|e, k)$. Bayes' rule tells us how to update the agent's belief in hypothesis h as new evidence arrives.

Proposition 6.4. (Bayes' rule) *As long as $P(e|k) \neq 0$,*

$$P(h|e \wedge k) = \frac{P(e|h \wedge k) \times P(h|k)}{P(e|k)}.$$

This is often written with the background knowledge k implicit. In this case, if $P(e) \neq 0$, then

$$P(h|e) = \frac{P(e|h) \times P(h)}{P(e)}.$$

$P(e|h)$ is the **likelihood** of the hypothesis h ; $P(h)$ is the **prior** of the hypothesis h . Bayes' rule states that the posterior is proportional to the likelihood times the prior.

Proof. The commutativity of conjunction means that $h \wedge e$ is equivalent to $e \wedge h$, and so they have the same probability given k . Using the rule for multiplication in two different ways,

$$\begin{aligned} P(h \wedge e|k) &= P(h|e \wedge k) \times P(e|k) \\ &= P(e|h \wedge k) \times P(h|k). \end{aligned}$$

The theorem follows from dividing the right-hand sides by $P(e|k)$. □

Often, Bayes' rule is used to compare various hypotheses (different h_i 's), where it can be noticed that the denominator $P(e|k)$ is a constant that does not depend on the particular hypothesis. When comparing the relative posterior probabilities of hypotheses, we can ignore the denominator. To get the posterior probability, the denominator can be computed by reasoning by cases. If H is an exclusive and covering set of propositions representing all possible hypotheses, then

$$\begin{aligned} P(e|k) &= \sum_{h \in H} P(e \wedge h|k) \\ &= \sum_{h \in H} P(e|h \wedge k) \times P(h|k). \end{aligned}$$

Thus, the denominator of Bayes' rule is obtained by summing the numerators for all the hypotheses. When the hypothesis space is large, computing the denominator can be computationally difficult.

Generally, one of $P(e|h, k)$ or $P(h|e, k)$ is much easier to estimate than the other. This often occurs when you have a causal theory of a domain, and the predictions of different hypotheses – the $P(e|h_i, k)$ for each hypothesis h_i – can be derived from the domain theory.

Example 6.5 Suppose the diagnostic assistant is interested in the diagnosis of the light switch s_1 of Figure 1.8 (page 34). You would expect that the modeler is able to specify how the output of a switch is a function of the input, the switch position, and the status of the switch (whether it is working, shorted, installed upside-down, etc.). Bayes' rule lets an agent infer the status of the switch given the other information.

Example 6.6 Suppose an agent has information about the reliability of fire alarms. It may know how likely it is that an alarm will work if there is a fire. If it must know the probability that there is a fire, given that there is an alarm, it can use Bayes' rule:

$$P(\text{fire}|\text{alarm}) = \frac{P(\text{alarm}|\text{fire}) \times P(\text{fire})}{P(\text{alarm})}$$

where $P(\text{alarm}|\text{fire})$ is the probability that the alarm worked, assuming that there was a fire. It is a measure of the alarm's reliability. The expression $P(\text{fire})$ is the probability of a fire given no other information. It is a measure of how fire-prone the building is. $P(\text{alarm})$ is the probability of the alarm sounding, given no other information.

6.1.4 Expected Values

You can use the probabilities to give the expected value of any numerical random variable (i.e., one whose domain is a subset of the reals). A variable's expected value is the variable's weighted average value, where its value in each possible world is weighted by the measure of the possible world.

Suppose V is a random variable whose domain is numerical, and ω is a possible world. Define $V(\omega)$ to be the value v in the domain of V such that $V|_{\omega} = V = v$. That is, we are treating a random variable as a function on worlds.

The **expected value** of numerical variable V , written $E(V)$, is

$$E(V) = \sum_{\omega \in \Omega} V(\omega) \times \mu(\omega)$$

when finitely many worlds exist. When infinitely many worlds exist, we must integrate.

Example 6.7 If *number of broken switches* is an integer-valued random variable,

$$E(\text{number of broken switches})$$

— — —
would give the expected number of broken switches. If the world acted according to the probability model, this would give the long-run average number of broken switches.

In a manner analogous to the semantic definition of conditional probability (page 226), the **conditional expected value** of variable X conditioned on evidence e , written $E(V|e)$, is

$$E(V|e) = \sum_{\omega \in \Omega} V(\omega) \times \mu_e(\omega).$$

Thus,

$$\begin{aligned} E(V|e) &= \frac{1}{P(e)} \sum_{\omega|e} V(\omega) \times P(\omega) \\ &= \sum_{\omega \in \Omega} V(\omega) \times P(\omega|e). \end{aligned}$$

Example 6.8 The expected number of broken switches given that light l_1 is not lit is given by

$$E(\text{number of broken switches} | \neg \text{lit}(l_1)).$$
 This is obtained by averaging the number of broken switches over all of the worlds in which light l_1 is not lit.

6.1.5 Information Theory

Probability forms the basis of information theory. In this section, we give a brief overview of information theory.

A **bit** is a binary digit. Because a bit has two possible values, it can be used to distinguish two items. Often the two values are written as 0 and 1, but they can be any two different values.

Two bits can distinguish four items, each associated with either 00, 01, 10, or 11. Similarly, three bits can distinguish eight items. In general, n bits can distinguish 2^n items. Thus, we can distinguish n items with $\log_2 n$ bits. It may be surprising, but we can do better than this by taking probabilities into account.

Example 6.9 Suppose you want to design a code to distinguish the elements of the set $\{a, b, c, d\}$, with $P(a)=\frac{1}{2}$, $P(b)=\frac{1}{4}$, $P(c)=\frac{1}{8}$, and $P(d)=\frac{1}{8}$. Consider the following code:

a	0	c	110
b	10	d	111

This code sometimes uses 1 bit and sometimes uses 3 bits. On average, it uses

$$\begin{aligned} &P(a) \times 1 + P(b) \times 2 + P(c) \times 3 + P(d) \times 3 \\ &= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{3}{8} = 1\frac{3}{4} \text{ bits.} \end{aligned}$$

For example, the string *aacabbda* with 8 characters has code 00110010101110, which uses 14 bits.

With this code, $-\log_2 P(a) = 1$ bit is required to distinguish a from the other symbols. To distinguish b , you must have $-\log_2 P(b) = 2$ bits. To distinguish c , you must have $-\log_2 P(c) = 3$ bits.

It is possible to build a code that, to identify x , requires $\log_2 P(x)$ bits (or the integer greater than this, if x is the only thing being transmitted). Suppose there is a sequence of symbols we want to transmit or store and we know the probability distribution over the symbols. A symbol x with probability $P(x)$ uses $\log_2 P(x)$ bits. To transmit a sequence, each symbol requires, on average,

$$\sum_x -P(x) \times \log_2 P(x)$$

bits to send it. This value just depends on the probability distribution of the symbols. This is called the **information content** or **entropy** of the distribution.

Analogously to conditioning in probability, the expected number of bits it takes to describe a distribution given evidence e is

$$I(e) = \sum_x -P(x|e) \times \log_2 P(x|e).$$

If a test exists that can distinguish the cases where a is true from the cases where a is false, the **information gain** from this test is

$$I(\text{true}) - (P(a) \times I(a) + P(\neg a) \times I(\neg a)),$$

where $I(\text{true})$ is the expected number of bits needed before the test and $P(a)$

$I(a) + P(\neg a) \times I(\neg a)$ is the expected number of bits after the test. ×

In later sections, we use the notion of information for a number of tasks:

- In diagnosis, an agent can choose a test that provides the most information.
- In decision-tree learning (page 298), information theory provides a useful criterion for choosing which property to split on: split on the property that provides the greatest information gain. The elements it must distinguish are the different values in the target concept, and the probabilities are obtained from the proportion of each value in the training set remaining at each node.
- In Bayesian learning (page 334), information theory provides a basis for deciding which is the best model given some data.

6.2 Independence

The axioms of probability are very weak and provide few constraints on allowable conditional probabilities. For example, if there are n binary variables, there are $2^n - 1$ numbers to be assigned to give a complete probability distribution from which arbitrary conditional probabilities can be derived. To determine any probability, you may have to start with an enormous database of conditional probabilities or of probabilities of possible worlds.

Two main approaches are used to overcome the need for so many numbers:

Independence Assume that the knowledge of the truth of one proposition, Y , does not affect the agent's belief in another proposition, X , in the context of other propositions Z . We say that X is independent of Y given Z . This is defined below.

Maximum entropy or random worlds Given no other knowledge, assume that everything is as random as possible. That is, the probabilities are distributed as uniformly as possible consistent with the available information.

We consider here in detail the first of these (but see the box on page 234).

As long as the value of $P(h|e)$ is not 0 or 1, the value of $P(h|e)$ does not constrain the value of $P(h|f \wedge e)$. This latter probability could have any value in the range $[0, 1]$: It is 1 when f implies h , and it is 0 if f implies $\neg h$. $\neg$

As far as probability theory is concerned, there is no reason why the name of the Queen of Canada should not be as significant as a light switch's position in determining whether the light is on. Knowledge of the domain, however, may tell us that it is irrelevant.

In this section we present a representation that allows us to model the structure of the world where relevant propositions are local and where not-directly-relevant variables can be ignored when probabilities are specified. This structure can be exploited for efficient reasoning.

A common kind of qualitative knowledge is of the form $P(h|e) = P(h|f \wedge e)$. This equality says that f is irrelevant to the probability of h given e . For example, the fact that Elizabeth is the Queen of Canada is irrelevant to the probability that w_2 is live given that switch s_1 is down. This idea can apply to random variables, as in the following definition:

Random variable X is **conditionally independent** of random variable Y **given** random variable Z if for all $x \in \text{dom}(X)$, for all $y \in \text{dom}(Y)$, for all $y^t \in \text{dom}(Y)$, and for all $z \in \text{dom}(Z)$, such that $P(Y = y \wedge Z = z) > 0$ and $P(Y = y^t \wedge Z = z) > 0$,

$$P(X = x | Y = y \wedge Z = z) = P(X = x | Y = y^t \wedge Z = z).$$

That is, given a value of Z , knowing Y 's value does not affect your belief in the value of X .

Proposition 6.5. *The following four statements are equivalent, as long as the conditional probabilities are well defined:*

1. X is conditionally independent of Y given Z .
2. Y is conditionally independent of X given Z .
3. $P(X|Y, Z) = P(X|Z)$. That is, in the context that you are given a value for Z , if you were given a value for Y , you would have the same belief in X as if you were not given a value for Y .
4. $P(X, Y|Z) = P(X|Z)P(Y|Z)$.

Reducing the

The distinction between allowing representations of independence and using maximum entropy or random worlds highlights an important difference between views of a knowledge representation:

- The first view is that a knowledge representation provides a high-level modeling language that lets us model a domain in a reasonably natural way. According to this view, it is expected that knowledge representation designers prescribe how to use the knowledge representation language. It is expected that they provide a user manual on how to describe domains of interest.
- The second view is that a knowledge representation should allow someone to add whatever knowledge they may have about a domain. The knowledge representation should fill in the rest in a commonsense manner. According to this view, it is unreasonable for a knowledge representation designer to specify how particular knowledge should be encoded.

Judging a knowledge representation by the wrong criteria does not result in a fair assessment.

A belief network is a representation for a particular independence among variables. Belief networks should be viewed as a modeling language. Many domains can be concisely and naturally represented by exploiting the independencies that belief networks compactly represent. This does not mean that we can just throw in lots of facts (or probabilities) and expect a reasonable answer. One must think about a domain and consider exactly what variables are involved and what the dependencies are among the variables. When judged by this criterion, belief networks form a useful representation scheme.

Once the network structure and the domains of the variables for a belief network are defined, exactly which numbers are required (the conditional probabilities) are prescribed. The user cannot simply add arbitrary conditional probabilities but must follow the network's structure. If the numbers that are required of a belief network are provided and are locally consistent, the whole network will be consistent.

In contrast, the maximum entropy or random worlds approaches infer the most random worlds that are consistent with a probabilistic knowledge base. They form a probabilistic knowledge representation of the second type. For the random worlds approach, any numbers that happen to be available can be added and used. However, if you allow someone to add arbitrary probabilities, it is easy for the knowledge to be inconsistent with the axioms of probability. Moreover, it is difficult to justify an answer as correct if the assumptions are not made explicit.

The proof is left as an exercise. [See Exercise 1 (page 275).]

Variables X and Y are **unconditionally independent** if $P(X, Y) = P(X)P(Y)$, that is, if they are conditionally independent given no observations. Note that X and Y being unconditionally independent does not imply they are conditionally independent given some other information Z .

Conditional independence is a useful assumption about a domain that is often natural to assess and can be exploited to give a useful representation.

6.3 Belief Networks

The notion of conditional independence can be used to give a concise representation of many domains. The idea is that, given a random variable X , a small set of variables may exist that directly affect the variable's value in the sense that X is conditionally independent of other variables given values for the directly affecting variables. The set of locally affecting variables is called the **Markov blanket**. This locality is what is exploited in a belief network. A **belief network** is a directed model of conditional dependence among a set of random variables. The precise statement of conditional independence in a belief network takes into account the directionality.

To define a belief network, start with a set of random variables that represent all of the features of the model. Suppose these variables are $X_1, \dots, X_n$. Next, select a total ordering of the variables, $X_1, \dots, X_n$.

The chain rule (Proposition 6.3 (page 227)) shows how to decompose a conjunction into conditional probabilities:

$$\begin{aligned} P(X_1 = v_1 \wedge X_2 = v_2 \wedge \dots \wedge X_n = v_n) \\ = \prod_{i=1}^n P(X_i = v_i | X_1 = v_1 \wedge \dots \wedge X_{i-1} = v_{i-1}). \end{aligned}$$

Or, in terms of random variables and probability distributions,

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i | X_1, \dots, X_{i-1}).$$

Define the **parents** of random variable X_i , written $parents(X_i)$, to be a minimal set of predecessors of X_i in the total ordering such that the other predecessors of X_i are conditionally independent of X_i given $parents(X_i)$. That is, $parents(X_i) \subseteq \{X_1, \dots, X_{i-1}\}$ such that

$$P(X_i | X_{i-1} \dots X_1) = P(X_i | parents(X_i)).$$

If more than one minimal set exists, any minimal set can be chosen to be the parents. There can be more than one minimal set only when some of the predecessors are deterministic functions of others.

We can put the chain rule and the definition of parents together, giving

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i | parents(X_i)).$$

The probability over all of the variables, $P(X_1, X_2, \dots, X_n)$, is called the **joint probability distribution**. A belief network defines a **factorization** of the joint probability distribution, where the conditional probabilities form factors that are multiplied together.

A **belief network**, also called a **Bayesian network**, is an acyclic directed graph (DAG), where the nodes are random variables. There is an arc from each element of $parents(X_i)$ into X_i . Associated with the belief network is a set of conditional probability distributions – the conditional probability of each variable given its parents (which includes the prior probabilities of those variables with no parents).

Thus, a belief network consists of

- a DAG, where each node is labeled by a random variable;
- a domain for each random variable; and
- a set of conditional probability distributions giving $P(X_i | parents(X_i))$ for each variable X_i .

A belief network is acyclic by construction. The way the chain rule decomposes the conjunction gives the ordering. A variable can have only predecessors as parents. Different decompositions can result in different belief networks.

Example 6.10 Suppose we want to use the diagnostic assistant to diagnose whether there is a fire in a building based on noisy sensor information and possibly conflicting explanations of what could be going on. The agent receives a report about whether everyone is leaving the building. Suppose the report sensor is noisy: It sometimes reports leaving when there is no exodus (a false positive), and it sometimes does not report when everyone is leaving (a false negative). Suppose the fire alarm going off can cause the leaving, but this is not a deterministic relationship. Either tampering or fire could affect the alarm. Fire also causes smoke to rise from the building.

Suppose we use the following variables, all of which are Boolean, in the following order:

- *Tampering* is true when there is tampering with the alarm.
- *Fire* is true when there is a fire.
- *Alarm* is true when the alarm sounds.
- *Smoke* is true when there is smoke.
- *Leaving* is true if there are many people leaving the building at once.
- *Report* is true if there is a report given by someone of people leaving. *Report* is false if there is no report of leaving.

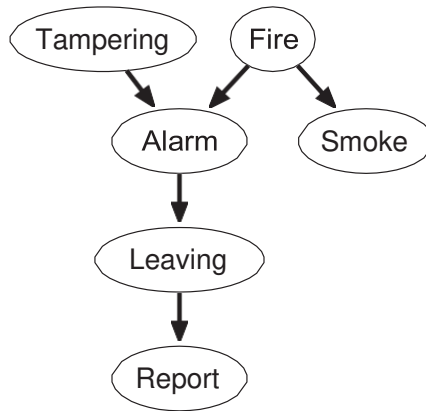


Figure 6.1: Belief network for report of leaving of Example [6.10](#)

The variable *Report* denotes the sensor report that people are leaving. This information is unreliable because the person issuing such a report could be playing a practical joke, or no one who could have given such a report may have been paying attention. This variable is introduced to allow conditioning on unreliable sensor data. The agent knows what the sensor reports, but it only has unreliable evidence about people leaving the building.

As part of the domain, assume the following conditional independencies:

- *Fire* is conditionally independent of *Tampering* (given no other information).
- *Alarm* depends on both *Fire* and *Tampering*. That is, we are making no independence assumptions about how *Alarm* depends on its predecessors given this variable ordering.
- *Smoke* depends only on *Fire* and is conditionally independent of *Tampering* and *Alarm* given whether there is a *Fire*.
- *Leaving* only depends on *Alarm* and not directly on *Fire* or *Tampering* or *Smoke*. That is, *Leaving* is conditionally independent of the other variables given *Alarm*.
- *Report* only directly depends on *Leaving*.

The belief network of Figure [6.1](#) expresses these dependencies. This network represents the factorization

$$\begin{aligned}
 &P(\textit{Tampering}, \textit{Fire}, \textit{Alarm}, \textit{Smoke}, \textit{Leaving}, \textit{Report}) \\
 &= P(\textit{Tampering}) \times P(\textit{Fire}) \times P(\textit{Alarm}|\textit{Tampering}, \textit{Fire}) \\
 &\quad \times P(\textit{Smoke}|\textit{Fire}) \times P(\textit{Leaving}|\textit{Alarm}) \times P(\textit{Report}|\textit{Leaving}).
 \end{aligned}$$

We also must define the domain of each variable. Assume that the variables are Boolean; that is, they have domain *true, false*. We use the lower-case variant of the variable to represent the true value and use negation for the false value. Thus, for example, *Tampering = true* is written as *tampering*, and *Tampering = false* is written as $\neg$ *tampering*.

The examples that follow assume the following conditional probabilities:

$$\begin{aligned}
 P(\textit{tampering}) &= 0.02 \\
 P(\textit{fire}) &= 0.01 \\
 P(\textit{alarm} | \textit{fire} \wedge \textit{tampering}) &= 0.5 \\
 P(\textit{alarm} | \textit{fire} \wedge \neg \textit{tampering}) &= 0.99 \\
 P(\textit{alarm} | \neg \textit{fire} \wedge \textit{tampering}) &= 0.85 \\
 P(\textit{alarm} | \neg \textit{fire} \wedge \neg \textit{tampering}) &= 0.0001 \\
 P(\textit{smoke} | \textit{fire}) &= 0.9 \\
 P(\textit{smoke} | \neg \textit{fire}) &= 0.01 \\
 P(\textit{leaving} | \textit{alarm}) &= 0.88 \\
 P(\textit{leaving} | \neg \textit{alarm}) &= 0.001 \\
 P(\textit{report} | \textit{leaving}) &= 0.75 \\
 P(\textit{report} | \neg \textit{leaving}) &= 0.01
 \end{aligned}$$

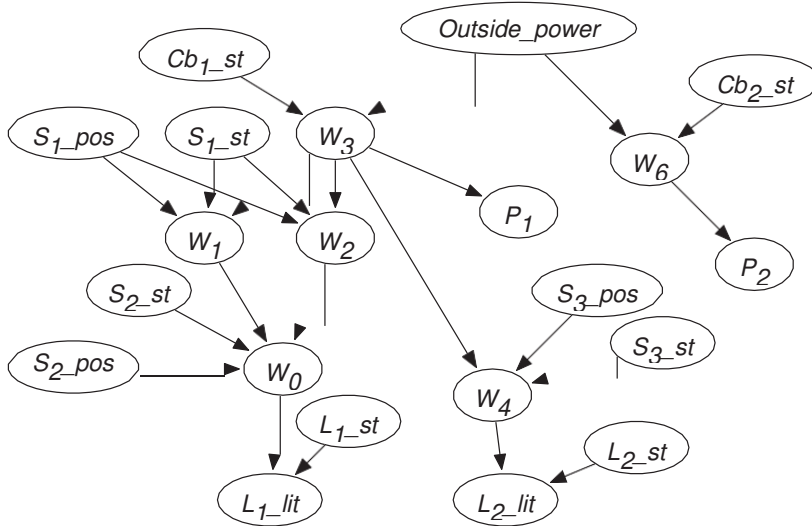
Example 6.11 Consider the wiring example of Figure 1.8 (page 34). Suppose we decide to have variables for whether lights are lit, for the switch positions, for whether lights and switches are faulty or not, and for whether there is power in the wires. The variables are defined in Figure 6.2.

Let's select an ordering where the causes of a variable are before the variable in the ordering. For example, the variable for whether a light is lit comes after variables for whether the light is working and whether there is power coming into the light.

Whether light l_1 is lit depends only on whether there is power in wire w_0 and whether light l_1 is working properly. Other variables, such as the position of switch s_1 , whether light l_2 is lit, or who is the Queen of Canada, are irrelevant. Thus, the parents of $L_1.lit$ are W_0 and $L_1.st$.

Consider variable W_0 , which represents whether there is power in wire w_0 . If we knew whether there was power in wires w_1 and w_2 , and we knew the position of switch s_2 and whether the switch was working properly, the value of the other variables (other than $L_1.lit$) would not affect our belief in whether there is power in wire w_0 . Thus, the parents of W_0 should be $S_2.Pos$, $S_2.st$, W_1 , and W_2 .

Figure 6.2 shows the resulting belief network after the independence of each variable has been considered. The belief network also contains the domains of the variables, as given in the figure, and conditional probabilities of each variable given its parents.



- For each wire w_i , there is a random variable, W_i , with domain $\{live, dead\}$, which denotes whether there is power in wire w_i . $W_i=live$ means wire w_i has power. $W_i=dead$ means there is no power in wire w_i .
- *Outside power* with domain $\{live, dead\}$ denotes whether there is power coming into the building.
- For each switch s_i , variable S_i_pos denotes the position of s_i . It has domain $\{up, down\}$.
- For each switch s_i , variable S_i_st denotes the state of switch s_i . It has domain $\{ok, upside\_down, short, intermittent, broken\}$. $S_i\_st=ok$ means switch s_i is working normally. $S_i\_st=upside\_down$ means switch s_i is installed upside-down. $S_i\_st=short$ means switch s_i is shorted and acting as a wire. $S_i\_st=broken$ means switch s_i is broken and does not allow electricity to flow.
- For each circuit breaker cb_i , variable Cb_i_st has domain $\{on, off\}$. $Cb_i\_st=on$ means power can flow through cb_i and $Cb_i\_st=off$ means that power can not flow through cb_i .
- For each light l_i , variable L_i_st with domain $\{ok, intermittent, broken\}$ denotes the state of the light. $L_i\_st=ok$ means light l_i will light if powered, $L_i\_st=intermittent$ means light l_i intermittently lights if powered, and $L_i\_st=broken$ means light l_i does not work.

Figure 6.2: Belief network for the electrical domain of Figure 1.8

For the variable W_1 , the following conditional probabilities must be specified:

$$P(W_1 = \text{live} | S_1 \text{ pos} = \text{up} \wedge S_1 \text{ st} = \text{ok} \wedge W_3 = \text{live})$$

$$P(W_1 = \text{live} | S_1 \text{ pos} = \text{up} \wedge S_1 \text{ st} = \text{ok} \wedge W_3 = \text{dead})$$

$$P(W_1 = \text{live} | S_1 \text{ pos} = \text{up} \wedge S_1 \text{ st} = \text{upside down} \wedge W_3 = \text{live})$$

$$\vdots$$

$$P(W_1 = \text{live} | S_1 \text{ pos} = \text{down} \wedge S_1 \text{ st} = \text{broken} \wedge W_3 = \text{dead}).$$

There are two values for $S_1 \text{ pos}$, five values for $S_1 \text{ ok}$, and two values for W_3 , so there are $2 \times 5 \times 2 = 20$ different cases where a value for $W_1 = \text{live}$ must be specified. As far as probability theory is concerned, the probability for $W_1 = \text{live}$ for these 20 cases could be assigned arbitrarily. Of course, knowledge of the domain constrains what values make sense. The values for $W_1 = \text{dead}$ can be computed from the values for $W_1 = \text{live}$ for each of these cases.

Because the variable $S_1 \text{ st}$ has no parents, it requires a prior distribution, which can be specified as the probabilities for all but one of the values; the remaining value can be derived from the constraint that all of the probabilities sum to 1. Thus, to specify the distribution of $S_1 \text{ st}$, four of the following five probabilities must be specified:

$$P(S_1 \text{ st} = \text{ok})$$

$$P(S_1 \text{ st} = \text{upside down})$$

$$P(S_1 \text{ st} = \text{short})$$

$$P(S_1 \text{ st} = \text{intermittent})$$

$$P(S_1 \text{ st} = \text{broken})$$

The other variables are represented analogously.

A belief network is a graphical representation of conditional independence. The independence allows us to depict direct effects within the graph and prescribes which probabilities must be specified. Arbitrary posterior probabilities can be derived from the network.

The independence assumption embedded in a belief network is as follows: *Each random variable is conditionally independent of its non-descendants given its parents.* That is, if X is a random variable with parents $Y_1, \dots, Y_n$, all random variables that are not descendants of X are conditionally independent of X given $Y_1, \dots, Y_n$:

$$P(X | Y_1, \dots, Y_n, R) = P(X | Y_1, \dots, Y_n),$$

if R does not involve a descendant of X . For this definition, we include X as a descendant of itself. The right-hand side of this equation is the form of the probabilities that are specified as part of the belief network. R may involve ancestors of X and other nodes as long as they are not descendants of X . The independence assumption states that all of the influence of non-descendant variables is captured by knowing the value of X 's parents.

Often, we refer to just the labeled DAG as a belief network. When this is done, it is important to remember that a domain for each variable and a set of conditional probability distributions are also part of the network.

The number of probabilities that must be specified for each variable is exponential in the number of parents of the variable. The independence assumption

Belief Networks and Causality

Belief networks have often been called **causal networks** and have been claimed to be a good representation of causality. Recall (page [204](#)) that a causal model predicts the result of interventions. Suppose you have in mind a causal model of a domain, where the domain is specified in terms of a set of random variables. For each pair of random variables X_1 and X_2 , if a direct causal connection exists from X_1 to X_2 (i.e., intervening to change X_1 in some context of other variables affects X_2 and this cannot be modeled by having some intervening variable), add an arc from X_1 to X_2 . You would expect that the causal model would obey the independence assumption of the belief network. Thus, all of the conclusions of the belief network would be valid.

You would also expect such a graph to be acyclic; you do not want something eventually causing itself. This assumption is reasonable if you consider that the random variables represent particular events rather than event types. For example, consider a causal chain that “being stressed” causes you to “work inefficiently,” which, in turn, causes you to “be stressed.” To break the apparent cycle, we can represent “being stressed” at different stages as different random variables that refer to different times. Being stressed in the past causes you to not work well at the moment which causes you to be stressed in the future. The variables should satisfy the **clarity principle** (page [114](#)) and have a well-defined meaning. The variables should not be seen as event types. The belief network itself has nothing to say about causation, and it can represent non-causal independence, but it seems particularly appropriate when there is causality in a domain. Adding arcs that represent local causality tends to produce a small belief network. The belief network of Figure [6.2](#) (page [239](#)) shows how this can be done for a simple domain.

A causal network models **interventions**. If someone were to artificially force a variable to have a particular value, the variable’s descendants – but no other nodes – would be affected.

Finally, you can see how the causality in belief networks relates to the causal and evidential reasoning discussed in Section [5.7](#) (page [204](#)). A causal belief network can be seen as a way of axiomatizing in a causal direction. Reasoning in belief networks corresponds to abducing to causes and then predicting from these. A direct mapping exists between the logic-based abductive view discussed in Section [5.7](#) (page [204](#)) and belief networks: Belief networks can be modeled as logic programs with probabilities over possible hypotheses. This is described in Section [14.3](#) (page [611](#)).

is useful insofar as the number of variables that directly affect another variable is small. You should order the variables so that nodes have as few parents as possible.

Note the restriction “each random variable is conditionally independent of its non-descendants given its parents” in the definition of the independence encoded in a belief network (page 240). If R contains a descendant of variable X , the independence assumption is not directly applicable.

Example 6.12 In Figure 6.2 (page 239), variables $S_3 pos$, $S_3 st$, and W_3 are the parents of variable W_4 . If you know the values of $S_3 pos$, $S_3 st$, and W_3 , knowing whether or not l_1 is lit or knowing the value of $Cb_1 st$ will not affect your belief in whether there is power in wire w_4 . However, even if you knew the values of $S_3 pos$, $S_3 st$, and W_3 , learning whether l_2 is lit potentially changes your belief in whether there is power in wire w_1 . The independence assumption is not directly applicable.

The variable $S_1 pos$ has no parents. Thus, the independence embedded in the belief network specifies that $P(S_1 pos = up | A) = P(S_1 pos = up)$ for any A that does not involve a descendant of $S_1 pos$. If A includes a descendant of $S_1 pos = up$ – for example, if A is $S_2 pos = up \wedge l_1 lit = true$ – the independence assumption cannot be directly applied.

A belief network specifies a joint probability distribution from which arbitrary conditional probabilities can be derived. A network can be queried by asking for the conditional probability of any variables conditioned on the values of any other variables. This is typically done by providing observations on some variables and querying another variable.

Example 6.13 Consider Example 6.10 (page 236). The prior probabilities (with no evidence) of each variable can be computed using the methods of the next section. The following conditional probabilities follow from the model of Example 6.10, to about three decimal places:

$$P(tampering) = 0.02$$

$$P(fire) = 0.01$$

$$P(report) = 0.028$$

$$P(smoke) = 0.0189$$

Observing the report gives the following:

$$P(tampering | report) = 0.399$$

$$P(fire | report) = 0.2305$$

$$P(smoke | report) = 0.215$$

As expected, the probability of both *tampering* and *fire* are increased by the report. Because *fire* is increased, so is the probability of *smoke*.

Suppose instead that *smoke* were observed:

$$P(tampering | smoke) = 0.02$$

$$P(fire | smoke) = 0.476$$

$$P(report | smoke) = 0.320$$

Note that the probability of *tampering* is not affected by observing *smoke*; however, the probabilities of *report* and *fire* are increased.

Suppose that both *report* and *smoke* were observed:

$$P(\text{tampering} | \text{report} \wedge \text{smoke}) = 0.0284$$

$$P(\text{fire} | \text{report} \wedge \text{smoke}) = 0.964$$

Observing both makes *fire* even more likely. However, in the context of the *report*, the presence of *smoke* makes *tampering* less likely. This is because the *report* is **explained away** by *fire*, which is now more likely.

Suppose instead that *report*, but not *smoke*, was observed:

$$P(\text{tampering} | \text{report} \wedge \neg \text{smoke}) = 0.501$$

$$P(\text{fire} | \text{report} \wedge \neg \text{smoke}) = 0.0294$$

In the context of the *report*, *fire* becomes much less likely and so the probability of *tampering* increases to explain the *report*.

This example illustrates how the belief net independence assumption gives commonsense conclusions and also demonstrates how explaining away is a consequence of the independence assumption of a belief network.

This network can be used in a number of ways:

- By conditioning on the knowledge that the switches and circuit breakers are ok, and on the values of the outside power and the position of the switches, this network can simulate how the lighting should work.
- Given values of the outside power and the position of the switches, the network can infer the likelihood of any outcome – for example, how likely it is that l_1 is lit.
- Given values for the switches and whether the lights are lit, the posterior probability that each switch or circuit breaker is in any particular state can be inferred.
- Given some observations, the network can be used to reason backward to determine the most likely position of switches.
- Given some switch positions, some outputs, and some intermediate values, the network can be used to determine the probability of any other variable in the network.

6.3.1 Constructing Belief Networks

To represent a domain in a belief network, the designer of a network must consider the following questions:

- What are the relevant variables? In particular, the designer must consider
 - what the agent may observe in the domain. Each feature that can be observed should be a variable, because the agent must be able to condition on all of its observations.
 - what information the agent is interested in knowing the probability of, given the observations. Each of these features should be made into a variable that can be queried.

- other **hidden variables** or **latent variables** that will not be observed or queried but that make the model simpler. These variables either account for dependencies or reduce the size of the specification of the conditional probabilities.
- What values should these variables take? This involves considering the level of detail at which the agent should reason to answer the sorts of queries that will be encountered.
- For each variable, the designer should specify what it means to take each value in its domain. What must be true in the world for a variable to have a particular value should satisfy the clarity principle (page 114). It is a good idea to explicitly document the meaning of all variables and their possible values. The only time the designer may not want to do this is when a hidden variable exists whose values the agent will want to learn from data [see Section 11.2.2 (page 460)].
- What is the relationship between the variables? This should be expressed in terms of local influence and be modeled using the parent relation.
- How does the distribution of a variable depend on the variables that locally influence it (its parents)? This is expressed in terms of the conditional probability distributions.

Example 6.14 Suppose you want the diagnostic assistant to be able to reason about the possible causes of a patient's wheezing and coughing, as in Example 5.30 (page 201).

- The agent can observe coughing, wheezing, and fever and can ask whether the patient smokes. There are thus variables for these.
- The agent may want to know about other symptoms of the patient and the prognosis of various possible treatments; if so, these should also be variables. (Although they are not used in this example).
- There are variables that are useful to predict the outcomes of patients. The medical community has named many of these and characterized their symptoms. Here we will use the variables *Bronchitis* and *Influenza*.
- Now consider what the variables directly depend on. Whether patients wheeze depends on whether they have bronchitis. Whether they cough depends on whether they have bronchitis. Whether patients have bronchitis depends on whether they have influenza and whether they smoke. Whether they have fever depends on whether they have influenza. Figure 6.3 depicts these dependencies.
- Choosing the values for the variables involves considering the level of detail at which to reason. You could encode the severity of each of the diseases and symptoms as values for the variables. You could, for example, use the values *severe*, *moderate*, *mild*, or *absent* for the *Wheezing* variable. You could even model the disease at a lower level of abstraction, for example, by representing all subtypes of the diseases. For ease of exposition, we will model the domain at a very abstract level, only considering the presence or absence of symptoms and diseases. Each of the variables will be

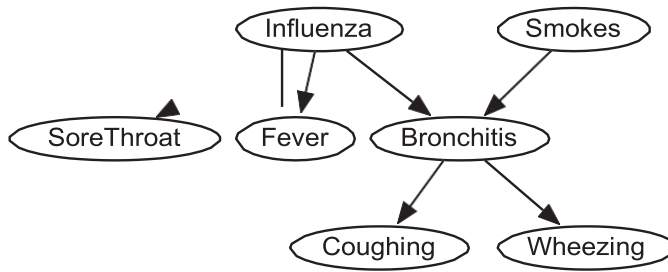


Figure 6.3: Belief network for Example 6.14

Boolean, with domain $\{true, false\}$, representing the presence or absence of the associated disease or symptom.

You assess how each variable depends on its parents, which is done by specifying the conditional probabilities of each variable given its parents:

$$\begin{aligned}
 P(\text{influenza}) &= 0.05 \\
 P(\text{smokes}) &= 0.2 \\
 P(\text{soreThroat}|\text{influenza}) &= 0.3 \\
 P(\text{soreThroat}|\neg\text{influenza}) &= 0.001 \\
 P(\text{fever}|\text{influenza}) &= 0.9 \\
 P(\text{fever}|\neg\text{influenza}) &= 0.05 \\
 P(\text{bronchitis}|\text{influenza} \wedge \text{smokes}) &= 0.99 \\
 P(\text{bronchitis}|\text{influenza} \wedge \neg\text{smokes}) &= 0.9 \\
 P(\text{bronchitis}|\neg\text{influenza} \wedge \text{smokes}) &= 0.7 \\
 P(\text{bronchitis}|\neg\text{influenza} \wedge \neg\text{smokes}) &= 0.0001 \\
 P(\text{coughing}|\text{bronchitis}) &= 0.8 \\
 P(\text{coughing}|\neg\text{bronchitis}) &= 0.07 \\
 P(\text{wheezing}|\text{bronchitis}) &= 0.6 \\
 P(\text{wheezing}|\neg\text{bronchitis}) &= 0.001
 \end{aligned}$$

The process of **diagnosis** is carried out by conditioning on the observed symptoms and deriving posterior probabilities of the faults or diseases.

This example also illustrates another example of explaining away and the preference for simpler diagnoses over complex ones.

Before any observations, we can compute (see the next section), to a few significant digits, $P(\text{smokes}) = 0.2$, $P(\text{influenza}) = 0.05$, and $P(\text{bronchitis}) = 0.18$. Once *wheezing* is observed, all three become more likely: $P(\text{smokes}|\text{wheezing}) = 0.79$, $P(\text{influenza}|\text{wheezing}) = 0.25$, and $P(\text{bronchitis}|\text{wheezing}) = 0.992$.

Suppose *wheezing* $\wedge$ *fever* is observed: $P(\text{smokes}|\text{wheezing} \wedge \text{fever}) = 0.32$, $P(\text{influenza}|\text{wheezing} \wedge \text{fever}) = 0.86$, and $P(\text{bronchitis}|\text{wheezing} \wedge \text{fever}) = 0.998$. Notice how, as in Example 5.30 (page 201), when *fever* is observed, *influenza* is indicated, and so *smokes* is **explained away**.

Example 6.15 Consider the belief network depicted in Figure 6.2 (page 239). Note the independence assumption embedded in this model: The DAG specifies that the lights, switches, and circuit breakers break independently. To model dependencies among how the switches break, you can add more arcs and perhaps more nodes. For example, if lights do not break independently because they come from the same batch, you can add an extra node that conveys the dependency. You would add a node that represents whether the lights come from a good or bad batch, which is made a parent of $L_1 st$ and $L_2 st$. The lights can now break dependently. When you have evidence that one light is broken, the probability that the batch is bad may increase and thus make it more likely that the other light is bad. If you are not sure whether the lights are indeed from the same batch, you can add a node representing this, too. The important point is that the belief network provides a specification of independence that lets us model dependencies in a natural and direct manner.

The model implies that no possibility exists of there being shorts in the wires or that the house is wired differently from the diagram. In particular, it implies that w_0 cannot be shorted to w_4 so that wire w_0 can get power from wire w_4 . You could add extra dependencies that let each possible short be modeled. An alternative is to add an extra node that indicates that the model is appropriate. Arcs from this node would lead to each variable representing power in a wire and to each light. When the model is appropriate, you can use the probabilities of Example 6.11 (page 238). When the model is inappropriate, you can, for example, specify that each wire and light works at random. When there are weird observations that do not fit in with the original model – they are impossible or extremely unlikely given the model – the probability that the model is inappropriate will increase.

Example 6.16 Suppose we want to develop a **help system** to determine what help page a user is interested in based on the keywords they give in a query to a help system.

The system will observe the words that the user gives. Suppose that we do not want to model the sentence structure, but assume that the set of words will be sufficient to determine the help page. The user can give multiple words. One way to represent this is to have a Boolean variable for each word. Thus, there will be nodes labeled “able”, “absent”, “add”, ..., “zoom” that have the value *true* when the user uses that word in a query and *false* when the user does not use that word.

We are interested in which help page the user wants. Suppose that the user is interested in one and only one help page. Thus, it seems reasonable to have a node H with domain the set of all help pages, $\{h_1, \dots, h_k\}$.

One way this can be represented is as a **naive Bayesian classifier**. A naive Bayesian classifier is a belief network that has a single node – the class – that directly influences the other variables, and the other variables are independent given the class. Figure 6.4 shows a naive Bayesian classifier for the help system where H , the help page the user is interested in, is the class, and the other nodes represent the words used in the query. In this network, the words used in a

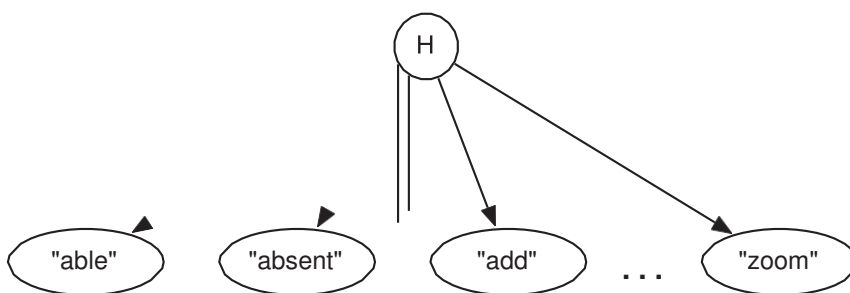


Figure 6.4: Naive belief network for Example 6.16

query depend on the help page the user is interested in, and the words are conditionally independent of each other given the help page.

This network requires $P(h_i)$ for each help page h_i , which specifies how likely it is that a user would want this help page given no information. This information could be obtained from how likely it is that users have particular problems they need help with. This network assumes the user is interested in exactly one help page, and so $\sum_i P(h_i) = 1$.

The network also requires, for each word w_j and for each help page h_i , the probability $P(w_j|h_i)$. These may seem more difficult to acquire but there are a few heuristics we can use. The average of these values should be the average number of words in a query divided by the total number of words. We would expect words that appear in the help page to be more likely to be used when asking for that help page than words not in the help page. There may also be keywords associated with the page that may be more likely to be used. There may also be some words that are just used more, independently of the help page the user is interested in. Example 7.13 (page 312) shows how the probabilities of this network can be learned from experience.

To condition on the set of words in a query, the words that appear in the query are observed to be true and the words that are not in the query are observed to be false. For example, if the help text was “the zoom is absent”, the words “the”, “zoom”, “is”, and “absent” would be observed to be true, and the other words would be observed to be false. The posterior for H can then be computed and the most likely few help topics can be shown to the user.

Some words, such as “the” and “is”, may not be useful in that they have the same conditional probability for each help topic and so, perhaps, would be omitted from the model. Some words that may not be expected in a query could also be omitted from the model.

Note that the conditioning was also on the words that were not in the query. For example, if page h_{73} was about printing problems, we may expect that everyone who wanted page h_{73} would use the word “print”. The non-existence of the word “print” in a query is strong evidence that the user did not want page h_{73} .

The independence of the words given the help page is a very strong assumption. It probably does not apply to words like “not”, where what “not”

is associated with is very important. If people are asking sentences, the words would not be conditionally independent of each other given the help they need, because the probability of a word depends on its context in the sentence. There may even be words that are complementary, in which case you would expect users to use one and not the other (e.g., “type” and “write”) and words you would expect to be used together (e.g., “go” and “to”); both of these cases violate the independence assumption. It is an empirical question as to how much violating the assumptions hurts the usefulness of the system.

6.4 Probabilistic Inference

The most common probabilistic inference task is to compute the posterior distribution of a query variable given some evidence. Unfortunately, even the problem of estimating the posterior probability in a belief network within an absolute error (of less than 0.5), or within a constant multiplicative factor, is NP-hard, so general efficient implementations will not be available.

- The main approaches for probabilistic inference in belief networks are
- exploiting the structure of the network. This approach is typified by the variable elimination algorithm detailed later.
 - search-based approaches. By enumerating some of the possible worlds, posterior probabilities can be estimated from the worlds generated. By computing the probability mass of the worlds not considered, a bound on the error in the posterior probability can be estimated. This approach works well when the distributions are extreme (all probabilities are close to zero or close to one), as occurs in engineered systems.
 - **variational inference**, where the idea is to find an approximation to the problem that is easy to compute. First choose a class of representations that are easy to compute. This class could be as simple as the set of disconnected belief networks (with no arcs). Next try to find a member of the class that is closest to the original problem. That is, find an easy-to-compute distribution that is as close as possible to the posterior distribution that must be computed. Thus, the problem reduces to an optimization problem of minimizing the error.
 - stochastic simulation. In these approaches, random cases are generated according to the probability distributions. By treating these random cases as a set of samples, the marginal distribution on any combination of variables can be estimated. Stochastic simulation methods are discussed in Section [6.4.2](#) (page [256](#)).

This book presents only the first and fourth methods.

6.4.1 Variable Elimination for Belief Networks

This section gives an algorithm for finding the posterior distribution for a variable in an arbitrarily structured belief network. Many of the efficient exact

$r(X, Y, Z) =$	X	Y	Z	val
	t	t	t	0.1
	t	t	f	0.9
	t	f	t	0.2
	t	f	f	0.8
	f	t	t	0.4
	f	t	f	0.6
	f	f	t	0.3
	f	f	f	0.7

$r(X=t, Y, Z) =$	Y	Z	val
	t	t	0.1
	t	f	0.9
	f	t	0.2
	f	f	0.8

$r(X=t, Y, Z=f) =$	Y	val
	t	0.9

$r(X=t, Y=f, Z=f) =$			0.8

Figure 6.5: An example factor and assignments

methods can be seen as optimizations of this algorithm. This algorithm can be seen as a variant of variable elimination (VE) for constraint satisfaction problems (CSPs) (page 127) or VE for soft constraints (page 147).

The algorithm is based on the notion that a belief network specifies a factorization of the joint probability distribution (page 236).

Before we give the algorithm, we define factors and the operations that will be performed on them. Recall that $P(X|Y)$ is a function from variables (or sets of variables) X and Y into the real numbers that, given a value for X and a value for Y , gives the conditional probability of the value for X , given the value for Y . This idea of a function of variables is generalized as the notion of a factor. The VE algorithm for belief networks manipulates factors to compute posterior probabilities.

A **factor** is a representation of a function from a tuple of random variables into a number. We will write factor f on variables $X_1, \dots, X_j$ as $f(X_1, \dots, X_j)$. The variables $X_1, \dots, X_j$ are the variables **of** the factor f , and f is a factor **on** $X_1, \dots, X_j$.

Suppose $f(X_1, \dots, X_j)$ is a factor and each v_i is an element of the domain of X_i . $f(X_1 = v_1, X_2 = v_2, \dots, X_j = v_j)$ is a number that is the value of f when each X_i has value v_i . Some of the variables of a factor can be assigned to values to make a new factor on the other variables. For example, $f(X_1 = v_1, X_2, \dots, X_j)$, sometimes written as $f(X_1, X_2, \dots, X_j)_{X_1=v_1}$, where v_1 is an element of the domain of variable X_1 , is a factor on $X_2, \dots, X_j$.

Example 6.17 Figure 6.5 shows a factor $r(X, Y, Z)$ on variables X, Y and Z as a table. This assumes that each variable is binary with domain t, f . The figure gives a table for the factor $r(X = t, Y, Z)$, which is a factor on Y, Z . Similarly, $r(X = t, Y, Z = f)$ is a factor on Y , and $r(X = t, Y = f, Z = f)$ is a number.

Factors can be multiplied together. Suppose f_1 and f_2 are factors, where f_1 is a factor that contains variables $X_1, \dots, X_i$ and $Y_1, \dots, Y_j$, and f_2 is a factor with variables $Y_1, \dots, Y_j$ and $Z_1, \dots, Z_k$, where $Y_1, \dots, Y_j$ are the variables in

Representations of Conditional Probabilities and Factors

A conditional probability distribution can be seen as a function of the variables involved. A factor is a representation of a function on variables; thus, a factor can be used to represent conditional probabilities.

When the variables have finite domains, these factors can be implemented as arrays. If there is an ordering of the variables (e.g., alphabetical) and the values in the domains are mapped into the non-negative integers, there is a canonical representation of each factor as a one-dimensional array that is indexed by natural numbers. Operations on factors can be implemented efficiently using such a representation. However, this form is not a good one to present to users because the structure of the conditional is lost.

Factors do not have to be implemented as arrays. The tabular representation is often too large when there are many parents. Often, more structure exists in conditional probabilities that can be exploited.

One such structure exploits **context-specific independence**, where one variable is conditionally independent of another, given a particular value of the third variable. For example, suppose the robot can go outside or get coffee. Whether it gets wet depends on whether there is rain in the context that it went out or on whether the cup was full if it got coffee. There are a number of ways to represent the conditional probability $P(Wet \mid Out, Rain, Full)$ —for example as a decision tree, as rules with probabilities, or as tables with contexts:

<pre> graph TD Out[Out] -- t --> Node1 Out -- f --> Node2 Node1 -- Rain --> P1[0.8] Node1 -- Full --> P2[0.1] Node2 -- Full --> P3[0.6] Node2 -- Rain --> P4[0.3] </pre>	<i>out :</i>	<table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th><i>Rain</i></th> <th><i>Wet</i></th> <th><i>Prob</i></th> </tr> </thead> <tbody> <tr><td><i>t</i></td><td><i>t</i></td><td>0.8</td></tr> <tr><td><i>t</i></td><td><i>f</i></td><td>0.2</td></tr> <tr><td><i>f</i></td><td><i>t</i></td><td>0.1</td></tr> <tr><td><i>f</i></td><td><i>f</i></td><td>0.9</td></tr> </tbody> </table>	<i>Rain</i>	<i>Wet</i>	<i>Prob</i>	<i>t</i>	<i>t</i>	0.8	<i>t</i>	<i>f</i>	0.2	<i>f</i>	<i>t</i>	0.1	<i>f</i>	<i>f</i>	0.9
<i>Rain</i>	<i>Wet</i>	<i>Prob</i>															
<i>t</i>	<i>t</i>	0.8															
<i>t</i>	<i>f</i>	0.2															
<i>f</i>	<i>t</i>	0.1															
<i>f</i>	<i>f</i>	0.9															
$wet \leftarrow out \wedge rain : 0.8$ $wet \leftarrow out \wedge \sim rain : 0.1$ $wet \leftarrow \sim out \wedge full : 0.6$ $wet \leftarrow \sim out \wedge \sim full : 0.3$	<i>~out :</i>	<table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th><i>Full</i></th> <th><i>Wet</i></th> <th><i>Prob</i></th> </tr> </thead> <tbody> <tr><td><i>t</i></td><td><i>t</i></td><td>0.6</td></tr> <tr><td><i>t</i></td><td><i>f</i></td><td>0.4</td></tr> <tr><td><i>f</i></td><td><i>t</i></td><td>0.3</td></tr> <tr><td><i>f</i></td><td><i>f</i></td><td>0.7</td></tr> </tbody> </table>	<i>Full</i>	<i>Wet</i>	<i>Prob</i>	<i>t</i>	<i>t</i>	0.6	<i>t</i>	<i>f</i>	0.4	<i>f</i>	<i>t</i>	0.3	<i>f</i>	<i>f</i>	0.7
<i>Full</i>	<i>Wet</i>	<i>Prob</i>															
<i>t</i>	<i>t</i>	0.6															
<i>t</i>	<i>f</i>	0.4															
<i>f</i>	<i>t</i>	0.3															
<i>f</i>	<i>f</i>	0.7															

Another common representation is a **noisy or**. For example, suppose the robot can get wet from rain, coffee, sprinkler, or kids. There can be a probability that it gets wet from rain if it rains, and a probability that it gets wet from coffee if it has coffee, and so on (these probabilities give the noise). The robot gets wet if it gets wet from one of them, giving the “or”.

The next chapter explores other representations that can be used for conditional probabilities.

$f_1=$	<table><tr><th>A</th><th>B</th><th>val</th></tr><tr><td>t</td><td>t</td><td>0.1</td></tr><tr><td>t</td><td>f</td><td>0.9</td></tr><tr><td>f</td><td>t</td><td>0.2</td></tr><tr><td>f</td><td>f</td><td>0.8</td></tr></table>	A	B	val	t	t	0.1	t	f	0.9	f	t	0.2	f	f	0.8	$f_1 \times f_2=$	<table><tr><th>A</th><th>B</th><th>C</th><th>val</th></tr><tr><td>t</td><td>t</td><td>t</td><td>0.03</td></tr><tr><td>t</td><td>t</td><td>f</td><td>0.07</td></tr><tr><td>t</td><td>f</td><td>t</td><td>0.54</td></tr><tr><td>t</td><td>f</td><td>f</td><td>0.36</td></tr><tr><td>f</td><td>t</td><td>t</td><td>0.06</td></tr><tr><td>f</td><td>t</td><td>f</td><td>0.14</td></tr><tr><td>f</td><td>f</td><td>t</td><td>0.48</td></tr><tr><td>f</td><td>f</td><td>f</td><td>0.32</td></tr></table>	A	B	C	val	t	t	t	0.03	t	t	f	0.07	t	f	t	0.54	t	f	f	0.36	f	t	t	0.06	f	t	f	0.14	f	f	t	0.48	f	f	f	0.32
	A	B	val																																																			
	t	t	0.1																																																			
	t	f	0.9																																																			
f	t	0.2																																																				
f	f	0.8																																																				
A	B	C	val																																																			
t	t	t	0.03																																																			
t	t	f	0.07																																																			
t	f	t	0.54																																																			
t	f	f	0.36																																																			
f	t	t	0.06																																																			
f	t	f	0.14																																																			
f	f	t	0.48																																																			
f	f	f	0.32																																																			
$f_2=$	<table><tr><th>B</th><th>C</th><th>val</th></tr><tr><td>t</td><td>t</td><td>0.3</td></tr><tr><td>t</td><td>f</td><td>0.7</td></tr><tr><td>f</td><td>t</td><td>0.6</td></tr><tr><td>f</td><td>f</td><td>0.4</td></tr></table>	B	C	val	t	t	0.3	t	f	0.7	f	t	0.6	f	f	0.4																																						
	B	C	val																																																			
	t	t	0.3																																																			
	t	f	0.7																																																			
f	t	0.6																																																				
f	f	0.4																																																				

Figure 6.6: Multiplying factors example

common to f_1 and f_2 . The **product** of f_1 and f_2 , written $f_1 \times f_2$, is a factor on the union of the variables, namely $X_1, \dots, X_i, Y_1, \dots, Y_j, Z_1, \dots, Z_k$, defined by:

$$(f_1 \times f_2)(X_1, \dots, X_i, Y_1, \dots, Y_j, Z_1, \dots, Z_k) \\ = f_1(X_1, \dots, X_i, Y_1, \dots, Y_j) \times f_2(Y_1, \dots, Y_j, Z_1, \dots, Z_k).$$

Example 6.18 Figure 6.6 shows the product of $f_1(A, B)$ and $f_2(B, C)$, which is a factor on A, B, C . Note that $(f_1 \times f_2)(A = t, B = f, C = f) = f_1(A = t, B = f) \times f_2(B = f, C = f) = 0.9 \times 0.4 = 0.36$.

The remaining operation is to sum out a variable in a factor. Given factor $f(X_1, \dots, X_j)$, summing out a variable, say X_1 , results in a factor on the other variables, $X_2, \dots, X_j$, defined by

$$(\sum_{X_1} f)(X_2, \dots, X_j) = f(X_1 = v_1, X_2, \dots, X_j) + \dots + f(X_1 = v_k, X_2, \dots, X_j),$$

where $\{v_1, \dots, v_k\}$ is the set of possible values of variable X_1 .

Example 6.19 Figure 6.7 (on the next page) gives an example of summing out variable B from a factor $f_3(A, B, C)$, which is a factor on A, C . Notice how

$$\begin{aligned} (\sum_B f_3)(A = t, C = f) &= f_3(A = t, B = t, C = f) + f_3(A = t, B = f, C = f) \\ &= 0.07 + 0.36 \\ &= 0.43 \end{aligned}$$

A conditional probability distribution $P(X|Y_1, \dots, Y_j)$ can be seen as a factor f on $X, Y_1, \dots, Y_j$, where

$$f(X = u, Y_1 = v_1, \dots, Y_j = v_j) = P(X = u | Y_1 = v_1 \wedge \dots \wedge Y_j = v_j).$$

Usually, humans prefer the $P(\cdot | \cdot)$ notation, but internally the computer just treats conditional probabilities as factors.

$f_3 =$	A	B	C	val	$\Sigma_B f_3 =$	A	C	val
	t	t	t	0.03		t	t	0.57
	t	t	f	0.07		t	f	0.43
	t	f	t	0.54		f	t	0.54
	t	f	f	0.36		f	f	0.46
	f	t	t	0.06				
	f	t	f	0.14				
	f	f	t	0.48				
	f	f	f	0.32				

Figure 6.7: Summing out a variable from a factor

The **belief network inference problem** is the problem of computing the posterior distribution of a variable, given some evidence.

The problem of computing posterior probabilities can be reduced to the problem of computing the probability of conjunctions. Given evidence $Y_1 = v_1, \dots, Y_j = v_j$, and query variable Z :

$$\begin{aligned}
 P(Z | Y_1 = v_1, \dots, Y_j = v_j) &= \frac{P(Z, Y_1 = v_1, \dots, Y_j = v_j)}{P(Y_1 = v_1, \dots, Y_j = v_j)} \\
 &= \frac{P(Z, Y_1 = v_1, \dots, Y_j = v_j)}{\sum_Z P(Z, Y_1 = v_1, \dots, Y_j = v_j)}
 \end{aligned}$$

So the agent computes the factor $P(Z, Y_1 = v_1, \dots, Y_j = v_j)$ and normalizes. Note that this is a factor only of Z ; given a value for Z , it returns a number that is the probability of the conjunction of the evidence and the value for Z .

Suppose the variables of the belief network are $X_1, \dots, X_n$. To compute the factor $P(Z, Y_1 = v_1, \dots, Y_j = v_j)$, sum out the other variables from the joint distribution. Suppose $Z_1, \dots, Z_k$ is an enumeration of the other variables in the belief network – that is,

$$\{Z_1, \dots, Z_k\} = \{X_1, \dots, X_n\} - \{Z\} - \{Y_1, \dots, Y_j\}.$$

The probability of Z conjoined with the evidence is

$$p(Z, Y_1 = v_1, \dots, Y_j = v_j) = \sum_{Z_k} \cdots \sum_{Z_1} P(X_1, \dots, X_n)_{Y_1 = v_1, \dots, Y_j = v_j}.$$

The order that the variables Z_i are summed out is an **elimination ordering**.

Note how this is related to the possible worlds semantics of probability (page 221). There is a possible world for each assignment of a value to each variable. The **joint probability distribution**, $P(X_1, \dots, X_n)$, gives the

probability (or measure) for each possible world. The VE algorithm thus selects the worlds with the observed values for the Y_i 's and sums over the possible worlds with the same value for Z . This corresponds to the definition of conditional probability (page 225). However, VE does this more efficiently than by summing over all of the worlds.

By the rule for conjunction of probabilities and the definition of a belief network,

$$P(X_1, \dots, X_n) = P(X_1 | \text{parents}(X_1)) \times \dots \times P(X_n | \text{parents}(X_n)),$$

where $\text{parents}(X_i)$ is the set of parents of variable X_i .

We have now reduced the belief network inference problem to a problem of summing out a set of variables from a product of factors. To solve this problem efficiently, we use the distribution law learned in high school: to compute a sum of products such as $xy + xz$ efficiently, distribute out the common factors (here x), which results in $x(y + z)$. This is the essence of the VE algorithm. We call the elements multiplied together “factors” because of the use of the term in algebra. Initially, the factors represent the conditional probability distributions, but the intermediate factors are just functions on variables that are created by adding and multiplying factors.

To compute the posterior distribution of a query variable given observations:

1. Construct a factor for each conditional probability distribution.
2. Eliminate each of the non-query variables:
 - if the variable is observed, its value is set to the observed value in each of the factors in which the variable appears,
 - otherwise the variable is summed out.
3. Multiply the remaining factors and normalize.

To sum out a variable Z from a product $f_1, \dots, f_k$ of factors, first partition the factors into those that do not contain Z , say $f_1, \dots, f_i$, and those that contain Z , $f_{i+1}, \dots, f_k$; then distribute the common factors out of the sum:

$$\sum_Z f_1 \times \dots \times f_k = f_1 \times \dots \times f_i \times \sum_Z f_{i+1} \times \dots \times f_k.$$

VE explicitly constructs a representation (in terms of a multidimensional array, a tree, or a set of rules) of the rightmost factor.

Figure 6.8 (on the next page) gives pseudocode for the VE algorithm. The elimination ordering can be given a priori or can be computed on the fly. It is worthwhile to select observed variables first in the elimination ordering, because eliminating these simplifies the problem.

This assumes that the query variable is not observed. If it is observed to have a particular value, its posterior probability is just 1 for the observed value and 0 for the other values.

```

1: procedure VE BN(Vs, Ps, O, Q)
2:   Inputs
3:     Vs: set of variables
4:     Ps: set of factors representing the conditional probabilities
5:     O: set of observations of values on some of the variables
6:     Q: a query variable
7:   Output
8:     posterior distribution on Q
9:   Local
10:    Fs: a set of factors
11:    Fs  $\leftarrow$  Ps
12:    for each X  $\in$  Vs  $- \{Q\}$  using some elimination ordering do
13:      if X is observed then
14:        for each F  $\in$  Fs that involves X do
15:          set X in F to its observed value in O
16:          project F onto remaining variables
17:        else
18:          Rs :=  $\{F \in Fs : F \text{ involves } X\}$ 
19:          let T be the product of the factors in Rs
20:          N :=  $\sum_X T$ 
21:          Fs := Fs  $\setminus$  Rs  $\cup \{N\}$ 
22:        let T be the product of the factors in Rs
23:        N :=  $\sum_Q T$ 
24:    return T/N

```

Figure 6.8: Variable elimination for belief networks

Example 6.20 Consider Example 6.10 (page 236) with the query

$$P(\text{Tampering} | \text{Smoke} = \text{true} \wedge \text{Report} = \text{true}).$$

After eliminating the observed variables, *Smoke* and *Report*, the following factors remain:

<i>ConditionalProbability</i>	<i>Factor</i>
$P(\text{Tampering})$	$f_0(\text{Tampering})$
$P(\text{Fire})$	$f_1(\text{Fire})$
$P(\text{Alarm} \text{Tampering}, \text{Fire})$	$f_2(\text{Tampering}, \text{Fire}, \text{Alarm})$
$P(\text{Smoke} = \text{yes} \text{Fire})$	$f_3(\text{Fire})$
$P(\text{Leaving} \text{Alarm})$	$f_4(\text{Alarm}, \text{Leaving})$
$P(\text{Report} = \text{yes} \text{Leaving})$	$f_5(\text{Leaving})$

The algorithm ignores the conditional probability reading and just works with the factors. The intermediate factors do not always represent a conditional probability.

Suppose *Fire* is selected next in the elimination ordering. To eliminate *Fire*, collect all of the factors containing *Fire* – $f_1(\text{Fire})$, $f_2(\text{Tampering}, \text{Fire}, \text{Alarm})$, and $f_3(\text{Fire})$ – multiply them together, and sum out *Fire* from the resulting factor. Call this factor $F_6(\text{Tampering}, \text{Alarm})$. At this stage, the following factors remain:

$$\begin{aligned} &f_0(\text{Tampering}), \\ &f_4(\text{Alarm}, \text{Leaving}), \\ &f_5(\text{Leaving}), \\ &f_6(\text{Tampering}, \text{Alarm}). \end{aligned}$$

Suppose *Alarm* is eliminated next. VE multiplies the factors containing *Alarm* and sums out *Alarm* from the product, giving a factor, call it f_7 :

$$f_7(\text{Tampering}, \text{Leaving}) = \sum_{\text{Alarm}} f_4(\text{Alarm}, \text{Leaving}) \times f_6(\text{Tampering}, \text{Alarm})$$

It then has the following factors:

$$\begin{aligned} &f_0(\text{Tampering}), \\ &f_5(\text{Leaving}), \\ &f_7(\text{Tampering}, \text{Leaving}). \end{aligned}$$

Eliminating *Leaving* results in the factor

$$f_8(\text{Tampering}) = \sum_{\text{Leaving}} f_5(\text{Leaving}) \times f_7(\text{Tampering}, \text{Leaving}).$$

To determine the distribution over *Tampering*, multiply the remaining factors, giving

$$f_9(\text{Tampering}) = f_0(\text{Tampering}) \times f_8(\text{Tampering}).$$

The posterior distribution over tampering is given by

$$\frac{f_9(\text{Tampering})}{\sum_{\text{Tampering}} f_9(\text{Tampering})}.$$

Note that the denominator is the prior probability of the evidence.

Example 6.21 Consider the same network as in the previous example but with the following query:

$$P(\text{Alarm} | \text{Fire} = \text{true}).$$

When *Fire* is eliminated, the factor $P(\text{Fire})$ becomes a factor of no variables; it is just a number, $P(\text{Fire} = \text{true})$.

Suppose *Report* is eliminated next. It is in one factor, which represents $P(\text{Report} | \text{Leaving})$. Summing over all of the values of *Report* gives a factor on *Leaving*, all of whose values are 1. This is because $P(\text{Report} = \text{true} | \text{Leaving} = v) + P(\text{Report} = \text{false} | \text{Leaving} = v) = 1$ for any value v of *Leaving*.

Similarly, if you eliminate *Leaving* next, you multiply a factor that is all 1 by a factor representing $P(\textit{Leaving} | \textit{Alarm})$ and sum out *Leaving*. This, again, results in a factor all of whose values are 1.

Similarly, eliminating *Smoke* results in a factor of no variables, whose value is 1. Note that even if smoke had been observed, eliminating smoke would result in a factor of no variables, which would not affect the posterior distribution on *Alarm*.

Eventually, there is only the factor on *Alarm* that represents its prior probability and a constant factor that will cancel in the normalization.

The complexity of the algorithm depends on a measure of complexity of the network. The size of a tabular representation of a factor is exponential in the number of variables in the factor. The **treewidth** of a network, given an elimination ordering, is the maximum number of variables in a factor created by summing out a variable, given the elimination ordering. The **treewidth** of a belief network is the minimum treewidth over all elimination orderings. The treewidth depends only on the graph structure and is a measure of the sparseness of the graph. The complexity of VE is exponential in the treewidth and linear in the number of variables. Finding the elimination ordering with minimum treewidth is NP-hard, but there is some good elimination ordering heuristics, as discussed for CSP VE (page 130).

There are two main ways to speed up this algorithm. Irrelevant variables can be pruned, given the observations and the query. Alternatively, it is possible to compile the graph into a secondary structure that allows for caching of values.

6.4.2 Approximate Inference Through Stochastic Simulation

Many problems are too big for exact inference, and one must resort to approximate inference. One of the most effective methods is based on generating random samples from the (posterior) distribution that the network specifies.

Stochastic simulation is based on the idea that a set of samples can be used to compute probabilities. For example, you could interpret the probability $P(a) = 0.14$ as meaning that, out of 1,000 samples, about 140 will have *a* true. You can go from (enough) samples into probabilities and from probabilities into samples.

We consider three problems:

- how to generate samples,
- how to incorporate observations, and
- how to infer probabilities from samples.

We examine three methods that use sampling to compute the posterior distribution of a variable: (1) rejection sampling, (2) importance sampling, and (3) particle filtering.

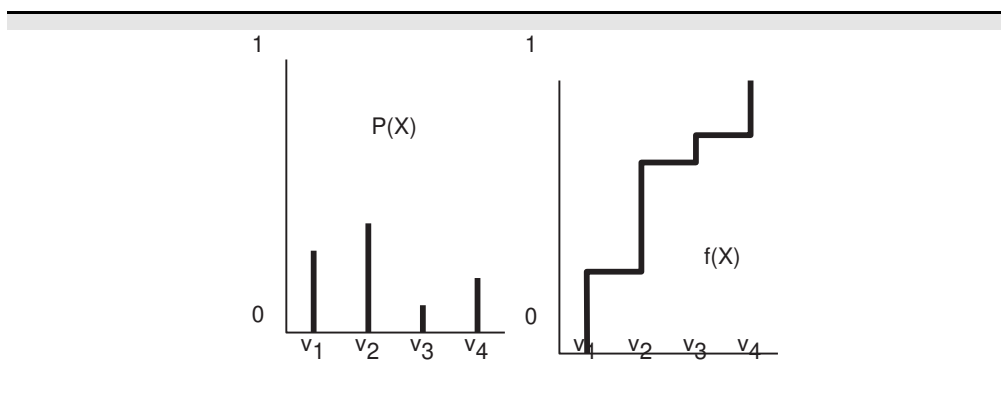


Figure 6.9: A cumulative probability distribution

Sampling from a Single Variable

To generate samples from a single discrete or real-valued variable, X , first totally order the values in the domain of X . For discrete variables, if there is no natural order, you can just create an arbitrary ordering. Given this ordering, the **cumulative probability distribution** is a function of x , defined by $f(x) = P(X \leq x)$.

To generate a random sample for X , select a random number y in the range $[0, 1]$. We select y from a uniform distribution to ensure that each number between 0 and 1 has the same chance of being chosen. Let v be the value of X that maps to y in the cumulative probability distribution. That is, v is the element of $\text{dom}(X)$ such that $f(v) = y$ or, equivalently, $v = f^{-1}(y)$. Then, $X = v$ is a random sample of X , chosen according to the distribution of X .

Example 6.22 Consider a random variable X with domain $\{v_1, v_2, v_3, v_4\}$. Suppose $P(X=v_1) = 0.3$, $P(X=v_2) = 0.4$, $P(X=v_3) = 0.1$, and $P(X=v_4) = 0.2$. First, totally order the values, say $v_1 < v_2 < v_3 < v_4$. Figure 6.9 shows $P(X)$, the distribution for X , and $f(X)$, the cumulative distribution for X . Consider value v_1 ; 0.3 of the range of f maps back to v_1 . Thus, if a sample is uniformly selected from the Y -axis, v_1 has a 0.3 chance of being selected, v_2 has a 0.4 chance of being selected, and so forth.

Forward Sampling in Belief Networks

Forward sampling is a way to generate a sample of every variable of a belief network so that each sample is generated in proportion to its probability. Suppose $X_1, \dots, X_n$ is a total ordering of the variables so that the parents of a variable come before the variable in the total order. Forward sampling draws a sample of all of the variables by drawing a sample of each variable $X_1, \dots, X_n$ in order. First, it samples X_1 using the aforementioned method. For each of the other variables, due to the total ordering of variables, when it comes time to

Sample	Tampering	Fire	Alarm	Smoke	Leaving	Report
s ₁	false	true	true	true	false	false
s ₂	false	false	false	false	false	false
s ₃	false	true	true	true	true	true
s ₄	false	false	false	false	false	true
s ₅	false	false	false	false	false	false
s ₆	false	false	false	false	false	false
s ₇	true	false	false	true	true	true
s ₈	true	false	false	false	false	true
...						
s ₁₀₀₀	true	false	true	true	false	false

Figure 6.10: Sampling for a belief network

sample X_i , it already has values for all of X_i 's parents. It now samples a value for X_i from the distribution of X_i given the values already assigned to the parents of X_i . Repeating this for every variable generates a sample containing values for all of the variables. The probability of selecting a particular assignment to all of the variables will be the probability of the assignment.

Example 6.23 Let us create a set of samples for the belief network of Figure 6.1 (page 237). Suppose the variables are ordered as follows: *Tampering*, *Fire*, *Alarm*, *Smoke*, *Leaving*, *Report*. First the algorithm samples *Tampering*, using the inverse of the cumulative distribution. Suppose it selects *Tampering* = *false*. Then it samples *Fire* using the same method. Suppose it selects *Fire* = *true*. Then it must select a value for *Alarm*, using the distribution $P(\textit{Alarm} \mid \textit{Tampering} = \textit{false}, \textit{Fire} = \textit{true})$. Suppose it selects *Alarm* = *true*. Next, it selects a value for *Smoke* using $P(\textit{Smoke} \mid \textit{Fire} = \textit{true})$. Then it selects a value for *Leaving* using the distribution for $P(\textit{Leaving} \mid \textit{Alarm} = \textit{true})$. Suppose it selects *Leaving* = *false*. Then it selects a value for *Report*, using the distribution $P(\textit{Report} \mid \textit{Leaving} = \textit{false})$. It has thus selected a value for each variable and created the first sample of Figure 6.10. Notice that it has selected a very unlikely combination of values. This does not happen very often; it happens in proportion to how likely the sample is. It can then repeat this until it has enough samples. In Figure 6.10, it generated 1,000 samples.

From Samples to Probabilities

Probabilities can be estimated from a set of examples using the sample average. The **sample average** of a proposition a is the number of samples where a is true divided by the total number of samples. The sample average approaches the true probability as the number of samples approaches infinity by the law of large numbers.

Hoeffding's inequality provides an estimate of the error of an unconditional probability given n samples:

Proposition 6.6 (Hoeffding). *Suppose p is the true probability, and s is the sample average from n independent samples; then*

$$P(|s - p| > c) \leq 2e^{-2nc^2}.$$

This theorem can be used to determine how many samples are required to guarantee a **probably approximately correct** estimate of the probability. To guarantee that the error is less than some $c < 0.5$, infinitely many samples are required. However, if you are willing to have an error greater than c in δ of the cases, you can solve $2e^{-2nc^2} < \delta$ for n , which gives

$$n > \frac{-\ln \frac{\delta}{2}}{2c^2}.$$

For example, suppose you want an error less than 0.1, nineteen times out of twenty; that is, you are only willing to tolerate an error bigger than 0.1, in 5% of the cases. You can use Hoeffding's bound by setting c to 0.1 and $\delta = 0.05$, which gives $n > 184$. Thus, you can guarantee such bounds on the error with 185 samples. If you want an error of less than 0.01 in at least 95% of the cases, 18,445 samples can be used. If you want an error of less than 0.1 in 99% of the cases, 265 samples can be used.

Rejection Sampling

Given some evidence e , rejection sampling estimates $P(h|e)$ using the formula

$$P(h|e) = \frac{P(h \wedge e)}{P(e)}.$$

This can be computed by considering only the samples where e is true and by determining the proportion of these in which h is true. The idea of **rejection sampling** is that samples are generated as before, but any sample where e is false is rejected. The proportion of the remaining, non-rejected, samples where h is true is an estimate of $P(h|e)$. If the evidence is a conjunction of assignments of values to variables, a sample can be rejected when any of the variables assigned in the sample are different from the observed value.

The error in the probability of h depends on the number of samples that are not rejected. The number of samples that are not rejected is proportional to $P(e)$. Thus, in Hoeffding's inequality, n is the number of non-rejected samples. Therefore, the error depends on $P(e)$.

Rejection sampling does not work well when the evidence is unlikely. This may not seem like that much of a problem because, by definition, unlikely

Sample	Tampering	Fire	Alarm	Smoke	Leaving	Report	
s_1	false	true	true	true	X		
s_2	false	false	false	false	false	false	X
s_3	false	true	true	true	X		
s_4	false	false	false	false	false	true	✓
s_5	false	false	false	false	false	false	X
s_6	false	false	false	false	false	false	X
s_7	true	false	false	true	X		
s_8	true	false	false	false	false	true	✓
...							
s_{1000}	true	false	true	true	X		

Figure 6.11: Rejection sampling for $P(\text{tampering} | \neg \text{smoke} \wedge \text{report})$

evidence is unlikely to occur. But, although this may be true for simple models, for complicated models with complex observations, every possible observation may be unlikely. Also, for many applications, such as in diagnosis, the user is interested in determining the probabilities because unusual observations are involved.

Example 6.24 Figure 6.11 shows how rejection sampling can be used to estimate $P(\text{tampering} | \text{smoke} \wedge \text{report})$. Any sample with $\text{Smoke} = \text{true}$ is rejected. The sample can be rejected without considering any more variables. Any sample with $\text{Report} = \text{false}$ is rejected. The sample average from the remaining samples (those marked with ✓) is used to estimate the posterior probability of *tampering*.

Because $P(\neg \text{smoke} \wedge \text{report}) = 0.0213$, we would expect about 21 samples out of the 1,000 to not be rejected. Thus, 21 can be used as n in Hoeffding’s inequality, which, for example, guarantees an error for any probability computed from these samples of less than 0.2 in about 63% of the cases.

Importance Sampling

Instead of creating a sample and then rejecting it, it is possible to mix sampling with inference to reason about the probability that a sample would be rejected. In importance sampling, each sample has a weight, and the sample average is computed using the weighted average of samples. The weights of samples come from two sources:

- The samples do not have to be selected in proportion to their probability, but they can be selected according to some other distribution, called the **proposal distribution**.
- Evidence is used to update the weights and is used to compute the probability that a sample would be rejected.

Example 6.25 Consider variable A with no parents; variable E has A as its only parent, but A has other children. Suppose $P(e|a) = 0.003$, $P(e|\bar{a}) = 0.63$, and e is observed. Consider the samples with $A = \text{true}$. Out of 1,000 such samples, only about 3 will not be rejected. Instead of rejecting 99.7% of the samples with $A = \text{true}$, each sample with $A = \text{true}$ can be weighted by 0.003. Thus, just one sample is able to convey the information of many rejections.

Suppose $P(a) = 0.98$. If the algorithm samples according to the probability, $A = \text{false}$ would only be true in about 20 samples out of 1,000. Instead of sampling according to the probability, suppose $A = \text{true}$ is sampled 50% of the time, but each sample is weighted as follows. Each sample with $A = \text{true}$ can be weighted by $0.98/0.5 = 1.96$ and each sample with $A = \text{false}$ can be weighted by $0.02/0.5 = 0.04$. It is easy to show that the weighted sample average is the same as the probability.

In rejection sampling, given the preceding probabilities and the observation of e , A will be true in 98% of the samples and 99.7% of these will be rejected due to the evidence of e . $A = \text{false}$ would be selected in 2% of the samples and 37% of these will be rejected. Rejection sampling would thus accept only $0.98 \times 0.003 + 0.02 \times 0.63 = 0.01554$ of the samples and would reject more than 98% of the samples.

If you combine the ideas in the first two paragraphs of this example, half of the examples will have $A = \text{true}$, and these will be weighted by $1.96 \times 0.003 = 0.00588$, and the other half of the samples will have $A = \text{false}$ with a weighting of $0.04 \times 0.63 = 0.0252$. Even two such samples convey the information of many rejected samples.

Importance sampling differs from rejection sampling in two ways:

- Importance sampling does not sample all variables, only some of them. The variables that are not sampled and are not observed are summed out (i.e., some exact inference is carried out).

In particular, you probably do not want it to sample the observed variables (although the algorithm does not preclude this). If all of the non-observed variables are sampled, it is easy to determine the probability of a sample given the evidence [see Exercise 6.10 (page 278)].

- Importance sampling does not have to sample the variables according to their prior probability; it can sample them using any distribution. The distribution that it uses to sample the variables is called the **proposal distribution**. Any distribution can be used as a proposal distribution as long as the proposal distribution does not have a zero probability for choosing some sample that is possible in the model (otherwise this part of the space will never be explored). Choosing a good proposal distribution is non-trivial.

In general, to sum over variables S from a product $f(S)q(S)$, you can choose a set of samples $\{s_1, \dots, s_N\}$ from the distribution $q(s)$. Then

$$\sum_S f(S)q(S) = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{s_i} f(s_i) , \quad (6.1)$$

which essentially computes the expected value (page 230) of $f(S)$, where the expectation is over the distribution $q(S)$.

In forward sampling, $q(s)$ is the uniform sample, but Equation (6.1) works for any distribution.

In importance sampling, let S be the set of variables that will be sampled. As in VE, we introduce some variables and sum them out; in this case, we sum over the sampled variables:

$$P(h|e) = \sum_S P(h|S, e)P(S|e).$$

Multiplying the top and the bottom by proposal distribution $q(S)$ gives

$$P(h|e) = \sum_S \frac{P(h|S, e)P(S|e)q(S)}{q(S)}.$$

Note that this does not give a divide-by-zero error; if $q(s) = 0$, s would never be chosen as a sample.

Using Equation (6.1), suppose $\{s_1, \dots, s_N\}$ is the set of all samples:

$$P(h|e) = \lim_{N \rightarrow \infty} \sum_{s_i} \frac{P(h|s_i, e)P(s_i|e)}{q(s_i)}.$$

Using Bayes' rule on $P(s_i|e)$, and noting that $P(e)$ is a constant, gives

$$P(h|e) = \lim_{N \rightarrow \infty} \frac{1}{k} \sum_{s_i} \frac{P(h|s_i, e)P(e|s_i)P(s_i)}{q(s_i)},$$

where k is a normalizing constant that ensures that the posterior probabilities of the values for a mutually exclusive and covering set of hypotheses sum to 1.

Thus, for each sample, the weighting $P(s_i)/q(s_i)$ acts like a prior that is multiplied by the probability of the evidence, given the sample, to get the weight of the sample. Given many samples, the preceding formula shows how to predict the posterior on any h by getting the weighted average prediction of each sample.

Note how importance sampling generalizes rejection sampling. Rejection sampling is the case with $q(s_i) = P(s_i)$ and S includes all of the variables, including the observed variables.

Figure 6.12 shows the details of the importance sampling algorithm for computing $P(Q|e)$ for query variable Q and evidence e . The first *for* loop (line 18) creates the sample s on S . The variable p (on line 21) is the weight of sample s . The algorithm updates the weight of each value of the query variable and adds the probability of the sample s to the variable *mass*, which represents the **probability mass** – the sum of probabilities for all of the values for the query variable. Finally, it returns the probability by dividing the weight of each value of the query variable by the probability mass.

```

1: procedure IS BN( $V_S, P, e, Q, S, q, n$ )
2:   Inputs
3:      $V_S$ : set of variables
4:      $P$ : a procedure to compute conditional probabilities
5:      $e$ : the evidence; an assignment of a value to some of the variables
6:      $Q$ : a query variable
7:      $S = \{S_1, \dots, S_k\}$  a set of variables to sample
8:      $q$ : a distribution on  $S$  (the proposal distribution)
9:      $n$ : number of samples to generate
10:  Output
11:    posterior distribution on  $Q$ 
12:  Local
13:    array  $v[k]$ , where  $v[i] \in \text{dom}(S_i)$ 
14:    real array  $\text{ans}[m]$  where  $m$  is the size of  $\text{dom}(Q)$ 
15:     $s$  assignment of a value to each element of  $S$ 
16:     $\text{mass} := 0$ 
17:    repeat  $n$  times
18:      for  $i = 1 : k$  do
19:        select  $v_i \in \text{dom}(S_i)$  using distribution  $q(S_i=v_i | S_0=v_0, \dots, S_{i-1}=v_{i-1})$ 
20:         $s :=$  assignment  $S_0=v_0, \dots, S_k=v_k$ 
21:         $p := P(e|s) \times P(s)/q(s)$ 
22:        for each  $v_i \in \text{dom}(Q)$  do
23:           $\text{ans}[i] := \text{ans}[i] + P(Q = v_i | s \wedge e) \times p$ 
24:           $\text{mass} := \text{mass} + p$ 
25:    return  $\text{ans}[]/\text{mass}$ 

```

Figure 6.12: Importance sampling for belief network inference

Example 6.26 Suppose we want to use importance sampling to compute $P(\text{alarm} | \text{smoke report})$. We must choose which variables to sample and the proposal distribution. Suppose we sample *Tampering*, *Fire*, and *Leaving*, and we use the following proposal distribution:

$$q(\text{tampering}) = 0.02$$

$$q(\text{fire}) = 0.5$$

$$q(\text{Alarm} | \text{Tampering}, \text{Alarm}) = P(\text{Alarm} | \text{Tampering}, \text{Alarm})$$

$$q(\text{Leaving} | \text{Alarm}) = P(\text{Leaving} | \text{Alarm})$$

Thus, the proposal distribution is the same as the original distribution, except for *Fire*.

The following table gives a few samples. In this table, s is the sample; e is *smoke report*; $P(e|s)$ is equal to $P(\text{smoke} | \text{Fire}) \times P(\text{report} | \text{Leaving})$, where the value for *Fire* and *Leaving* are from the sample; $P(s)/q(s)$ is 0.02 when

$Fire = true$ in the sample and is 1.98 when $Fire = false$; $p = P(e \mid s) \times P(s)/q(s)$ is the weight of the sample.

<i>Tampering</i>	<i>Fire</i>	<i>Alarm</i>	<i>Leaving</i>	$P(e s)$	$P(s)/q(s)$	p
false	true	false	true	0.675	0.02	0.0135
true	true	true	false	0.009	0.02	0.00018
false	false	false	true	0.0075	1.98	0.01485
false	true	false	false	0.009	0.02	0.00018

$P(alarm | smoke \wedge report)$ is the weighted proportion of the samples that have *Alarm* true.

The efficiency of this algorithm, in terms of how accuracy depends on the run time, depends on

- the proposal distribution. To get the best result, the proposal distribution should be as close as possible to the posterior distribution. However, an agent typically cannot sample directly from the posterior distribution; if it could, it could produce posterior probabilities much more simply.
- which variables to sample. Sampling fewer variables means that there is more information from each sample, but each sample requires more time to compute the probability of the sample.

Determining the proposal distribution and which variables to sample is an art.

Particle Filtering

Importance sampling enumerates the samples one at a time and, for each sample, assigns a value to each variable. It is also possible to start with all of the samples and, for each variable, generate a value for that variable for each of the samples. For example, for the data of Figure 6.10 (page 258), the same data could be generated by generating all of the values for *Tampering* before generating the values for *Fire*. The **particle filtering** algorithm generates all the samples for one variable before moving to the next variable. It does one sweep through the variables, and for each variable it does a sweep through all of the samples. This algorithm has an advantage when variables are generated dynamically and there can be unboundedly many variables. It also allows for a new operation of resampling.

Given a set of samples on some of the variables, **resampling** consists of taking n samples, each with their own weighting, and generating a new set of n samples, each with the same weight. Resampling can be implemented in the same way that random samples for a single random variable are generated (page 257), but samples, rather than values, are selected. Some of the samples are selected multiple times and some are not selected.

A **particle** consists of an assignment of a value to a set of variables and an associated weight. The probability of a proposition, given some evidence, is proportional to the weighted proportion of the weights of the particles in which the proposition is true. A set of particles is a **population**.

Particle filtering is a sampling method that starts with a population of particles, each of which assigns a value to no variables, and has a weight of 1. At each step it can

- select a variable that has not been sampled or summed out and is not observed. For each particle, it samples the variable according to some proposal distribution. The weight of the particle is updated as in importance sampling.
- select a piece of evidence to absorb. This evidence should not have been absorbed before. The weight of the particle is multiplied by the probability of the evidence given the values of the particle (and summing out any variables that are relevant and have not been sampled).
- resample the population. Resampling constructs a new population of particles, each with the same weight, by selecting particles from the population, where each particle is chosen with probability proportional to the weight of the particle. Some particles may be forgotten and some may be duplicated.

Importance sampling can be seen as being equivalent to particle filtering without resampling, but the principal difference is the order in which the particles are generated. In particle filtering, each variable is sampled for all particles, whereas, in importance sampling, each particle (sample) is sampled for all variables before the next particle is considered.

Particle filtering has two main advantages over importance sampling. First, it can be used for an unbounded number of variables (which we will see later). Second, the particles better cover the hypothesis space. Whereas importance sampling will involve some particles that have very low probability, with only a few of the particles covering most of the probability mass, resampling lets many particles more uniformly cover the probability mass.

Example 6.27 Consider using particle filtering to compute $P(\text{Report smoke})$ for the belief network of Figure 6.1 (page 237). First generate the particles $s_1, \dots, s_{1000}$. For this example, we use the conditional probability of the variable being sampled given particle as the proposal distribution. Suppose it first samples *Fire*. Out of the 1,000 particles, about 10 will have *Fire* = true and about 990 will have *Fire* = false (as $P(\text{fire}) = 0.01$). It can then absorb the evidence *Smoke* = true. Those particles with *Fire* = true will be weighted by 0.9 [as $P(\text{smoke} | \text{fire}) = 0.9$] and those particles with *Fire* = false will be weighted by 0.01 [as $P(\text{smoke} | \text{fire}) = 0.01$]. It can then resample; each particle can be chosen in proportion to its weight. The particles with *Fire* = true will be chosen in the ratio 990 : 0.01 : 10 : 0.9. Thus, about 524 particles will be chosen with *Fire* = true, and the remainder with *Fire* = false. The other variables can be sampled, in turn, until *Report* is sampled.

Note that in particle filtering the particles are not independent, so Hoeffding's inequality (page 259), is not directly applicable.

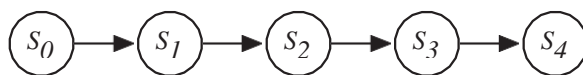


Figure 6.13: A Markov chain as a belief network

6.5 Probability and Time

We can model a dynamic system as a belief network by treating a feature at a particular time as a random variable. We first give a model in terms of states and then show how it can be extended to features.

6.5.1 Markov Chains

A **Markov chain** is a special sort of belief network used to represent sequences of values, such as the sequence of states in a dynamic system or the sequence of words in a sentence.

Figure 6.13 shows a generic Markov chain as a belief network. The network does not have to stop at stage s_4 , but it can be extended indefinitely. The belief network conveys the independence assumption

$$P(S_{t+1}|S_0, \dots, S_t) = P(S_{t+1}|S_t),$$

which is called the **Markov assumption**.

Often, S_t represents the **state** at time t . Intuitively, S_t conveys all of the information about the history that can affect the future states. At S_t , you can see that “the future is conditionally independent of the past given the present.”

A Markov chain is **stationary** if the transition probabilities are the same for each time point [i.e., for all $t > 0$, $t' > 0$, $P(S_{t+1}|S_t) = P(S_{t'+1}|S_{t'})$]. To specify a stationary Markov chain, two conditional probabilities must be specified:

- $P(S_0)$ specifies initial conditions.
- $P(S_{t+1}|S_t)$ specifies the dynamics, which is the same for each $t \geq 0$.

Stationary Markov chains are of interest because

- They provide a simple model that is easy to specify.
- The assumption of stationarity is often the natural model, because the dynamics of the world typically does not change in time. If the dynamics does change in time, it is usually because some other feature exists that could also be modeled.
- The network can extend indefinitely. Specifying a small number of parameters can give an infinite network. You can ask queries or make observations about any arbitrary points in the future or the past.

To determine the probability distribution of state S_i , VE can be used to sum out the preceding variables. Note that the variables after S_i are irrelevant to the probability of S_i and need not be considered. This computation is normally specified as matrix multiplication, but note that matrix multiplication

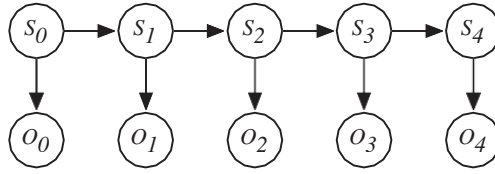


Figure 6.14: A hidden Markov model as a belief network

is a simple form of VE. Similarly, to compute $P(S_i|S_k)$, where $k > i$, only the variables before S_k need to be considered.

6.5.2 HiddenMarkov Models

A **hidden Markov model (HMM)** is an augmentation of the Markov chain to include observations. Just like the state transition of the Markov chain, an HMM also includes observations of the state. These observations can be **partial** in that different states can map to the same observation and **noisy** in that the same state can stochastically map to different observations at different times.

The assumptions behind an HMM are that the state at time $t + 1$ only depends on the state at time t , as in the Markov chain. The observation at time t only depends on the state at time t . The observations are modeled using the variable O_t for each time t whose domain is the set of possible observations. The belief network representation of an HMM is depicted in Figure 6.14. Although the belief network is shown for four stages, it can proceed indefinitely.

A stationary HMM includes the following probability distributions:

- $P(S_0)$ specifies initial conditions.
- $P(S_{t+1}|S_t)$ specifies the dynamics.
- $P(O_t|S_t)$ specifies the sensor model.

There are a number of tasks that are common for HMMs.

The problem of **filtering** or belief-state **monitoring** is to determine the current state based on the current and previous observations, namely to determine

$$P(S_i|O_0, \dots, O_i).$$

Note that all state and observation variables after S_i are irrelevant because they are not observed and can be ignored when this conditional distribution is computed.

The problem of **smoothing** is to determine a state based on past and future observations. Suppose an agent has observed up to time k and wants to determine the state at time i for $i < k$; the smoothing problem is to determine

$$P(S_i|O_0, \dots, O_k).$$

All of the variables S_i and V_i for $i > k$ can be ignored.

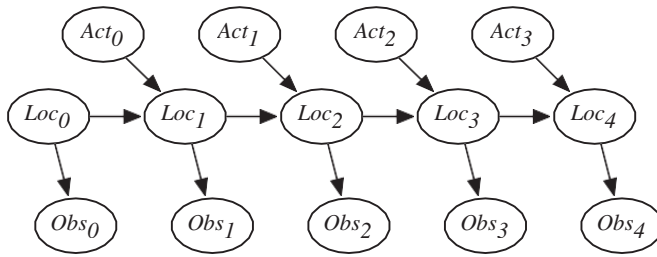


Figure 6.15: A belief network for localization

Localization

Suppose a robot wants to determine its location based on its history of actions and its sensor readings. This is the problem of **localization**. Figure 6.15 shows a belief-network representation of the localization problem. There is a variable Loc_i for each time i , which represents the robot's location at time i . There is a variable Obs_i for each time i , which represents the robot's observation made at time i . For each time i , there is a variable Act_i that represents the robot's action at time i . In this section, assume that the robot's actions are observed (we consider the case in which the robot chooses its actions in Chapter 9).

This model assumes the following dynamics: At time i , the robot is at location Loc_i , it observes Obs_i , then it acts, it observes its action Act_i , and time progresses to time $i + 1$, where it is at location Loc_{i+1} . Its observation at time t only depends on the state at time t . The robot's location at time $t + 1$ depends on its location at time t and its action at time t . Its location at time $t + 1$ is conditionally independent of previous locations, previous observations, and previous actions, given its location at time t and its action at time t .

The localization problem is to determine the robot's location as a function of its observation history:

$$P(Loc_t | Obs_0, Act_0, Obs_1, Act_1, \dots, Act_{t-1}, Obs_t).$$

Example 6.28 Consider the domain depicted in Figure 6.16. There is a circular corridor, with 16 locations numbered 0 to 15. The robot is at one of these locations at each time. This is modeled with, for every time i , a variable Loc_i with domain $\{0, 1, \dots, 15\}$.

- There are doors at positions 2, 4, 7, and 11 and no doors at other locations.
- The robot has a sensor that can noisily sense whether or not it is in front of a door. This is modeled with a variable Obs_i for each time i , with domain $\{door, nodoor\}$. Assume the following conditional probabilities:

$$P(Obs=door \mid atDoor) = 0.8$$

$$P(Obs=door \mid notAtDoor) = 0.1$$

where $atDoor$ is true at states 2, 4, 7, and 11 and $notAtDoor$ is true at the other states.

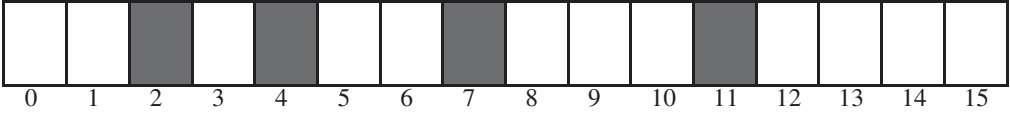


Figure 6.16: Localization domain

Thus, the observation is partial in that many states give the same observation and it is noisy in the following way: In 20% of the cases in which the robot is at a door, the sensor falsely gives a negative reading. In 10% of the cases where the robot is not at a door, the sensor records that there is a door.

- The robot can, at each time, move left, move right, or stay still. Assume that the *stay still* action is deterministic, but the dynamics of the moving actions are stochastic. Just because it carries out the *goRight* action does not mean that it actually goes one step to the right – it is possible that it stays still, goes two steps right, or even ends up at some arbitrary location (e.g., if someone picks up the robot and moves it). Assume the following dynamics, for each location L :

$$P(\text{Loc}_{t+1}=L|\text{Act}_t=\text{goRight} \wedge \text{Loc}_t=L)=0.1$$

$$P(\text{Loc}_{t+1}=L+1|\text{Act}_t=\text{goRight} \wedge \text{Loc}_t=L)=0.8$$

$$P(\text{Loc}_{t+1}=L+2|\text{Act}_t=\text{goRight} \wedge \text{Loc}_t=L)=0.074$$

$$P(\text{Loc}_{t+1}=L^t|\text{Act}_t=\text{goRight} \wedge \text{Loc}_t=L)=0.002 \text{ for any other location } L^t.$$

All location arithmetic is modulo 16. The action *goLeft* works the same but to the left.

The robot starts at an unknown location and must determine its location.

It may seem as though the domain is too ambiguous, the sensors are too noisy, and the dynamics is too stochastic to do anything. However, it is possible to compute the probability of the robot's current location given its history of actions and observations.

Figure 6.17 (on the next page) gives the robot's probability distribution over its locations, assuming it starts with no knowledge of where it is and experiences the following observations: observe door, go right, observe no door, go right, and then observe door. Location 4 is the most likely current location, with posterior probability of 0.42. That is, in terms of the network of Figure 6.15:

$$P(\text{Loc}_2=4 \mid \text{Obs}_0=\text{door}, \text{Act}_0=\text{goRight}, \text{Obs}_1=\text{nodoor}, \\ \text{Act}_1=\text{goRight}, \text{Obs}_2=\text{door})=0.42$$

Location 7 is the second most likely current location, with posterior probability of 0.141. Locations 0, 1, 3, 8, 12, and 15 are the least likely current locations, with posterior probability of 0.011.

You can see how well this works for other sequences of observations by using the applet at the book web site.

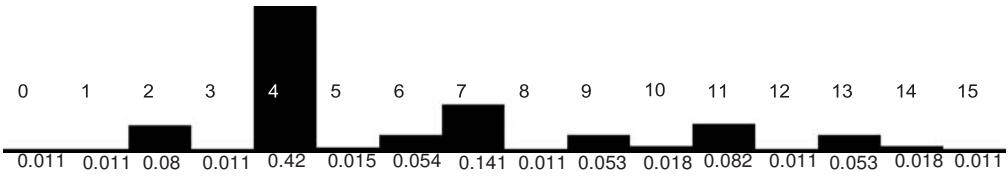


Figure 6.17: A distribution over locations. The locations are numbered from 0 to 15. The number at the bottom gives the posterior probability that the robot is at the location after the particular sequence of actions and observations given in Example 6.28 (page 268). The height of the bar is proportional to the posterior probability.

Example 6.29 Let us augment Example 6.28 (page 268) with another sensor. Suppose that, in addition to a door sensor, there is also a light sensor. The light sensor and the door sensor are conditionally independent given the state. Suppose the light sensor is not very informative; it can only give yes-or-no information about whether it can detect any light, and that this is very noisy, and depends on the location.

This is modeled in Figure 6.18 using the following variables:

- Loc_t is the robot's location at time t .
- Act_t is the robot's action at time t .
- D_t is the door sensor value at time t .
- L_t is the light sensor value at time t .

Conditioning on both L_i and D_i lets it combine information from the light sensor and the door sensor. This is an instance of **sensor fusion**. It is not necessary to define any new mechanisms for sensor fusion given the belief-network model; standard probabilistic inference combines the information from both sensors.

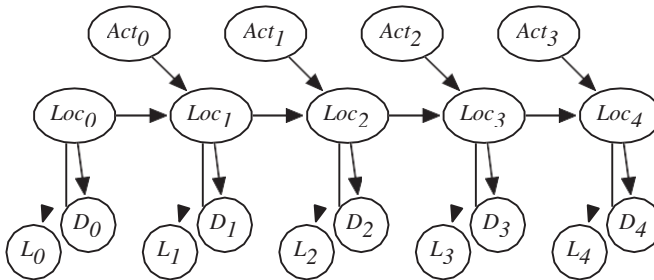


Figure 6.18: Localization with multiple sensors

6.5.3 Algorithms for Monitoring and Smoothing

You can use any standard belief-network algorithms, such as VE or particle filtering, to carry out monitoring or smoothing. However, you can take advantage of the fact that time moves forward and that you are getting observations in time and are interested in the state at the current time.

In **belief monitoring** or **filtering**, an agent computes the probability of the current state given the history of observations. In terms of the HMM of Figure 6.14 (page 267), for each i , the agent wants to compute $P(S_i | o_0, \dots, o_i)$, which is the distribution over the state at time i given the particular observation of $o_0, \dots, o_i$. This can easily be done using VE:

$$\begin{aligned}
 P(S_i | o_0, \dots, o_i) &\propto P(S_i, o_0, \dots, o_i) \\
 &= P(o_i | S_i) P(S_i, o_0, \dots, o_{i-1}) \\
 &= P(o_i | S_i) \sum_{S_{i-1}} P(S_i, S_{i-1}, o_0, \dots, o_{i-1}) \\
 &= P(o_i | S_i) \sum_{S_{i-1}} P(S_i | S_{i-1}) P(S_{i-1}, o_0, \dots, o_{i-1}) \\
 &\propto P(o_i | S_i) \sum_{S_{i-1}} P(S_i | S_{i-1}) P(S_{i-1} | o_0, \dots, o_{i-1}). \tag{6.2}
 \end{aligned}$$

Suppose the agent has computed the previous belief based on the observations received up until time $i-1$. That is, it has a factor representing $P(S_{i-1} | o_0, \dots, o_{i-1})$. Note that this is just a factor on S_{i-1} . To compute the next belief, it multiplies this by $P(S_i | S_{i-1})$, sums out S_{i-1} , multiplies this by the factor $P(o_i | S_i)$, and normalizes.

Multiplying a factor on S_{i-1} by the factor $P(S_i | S_{i-1})$ and summing out S_{i-1} is **matrix multiplication**. Multiplying the result by $P(o_i | S_i)$ is called the **dot product**. Matrix multiplication and dot product are simple instances of VE.

Example 6.30 Consider the domain of Example 6.28 (page 268). An observation of a door involves multiplying the probability of each location L by $P(\text{door} | \text{Loc} = L)$ and renormalizing. A move right involves, for each state, doing a forward simulation of the move-right action in that state weighted by the probability of being in that state.

For many problems the state space is too big for exact inference. For these domains, particle filtering (page 264) is often very effective. With temporal models, resampling typically occurs at every time step. Once the evidence has been observed, and the posterior probabilities of the samples have been computed, they can be resampled.

Smoothing is the problem of computing the probability distribution of a state variable in an HMM given past and future observations. The use of future observations can make for more accurate predictions. Given a new observation it is possible to update all previous state estimates with one sweep through the states using VE; see Exercise 6.11 (page 279).

6.5.4 Dynamic Belief Networks

You do not have to represent the state at a particular time as a single variable. It is often more natural to represent the state in terms of features (page 112).

A **dynamic belief network (DBN)** is a belief network with regular repeated structure. It is like a (hidden) Markov model, but the states and the observations are represented in terms of features. Assume that time is discrete (page 46). If F is a feature, we write F_t as the random variable that represented the value of variable F at time t . A dynamic belief network makes the following assumptions:

- The set of features is the same at each time.
- For any time $t > 0$, the parents of variable F_t are variables at time t or time $t - 1$, such that the graph for any time is acyclic. The structure does not depend on the value of t (except $t = 0$ is a special case).
- The conditional probability distribution of how each variable depends on its parents is the same for every time $t > 0$.

Thus, a dynamic belief network specifies a belief network for time $t = 0$, and for each variable F_t specifies $P(F_t | \text{parents}(F_t))$, where the parents of F_t are in the same or previous time step. This is specified for t as a free parameter; the conditional probabilities can be used for any time $t > 0$. As in a belief network, directed cycles are not allowed.

The model for a dynamic belief network can be represented as a **two-step belief network** that represents the variables at the first two times (times 0 and 1). That is, for each feature F there are two variables, F_0 and F_1 ; $\text{parents}(F_0)$ only include variables for time 0, and $\text{parents}(F_1)$ can be variables at time 0 or 1, as long as the resulting graph is acyclic. Associated with the network are the probabilities $P(F_0 | \text{parents}(F_0))$ and $P(F_1 | \text{parents}(F_1))$. Because of the repeated structure, $P(F_i | \text{parents}(F_i))$, for $i > 1$, has exactly the same structure and the same conditional probability values as $P(F_1 | \text{parents}(F_1))$.

Example 6.31 Suppose the trading agent (page 37) wants to model the dynamics of the price of a commodity such as printer paper. To represent this domain, the designer models what variables affect the price and the other variables. Suppose the cost of pulp and the transportation costs directly affect the price of paper. The transportation costs are affected by the weather. The pulp cost is affected by the prevalence of tree pests, which in turn depend on the weather. Suppose that each variable depends on the values of the previous time step. A two-stage dynamic belief network representing these dependencies is shown in Figure 6.19.

Note that, in this figure, the variables are initially independent.

This two-stage dynamic belief network can be expanded into a regular dynamic belief network by replicating the nodes for each time step, and the parents for future steps are a copy of the parents for the time 1 variables. An expanded belief network is shown in Figure 6.20. The subscripts represent the time that the variable is referring to.

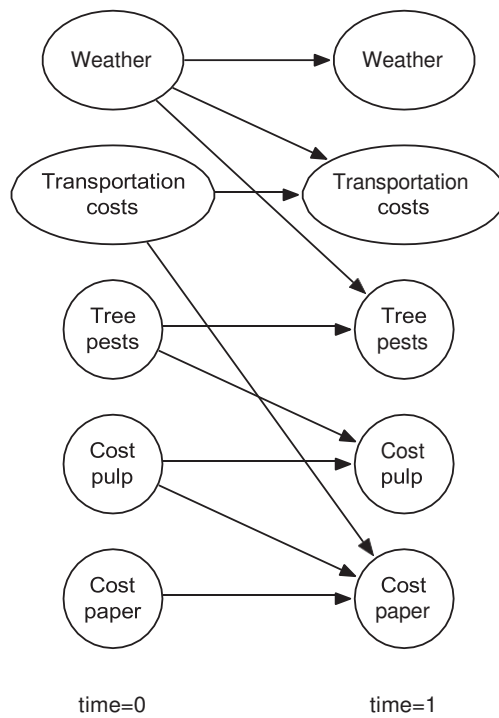


Figure 6.19: Two-stage dynamic belief network for paper pricing

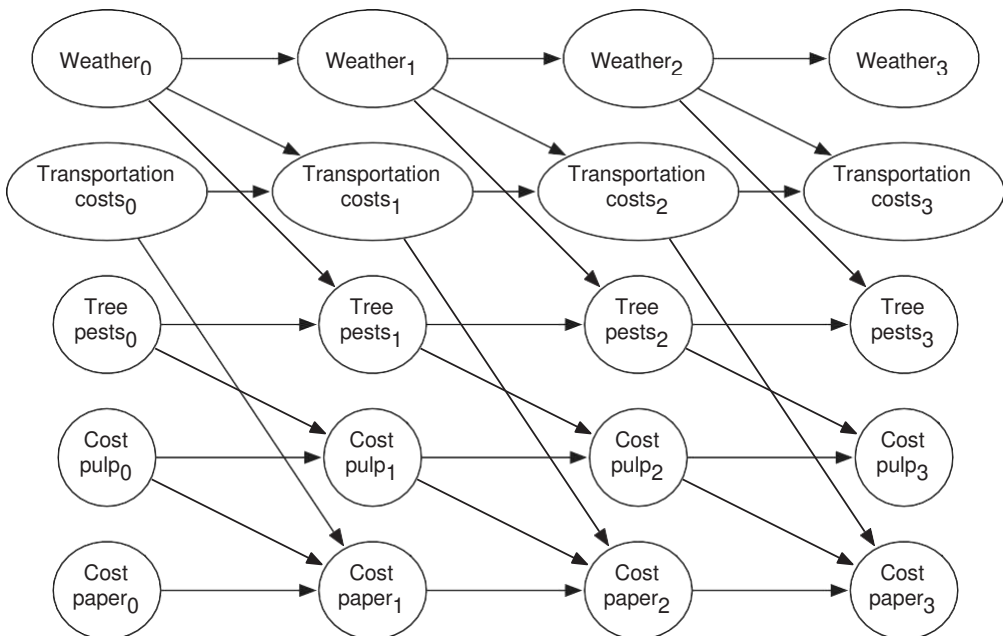


Figure 6.20: Expanded dynamic belief network for paper pricing

6.5.5 Time Granularity

One of the problems with the definition of an HMM or a dynamic belief network is that the model depends on the time granularity. The time granularity can either be fixed, for example each day or each thirtieth of a second, or it can be event-based, where a time step exists when something interesting occurs. If the time granularity were to change, for example from daily to hourly, the conditional probabilities must be changed.

One way to model the dynamics independently of the time granularity is to model, for each variable and each value for the variable,

- how long the variable is expected to keep that value and
- what value it will transition to when its value changes.

Given a discretization of time, and a time model for state transitions, such as an exponential decay, the dynamic belief network can be constructed from this information. If the discretization of time is fine enough, ignoring multiple value transitions in each time step will only result in a small error. See Exercise [6.12](#) (page [279](#)).

6.6 Review

The following are the main points you should have learned from this chapter:

- Probability can be used to make decisions under uncertainty.
- The posterior probability is used to update an agent's beliefs based on evidence.
- A Bayesian belief network can be used to represent independence in a domain.
- Exact inference can be carried out for sparse graphs (with low treewidth).
- Stochastic simulation can be used for approximate inference.
- A hidden Markov model or a dynamic belief network can be used for probabilistic reasoning in time, such as for localization.

6.7 References and Further Reading

Introductions to probability theory from an AI perspective, and belief (Bayesian) networks, are by [Darwiche \[2009\]](#), [\[Koller and Friedman, 2009\]](#), [Pearl \[1988\]](#), [Jensen \[1996\]](#), and [Castillo, Gutiérrez, and Hadi \[1996\]](#). [Halpern \[1997\]](#) reviews the [relationship between logic and probability](#). [Bacchus, Grove, Halpern, and Koller \[1996\]](#) present a random worlds approach to probabilistic reasoning.

Variable [elimination for evaluating belief networks is presented in](#) [Zhang and Poole \[1994\]](#) and [Dechter \[1996\]](#). Treewidth is discussed by [Bodlaender \[1993\]](#).

For comprehensive reviews of information theory, see [Cover and Thomas \[1991\]](#) and [Grünwald \[2007\]](#).

For discussions of causality, see [Pearl \[2000\]](#) and [Spirtes et al. \[2000\]](#).

For introductions to stochastic [simulation](#), see [Rubinstein \[1981\]](#) and [Andrieu, de Freitas, Doucet, and Jordan \[2003\]](#). The forward sampling in belief networks is based on [Henrion \[1988\]](#), who called it logic sampling. The use of importance [sampling in belief networks described here is based on Cheng and Druzdzel \[2000\]](#), who also consider how to learn the proposal distribution. There is [a collection of articles on particle filtering in](#) Doucet, de Freitas, and Gordon [2001].

HMMs are described by [Rabiner \[1989\]](#). Dynamic Bayesian networks were introduced by [Dean and Kanazawa \[1989\]](#). Markov localization and other issues on the relationship [of probability and robotics are described by](#) Thrun, Burgard, and Fox [2005]. The use of particle filtering for localization is due to [Dellaert, Fox, Burgard, and Thrun \[1999\]](#).

The annual Conference on Uncertainty in Artificial Intelligence, and the general AI conferences, provide up-to-date research results.

6.8 Exercises

Exercise 6.1 Using only the axioms of probability and the definition of conditional independence, prove Proposition [6.5](#) (page [233](#)).

Exercise 6.2 Consider the belief network of Figure [6.21](#) (on the next page), which extends the electrical domain to include an overhead projector. Answer the following questions about how knowledge of the values of some variables would affect the probability of another variable:

- (a) Can knowledge of the value of *Projector plugged in* affect your belief in the value of *Sam reading book*? Explain.
- (b) Can knowledge of *Screen lit up* affect your belief in *Sam reading book*? Explain.
- (c) Can knowledge of *Projector plugged in* affect your belief in *Sam reading book* given that you have observed a value for *Screen lit up*? Explain.
- (d) Which variables could have their probabilities changed if just *Lamp works* was observed?
- (e) Which variables could have their probabilities changed if just *Power in projector* was observed?

Exercise 6.3 Represent the same scenario as in Exercise [5.8](#) (page [211](#)) using a belief network. Show the network structure and shade the observed nodes. Give all of the initial factors, making reasonable assumptions about the conditional probabilities (they should follow the story given in that exercise, but allow some noise).

Exercise 6.4 Suppose we want to diagnose the errors school students make when adding multidigit binary numbers. Suppose we are only considering adding two two-digit numbers to form a three-digit number.

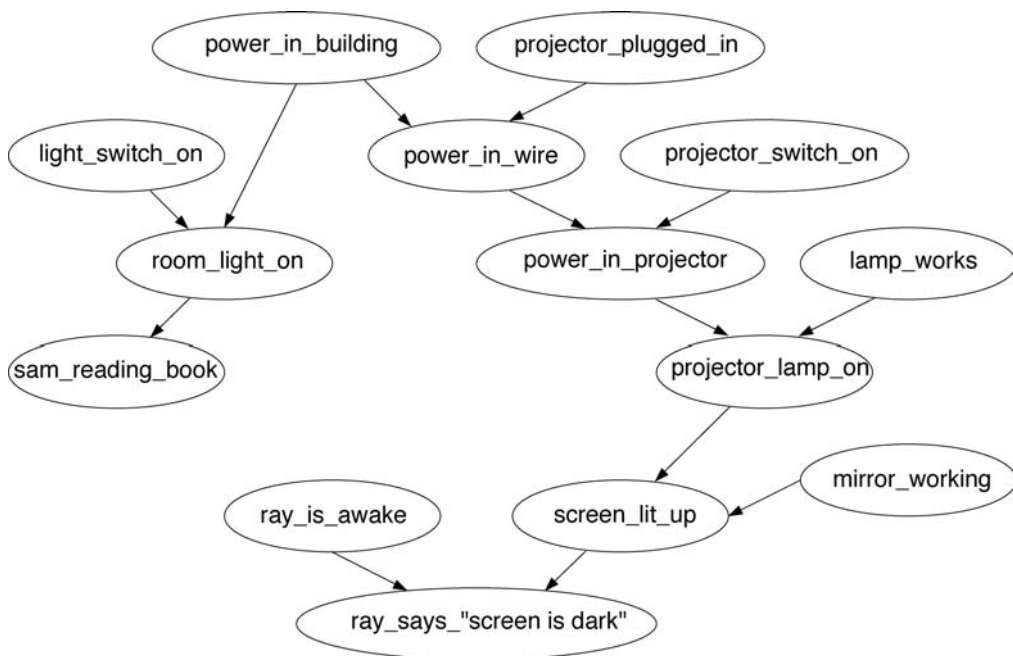


Figure 6.21: Belief network for overhead projector

That is, the problem is of the form:

$$\begin{array}{r} A_1 \ A_0 \\ + \ B_1 \ B_0 \\ \hline C_2 \ C_1 \ C_0 \end{array}$$

where A_i , B_i , and C_i are all binary digits.

- (a) Suppose we want to model whether students know binary addition and whether they know how to carry. If they know how, they usually get the correct answer, but sometimes they make mistakes. If they don't know how to do the appropriate task, they simply guess.

What variables are necessary to model binary addition and the errors students could make? You must specify, in words, what each of the variables represents. Give a DAG that specifies the dependence of these variables.

- (b) What are reasonable conditional probabilities for this domain?
- (c) Implement this, perhaps by using the [Alspace.org](https://alspace.org) belief-network tool. Test your representation on a number of different cases.

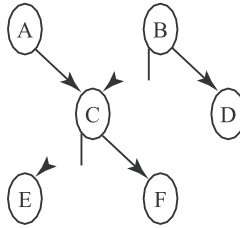
You must give the graph, explain what each variable means, give the probability tables, and show how it works on a number of examples.

Exercise 6.5 In this question, you will build a belief network representation of the Deep Space 1 (DS1) spacecraft considered in Exercise 5.10 (page 212). Figure 5.14 (page 213) depicts a part of the actual DS1 engine design.

Suppose the following scenario:

- Valves can be *open* or *closed*.
 - A valve can be *ok*, in which case the gas will flow if the valve is open and not if it is closed; *broken*, in which case gas never flows; *stuck*, in which case gas flows independently of whether the valve is open or closed; or *leaking*, in which case gas flowing into the valve leaks out instead of flowing through.
 - There are three gas sensors that can detect gas leaking (but not which gas); the first gas sensor detects gas from the rightmost valves ($v1 \dots v4$), the second gas sensor detects gas from the center valves ($v5 \dots v12$), and the third gas sensor detects gas from the leftmost valves ($v13 \dots v16$).
- (a) Build a belief-network representation of the domain. You only must consider the topmost valves (those that feed into engine $e1$). Make sure there are appropriate probabilities.
- (b) Test your model on some non-trivial examples.

Exercise 6.6 Consider the following belief network:



with Boolean variables (we write $A = \text{true}$ as a and $A = \text{false}$ as $\neg a$) and the following conditional probabilities:

$P(a) = 0.9$	$P(d b) = 0.1$
$P(b) = 0.2$	$P(d \neg b) = 0.8$
$P(c a, b) = 0.1$	$P(e c) = 0.7$
$P(c a, \neg b) = 0.8$	$P(e \neg c) = 0.2$
$P(c \neg a, b) = 0.7$	$P(f c) = 0.2$
$P(c \neg a, \neg b) = 0.4$	$P(f \neg c) = 0.9$

- (a) Compute $P(e)$ using VE. You should first prune irrelevant variables. Show the factors that are created for a given elimination ordering.
- (b) Suppose you want to compute $P(e, f)$ using VE. How much of the previous computation can be reused? Show the factors that are different from those in part (a).

Exercise 6.7 Explain how to extend VE to allow for more general observations and queries. In particular, answer the following:

- (a) How can the VE algorithm be extended to allow observations that are disjunctions of values for a variable (e.g., of the form $X = a \vee X = b$)?

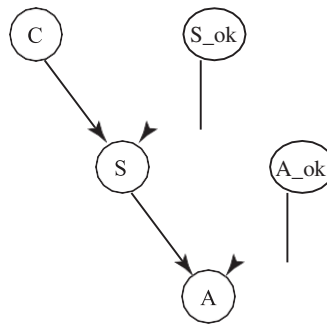


Figure 6.22: Belief network for a nuclear submarine

- (b) How can the VE algorithm be extended to allow observations that are disjunctions of values for different variables (e.g., of the form $X = a \vee Y = b$)?
- (c) How can the VE algorithm be extended to allow for the marginal probability on a set of variables (e.g., asking for the marginal $P(XY|e)$)?

Exercise 6.8 In a nuclear research submarine, a sensor measures the temperature of the reactor core. An alarm is triggered ($A = \text{true}$) if the sensor reading is abnormally high ($S = \text{true}$), indicating an overheating of the core ($C = \text{true}$). The alarm and/or the sensor can be defective ($S_{ok} = \text{false}$, $A_{ok} = \text{false}$), which can cause them to malfunction. The alarm system can be modeled by the belief network of Figure 6.22.

- (a) What are the initial factors for this network? For each factor, state what it represents and what variables it is a function of.
- (b) Show how VE can be used to compute the probability that the core is overheating, given that the alarm does not go off; that is, $P(c|\neg a)$.
For each variable eliminated, show which variable is eliminated, which factor(s) are removed, and which factor(s) are created, including what variables each factor is a function of. Explain how the answer can be derived from the final factor.
- (c) Suppose we add a second, identical sensor to the system and trigger the alarm when either of the sensors reads a high temperature. The two sensors break and fail independently. Give the corresponding extended belief network.

Exercise 6.9 Let's continue Exercise 5.14 (page 215).

- (a) Explain what knowledge (about physics and about students) a belief-network model requires.
- (b) What is the main advantage of using belief networks (over using abductive diagnosis or consistency-based diagnosis) in this domain?
- (c) What is the main advantage of using abductive diagnosis or consistency-based diagnosis compared to using belief networks in this domain?

Exercise 6.10 In importance sampling, every non-observed variable is sampled; a full implementation of VE is not needed. Explain how to compute the probability

of a sample given the evidence in this situation. [Hint: remember that it is possible to sample children as well as parents of observed variables.]

Exercise 6.11 Consider the problem of filtering in HMMs (page [271](#)).

- (a) Give a formula for the probability of some variable X_j given future and past observations. This should involve obtaining a factor from the previous state and a factor from the next state and combining them to determine the posterior probability of X_k . How can the factor needed by X_{j-1} be computed without recomputing the message from X_{j+1} ? [Hint: consider how VE, eliminating from the leftmost variable and eliminating from the rightmost variable, can be used to compute the posterior distribution for X_j .]
- (b) Suppose you have computed the probability distribution for each state $S_1, \dots, S_k$, and then you get an observation for time $k + 1$. How can the posterior probability of each variable be updated in time linear in k ? [Hint: you may need to store more than just the distribution over each S_i .]

Exercise 6.12 Consider the problem of generating a dynamic belief network given a particular discretization of time and given a representation in terms of transition time, and the state transition, as in Section [6.5.5](#) (page [274](#)). Suppose that there is an exponential distribution of how long each variable remains in a state and that the half-life of each variable value is specified. Give the dynamic belief network representation, assuming only a single transition in each time step.

Exercise 6.13 Suppose you get a job where the boss is interested in localization of a robot that is carrying a camera around a factory. The boss has heard of variable elimination, rejection sampling, and particle filtering and wants to know which would be most suitable for this task. You must write a report for your boss (using proper English sentences), explaining which one of these technologies would be most suitable. For the two technologies that are not the most suitable, explain why you rejected them. For the one that is most suitable, explain what information is required by that technology to use it for localization:

- (a) VE (i.e., exact inference as used in HMMs),
- (b) rejection sampling, or
- (c) particle filtering.

Part III

Learning and Planning

Learning: Overview and Supervised Learning

Whoso neglects learning in his youth, loses the past and is dead for the future.

– Euripides (484 BC – 406 BC), Phrixus, Frag. 927

Learning is the ability of an agent to improve its behavior based on experience. This could mean the following:

- The range of behaviors is expanded; the agent can do more.
- The accuracy on tasks is improved; the agent can do things better.
- The speed is improved; the agent can do things faster.

The ability to learn is essential to any intelligent agent. As Euripides pointed, learning involves an agent remembering its past in a way that is useful for its future.

This chapter considers supervised learning: given a set of training examples made up of input-output pairs, predict the output of a new input. We show how such learning may be based on one of four possible approaches: choosing a single hypothesis that fits the training examples well, predicting directly from the training examples, selecting the subset of a hypothesis space consistent with the training examples, or finding the posterior probability distribution of hypotheses conditioned on the training examples.

Chapter [11](#) goes beyond supervised learning and considers clustering (often called unsupervised learning), learning probabilistic models, and reinforcement learning. Section [14.2](#) (page [606](#)) considers learning relational representations.

7.1 Learning Issues

The following components are part of any learning problem:

task The behavior or task that is being improved

data The experiences that are used to improve performance in the task

measure of improvement How the improvement is measured – for example, new skills that were not present initially, increasing accuracy in prediction, or improved speed

Consider the agent internals of Figure 2.9 (page 60). The problem of **learning** is to take in prior knowledge and data (e.g., about the experiences of the agent) and to create an internal representation (the knowledge base) that is used by the agent as it acts.

This internal representation could be the raw experiences themselves, but it is typically a compact representation that summarizes the data. The problem of inferring an internal representation based on examples is often called **induction** and can be contrasted with deduction (page 167), which is deriving consequences of a knowledge base, and abduction (page 199), which is hypothesizing what may be true about a particular case.

There are two principles that are at odds in choosing a representation scheme:

- *The richer the representation scheme, the more useful it is for subsequent problems solving.* For an agent to learn a way to solve a problem, the representation must be rich enough to express a way to solve the problem.
- *The richer the representation, the more difficult it is to learn.* A very rich representation is difficult to learn because it requires a great deal of data, and often many different hypotheses are consistent with the data.

The representations required for intelligence are a compromise between many desiderata [see Section 1.4 (page 11)]. The ability to learn the representation is one of them, but it is not the only one.

Learning techniques face the following issues:

Task Virtually any task for which an agent can get data or experiences can be learned. The most commonly studied learning task is **supervised learning**: given some input features, some target features, and a set of **training examples** where the input features and the target features are specified, predict the target features of a new example for which the input features are given. This is called **classification** when the target variables are discrete and **regression** when the target features are continuous.

Other learning tasks include learning classifications when the examples are not already classified (unsupervised learning), learning what to do based on rewards and punishments (reinforcement learning), learning to reason faster (analytic learning), and learning richer representations such as logic programs (inductive logic programming) or Bayesian networks.

Feedback Learning tasks can be characterized by the feedback given to the learner. In **supervised learning**, what has to be learned is specified for each

example. Supervised classification occurs when a trainer provides the classification for each example. Supervised learning of actions occurs when the agent is given immediate feedback about the value of each action. **Unsupervised learning** occurs when no classifications are given and the learner must discover categories and regularities in the data. Feedback often falls between these extremes, such as in **reinforcement learning**, where the feedback in terms of rewards and punishments occurs after a sequence of actions. This leads to the **credit-assignment problem** of determining which actions were responsible for the rewards or punishments. For example, a user could give rewards to the delivery robot without telling it exactly what it is being rewarded for. The robot then must either learn what it is being rewarded for or learn which actions are preferred in which situations. It is possible that it can learn what actions to perform without actually determining which consequences of the actions are responsible for rewards.

Representation For an agent to use its experiences, the experiences must affect the agent's internal representation. Much of machine learning is studied in the context of particular representations (e.g., decision trees, neural networks, or case bases). This chapter presents some standard representations to show the common features behind learning.

Online and offline In **offline learning**, all of the training examples are available to an agent before it needs to act. In **online learning**, training examples arrive as the agent is acting. An agent that learns online requires some representation of its previously seen examples before it has seen all of its examples. As new examples are observed, the agent must update its representation. Typically, an agent never sees all of its examples. **Active learning** is a form of online learning in which the agent acts to acquire useful examples from which to learn. In active learning, the agent reasons about which examples would be useful to learn from and acts to collect these examples.

Measuring success Learning is defined in terms of improving performance based on some measure. To know whether an agent has learned, we must define a measure of success. The measure is usually not how well the agent performs on the training experiences, but how well the agent performs for new experiences.

In classification, being able to correctly classify all training examples is not the problem. For example, consider the problem of predicting a Boolean feature based on a set of examples. Suppose that there were two agents *P* and *N*. Agent *P* claims that all of the negative examples seen were the only negative examples and that every other instance is positive. Agent *N* claims that the positive examples in the training set were the only positive examples and that every other instance is negative. Both of these agents correctly classify every example in the training set but disagree on every other example. Success in learning should not be judged on correctly classifying the training set but on being able to correctly classify unseen examples. Thus, the learner must **generalize**: go beyond the specific given examples to classify unseen examples.

A standard way to measure success is to divide the examples into a training set and a test set. A representation is built using the training set, and then

the predictive accuracy is measured on the test set. Of course, this is only an approximation of what is wanted; the real measure is its performance on some future task.

Bias The tendency to prefer one hypothesis over another is called a **bias**. Consider the agents N and P defined earlier. Saying that a hypothesis is better than N 's or P 's hypothesis is not something that is obtained from the data – both N and P accurately predict all of the data given – but is something external to the data. Without a bias, an agent will not be able to make any predictions on unseen examples. The hypotheses adopted by P and N disagree on all further examples, and, if a learning agent cannot choose some hypotheses as better, the agent will not be able to resolve this disagreement. To have any inductive process make predictions on unseen data, an agent requires a bias. What constitutes a good bias is an empirical question about which biases work best in practice; we do not imagine that either P 's or N 's biases work well in practice.

Learning as search Given a representation and a bias, the problem of learning can be reduced to one of search. Learning is a search through the space of possible representations, trying to find the representation or representations that best fits the data given the bias. Unfortunately, the search spaces are typically prohibitively large for systematic search, except for the simplest of examples. Nearly all of the search techniques used in machine learning can be seen as forms of local search (page [130](#)) through a space of representations. The definition of the learning algorithm then becomes one of defining the search space, the evaluation function, and the search method.

Noise In most real-world situations, the data are not perfect. Noise exists in the data (some of the features have been assigned the wrong value), there are inadequate features (the features given do not predict the classification), and often there are examples with missing features. One of the important properties of a learning algorithm is its ability to handle noisy data in all of its forms.

Interpolation and extrapolation For cases in which there is a natural interpretation of “between,” such as where the prediction is about time or space, interpolation involves making a prediction between cases for which there are data. Extrapolation involves making a prediction that goes beyond the seen examples. Extrapolation is usually much more inaccurate than interpolation. For example, in ancient astronomy, the Ptolemaic system and heliocentric system of Copernicus made detailed models of the movement of solar system in terms of epicycles (cycles within cycles). The parameters for the models could be made to fit the data very well and they were very good at interpolation; however, the models were very poor at extrapolation. As another example, it is often easy to predict a stock price on a certain day given data about the prices on the days before and the days after that day. It is very difficult to predict the price that a stock will be tomorrow, and it would be very profitable to be able to do so. An agent must be careful if its test cases mostly involve interpolating between data points, but the learned model is used for extrapolation.

Why Should We Believe an Inductive Conclusion?

When learning from data, an agent makes predictions beyond what the data give it. From observing the sun rising each morning, people predict that the sun will rise tomorrow. From observing unsupported objects repeatedly falling, a child may conclude that unsupported objects always fall (until she comes across helium-filled balloons). From observing many swans, all of which were black, someone may conclude that all swans are black. From the data of Figure 7.1 (page 289), the algorithms that follow learn a representation that predicts the user action for a case where the author is unknown, the thread is new, the length is long, and it was read at work. The data do not tell us what the user does in this case. The question arises of why an agent should ever believe any conclusion that is not a logical consequence of its knowledge. When an agent adopts a bias, or chooses a hypothesis, it is going beyond the data – even making the same prediction about a new case that is identical to an old case in all measured respects goes beyond the data. So why should an agent believe one hypothesis over another? By what criteria can it possibly go about choosing a hypothesis?

The most common technique is to choose the simplest hypothesis that fits the data by appealing to **Ockham's razor**. William of Ockham was an English philosopher who was born in about 1285 and died, apparently of the plague, in 1349. (Note that "Occam" is the French spelling of the English town "Ockham" and is often used.) He argued for economy of explanation: "What can be done with fewer [assumptions] is done in vain with more" [Edwards, 1967, Vol. 8, p. 307].

Why should one believe the simplest hypothesis, especially because which hypothesis is simplest depends on the language used to express the hypothesis?

First, it is reasonable to assume that there is structure in the world and that an agent should discover this structure to act appropriately. A reasonable way to discover the structure of the world is to search for it. An efficient search strategy is to search from simpler hypotheses to more complicated ones. If there is no structure to be discovered, nothing will work! The fact that much structure has been found in the world (e.g., all of the structure discovered by physicists) would lead us to believe that this is not a futile search.

The fact that simplicity is language dependent should not necessarily make us suspicious. Language has evolved because it is useful; it allows people to express the structure of the world. Thus, we would expect that simplicity in everyday language would be a good measure of complexity.

The most important reason for believing inductive hypotheses is that it is useful to believe them. They help agents interact with the world and to avoid being killed; an agent that does not learn that it should not fling itself from heights will not survive long. The "simplest hypothesis" heuristic is useful because it works.

7.2 Supervised Learning

An abstract definition of **supervised learning** is as follows. Assume the learner is given the following data:

- a set of **input features**, $X_1, \dots, X_n$;
- a set of **target features**, $Y_1, \dots, Y_k$;
- a set of **training examples**, where the values for the input features and the target features are given for each example; and
- a set of **test examples**, where only the values for the input features are given.

The aim is to predict the values of the target features for the test examples and as-yet-unseen examples. Typically, learning is the creation of a representation that can make predictions based on descriptions of the input features of new examples.

If e is an example, and F is a feature, let $val(e, F)$ be the value of feature F in example e .

Example 7.1 Figure 7.1 shows training and test examples typical of a classification task. The aim is to predict whether a person reads an article posted to a bulletin board given properties of the article. The input features are *Author*, *Thread*, *Length*, and *WhereRead*. There is one target feature, *UserAction*. There are eighteen training examples, each of which has a value for all of the features.

In this data set, $val(e_{11}, Author) = unknown$, $val(e_{11}, Thread) = followUp$, and $val(e_{11}, UserAction) = skips$.

The aim is to predict the user action for a new example given its values for the input features.

The most common way to learn is to have a **hypothesis space** of all possible representations. Each possible representation is a **hypothesis**. The hypothesis space is typically a large finite, or countably infinite, space. A prediction is made using one of the following:

- the best hypothesis that can be found in the hypothesis space according to some measure of better,
- all of the hypotheses that are consistent with the training examples, or
- the posterior probability of the hypotheses given the evidence provided by the training examples.

One exception to this paradigm is in case-based reasoning, which uses the examples directly.

7.2.1 Evaluating Predictions

If e is an example, a **point estimate** for target feature Y is a prediction of a particular value for Y on e . Let $pval(e, Y)$ be the predicted value for target feature Y on example e . The **error** for this example on this feature is a measure of

<i>Example</i>	<i>Author</i>	<i>Thread</i>	<i>Length</i>	<i>WhereRead</i>	<i>UserAction</i>
e_1	known	new	long	home	skips
e_2	unknown	new	short	work	reads
e_3	unknown	follow Up	long	work	skips
e_4	known	follow Up	long	home	skips
e_5	known	new	short	home	reads
e_6	known	follow Up	long	work	skips
e_7	unknown	follow Up	short	work	skips
e_8	unknown	new	short	work	reads
e_9	known	follow Up	long	home	skips
e_{10}	known	new	long	work	skips
e_{11}	unknown	follow Up	short	home	skips
e_{12}	known	new	long	work	skips
e_{13}	known	follow Up	short	home	reads
e_{14}	known	new	short	work	reads
e_{15}	known	new	short	home	reads
e_{16}	known	follow Up	short	work	reads
e_{17}	known	new	short	home	reads
e_{18}	unknown	new	short	work	reads
e_{19}	unknown	new	long	work	?
e_{20}	unknown	follow Up	long	home	?

Figure 7.1: Examples of a user's preferences. These are some training and test examples obtained from observing a user deciding whether to read articles posted to a threaded discussion board depending on whether the author is known or not, whether the article started a new thread or was a follow-up, the length of the article, and whether it is read at home or at work. $e_1, \dots, e_{18}$ are the training examples. The aim is to make a prediction for the user action on e_{19}, e_{20} , and other, currently unseen, examples.

how close $pval(e, Y)$ is to $val(e, Y)$, where $val(e, Y)$ is the actual value for feature Y in e .

For regression, when the target feature Y is real valued, both $pval(e, Y)$ and $val(e, Y)$ are real numbers that can be compared arithmetically.

For classification, when the target feature Y is a discrete variable, a number of alternatives exist:

- When Y is binary, one value can be associated with 0, the other value with 1, and a prediction can be some real number. The predicted and actual values can be compared numerically.
- When the domain of Y has more than two values, sometimes the values are totally ordered and can be scaled so a real number can be associated with each value of the domain of Y . In this case, the predicted and actual values can be compared on this scale. Often, this is not appropriate even when the

values are totally ordered; for example, suppose the values are *short*, *medium*, and *long*. The prediction that the value is *short long* is very different from the prediction that the value is *medium*.

- When the domain of Y is $v_1, \dots, v_k$, where $k > 2$, a separate prediction can be made for each v_i . This can be modeled by having a binary **indicator variable** (page 141) associated with each v_i which, for each example, has value 1 when the example has value v_i and the indicator variable has value 0 otherwise. For each training example, exactly one of the indicator variables associated with Y will be 1 and the others will be 0. A prediction gives k real numbers – one real number for each v_i .

Example 7.2 Suppose the trading agent wants to learn a person's preference for the length of holidays. Suppose the holiday can be for 1, 2, 3, 4, 5, or 6 days.

One representation is to have a real-valued variable Y that is the number of days in the holiday.

Another representation is to have six real-valued variables, $Y_1, \dots, Y_6$, where Y_i represents the proposition that the person would like to stay for i days. For each example, $Y_i=1$ when there are i days in the holiday, and $Y_i=0$ otherwise.

The following is a sample of five data points using the two representations:

Example	Y	Example	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
e_1	1	e_1	1	0	0	0	0	0
e_2	6	e_2	0	0	0	0	0	1
e_3	6	e_3	0	0	0	0	0	1
e_4	2	e_4	0	1	0	0	0	0
e_5	1	e_5	1	0	0	0	0	0

A prediction for a new example in the first representation can be any real number, such as $Y=3.2$.

In the second representation, the learner would predict a value for each Y_i for each example. One such prediction may be $Y_1=0.5$, $Y_2=0.3$, $Y_3=0.1$, $Y_4=0.1$, $Y_5=0.1$, and $Y_6=0.5$. This is a prediction that the person may like 1 day or 6 days, but will not like a stay of 3, 4, or 5 days.

In the following definitions, E is the set of all examples and T is the set of target features.

There are a number of prediction measures that can be defined:

- The **absolute error** on E is the sum of the absolute errors of the predictions on each example. That is,

$$\sum_{e \in E} \sum_{Y \in T} |val(e, Y) - pval(e, Y)|.$$

This is always non-negative, and is only zero when the predictions exactly fit the observed values.

- The **sum-of-squares error** on E is

$$\sum_{e \in E} \sum_{Y \in T} (val(e, Y) - pval(e, Y))^2.$$

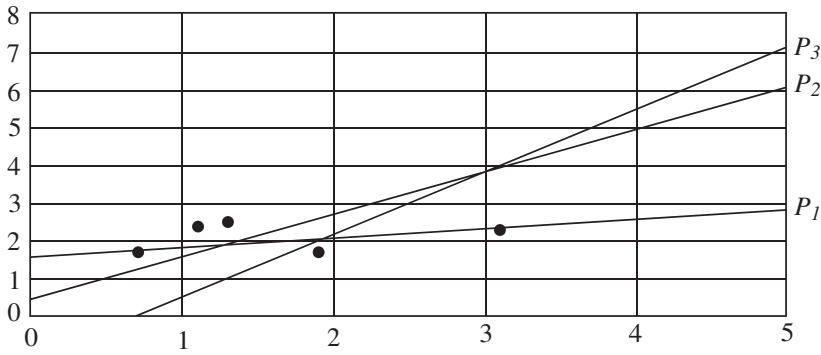


Figure 7.2: Linear predictions for a simple prediction example. Filled circles are the training examples. P_1 is the prediction that minimizes the absolute error of the training examples. P_2 is the prediction that minimizes the sum-of-squares error of the training examples. P_3 is the prediction that minimizes the worst-case error of the training examples. See Example 7.3.

This measure treats large errors as worse than small errors. An error twice as big is four times as bad, and an error 10 times as big is 100 times worse.

- The **worst-case error** on E is the maximum absolute error:

$$\max_{e \in E} \max_{Y \in \mathcal{T}} |val(e, Y) - pval(e, Y)|.$$

In this case, the learner is evaluated by how bad it can be.

Example 7.3 Suppose there is a real-valued target feature, Y , that is based on a single real-valued input feature, X . Suppose the data contains the following (X, Y) points:

$$(0.7, 1.7), (1.1, 2.4), (1.3, 2.5), (1.9, 1.7), (2.6, 2.1), (3.1, 2.3), (3.9, 7).$$

Figure 7.2 shows a plot of the training data (filled circles) and three lines, P_1 , P_2 , and P_3 , that predict the Y -value for all X points. P_1 is the line that minimizes the absolute error, P_2 is the line that minimizes the sum-of-squares error, and P_3 minimizes the worst-case error of the training examples.

Lines P_1 and P_2 give similar predictions for $X=1.1$; namely, P_1 predicts 1.805 and P_2 predicts 1.709, whereas the data contain a data point $(1.1, 2.4)$. P_3 predicts 0.7. They give predictions within 1.5 of each other when interpolating in the range $[1, 3]$. Their predictions diverge when extrapolating from the data. P_1 and P_3 give very different predictions for $X=10$.

The difference between the lines that minimize the various error measures is most pronounced in how they handle the outlier examples, in this case the point $(3.9, 7)$. The other points are approximately in a line.

The prediction with the least worst-case error for this example, P_3 , only depends on three data points, $(1.1, 2.4)$, $(3.1, 2.3)$, and $(3.9, 7)$, each of which has the same worst-case error for prediction P_3 . The other data points could be

at different locations, as long as they are not farther away from P_3 than these three points.

In contrast, the prediction that minimizes the absolute error, P_1 , does not change as a function of the actual Y -value of the training examples, as long as the points above the line stay above the line, and those below the line stay below. For example, the prediction that minimizes the absolute error would be the same, even if the last data point was $(3.9, 107)$ instead of $(3.9, 7)$.

Prediction P_2 is sensitive to all of the data points; if the Y -value for any point changes, the line that minimizes the sum-of-squares error will change.

There are a number of prediction measures that can be used for the special case where the domain of Y is 0, 1, and the prediction is in the range $[0, 1]$. These measures can be used for Boolean domains where *true* is treated as 1, and *false* is treated as 0.

- The **likelihood of the data** is the probability of the data when the predicted value is interpreted as a probability:

$$\prod_{e \in E} \prod_{Y \in T} pval(e, Y)^{val(e, Y)} (1 - pval(e, Y))^{(1 - val(e, Y))}.$$

One of $val(e, Y)$ and $(1 - val(e, Y))$ is 1, and the other is 0. Thus, this product uses $pval(e, Y)$ when $val(e, Y)=1$ and $(1 - pval(e, Y))$ when $val(e, Y)=0$. A better prediction is one with a higher likelihood. The model with the greatest likelihood is the **maximum likelihood model**.

- The **entropy** of the data is the number of bits it will take to encode the data given a code that is based on $pval(e, Y)$ treated as a probability. The entropy is

$$- \sum_{e \in E} \sum_{Y \in T} [val(e, Y) \log pval(e, Y) + (1 - val(e, Y)) \log(1 - pval(e, Y))].$$

A better prediction is one with a lower entropy.

A prediction that minimizes the entropy is a prediction that maximizes the likelihood. This is because the entropy is the negative of the logarithm of the likelihood.

- Suppose the predictions are also restricted to be $\{0, 1\}$. A **false-positive error** is a positive prediction that is wrong (i.e., the predicted value is 1, and the actual value is 0). A **false-negative error** is a negative prediction that is wrong (i.e., the predicted value is 0, and the actual value is 1). Often different costs are associated with the different sorts of errors. For example, if there are data about whether a product is safe, there may be different costs for claiming it is safe when it is not safe, and for claiming it is not safe when it is safe.

We can separate the question of whether the agent has a good learning algorithm from whether it makes good predictions based on preferences that are outside of the learner. The predicting agent can at one extreme choose to only claim a positive prediction when it is sure the prediction is positive. At the other extreme, it can claim a positive prediction unless it is sure the

prediction should be negative. It can often make predictions between these extremes.

One way to test the prediction independently of the decision is to consider the four cases between the predicted value and the actual value:

	actual positive	actual negative
predict positive	true positive (tp)	false positive (fp)
predict negative	false negative (fn)	true negative (tn)

Suppose tp is the number of true positives, fp is the number of false negatives, fn is the number of false negatives, and tn is the number of true negatives. The **precision** is $\frac{tp}{tp+fp}$, which is the proportion of positive predictions that are actual positives. The **recall** or **true-positive rate** is $\frac{tp}{tp+fn}$, which is the proportion of actual positives that are predicted to be positive. The **false-positive error** rate is $\frac{fp}{fp+tn}$ which is the proportion of actual negatives predicted to be positive.

An agent should try to maximize precision and recall and to minimize the false-positive rate; however, these goals are incompatible. An agent can maximize precision and minimize the false-positive rate by only making positive predictions it is sure about. However, this choice worsens recall. To maximize recall, an agent can be risky in making predictions, which makes precision smaller and the false-positive rate larger. The predicting agent often has parameters that can vary a threshold of when to make positive predictions. A **precision-recall curve** plots the precision against the recall as these parameters change. An **ROC curve**, or receiver operating characteristic curve, plots the false-positive rate against the false-negative rate as this parameter changes. Each of these approaches may be used to compare learning algorithms independently of the actual claim of the agent.

- The prediction can be seen as an action of the predicting agent. The agent should choose the action that maximizes a preference function that involves a trade-off among the costs associated with its actions. The actions may be more than true or false, but may be more complex, such as “proceed with caution” or “definitely true.” What an agent should do when faced with uncertainty is discussed in Chapter 9.

Example 7.4 Consider the data of Example 7.2 (page 290). Suppose there are no input features, so all of the examples get the same prediction.

In the first representation, the prediction that minimizes the sum of absolute errors on the training data presented in Example 7.2 (page 290) is 2, with an error of 10. The prediction that minimizes the sum-of-squares error on the training data is 3.2. The prediction the minimizes the worst-case error is 3.5.

For the second representation, the prediction that minimizes the sum of absolute errors for the training examples is to predict 0 for each Y_i . The prediction that minimizes the sum-of-squares error for the training examples is $Y_1=0.4$, $Y_2=0.1$, $Y_3=0$, $Y_4=0$, $Y_5=0$, and $Y_6=0.4$. This is also the prediction

that minimizes the entropy and maximizes the likelihood of the training data. The prediction that minimizes the worst-case error for the training examples is to predict 0.5 for Y_1 , Y_2 , and Y_6 and to predict 0 for the other features.

Thus, whichever prediction is preferred depends on how the prediction will be evaluated.

7.2.2 Point Estimates with No Input Features

The simplest case for learning is when there are no input features and where there is a single target feature. This is the base case for many of the learning algorithms and corresponds to the case where all inputs are ignored. In this case, a learning algorithm predicts a single value for the target feature for all of the examples. The prediction that minimizes the error depends on the error that is being minimized.

Suppose E is a set of examples and Y is a numeric feature. The best an agent can do is to make a single point estimate for all examples. Note that it is possible for the agent to make stochastic predictions, but these are not better; see Exercise 7.2 (page 342).

The sum-of-squares error on E of prediction v is

$$\sum_{e \in E} (\text{val}(e, Y) - v)^2.$$

The absolute error on E of prediction v is

$$\sum_{e \in E} |\text{val}(e, Y) - v|.$$

The worst-case error on E of prediction v is

$$\max_{e \in E} |\text{val}(e, Y) - v|.$$

Proposition 7.1. Suppose V is the multiset of values of $\text{val}(e, Y)$ for $e \in E$.

- (a) The prediction that minimizes the sum-of-squares error on E is the mean of V (the average value).
- (b) The value that minimizes the absolute error is the median of V . In particular, any number v such that there is the same number of values of V less than v as there are values greater than v minimizes the error.
- (c) The value that minimizes the worst-case error is $(\max + \min)/2$, where $\max$ is the maximum value and $\min$ is the minimum value.

Proof. The details of the proof are left as an exercise. The basic idea follows:

- (a) Differentiate the formula for the sum-of-squares error with respect to v and set to zero. This is elementary calculus. To make sure the point(s) with a derivative of zero is(are) a minimum, the end points also must be checked.

Prediction measure	Measure of prediction p for the training data	Optimal prediction for training data
absolute error	$n_0p + n_1(1 - p)$	$\frac{n_1}{n_0 + n_1}$
sum squares	$n_0p^2 + n_1(1 - p)^2$	$\frac{n_1}{n_0 + n_1}$
worst case	$\square$ p if $n_1 = 0$ $\square$ $1 - p$ if $n_0 = 0$ $\square$ $\max(p, 1 - p)$ otherwise	$\square$ 0 if $n_1 = 0$ $\square$ 1 if $n_0 = 0$ $\square$ 0.5 otherwise
likelihood	$p^{n_1}(1 - p)^{n_0}$	$\frac{n_1}{n_0 + n_1}$
entropy	$-n_1 \log p - n_0 \log(1 - p)$	$\frac{n_1}{n_0 + n_1}$

Figure 7.3: Optimal prediction for binary classification where the training data consist of n_0 examples of 0 and n_1 examples of 1, with no input features. $\text{median}(n_0, n_1)$ is 0 if $n_0 > n_1$, 1 if $n_0 < n_1$, and any value in $[0, 1]$ if $n_0 = n_1$.

- (b) The absolute error is a piecewise linear function of v . The slope for a value that is not in V depends on the number of elements greater minus the number of elements less than that value: v is a minimum if there are the same number of elements greater than v as there are less than v .
- (c) This prediction has an error of $(\text{max\_min})/2$; increasing or decreasing the prediction will increase the error. $\square$

When the target feature has domain $\{0, 1\}$, the training examples can be summarized in two numbers: n_0 , the number of examples with the value 0, and n_1 , the number of examples with value 1. The prediction for each new case is the same number, p .

The optimal prediction p depends on the optimality criteria. The value of the optimality criteria for the training examples can be computed analytically and can be optimized analytically. The results are summarized in Figure 7.3.

Notice that optimizing the absolute error means predicting the median, which in this case is also the mode; this should not be surprising because the error is linear in p .

The optimal prediction for the training data for the other criteria is to predict the **empirical frequency**: the proportion of 1's in the training data, namely $\frac{n_1}{n_0 + n_1}$. This can be seen as a prediction of the **probability**. The empirical fre-

quency is often called the **maximum-likelihood estimate**.

This analysis does not specify the optimal prediction for the test data. We would *not* expect the empirical frequency of the training data to be the optimal prediction for the test data for maximizing the likelihood or minimizing the entropy. If $n_0 = 0$ or if $n_1 = 0$, all of the training data are classified the same. However, if just one of the test examples is not classified in this way, the likelihood would be 0 (its lowest possible value) and the entropy would be infinite. See

Exercise [1](#) (page [342](#)).

7.2.3 Learning Probabilities

For many of the prediction measures, the optimal prediction on the training data is the empirical frequency. Thus, making a point estimate can be interpreted as learning a probability. However, the empirical frequency is typically not a good estimate of the probability of new cases; just because an agent has not observed some value of a variable does not mean that the value should be assigned a probability of zero. A probability of zero means that the value is impossible.

Typically, we do not have data without any prior knowledge. There is typically a great deal of knowledge about a domain, either in the meaning of the symbols or in experience with similar examples that can be used to improve predictions.

A standard way both to solve the zero-probability problem and to take prior knowledge into account is to use a **pseudocount** or **prior count** for each value to which the training data is added.

Suppose there is a binary feature Y , and an agent has observed n_0 cases where $Y=0$ and n_1 cases where $Y=1$. The agent can use a pseudocount $c_0 \geq 0$ for $Y=0$ and a pseudocount $c_1 \geq 0$ for $Y=1$ and estimate the probability as

$$P(Y=1) = (n_1 + c_1) / (n_0 + c_0 + n_1 + c_1).$$

This takes into account both the data and the prior knowledge. This formula can be justified in terms of a prior on the parameters [see Section 7.8 (page 334)]. Choosing pseudocounts is part of designing the learner.

More generally, suppose Y has domain $\{y_1, \dots, y_k\}$. The agent starts with a pseudocount c_i for each y_i . These counts are chosen before the agent has seen any of the data. Suppose the agent observes some training examples, where n_i is the number of data points with $Y=y_i$. It can then use

$$P(Y=y_i) = \frac{c_i + n_i}{\sum_{it} c_{it} + n_{it}}.$$

To determine the pseudocounts, consider the question, “How much more should an agent believe y_i if it had seen one example with y_i true than if it had seen no examples with y_i true?” If, with no examples of y_i true, the agent believes that y_i is impossible, c_i should be zero. If not, the ratio chosen in answer to that question should be equal to the ratio $(1 + c_i) : c_i$. If the pseudocount is 1, a value that has been seen once would be twice as likely as one that has been seen no times. If the pseudocount is 10, a value observed once would be 10% more likely than a value observed no times. If the pseudocount is 0.1, a value observed once would be 11 times more likely than a value observed no times. If there is no reason to choose one value in the domain of Y over another, all of the values of c_i should be equal.

If there is no prior knowledge, [Laplace \[1812\]](#) suggested that it is reasonable to set $c_i=1$. See Section 7.8 (page 334) for a justification of why this may be reasonable. We will see examples where it is not appropriate, however.

The same idea can be used to learn a **conditional probability distribution** (page 227). To estimate a conditional distribution, $P(Y|X)$, of variable Y conditioned on variable(s) X , the agent can maintain a count for each pair of a value for Y and a value for X . Suppose c_{ij} is a non-negative number that will be used as a pseudocount for $Y=y_i \wedge X=x_j$. Suppose n_{ij} is the number of observed cases of $Y=y_i \wedge X=x_j$. The agent can use

$$P(Y=y_i | X=x_j) = \frac{c_{ij} + n_{ij}}{\sum_i c_{ij} + n_{ij}}$$

but this does not work well when the denominator is small, which occurs when some values of X are rare. When X has structure – for example, when it is composed of other variables – it is often the case that some assignments to X are very rare or even do not appear in the training data. In these cases, the learner must use other methods that are discussed in this chapter.

Probabilities from Experts

The use of pseudocounts also gives us a way to combine **expert opinion** and data. Often a single agent does not have good data but may have access to multiple experts who have varying levels of expertise and who give different probabilities.

There are a number of problems with obtaining probabilities from experts:

- experts' reluctance to give an exact probability value that cannot be refined,
- representing the uncertainty of a probability estimate,
- combining the numbers from multiple experts, and
- combining expert opinion with actual data.

Rather than expecting experts to give probabilities, the experts can provide counts. Instead of giving a real number such as 0.667 for the probability of A , an expert can give a pair of numbers as n, m that can be interpreted as though the expert had observed n A 's out of m trials. Essentially, the experts provide not only a probability but also an estimate of the size of the data set on which their opinion is based.

The counts from different experts can be combined together by adding the components to give the pseudocounts for the system. Whereas the ratio reflects the probability, different levels of confidence can be reflected in the absolute values: 2, 3 reflects extremely low confidence that would quickly be dominated by data or other experts' estimates. The pair 20, 30 reflects more confidence – a few examples would not change it much, but tens of examples would. Even hundreds of examples would have little effect on the prior counts of the pair 2000, 3000. However, with millions of data points, even these prior counts would have little impact on the resulting probability.

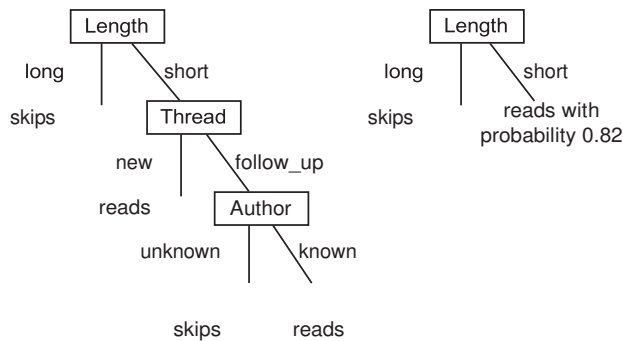


Figure 7.4: Two decision trees

7.3 Basic Models for Supervised Learning

A learned model is a representation of a function from the input features to the target features. Most supervised learning methods take the input features, the target features, and the training data and return a model that can be used for future prediction. Many of the learning methods differ in what representations are considered for representing the function. We first consider some basic models from which other composite models are built. Section 7.4 (page 313) considers more sophisticated models that are built from these basic models.

7.3.1 Learning Decision Trees

A decision tree is a simple representation for classifying examples. Decision tree learning is one of the most successful techniques for supervised classification learning. For this section, assume that all of the features have finite discrete domains, and there is a single target feature called the **classification**. Each element of the domain of the classification is called a **class**.

A **decision tree** or a **classification tree** is a tree in which each internal (non-leaf) node is labeled with an input feature. The arcs coming from a node labeled with a feature are labeled with each of the possible values of the feature. Each leaf of the tree is labeled with a class or a probability distribution over the classes.

To classify an example, filter it down the tree, as follows. For each feature encountered in the tree, the arc corresponding to the value of the example for that feature is followed. When a leaf is reached, the classification corresponding to that leaf is returned.

Example 7.5 Figure 7.4 shows two possible decision trees for the example of Figure 7.1 (page 289). Each decision tree can be used to classify examples according to the user's action. To classify a new example using the tree on the left, first determine the length. If it is long, predict skips. Otherwise, check the thread. If the thread is new, predict reads. Otherwise, check the author and

predict read only if the author is known. This decision tree can correctly classify all examples in Figure 7.1 (page 289).

The tree on the right makes probabilistic predictions when the length is short. In this case, it predicts reads with probability 0.82 and so skips with probability 0.18.

A deterministic decision tree, in which all of the leaves are classes, can be mapped into a set of rules, with each leaf of the tree corresponding to a rule. The example has the classification at the leaf if all of the conditions on the path from the root to the leaf are true.

Example 7.6 The leftmost decision tree of Figure 7.4 can be represented as the following rules:

$skips \leftarrow long.$
 $reads \leftarrow short \wedge new.$
 $reads \leftarrow short \wedge followUp \wedge known.$
 $skips \leftarrow short \wedge followUp \wedge unknown.$

With negation as failure (page 194), the rules for either *skips* or *reads* can be omitted, and the other can be inferred from the negation.

To use decision trees as a target representation, there are a number of questions that arise:

- Given some training examples, what decision tree should be generated? Because a decision tree can represent any function of the input features, the bias that is necessary to learn is incorporated into the preference of one decision tree over another. One proposal is to prefer the smallest tree that is consistent with the data, which could mean the tree with the least depth or the tree with the fewest nodes. Which decision trees are the best predictors of unseen data is an empirical question.
- How should an agent go about building a decision tree? One way is to search the space of decision trees for the smallest decision tree that fits the data. Unfortunately the space of decision trees is enormous (see Exercise 7.7 (page 344)). A practical solution is to carry out a local search on the space of decision trees, with the goal of minimizing the error. This is the idea behind the algorithm described below.

Searching for a Good Decision Tree

A decision tree can be incrementally built from the top down by recursively selecting a feature to split on and partitioning the training examples with respect to that feature. In Figure 7.5 (on the next page), the procedure *DecisionTreeLearner* learns a decision tree for binary attributes. The decisions regarding when to stop and which feature to split on are left undefined. The procedure *DecisionTreeClassify* takes in a decision tree produced by the learner and makes predictions for a new example.

```

1: procedure DecisionTreeLerner( $X, Y, E$ )
2:   Inputs
3:      $X$ : set of input features,  $X = \{X_1, \dots, X_n\}$ 
4:      $Y$ : target feature
5:      $E$ : set of training examples
6:   Output
7:     decision tree
8:   if stopping criterion is true then
9:     return pointEstimate( $Y, E$ )
10:  else
11:    Select feature  $X_i \in X$ , with domain  $\{v_1, v_2\}$ 
12:    let  $E_1 = \{e \in E : \text{val}(e, X_i) = v_1\}$ 
13:    let  $T_1 = \text{DecisionTreeLerner}(X \setminus \{X_i\}, Y, E_1)$ 
14:    let  $E_2 = \{e \in E : \text{val}(e, X_i) = v_2\}$ 
15:    let  $T_2 = \text{DecisionTreeLerner}(X \setminus \{X_i\}, Y, E_2)$ 
16:    return ( $X_i = v_1, T_1, T_2$ )
17:
18: procedure DecisionTreeClassify( $e, X, Y, DT$ )
19:   Inputs
20:      $X$ : set of input features,  $X = \{X_1, \dots, X_n\}$ 
21:      $Y$ : target feature
22:      $e$ : example to classify
23:      $DT$ : decision tree
24:   Output
25:     prediction on  $Y$  for example  $e$ 
26:   Local
27:      $S$  subtree of  $DT$ 
28:      $S := DT$ 
29:   while  $S$  is an internal node of the form  $X_i = v, T_1, T_2$  do
30:     if  $\text{val}(e, X_i) = v$  then
31:        $S := T_1$ 
32:     else
33:        $S := T_2$ 
34:   return  $S$ 

```

Figure 7.5: Decision tree learning and classification for binary features

The algorithm *DecisionTreeLerner* builds a decision tree from the top down as follows: The input to the algorithm is a set of input features, a target feature, and a set of examples. The learner first tests if some stopping criterion is true. If the stopping criterion is true, it returns a point estimate (page 288) for Y , which is either a value for Y or a probability distribution over the values for Y . If the stopping criterion is not true, the learner selects a feature X_i to split

on, and for each value v of this feature, it recursively builds a subtree for those examples with $X_i=v$. The returned tree is represented here in terms of triples representing an if-then-else structure.

Example 7.7 Consider applying *DecisionTreeLearner* to the classification data of Figure 7.1 (page 289). The initial call is

decisionTreeLearner([*Author*, *Thread*, *Length*, *WhereRead*], *UserAction*,
[$e_1, e_2, \dots, e_{18}$]).

Suppose the stopping criterion is not true and the algorithm selects the feature *Length* to split on. It then calls

decisionTreeLearner([*WhereRead*, *Thread*, *Author*], *UserAction*,
[$e_1, e_3, e_4, e_6, e_9, e_{10}, e_{12}$]).

All of these examples agree on the user action; therefore, the algorithm returns the prediction *skips*. The second step of the recursive call is

decisionTreeLearner([*WhereRead*, *Thread*, *Author*], *UserAction*,
[$e_2, e_5, e_7, e_8, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}$]).

Not all of the examples agree on the user action, so the algorithm selects a feature to split on. Suppose it selects *Thread*. Eventually, this recursive call returns the subtree for the case when *Length* is *short*, such as

(*Thread*=*new*, *reads*, (*Author*=*unknown*, *skips*, *reads*)) .

The final result is

(*Length*=*long*, *skips*,
(*Thread*=*new*, *reads*, (*Author*=*unknown*, *skips*, *reads*))) ,

which is a representation of the tree of Figure 7.4 (page 298).

The learning algorithm of Figure 7.5 leaves three choices unspecified:

- The stopping criterion is not defined. The learner should stop when there are no input features, when all of the examples have the same classification, or when no splitting would improve the classification ability of the resulting tree. The last is the most difficult criterion to test for; see below.
- What should be returned at the leaves is not defined. This is a point estimate (page 288) because, at this step, all of the other input features are ignored. This prediction is typically the most likely classification, the median or mean value, or a probability distribution over the classifications. [See Exercise 7.9 (page 345).]
- Which feature to select to split on is not defined. The aim is to choose the feature that will result in the smallest tree. The standard way to do this is to choose the **myopically optimal** split: if the learner were only allowed one split, which single split would result in the best classification? With the sum-of-squares error, for each feature, determine the error of the resulting tree based on a single split on that feature. For the likelihood or the entropy, the

myopically optimal split is the one that gives the maximum **information gain** (page 232). Sometimes information gain is used even when the optimality criterion is the sum-of-squares error. An alternative, the **Gini index**, is investigated in Exercise 7.10 (page 345).

Example 7.8 Consider learning the user action from the data of Figure 7.1 (page 289), where we split on the feature with the maximum information gain or we myopically choose the split that minimizes the entropy or maximizes the likelihood of the data. See Section 6.1.5 (page 231) for the definition of information used here.

The information content of all examples with respect to feature *UserAction* is 1.0, because there are 9 examples with *UserAction=reads* and 9 examples with *UserAction=skips*.

Splitting on *Author* partitions the examples into $[e_1, e_4, e_5, e_6, e_9, e_{10}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}]$ with *Author=known* and $[e_2, e_3, e_7, e_8, e_{11}, e_{18}]$ with *Author=unknown*, each of which is evenly split between the different user actions. The information gain for the test *Author* is zero. In this case, finding out whether the author is known, by itself, provides no information about what the user action will be.

Splitting on *Thread* partitions the examples into $[e_1, e_2, e_5, e_8, e_{10}, e_{12}, e_{14}, e_{15}, e_{17}, e_{18}]$ and $[e_3, e_4, e_6, e_7, e_9, e_{11}, e_{13}, e_{16}]$. The first set of examples, those with *Thread=new*, contains 3 examples with *UserAction=skips* and 7 examples with *UserAction=reads*; thus, the information content of this set with respect to the user action is

$$-0.3 \times \log_2 0.3 - 0.7 \times \log_2 0.7 = 0.881$$

and so the information gain is 0.119.

Similarly, the examples with *Thread=old* divide up 6 : 2 according to the user action and thus have information content 0.811. The expected information gain is thus $1.0 \left[\frac{10}{18} \cdot 0.881 + \frac{8}{18} \cdot 0.811 \right] = 0.150$.

The test *Length* divides the examples into $[e_1, e_3, e_4, e_6, e_9, e_{10}, e_{12}]$ and $[e_2, e_5, e_7, e_8, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}]$. The former all agree on the value of *UserAction* and so have information content zero. The user action divides the second set 9 : 2, and so the information is 0.684. Thus, the expected information gain by the test *length* is $1.0 \frac{11}{18} \cdot 0.684 = 0.582$. This is the highest information gain of any test and so *Length* is chosen to split on.

In choosing which feature to split on, the information content before the test is the same for all tests, and so the learning agent can choose the test that results in the minimum expected information after the test.

The algorithm of Figure 7.5 (page 300) assumes each input feature has only two values. This restriction can be lifted in two ways:

- Allow a multiway split. To split on a multivalued variable, there would be a child for each value in the domain of the variable. This means that the representation of the decision tree becomes more complicated than the simple if-then-else form used for binary features. There are two main problems with this approach. The first is what to do with values for which

there are no training examples. The second is that, for most myopic splitting heuristics, including information gain, it is generally better to split on a variable with a larger domain because it produces more children and so can fit the data better than splitting on a feature with a smaller domain. [See Exercise 7.8 (page 344).] However, splitting on a feature with a smaller domain keeps the representation more compact.

- Partition the domain into two disjoint subsets. When the domain is totally ordered, such as if the domain is a subset of the real numbers, the domain can be split into values less than some threshold and those greater than the threshold. For example, the children could correspond to $X < v$ and $X \geq v$ for some value v in the domain of X . A myopically optimal value for v can be chosen in one sweep through the data by sorting the data on the value of X and considering each split that partitions the values. When the domain does not have a natural ordering, a split can be performed on arbitrary subsets of the domain. In this case, the myopically optimal split can be found by sorting values that appear in the data on the probability of classification.

If there is noise in the data, a major problem of the preceding algorithm is **overfitting** the data. Overfitting occurs when the algorithm tries to fit distinctions that appear in the training data but do not appear in the unseen examples. This occurs when random correlations exist in the training data that are not reflected in the data set as a whole. Section 7.5 (page 320) discusses ways to detect overfitting. There are two ways to overcome the problem of overfitting in decision trees:

- Restrict the splitting to split only when the split is useful.
- Allow unrestricted splitting and then prune the resulting tree where it makes unwarranted distinctions.

The second method seems to work better in practice. One reason is that it is possible that two features together predict well but one of them, by itself, is not very useful, as shown in the following example.

Example 7.9 Suppose the aim is to predict whether a game of matching pennies is won or not. The input features are A , whether the first coin is heads or tails; B , whether the second coin is heads or tails; and C , whether there is cheering. The target feature, W , is true when there is a win, which occurs when both coins are heads or both coins are tails. Suppose cheering is correlated with winning. This example is tricky because A by itself provides no information about W , and B by itself provides no information about W . However, together they perfectly predict W . A myopic split may first split on C , because this provides the most myopic information. If all the agent is told is C , this is much more useful than A or B . However, if the tree eventually splits on A and B , the split on C is not needed. Pruning can remove C as being useful, whereas stopping early will keep the split on C .

A discussion of how to trade off model complexity and fit to the data is presented in Section 7.5 (page 320).

7.3.2 Linear Regression and Classification

Linear functions provide the basis for many learning algorithms. In this section, we first cover regression – the problem of predicting a real-valued function from training examples. Then we consider the discrete case of classification.

Linear regression is the problem of fitting a linear function to a set of input-output pairs given a set of training examples, in which the input and output features are numeric.

Suppose the input features are $X_1, \dots, X_n$. A **linear function** of these features is a function of the form

$$f^{\bar{w}}(X_1, \dots, X_n) = w_0 + w_1 \times X_1 + \dots + w_n \times X_n,$$

where $\bar{w} = (w_0, w_1, \dots, w_n)$ is a tuple of weights. To make w_0 not be a special case, we invent a new feature, X_0 , whose value is always 1.

We will learn a function for each target feature independently, so we consider only one target, Y . Suppose a set E of examples exists, where each example $e \in E$ has values $val(e, X_i)$ for feature X_i and has an observed value $val(e, Y)$.

The predicted value is thus

$$\begin{aligned} pval^{\bar{w}}(e, Y) &= w_0 + w_1 \times val(e, X_1) + \dots + w_n \times val(e, X_n) \\ &= \sum_{i=0}^n w_i \times val(e, X_i), \end{aligned}$$

where we have made it explicit that the prediction depends on the weights, and where $val(e, X_0)$ is defined to be 1.

The sum-of-squares error on examples E for target Y is

$$\begin{aligned} Error_E(\bar{w}) &= \sum_{e \in E} (val(e, Y) - pval^{\bar{w}}(e, Y))^2 \\ &= \sum_{e \in E} \left(val(e, Y) - \sum_{i=0}^n w_i \times val(e, X_i) \right)^2. \end{aligned} \quad (7.1)$$

In this linear case, the weights that minimize the error can be computed analytically [see Exercise 7.5 (page 344)]. A more general approach, which can be used for wider classes of functions, is to compute the weights iteratively.

Gradient descent (page 149) is an iterative method to find the minimum of a function. Gradient descent starts with an initial set of weights; in each step, it decreases each weight in proportion to its partial derivative:

$$w_i := w_i - \eta \times \frac{\partial Error_E(\bar{w})}{\partial w_i}$$

where η , the gradient descent step size, is called the **learning rate**. The learning rate, as well as the features and the data, is given as input to the learning algorithm. The partial derivative specifies how much a small change in the weight would change the error.

```

1: procedure LinearLerner( $X, Y, E, \eta$ )
2:   Inputs
3:      $X$ : set of input features,  $X = \{X_1, \dots, X_n\}$ 
4:      $Y$ : target feature
5:      $E$ : set of examples from which to learn
6:      $\eta$ : learning rate
7:   Output
8:     parameters  $w_0, \dots, w_n$ 
9:   Local
10:     $w_0, \dots, w_n$ : real numbers
11:     $pval^w(e, Y) = w_0 + w_1 \times val(e, X_1) + \dots + w_n \times val(e, X_n)$ 
12:    initialize  $w_0, \dots, w_n$  randomly
13:    repeat
14:      for each example  $e$  in  $E$  do
15:         $\delta := val(e, Y) - pval^w(e, Y)$ 
16:        for each  $i \in [0, n]$  do
17:           $w_i := w_i + \eta \times \delta \times val(e, X_i)$ 
18:    until termination
19:    return  $w_0, \dots, w_n$ 

```

Figure 7.6: Gradient descent for learning a linear function

Consider minimizing the sum-of-squares error. The error is a sum over the examples. The partial derivative of a sum is the sum of the partial derivatives. Thus, we can consider each example separately and consider how much it changes the weights. The error with respect to example e has a partial derivative with respect to weight of w_i of $2[val(e, Y) - pval^w(e, Y)]val(e, X_i)$. For each example e , let $\delta = val(e, Y) - pval^w(e, Y)$. Thus, each example e updates each weight w_i :

$$w_i := w_i + \eta \times \delta \times val(e, X_i), \quad (7.2)$$

where we have ignored the constant 2, because we assume it is absorbed into the constant η .

Figure 7.6 gives an algorithm, *LinearLerner*(X, Y, E, η), for learning a linear function for minimizing the sum-of-squares error. Note that, in line 17, $val(e, X_0)$ is 1 for all e . Termination is usually after some number of steps, when the error is small or when the changes get small.

The algorithm presented in Figure 7.6 is sometimes called **incremental gradient descent** because of the weight change while it iterates through the examples. An alternative is to save the weights at each iteration of the while loop, use the saved weights for computing the function, and then update these saved weights after all of the examples. This process computes the true derivative of the error function, but it is more complicated and often does not work as well.

The same algorithm can be used for other error functions. For the absolute error, which is not actually differentiable at zero, the derivative can be defined to be zero at that point because the error is already at a minimum and the parameters do not have to change. See Exercise 7.12 (page 346).

Squashed Linear Functions

The use of a linear function does not work well for classification tasks. When there are only two values, say 0 and 1, a learner should never make a prediction of greater than 1 or less than 0. However, a linear function could make a prediction of, say, 3 for one example just to fit other examples better.

Initially let's consider binary classification, where the domain of the target variable is $\{0, 1\}$. If multiple binary target variables exist, they can be learned separately.

For classification, we often use a **squashed linear function** of the form

$$f^{\bar{w}}(X_1, \dots, X_n) = f(w_0 + w_1 \times X_1 + \dots + w_n \times X_n),$$

where f is an **activation function**, which is a function from real numbers into $[0, 1]$. Using a squashed linear function to predict a value for the target feature means that the prediction for example e for target feature Y is

$$pval^{\bar{w}}(e, Y) = f(w_0 + w_1 \times val(e, X_1) + \dots + w_n \times val(e, X_n)).$$

A simple activation function is the **step function**, $f(x)$, defined by

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

A step function was the basis for the **perceptron** [Rosenblatt, 1958], which was one of the first methods developed for learning. It is difficult to adapt gradient descent to step functions because gradient descent takes derivatives and step functions are not differentiable.

If the activation is differentiable, we can use gradient descent to update the weights. The sum-of-squares error is

$$Error_E(w) = \sum_{e \in E} \frac{1}{2} (val(e, Y) - f(\sum_i w_i \times val(e, X_i)))^2.$$

The partial derivative with respect to weight w_i for example e is

$$\frac{\partial Error_E(w)}{\partial w_i} = -2 \times \delta \times f'(\sum_i w_i \times val(e, X_i)) \times val(e, X_i),$$

where $\delta = val(e, Y) - pval^{\bar{w}}(e, Y)$, as before. Thus, each example e updates each weight w_i as follows:

$$w_i := w_i + \eta \times \delta \times f'(\sum_i w_i \times val(e, X_i)) \times val(e, X_i).$$

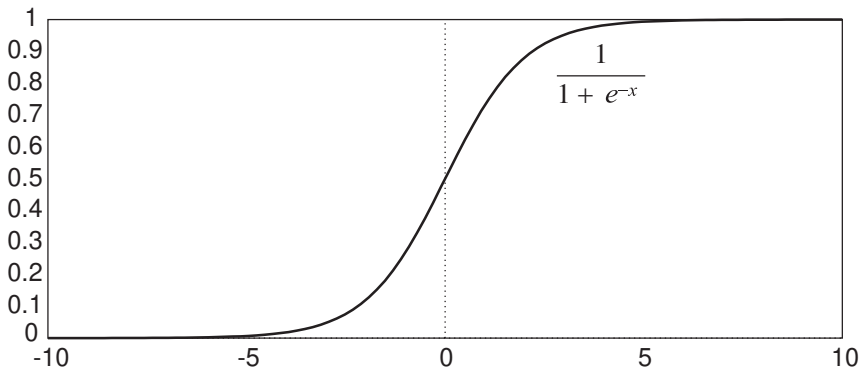


Figure 7.7: The sigmoid or logistic function

A typical differentiable activation function is the **sigmoid** or **logistic function**:

$$f(x) = \frac{1}{1 + e^{-x}}.$$

This function, depicted in Figure 7.7, squashes the real line into the interval $(0, 1)$, which is appropriate for classification because we would never want to make a prediction of greater than 1 or less than 0. It is also differentiable, with a simple derivative - namely, $f'(x) = f(x) \times (1 - f(x))$ - which can be computed using just the values of the outputs.

The *LinearLearner* algorithm of Figure 7.6 (page 305) can be changed to use the sigmoid function by changing line 17 to

$$w_i := w_i + \eta \times \delta \times pval^{\bar{w}}(e, Y) \times [1 - pval^{\bar{w}}(e, Y)] \times val(e, X_i).$$

where $pval^{\bar{w}}(e, Y) = f(\sum_i w_i val(e, X_i))$ is the predicted value of feature Y for example e .

Example 7.10 Consider learning a squashed linear function for classifying the data of Figure 7.1 (page 289). One function that correctly classifies the examples is

$$Reads = f(-8 + 7 \times Short + 3 \times New + 3 \times Known),$$

where f is the sigmoid function. A function similar to this can be found with about 3,000 iterations of gradient descent with a learning rate $\eta = 0.05$. According to this function, *Reads* is true (the predicted value is closer to 1 than 0) if and only if *Short* is true and either *New* or *Known* is true. Thus, the linear classifier learns the same function as the decision tree learner. To see how this works, see the “mail reading” example of the Neural Alspace.org applet.

This algorithm with the sigmoid function as the activation function can learn any linearly separable classification in the sense that the error can be

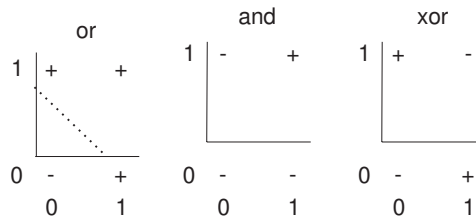


Figure 7.8: Linear separators for Boolean functions

made arbitrarily small on arbitrary sets of examples if, and only if, the target classification is linearly separable. A classification is **linearly separable** if there exists a hyperplane where the classification is true on one side of the hyperplane and false on the other side. The hyperplane is defined as where the predicted value, $f^w(X_1, \dots, X_n) = f(w_0 + w_1 \text{val}(e, X_1) + \dots + w_n \text{val}(e, X_n))$, is 0.5. For the sigmoid activation function, this occurs when $w_0 + w_1 \text{val}(e, X_1) + \dots + w_n \text{val}(e, X_n) = 0$ for the learned weights w . On one side of this hyperplane, the prediction is greater than 0.5; on the other side, it is less than 0.5.

Figure 7.8 shows linear separators for “or” and “and”. The dashed line separates the positive (true) cases from the negative (false) cases. One simple function that is not linearly separable is the **exclusive-or** (xor) function, shown on the right. There is no straight line that separates the positive examples from the negative examples. As a result, a linear classifier cannot represent, and therefore cannot learn, the exclusive-or function.

Often it is difficult to determine a priori whether a data set is linearly separable.

Example 7.11 Consider the data set of Figure 7.9, which is used to predict whether a person likes a holiday as a function of whether there is culture, whether the person has to fly, whether the destination is hot, whether there is music, and whether there is nature. In this data set, the value 1 means true and 0 means false. The linear classifier requires the numerical representation.

After 10,000 iterations of gradient descent with a learning rate of 0.05, the optimal prediction is (to one decimal point)

$$\begin{aligned} \text{Likes} = & f(2.3 \times \text{Culture} + 0.01 \times \text{Fly} - 9.1 \times \text{Hot} \\ & - 4.5 \times \text{Music} + 6.8 \times \text{Nature} + 0.01) , \end{aligned}$$

which approximately predicts the target value for all of the tuples in the training set except for the last and the third-to-last tuple, for which it predicts a value of about 0.5. This function seems to be quite stable with different initializations. Increasing the number of iterations makes it predict the other tuples better.

When the domain of the target variable is greater than 2 – there are more than two classes – indicator variables (page 290) can be used to convert the classification to binary variables. These binary variables could be learned separately. However, the outputs of the individual classifiers must be combined to give a prediction

for the target variable. Because exactly one of the values must be

<i>Culture</i>	<i>Fly</i>	<i>Hot</i>	<i>Music</i>	<i>Nature</i>	<i>Likes</i>
0	0	1	0	0	0
0	1	1	0	0	0
1	1	1	1	1	0
0	1	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	1
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
1	1	0	1	1	1
1	1	0	0	0	1
1	0	1	0	1	1
0	0	0	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0
1	0	0	1	0	0
1	1	1	0	1	0
0	0	0	0	1	1
0	1	0	0	0	1

Figure 7.9: Training data for which holiday a person likes

true for each example, the learner should not predict that more than one will be true or that none will be true. A classifier that predicts a probability distribution can normalize the predictions of the individual predictions. A learner that must make a definitive prediction can use the mode.

7.3.3 Bayesian Classifiers

A **Bayesian classifier** is based on the idea that the role of a (natural) class is to predict the values of features for members of that class. Examples are grouped in classes because they have common values for the features. Such classes are often called **natural kinds**. In this section, the target feature corresponds to a discrete **class**, which is not necessarily binary.

The idea behind a Bayesian classifier is that, if an agent knows the class, it can predict the values of the other features. If it does not know the class, Bayes' rule (page 227) can be used to predict the class given (some of) the feature values. In a Bayesian classifier, the learning agent builds a probabilistic model of the features and uses that model to predict the classification of a new example. A **latent variable** is a probabilistic variable that is not observed. A Bayesian classifier is a probabilistic model where the classification is a latent variable that is probabilistically related to the observed variables. Classification then become inference in the probabilistic model.

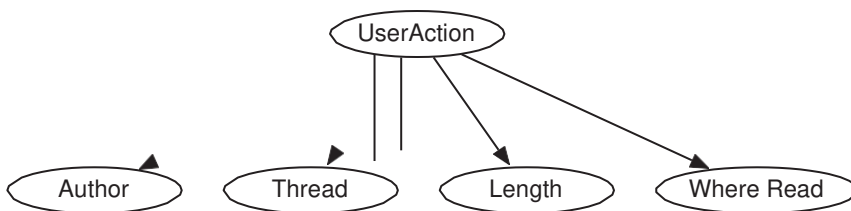


Figure 7.10: Belief network corresponding to a naive Bayesian classifier

The simplest case is the **naive Bayesian classifier**, which makes the independence assumption that the input features are conditionally independent of each other given the classification. (See page 233 for a definition of conditional independence.)

The independence of the naive Bayesian classifier is embodied in a particular belief network (page 235) where the features are the nodes, the target variable (the classification) has no parents, and the classification is the only parent of each input feature. This belief network requires the probability distributions $P(Y)$ for the target feature Y and $P(X_i|Y)$ for each input feature X_i . For each example, the prediction can be computed by conditioning on observed values for the input features and by querying the classification.

Given an example with inputs $X_1=v_1, \dots, X_k=v_k$, Bayes' rule (page 227) is used to compute the posterior probability distribution of the example's classification, Y :

$$\begin{aligned}
 P(Y|X_1=v_1, \dots, X_k=v_k) &= \frac{P(X_1=v_1, \dots, X_k=v_k|Y) \times P(Y)}{P(X_1=v_1, \dots, X_k=v_k)} \\
 &= \frac{P(X_1=v_1|Y) \times \dots \times P(X_k=v_k|Y) \times P(Y)}{\sum_Y P(X_1=v_1|Y) \times \dots \times P(X_k=v_k|Y) \times P(Y)}
 \end{aligned}$$

where the denominator is a normalizing constant to ensure the probabilities sum to 1. The denominator does not depend on the class and, therefore, it is not needed to determine the most likely class.

To learn a classifier, the distributions of $P(Y)$ and $P(X_i|Y)$ for each input feature can be learned from the data, as described in Section 7.2.3 (page 296). The simplest case is to use the empirical frequency in the training data as the probability (i.e., use the proportion in the training data as the probability). However, as shown below, this approach is often not a good idea when this results in zero probabilities.

Example 7.12 Suppose an agent wants to predict the user action given the data of Figure 7.1 (page 289). For this example, the user action is the classification. The naive Bayesian classifier for this example corresponds to the belief network of Figure 7.10. The training examples are used to determine the probabilities required for the belief network.

Suppose the agent uses the empirical frequencies as the probabilities for this example. The probabilities that can be derived from these data are

$$\begin{aligned}
P(\text{UserAction}=\text{reads}) &= \frac{9}{18} = 0.5 \\
P(\text{Author}=\text{known}|\text{UserAction}=\text{reads}) &= \frac{2}{3} \\
P(\text{Author}=\text{known}|\text{UserAction}=\text{skips}) &= \frac{2}{3} \\
P(\text{Thread}=\text{new}|\text{UserAction}=\text{reads}) &= \frac{7}{9} \\
P(\text{Thread}=\text{new}|\text{UserAction}=\text{skips}) &= \frac{1}{3} \\
P(\text{Length}=\text{long}|\text{UserAction}=\text{reads}) &= 0 \\
P(\text{Length}=\text{long}|\text{UserAction}=\text{skips}) &= \frac{7}{9} \\
P(\text{WhereRead}=\text{home}|\text{UserAction}=\text{reads}) &= \frac{4}{9} \\
P(\text{WhereRead}=\text{home}|\text{UserAction}=\text{skips}) &= \frac{4}{9}.
\end{aligned}$$

Based on these probabilities, the features *Author* and *WhereRead* have no predictive power because knowing either does not change the probability that the user will read the article. The rest of this example ignores these features.

To classify a new case where the author is unknown, the thread is a follow-up, the length is short, and it is read at home,

$$\begin{aligned}
&P(\text{UserAction}=\text{reads}|\text{Thread}=\text{followUp} \wedge \text{Length}=\text{short}) \\
&= \frac{P(\text{followUp}|\text{reads}) \times P(\text{short}|\text{reads}) \times P(\text{reads})}{c} \\
&= \frac{\frac{2}{9} \times 1 \times \frac{1}{2}}{\frac{9}{27} \times c} \\
&= \frac{1}{9} \times c
\end{aligned}$$

$$\begin{aligned}
&P(\text{UserAction}=\text{skips}|\text{Thread}=\text{followUp} \wedge \text{Length}=\text{short}) \\
&= \frac{P(\text{followUp}|\text{skips}) \times P(\text{short}|\text{skips}) \times P(\text{skips})}{c} \\
&= \frac{\frac{2}{3} \times \frac{2}{9} \times \frac{1}{2}}{\frac{9}{27} \times c} \\
&= \frac{2}{27} \times c
\end{aligned}$$

where c is a normalizing constant that ensures these add up to 1. Thus, c must be $\frac{27}{5}$, so

$$P(\text{UserAction}=\text{reads}|\text{Thread}=\text{followUp} \wedge \text{Length}=\text{short}) = 0.6 .$$

This prediction does not work well on example e_{11} , which the agent skips, even though it is a *followUp* and is *short*. The naive Bayesian classifier summarizes the data into a few parameters. It predicts the article will be read because being short is a stronger indicator that the article will be read than being a follow-up is an indicator that the article will be skipped.

A new case where the length is long has $P(\text{length}=\text{long} \mid \text{UserAction}=\text{reads})=0$. Thus, the posterior probability that the *UserAction=reads* is zero, no matter what the values of the other features are.

The use of zero probabilities can imply some unexpected behavior. First, some features become predictive: knowing just one feature value can rule out a category. If we allow zero probabilities, it is possible that some combinations of observations are impossible. See Exercise 7.13 (page 346). This is a problem not necessarily with using a Bayesian classifier but rather in using empirical frequencies as probabilities. The alternative to using the empirical frequencies is to incorporate pseudocounts (page 296). A designer of the learner should carefully choose pseudocounts, as shown in the following example.

Example 7.13 Consider how to learn the probabilities for the **help system** of Example 6.16 (page 246), where a helping agent infers what help page a user is interested in based on the keywords given by the user. The helping agent must learn the prior probability that each help page is wanted and the probability of each keyword given the help page wanted. These probabilities must be learned, because the system designer does not know a priori what words users will use. The agent can learn from the words users actually use when looking for help.

The learner must learn $P(H)$. To do this, it starts with a pseudocount (page 296) for each h_i . Pages that are a priori more likely can have a higher pseudocount. If the designer did not have any prior belief about which pages were more likely, the agent could use the same pseudocount for each page. To think about what count to use, the designer should consider how much more the agent would believe a page is the correct page after it has seen the page once; see Section 7.2.3 (page 296). It is possible to learn this pseudocount, if the designer has built another help system, by optimizing the pseudocount over the training data for that help system [or by using what is called a hierarchical Bayesian model (page 338)]. Given the pseudocounts and some data, $P(h_i)$ can be computed by dividing the count (the empirical count plus the pseudocount) associated with h_i by the sum of the counts for all of the pages.

For each word w_j and for each help page h_i , the helping agent requires two counts – the number of times w_j was used when h_i was the appropriate page (call this c_{ij}^+) and the number of times w_j was not used when h_i was the appropriate page (call this c_{ij}^-). Neither of these counts should be zero. We expect c_{ij}^- to be bigger on average than c_{ij}^+ , because we expect the average query to use a small portion of the total number of words. We may want to use different

counts for those words that appear in the help page h_i than for those words that do not appear in h_i , so that the system starts with sensible behavior.

Every time a user claims they have found the help page they are interested in, the counts for that page and the conditional counts for all of the words can be updated. That is, if the user says that h_i is the correct page, the count associated with h_i can be incremented, c_{ij}^+ is incremented for each word w_j used in the query, and c_{ij}^- is incremented for each w_j not in the query.

This model does not use information about the wrong page. If the user claims that a page is not the correct page, this information is not used until the correct page is found.

The biggest challenge in building such a help system is not in the learning but in acquiring useful data. In particular, users may not know whether they have found the page they were looking for. Thus, users may not know when to stop and provide the feedback from which the system learns. Some users may never be satisfied with a page. Indeed, there may not exist a page they are satisfied with, but that information never gets fed back the learner. Alternatively, some users may indicate they have found the page they were looking for, even though there may be another page that was more appropriate. In the latter case, the correct page may have its count reduced so that it is never discovered.

Although there are some cases where the naive Bayesian classifier does not produce good results, it is extremely simple, it is easy to implement, and often it works very well. It is a good method to try for a new problem.

In general, the naive Bayesian classifier works well when the independence assumption is appropriate, that is, when the class is a good predictor of the other features and the other features are independent given the class. This may be appropriate for **natural kinds**, where the classes have evolved because they are useful in distinguishing the objects that humans want to distinguish. Natural kinds are often associated with nouns, such as the class of dogs or the class of chairs.

7.4 Composite Models

Decision trees, (squashed) linear functions, and Bayesian classifiers provide the basis for many other supervised learning techniques. Linear classifiers are very restricted in what they can represent. Although decision trees can represent any discrete function, many simple functions have very complicated decision trees. Bayesian classifiers make a priori modeling assumptions that may not be valid.

One way to make the linear function more powerful is to have the inputs to the linear function be some non-linear function of the original inputs. Adding these new features can increase the dimensionality, making some functions that were not linear (or linearly separable) in the lower-dimensional space linear in the higher-dimensional space.

Example 7.14 The exclusive-or function, $x_1 \text{ xor } x_2$, is linearly separable in the space where the dimensions are X_1 , X_2 , and x_1x_2 , where x_1x_2 is a feature that is true when both x_1 and x_2 are true. To visualize this, consider Figure 7.8 (page 308); with the product as the third dimension, the top-right point will be lifted out of the page, allowing for a linear separator (in this case a plane) to go underneath it.

A **support vector machine (SVM)** is used for classification. It uses functions of the original inputs as the inputs of the linear function. These functions are called **kernel functions**. Many possible kernel functions exist. An example kernel function is the product of original features; adding the products of features is enough to enable the representation of the exclusive-or function. Increasing the dimensionality can, however, cause overfitting. An SVM constructs a decision surface, which is a hyperplane that divides the positive and negative examples in this higher-dimensional space. Define the **margin** to be the minimum distance from the decision surface to any of the examples. An SVM finds the decision surface with maximum margin. The examples that are closest to the decision surface are those that support (or hold up) the decision surface. In particular, these examples, if removed, would change the decision surface. Overfitting is avoided because these support vectors define a surface that can be defined in fewer parameters than there are examples. For detailed description of SVMs see the references at the end of this chapter.

Neural networks allow the inputs to the (squashed) linear function to be a squashed linear function with parameters to be tuned. Having multiple layers of squashed linear functions as inputs to (squashed) linear functions that predict the target variables allows more complex functions to be represented. Neural networks are described in more detail later.

Another nonlinear representation is a **regression tree**, which is a decision tree with a (squashed) linear function at the leaves of the decision tree. This can represent a piecewise linear approximation. It is even possible to have neural networks or SVMs at the leaves of the decision tree. To classify a new example, the example is filtered down the tree, and the classifier at the leaves is then used to classify the example.

The naive Bayesian classifier can be expanded to allow some input features to be parents of the classification and to allow some to be children. The probability of the classification given its parents can be represented as a decision tree or a squashed linear function or a neural network. The children of the classification do not have to be independent. One representation of the children is as a **tree augmented naive Bayesian (TAN) network**, where the children are allowed to have exactly one other parent other than the classification (as long as the resulting graph is acyclic). This allows for a simple model that accounts for interdependencies among the children. An alternative is to put structure in the class variable. A **latent tree model** decomposes the class variable into a number of latent variables that are connected together in a

tree structure. Each observed variable is a child of one of the latent variables. The latent variables allow a model of the dependence between the observed variables.

Another possibility is to use a number of classifiers that have each been trained on the data and to combine these using some mechanism such as voting or a linear function. These techniques are known as ensemble learning (page [319](#)).

7.4.1 Neural Networks

Neural networks are a popular target representation for learning. These networks are inspired by the neurons in the brain but do not actually simulate neurons. Artificial neural networks typically contain many fewer than the approximately 10^{11} neurons that are in the human brain, and the artificial neurons, called **units**, are much simpler than their biological counterparts.

Artificial neural networks are interesting to study for a number of reasons:

- As part of neuroscience, to understand real neural systems, researchers are simulating the neural systems of simple animals such as worms, which promises to lead to an understanding about which aspects of neural systems are necessary to explain the behavior of these animals.
- Some researchers seek to automate not only the functionality of intelligence (which is what the field of artificial intelligence is about) but also the mechanism of the brain, suitably abstracted. One hypothesis is that the only way to build the functionality of the brain is by using the mechanism of the brain. This hypothesis can be tested by attempting to build intelligence using the mechanism of the brain, as well as without using the mechanism of the brain. Experience with building other machines – such as flying machines (page [9](#)), which use the same principles, but not the same mechanism, that birds use to fly – would indicate that this hypothesis may not be true. However, it is interesting to test the hypothesis.
- The brain inspires a new way to think about computation that contrasts with currently available computers. Unlike current computers, which have a few processors and a large but essentially inert memory, the brain consists of a huge number of asynchronous distributed processes, all running concurrently with no master controller. One should not think that the current computers are the only architecture available for computation.
- As far as learning is concerned, neural networks provide a different measure of simplicity as a learning bias than, for example, decision trees. Multilayer neural networks, like decision trees, can represent any function of a set of discrete features. However, the functions that correspond to simple neural networks do not necessarily correspond to simple decision trees. Neural network learning imposes a different bias than decision tree learning. Which is better, in practice, is an empirical question that can be tested on different domains.

There are many different types of neural networks. This book considers one kind of neural network, the **feed-forward neural network**. Feed-forward networks can be seen as cascaded squashed linear functions. The inputs feed into a layer of hidden units, which can feed into layers of more hidden units, which eventually feed into the output layer. Each of the hidden units is a squashed linear function of its inputs.

Neural networks of this type can have as inputs any real numbers, and they have a real number as output. For regression, it is typical for the output units to be a linear function of their inputs. For classification it is typical for the output to be a sigmoid function of its inputs (because there is no point in predicting a value outside of $[0,1]$). For the hidden layers, there is no point in having their output be a linear function of their inputs because a linear function of a linear function is a linear function; adding the extra layers gives no added functionality. The output of each hidden unit is thus a squashed linear function of its inputs.

Associated with a network are the parameters for all of the linear functions. These parameters can be tuned simultaneously to minimize the prediction error on the training examples.

Example 7.15 Figure 7.11 shows a neural network with one hidden layer for the classification data of Figure 7.9 (page 309). As explained in Example 7.11 (page 308), this data set is not linearly separable. In this example, five Boolean inputs correspond to whether there is culture, whether the person has to fly, whether the destination is hot, whether there is music, and whether there is nature, and a single output corresponds to whether the person likes the holiday. In this network, there is one hidden layer, which contains two hidden units that have no a priori meaning. The network represents the following equations:

$$\begin{aligned} pval(e, Likes) &= f(w_0 + w_1 \times val(e, H1) + w_2 \times val(e, H2)) \\ val(e, H1) &= f(w_3 + w_4 \times val(e, Culture) + w_5 \times val(e, Fly) \\ &\quad + w_6 \times val(e, Hot) + w_7 \times val(e, Music) + w_8 \times val(e, Nature)) \\ val(e, H2) &= f(w_9 + w_{10} \times val(e, Culture) + w_{11} \times val(e, Fly) \\ &\quad + w_{12} \times val(e, Hot) + w_{13} \times val(e, Music) \\ &\quad + w_{14} \times val(e, Nature)) , \end{aligned}$$

where $f(x)$ is an activation function.

For this example, there are 15 real numbers to be learned ($w_0, \dots, w_{14}$). The hypothesis space is thus a 15-dimensional real space. Each point in this 15-dimensional space corresponds to a function that predicts a value for *Likes* for every example with *Culture*, *Fly*, *Hot*, *Music*, and *Nature* given.

Given particular values for the parameters, and given values for the inputs, a neural network predicts a value for each target feature. The aim of neural network learning is, given a set of examples, to find parameter settings that minimize the error. If there are m parameters, finding the parameter settings

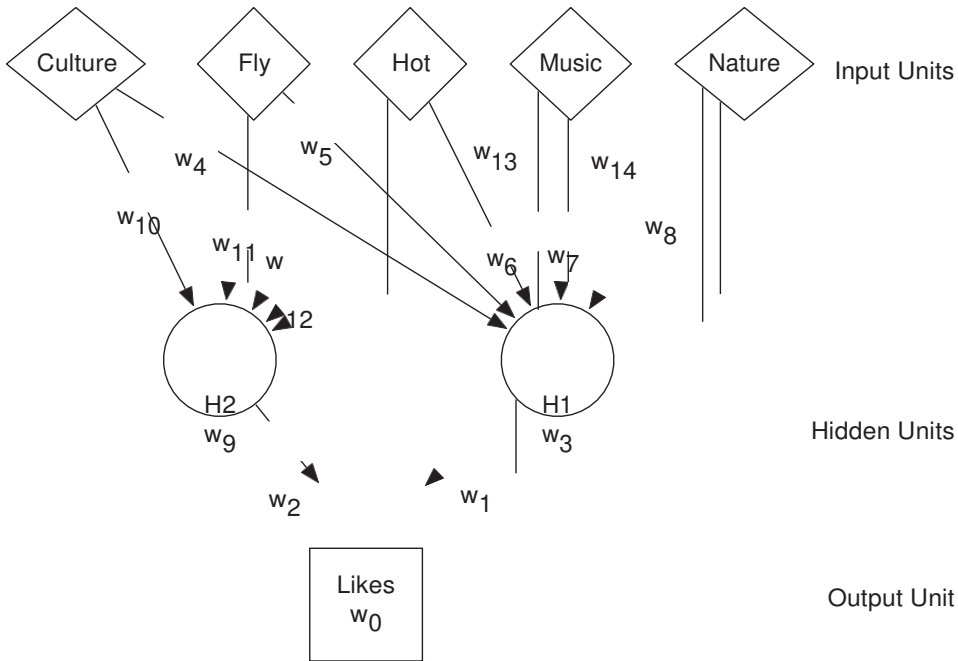


Figure 7.11: A neural network with one hidden layer. The w_i are weights. The weight inside the nodes is the weight that does not depend on an input; it is the one multiplied by 1. The meaning of this network is given in Example 7.15.

with minimum error involves searching through an m -dimensional Euclidean space.

Back-propagation learning is gradient descent search (page 149) through the parameter space to minimize the sum-of-squares error.

Figure 7.12 (on the next page) gives the incremental gradient descent version of back-propagation for networks with one layer of hidden units. This approach assumes n input features, k output features, and n_h hidden units. Both hw and ow are two-dimensional arrays of weights. Note that $0 : n_k$ means the index ranges from 0 to n_k (inclusive) and $1 : n_k$ means the index ranges from 1 to n_k (inclusive). This algorithm assumes that $val(e, X_0) = 1$ for all e .

The back-propagation algorithm is similar to the linear learner of Figure 7.6 (page 305), but it takes into account multiple layers and the activation function. Intuitively, for each example it involves simulating the network on that example, determining first the value of the hidden units (line 23) then the value of the output units (line 25). It then passes the error back through the network, computing the error on the output nodes (line 26) and the error on the hidden nodes (line 28). It then updates all of the weights based on the derivative of the error.

Gradient descent search (page 149) involves repeated evaluation of the

function to be minimized – in this case the error – and its derivative. An

```

1: procedure BackPropagationLearner( $X, Y, E, n_h, \eta$ )
2:   Inputs
3:      $X$ : set of input features,  $X = \{X_1, \dots, X_n\}$ 
4:      $Y$ : set of target features,  $Y = \{Y_1, \dots, Y_k\}$ 
5:      $E$ : set of examples from which to learn
6:      $n_h$ : number of hidden units
7:      $\eta$ : learning rate
8:   Output
9:     hidden unit weights  $hw[0 : n, 1 : n_h]$ 
10:    output unit weights  $ow[0 : n_h, 1 : k]$ 
11:   Local
12:      $hw[0 : n, 1 : n_h]$  weights for hidden units
13:      $ow[0 : n_h, 1 : k]$  weights for output units
14:      $hid[0 : n_h]$  values for hidden units
15:      $hErr[1 : n_h]$  errors for hidden units
16:      $out[1 : k]$  predicted values for output units
17:      $oErr[1 : k]$  errors for output units
18:   initialize  $hw$  and  $ow$  randomly
19:    $hid[0] := 1$ 
20:   repeat
21:     for each example  $e$  in  $E$  do
22:       for each  $h \in \{1, \dots, n_h\}$  do
23:          $hid[h] := \sum_{i=0}^n hw[i, h] \times val(e, X_i)$ 
24:       for each  $o \in \{1, \dots, k\}$  do
25:          $out[o] := \sum_{h=0}^{n_h} ow[h, o] \times hid[h]$ 
26:          $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ 
27:       for each  $h \in \{0, \dots, n_h\}$  do
28:          $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^k ow[h, o] \times oErr[o]$ 
29:       for each  $i \in \{0, \dots, n\}$  do
30:          $hw[i, h] := hw[i, h] + \eta \times hErr[h] \times val(e, X_i)$ 
31:       for each  $o \in \{1, \dots, k\}$  do
32:          $ow[h, o] := ow[h, o] + \eta \times oErr[o] \times hid[h]$ 
33:   until termination
34:   return  $w_0, \dots, w_n$ 

```

Figure 7.12: Back-propagation for learning a neural network with a single hidden layer

evaluation of the error involves iterating through all of the examples. Back-propagation learning thus involves repeatedly evaluating the network on all examples. Fortunately, with the logistic function the derivative is easy to determine given the value of the output for each unit.

Example 7.16 The network of Figure 7.11 (page 317), with one hidden layer containing two units, trained on the data of Figure 7.9 (page 309), can perfectly fit this data.

One run of back-propagation with the learning rate $\eta = 0.05$, and taking 10,000 steps, gave weights that accurately predicted the training data:

$$H1 = f(-2.0 \times Culture - 4.43 \times Fly + 2.5 \times Hot \\ + 2.4 \times Music - 6.1 \times Nature + 1.63)$$

$$H2 = f(-0.7 \times Culture + 3.0 \times Fly + 5.8 \times Hot \\ + 2.0 \times Music - 1.7 \times Nature - 5.0)$$

$$Likes = f(-8.5 \times H1 - 8.8 \times H2 + 4.36)$$

The use of neural networks may seem to challenge the **physical symbol system hypothesis** (page 15), which relies on symbols having meaning. Part of the appeal of neural networks is that, although meaning is attached to the input and output units, the designer does not associate a meaning with the hidden units. What the hidden units actually represent is something that is learned. After a neural network has been trained, it is often possible to look inside the network to determine what a particular hidden unit actually represents. Sometimes it is easy to express concisely in language what it represents, but often it is not. However, arguably, the computer has an internal meaning; it can explain its internal meaning by showing how examples map into the values of the hidden unit.

7.4.2 Ensemble Learning

In **ensemble learning**, an agent takes a number of learning algorithms and combines their output to make a prediction. The algorithms being combined are called **base-level algorithms**.

The simplest case of ensemble learning is to train the base-level algorithms on random subsets of the data and either let these vote for the most popular classification (for definitive predictions) or average the predictions of the base-level algorithm. For example, one could train a number of decision trees, each on random samples of, say, 50% of the training data, and then either vote for the most popular classification or average the numerical predictions. The outputs of the decision trees could even be inputs to a linear classifier, and the weights of this classifier could be learned.

This approach works well when the base-level algorithms are **unstable**: they tend to produce different representations depending on which subset of the data is chosen. Decision trees and neural networks are unstable, but linear classifiers tend to be stable and so would not work well with ensembles.

In **bagging**, if there are m training examples, the base-level algorithms are trained on sets of m randomly drawn, with replacement, sets of the training examples. In each of these sets, some examples are not chosen, and some are duplicated. On average, each set contains about 63% of the original examples.

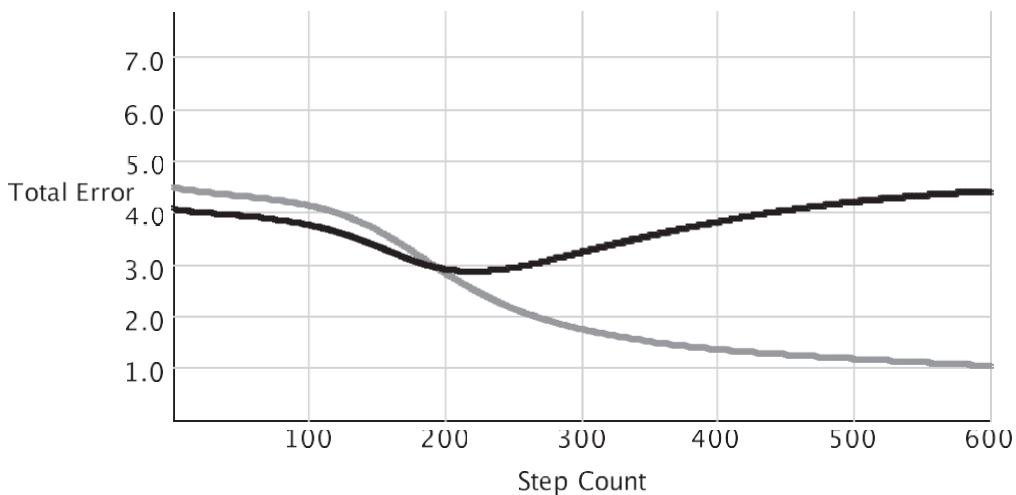


Figure 7.13: Error as a function of training time. On the x -axis is the step count of a run of back-propagation with three hidden units on the data of Figure 7.9 (page 309), using unseen examples as the test set. On the y -axis is the sum-of-squares error for the training set (gray line) and the test set (black line).

In **boosting** there is a sequence of classifiers in which each classifier uses a weighted set of examples. Those examples that the previous classifiers misclassified are weighted more. Weighting examples can either be incorporated into the base-level algorithms or can affect which examples are chosen as training examples for the future classifiers.

Another way to create base-level classifiers is to manipulate the input features. Different base-level classifiers can be trained on different features. Often the sets of features are hand-tuned.

Another way to get diverse base-level classifiers is to randomize the algorithm. For example, neural network algorithms that start at different parameter settings may find different local minima, which make different predictions. These different networks can be combined.

7.5 Avoiding Overfitting

Overfitting can occur when some regularities appear in the training data that do not appear in the test data, and when the learner uses those regularities for prediction.

Example 7.17 Figure 7.13 shows a typical plot of how the sum-of-squares error changes with the number of iterations of linear regression. The sum-of-squares error on the training set decreases as the number of iterations increases. For the test set, the error reaches a minimum and then increases as the number of iterations increases. The same behavior occurs in decision-tree learning as a function of the number of splits.

We discuss two ways to avoid overfitting. The first is to have an explicit trade-off between model complexity and fitting the data. The second approach is to use some of the training data to detect overfitting.

7.5.1 Maximum A Posteriori Probability and Minimum Description Length

One way to trade off model complexity and fit to the data is to choose the model that is most likely, given the data. That is, choose the model that maximizes the probability of the model given the data, $P(\text{model}|\text{data})$. The model that maximizes $P(\text{model}|\text{data})$ is called the **maximum a posteriori probability** model, or the **MAP model**.

The probability of a model (or a hypothesis) given some data is obtained by using Bayes' rule (page 227):

$$P(\text{model}|\text{data}) = \frac{P(\text{data}|\text{model}) \times P(\text{model})}{P(\text{data})}. \quad (7.3)$$

The likelihood, $P(\text{data}|\text{model})$, is the probability that this model would have produced this data set. It is high when the model is a good fit to the data, and it is low when the model would have predicted different data. The prior $P(\text{model})$ encodes the **learning bias** and specifies which models are a priori more likely. The prior probability of the model, $P(\text{model})$, is required to bias the learning toward simpler models. Typically simpler models have a higher prior probability. The denominator $P(\text{data})$ is a normalizing constant to make sure that the probabilities sum to 1.

Because the denominator of Equation (7.3) is independent of the model, it can be ignored when choosing the most likely model. Thus, the MAP model is the model that maximizes

$$P(\text{data}|\text{model}) \times P(\text{model}). \quad (7.4)$$

One alternative is to choose the **maximum likelihood model** – the model that maximizes $P(\text{data}|\text{model})$. The problem with choosing the most likely model is that, if the space of models is rich enough, a model exists that specifies that this particular data set will be produced, which has $P(\text{data}|\text{model}) = 1$. Such a model may be a priori very unlikely. However, we do not want to exclude it, because it may be the true model. Choosing the maximum-likelihood model is equivalent to choosing the maximum a posteriori model with a uniform prior over hypotheses.

MAP Learning of Decision Trees

To understand MAP learning, consider how it can be used to learn decision trees (page 298). If there are no examples with the same values for the input features and different values for the target features, there are always decision trees that fit the data perfectly. If the training examples do not cover all of the assignments to the input variables, multiple trees will fit the data perfectly.

However, with noise, none of these may be the best model. Not only do we want to compare the models that fit the data perfectly; we also want to compare those models with the models that do not necessarily fit the data perfectly. MAP learning provides a way to compare these models.

Suppose there are multiple decision trees that accurately fit the data. If $model$ denotes one of those decision trees, $P(data|model) = 1$. The preference for one decision tree over another depends on the prior probabilities of the decision trees; the prior probability encodes the learning bias (page 284). The preference for simpler decision trees over more complicated decision trees occurs because simpler decision trees have a higher prior probability.

Bayes' rule gives a way to trade off simplicity and ability to handle noise. Decision trees can handle noisy data by having probabilities at the leaves. When there is noise, larger decision trees fit the training data better, because the tree can account for random regularities (noise) in the training data. In decision-tree learning, the likelihood favors bigger decision trees; the more complicated the tree, the better it can fit the data. The prior distribution can favor smaller decision trees. When there is a prior distribution over decision trees, Bayes' rule specifies how to trade off model complexity and accuracy: The posterior probability of the model given the data is proportional to the product of the likelihood and the prior.

Example 7.18 Consider the data of Figure 7.1 (page 289), where the learner is to predict the user's actions.

One possible decision tree is the one given on the left of Figure 7.4 (page 298). Call this decision tree d_2 . The likelihood of the data is $P(data|d_2) = 1$. That is, d_2 accurately fits the data.

Another possible decision tree is one with no internal nodes, and a leaf that says to predict *reads* with probability $\frac{1}{2}$. This is the most likely tree with no internal nodes, given the data. Call this decision tree d_0 . The likelihood of the data given this model is

$$P(data|d_0) = \frac{1}{2}^{\sum_9} \times \frac{1}{2}^{\sum_9} \approx 0.00000149.$$

Another possible decision tree is one on the right of Figure 7.4 (page 298), which just splits on *Length*, and with probabilities on the leaves given by $P(reads|Length=long) = 0$ and $P(reads|Length=short) = \frac{9}{11}$. Note that $\frac{9}{11}$ is the empirical frequency of *reads* among the training set with *Length=short*. Call this decision tree d_{1a} . The likelihood of the data given this model is

$$P(data|d_{1a}) = 1^7 \times \frac{9}{11}^{\sum_9} \times \frac{2}{11}^{\sum_2} \approx 0.0543.$$

Another possible decision tree is one that just splits on *Thread*, and with probabilities on the leaves given by $P(reads|Thread=new) = \frac{7}{10}$ (as

7 out of the 10 examples with *Thread=new* have *UserAction=reads*), and

$P(\text{reads}|\text{Thread}=\text{followUp}) = \frac{2}{8}$ Call this decision tree d_{1t} . The likelihood of the data given d_{1t} is

$$P(\text{data}|d_{1t}) = \frac{7^{\Sigma_7}}{10} \times \frac{3^{\Sigma_3}}{10} \times \frac{6^{\Sigma_6}}{8} \times \frac{2^{\Sigma_2}}{8} \approx 0.000025.$$

These are just four of the possible decision trees. Which is best depends on the prior on trees. The likelihood of the data is multiplied by the prior probability of the decision trees to determine the posterior probability of the decision tree.

Description Length

The negative of the logarithm (base 2) of Formula (7.4) is

$$(-\log_2 P(\text{data}|\text{model})) + (-\log_2 P(\text{model})).$$

This can be interpreted in terms of **information theory** (page 231). The left-hand side of this expression is the number of bits it takes to describe the data given the model. The right-hand side is the number of bits it takes to describe the model. A model that minimizes this sum is a **minimum description length (MDL) model**. The **MDL principle** is to choose the model that minimizes the number of bits it takes to describe both the model and the data given the model.

One way to think about the MDL principle is that the aim is to communicate the data as succinctly as possible. The use of the model is to make communication shorter. To communicate the data, first communicate the model, then communicate the data in terms of the model. The number of bits it takes to communicate the data using a model is the number of bits it takes to communicate the model plus the number of bits it takes to communicate the data in terms of the model. The MDL principle is used to choose the model that lets us communicate the data in as few bits as possible.

As the logarithm function is monotonically increasing, the MAP model is the same the MDL model. The idea of choosing a model with the highest posterior probability is the same as choosing a model with a minimum description length.

Example 7.19 In Example 7.18, the definition of the priors on decision trees was left unspecified. The notion of a description length provides a basis for assigning priors to decision trees; consider how many bits it takes to describe a decision tree [see Exercise 7.11 (page 346)]. One must be careful defining the codes, because each code should describe a decision tree, and each decision tree should be described by a code.

7.5.2 Cross Validation

The problem with the previous methods is that they require a notion of simplicity to be known before the agent has seen any data. It would seem as though

an agent should be able to determine, from the data, how complicated a model needs to be. Such a method could be used when the learning agent has no prior information about the world.

The idea of **cross validation** is to split the training set into two: a set of examples to train with, and a **validation set**. The agent trains using the new training set. Prediction on the validation set is used to determine which model to use.

Consider a graph such as the one in Figure 7.13 (page 320). The error of the training set gets smaller as the size of the tree grows. The idea of cross validation is to choose the representation in which the error of the validation set is a minimum. In these cases, learning can continue until the error of the validation set starts to increase.

The validation set that is used as part of training is not the same as the test set. The test set is used to evaluate how well the learning algorithm works as a whole. It is cheating to use the test set as part of learning. Remember that the aim is to predict examples that the agent has not seen. The test set acts as a surrogate for these unseen examples, and so it cannot be used for training or validation.

Typically, we want to train on as many examples as possible, because then we get better models. However, having a small validation set means that the validation set may fit well, or not fit well, just by luck. There are various methods that have been used to reuse examples for both training and validation.

One method, *k*-fold cross validation, is used to determine the best model complexity, such as the depth of a decision tree or the number of hidden units in a neural network. The method of ***k*-fold cross validation** partitions the training set into *k* sets. For each model complexity, the learner trains *k* times, each time using one of the sets as the validation set and the remaining sets as the training set. It then selects the model complexity that has the smallest average error on the validation set (averaging over the *k* runs). It can return the model with that complexity, trained on all of the data.

7.6 Case-Based Reasoning

The previous methods tried to find a compact representation of the data that can be used for future prediction. In **case-based reasoning**, the training examples – the cases – are stored and accessed to solve a new problem. To get a prediction for a new example, those cases that are similar, or close to, the new example are used to predict the value of the target features of the new example. This is at one extreme of the learning problem where, unlike decision trees and neural networks, relatively little work must be done offline, and virtually all of the work is performed at query time.

Case-based reasoning can be used for classification and regression. It is also applicable when the cases are complicated, such as in legal cases, where the

cases are complex legal rulings, and in planning, where the cases are previous solutions to complex problems.

If the cases are simple, one algorithm that works well is to use the ***k*-nearest neighbors** for some given number k . Given a new example, the k training examples that have the input features closest to that example are used to predict the target value for the new example. The prediction can be the mode, average, or some interpolation between the prediction of these k training examples, perhaps weighting closer examples more than distant examples.

For this method to work, a distance metric is required that measures the closeness of two examples. First define a metric for the domain of each feature, in which the values of the features are converted to a numerical scale that can be used to compare values. Suppose $val(e, X_i)$ is a numerical representation of the value of feature X_i for the example e . Then $(val(e_1, X_i) - val(e_2, X_i))$ is the difference between example e_1 and e_2 on the dimension defined by feature X_i . The **Euclidean distance**, the square root of the sum of the squares of the dimension differences, can be used as the distance between two examples. One important issue is the relative scales of different dimensions; increasing the scale of one dimension increases the importance of that feature. Let w_i be a non-negative real-valued parameter that specifies the weight of feature X_i . The distance between examples e_1 and e_2 is then

$$d(e_1, e_2) = \sqrt{\sum_i w_i \times (val(e_1, X_i) - val(e_2, X_i))^2}.$$

The feature weights can be provided as input. It is also possible to learn these weights. The learning agent can try to find a parameter setting that minimizes the error in predicting the value of each element of the training set, based on every other instance in the training set. This is called the **leave-one-out cross-validation** error measure.

Example 7.20 Consider using case-based reasoning on the data of Figure 7.1 (page 289). Rather than converting the data to a secondary representation as in decision-tree or neural-network learning, case-based reasoning uses the examples directly to predict the value for the user action in a new case.

Suppose a learning agent wants to classify example e_{20} , for which the author is unknown, the thread is a follow-up, the length is short, and it is read at home. First the learner tries to find similar cases. There is an exact match in example e_{11} , so it may want to predict that the user does the same action as for example e_{11} and thus skips the article. It could also include other close examples.

Consider classifying example e_{19} , where the author is unknown, the thread is new, the length is long, and it was read at work. In this case there are no exact matches. Consider the close matches. Examples e_2 , e_8 , and e_{18} agree on the features *Author*, *Thread*, and *WhereRead*. Examples e_{10} and e_{12} agree on *Thread*, *Length*, and *WhereRead*. Example e_3 agrees on *Author*, *Length*, and *WhereRead*. Examples e_2 , e_8 , and e_{18} predict *Reads*, but the other examples predict *Skips*. So what should be predicted? The decision-tree algorithm says that *Length* is the

best predictor, and so e_2 , e_8 , and e_{18} should be ignored. For the sigmoid linear learning algorithm, the parameter values in Example 7.10 (page 307) similarly predict that the reader skips the article. A case-based reasoning algorithm to predict whether the user will or will not read this article must determine the relative importance of the dimensions.

One of the problems in case-based reasoning is accessing the relevant cases. A *kd-tree* is a way to index the training examples so that training examples that are close to a given example can be found quickly. Like a decision tree, a *kd-tree* splits on input features, but at the leaves are subsets of the training examples. In building a *kd-tree* from a set of examples, the learner tries to find an input feature that partitions the examples into set of approximately equal size and then builds *kd-trees* for the examples in each partition. This division stops when all of the examples at a leaf are the same. A new example can be filtered down the tree, as in a decision tree. The exact matches will be at the leaf found. However, the examples at the leaves of the *kd-tree* could possibly be quite distant from the example to be classified; they agree on the values down the branch of the tree but could disagree on the values of all other features. The same tree can be used to search for those examples that have one feature different from the ones tested in the tree. See Exercise 7.16 (page 347).

Case-based reasoning is also applicable when the cases are more complicated, for example, when they are legal cases or previous solutions to planning problems. In this scenario, the cases can be carefully chosen and edited to be useful. Case-based reasoning can be seen as a cycle of the following four tasks.

Retrieve: Given a new case, retrieve similar cases from the case base.

Reuse: Adapt the retrieved cases to fit to the new case.

Revise: Evaluate the solution and revise it based on how well it works.

Retain: Decide whether to retain this new case in the case base.

The revision can involve other reasoning techniques, such as using the proposed solution as a starting point to search for a solution, or a human could do the adaptation in an interactive system. Retaining can then save the new case together with the solution found.

Example 7.21 A common example of a case-based reasoning system is a helpdesk that users call with problems to be solved. For example, case-based reasoning could be used by the diagnostic assistant to help users diagnose problems on their computer systems. When a user gives a description of their problem, the closest cases in the case base are retrieved. The diagnostic assistant can recommend some of these to the user, adapting each case to the user's particular situation. An example of adaptation is to change the recommendation based on what software the user has, what method they use to connect to the Internet, and the brand of printer. If one of the cases suggested works, that can be recorded in the case base to make that case be more important when

another user asks a similar question. If none of the cases found works, some other problem solving can be done to solve the problem, perhaps by adapting other cases or having a human help diagnose the problem. When the problem is finally fixed, what worked in that case can be added to the case base.

7.7 Learning as Refining the Hypothesis Space

So far, learning is either choosing the best representation – for example, the best decision tree or the best values for parameters in a neural network – or predicting the value of the target features of a new case from a database of previous cases. This section considers a different notion of learning, namely learning as delineating those hypotheses that are consistent with the examples. Rather than choosing a hypothesis, the aim is to find all hypotheses that are consistent. This investigation will shed light on the role of a bias and provide a mechanism for a theoretical analysis of the learning problem.

We make three assumptions:

- There is a single target feature, Y , that is Boolean. This is not really a restriction for classification, because any discrete feature can be made into Boolean features using indicator variables (page 141).
- The hypotheses make definitive predictions, predicting true or false for each example, rather than probabilistic prediction.
- There is no noise in the data.

Given these assumptions, it is possible to write a hypothesis in terms of a proposition, where the primitive propositions are assignments to the input features.

Example 7.22 The decision tree of Figure 7.4 (page 298) can be seen as a representation *reads* defined by the proposition

$$pval(e, Reads) = val(e, Short) \wedge (val(e, New) \vee val(e, Known)) .$$

For the rest of this section, we write this more simply as

$$reads \leftrightarrow short \wedge (new \vee known) .$$

The goal is to try to find a proposition on the input features that correctly classifies the training examples.

Example 7.23 Consider the trading agent trying to infer which books or articles the user reads based on keywords supplied in the article. Suppose the learning agent has the following data:

article	Crime	Academic	Local	Music	Reads
<i>a</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
1	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>a</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>
2	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>a</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
3	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>

The aim is to learn which articles the user reads.

In this example, *reads* is the target feature, and the aim is to find a definition such as

$$reads \leftrightarrow crime \wedge (\neg academic \vee \neg music) .$$

This definition may be used to classify the training examples as well as future examples.

Hypothesis space learning assumes the following sets:

- I , the **instance space**, is the set of all possible examples.
- H , the **hypothesis space**, is a set of Boolean functions on the input features.
- $E \subseteq I$ is the set of **training examples**. Values for the input features and the target feature are given for the training example.

If $h \in H$ and $i \in I$, we write $h(i)$ to mean the value that h predicts for i on the target variable Y .

Example 7.24 In Example 7.23, I is the set of the $2^5 = 32$ possible examples, one for each combination of values for the features.

The hypothesis space H could be all Boolean combinations of the input features or could be more restricted, such as conjunctions or propositions defined in terms of fewer than three features.

In Example 7.23, the training examples are $E = \{a_1, a_2, a_3, a_4, a_5\}$. The target feature is *Reads*. Because the table specifies some of the values of this feature, and the learner will make predictions on unseen cases, the learner requires a bias (page 286). In hypothesis space learning, the bias is imposed by the hypothesis space.

Hypothesis h is **consistent** with a set of training examples E if $e \in E$, h accurately predicts the target feature of e . That is, $h(e) = val(e, Y)$; the predicted value is the same as the actual value for each example. The problem is to find the subset of or just an element of consistent with all of the training examples.

Example 7.25 Consider the data of Example 7.23, and suppose H is the set of conjunctions of literals. An example hypothesis in H that is consistent with $\{a_1\}$ is $\neg academic \wedge music$. This hypothesis means that the person reads an article if and only if $\neg academic \wedge music$ is true of the article. This concept is not the target concept because it is inconsistent with $\{a_1, a_2\}$.

7.7.1 Version-Space Learning

Rather than enumerating all of the hypotheses, the subset of consistent with the examples can be found more efficiently by imposing some structure on the hypothesis space.

Hypothesis h_1 is a **more general hypothesis** than hypothesis h_2 if h_2 implies h_1 . In this case, h_2 is a **more specific hypothesis** than h_1 . Any hypothesis is both more general than itself and more specific than itself.

Example 7.26 The hypothesis $\neg \text{academic} \wedge \text{music}$ is more specific than music and is also more specific than $\neg \text{academic}$. Thus, music is more general than $\neg \text{academic} \wedge \text{music}$. The most general hypothesis is *true*. The most specific hypothesis is *false*.

The “more general than” relation forms a partial ordering over the hypothesis space. The version-space algorithm that follows exploits this partial ordering to search for hypotheses that are consistent with the training examples.

Given hypothesis space H and examples E , the **version space** is the subset of H that is consistent with the examples.

The **general boundary** of a version space, G , is the set of maximally general members of the version space (i.e., those members of the version space such that no other element of the version space is more general). The **specific boundary** of a version space, S , is the set of maximally specific members of the version space.

These concepts are useful because the general boundary and the specific boundary completely determine the version space:

Proposition 7.2. *The version space, given hypothesis space H and examples E , can be derived from its general boundary and specific boundary. In particular, the version space is the set of $h \in H$ such that h is more general than an element of S and more specific than an element of G .*

Candidate Elimination Algorithm

The **candidate elimination algorithm** incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example. This is described in Figure 7.14 (on the next page).

Example 7.27 Consider how the candidate elimination algorithm handles Example 7.23 (page 327), where H is the set of conjunctions of literals.

Before it has seen any examples, $G_0 = \text{true}$ – the user reads everything – and $S_0 = \text{false}$ – the user reads nothing. Note that *true* is the empty conjunction and *false* is the conjunction of an atom and its negation.

After considering the first example, a_1 , $G_1 = \{\text{true}\}$ and

$$S_1 = \{\text{crime} \wedge \neg \text{academic} \wedge \neg \text{local} \wedge \text{music}\}.$$

Thus, the most general hypothesis is that the user reads everything, and the most specific hypothesis is that the user only reads articles exactly like this one.

```

1: procedure CandidateEliminationLearner( $X, Y, E, \mathbf{H}$ )
2:   Inputs
3:      $X$ : set of input features,  $X = \{X_1, \dots, X_n\}$ 
4:      $Y$ : target feature
5:      $E$ : set of examples from which to learn
6:      $\mathbf{H}$ : hypothesis space
7:   Output
8:     general boundary  $G \subseteq \mathbf{H}$ 
9:     specific boundary  $S \subseteq \mathbf{H}$  consistent with  $E$ 
10:  Local
11:     $G$ : set of hypotheses in  $\mathbf{H}$ 
12:     $S$ : set of hypotheses in  $\mathbf{H}$ 
13:    Let  $G = \{true\}$   $S = \{false\}$ 
14:    for each  $e \in E$  do
15:      if  $e$  is a positive example then
16:        Elements of  $G$  that classify  $e$  as negative are removed from  $G$ ;
17:      Each element  $s$  of  $S$  that classifies  $e$  as negative is removed and replaced
18:        by the minimal generalizations of  $s$  that classify  $e$  as positive and are less
19:        general than some member of  $G$ ;
20:      Non-maximal hypotheses are removed from  $S$ ;
21:      else
22:        Elements of  $S$  that classify  $e$  as positive are removed from  $S$ ;
23:      Each element  $g$  of  $G$  that classifies  $e$  as positive is removed and replaced
24:        by the minimal specializations of  $g$  that classifies  $e$  as negative and are more
25:        general than some member of  $S$ .
26:      Non-minimal hypotheses are removed from  $G$ .

```

Figure 7.14: Candidate elimination algorithm

After considering the first two examples, $G_2 = \{true\}$ and

$$S_2 = \{crime \wedge \neg academic \wedge \neg local\}.$$

Since a_1 and a_2 disagree on music, it has concluded that music cannot be relevant.

After considering the first three examples, the general boundary becomes

$$G_3 = \{crime, \neg academic\}$$

and $S_3 = S_2$. Now there are two most general hypotheses; the first is that the user reads anything about crime, and the second is that the user reads anything non-academic.

After considering the first four examples,

$$G_4 = \{crime, \neg academic \wedge \neg local\}$$

and $S_4 = S_3$.

After considering all five examples, we have

$$G_5 = \{crime\},$$

$$S_5 = \{crime \wedge \neg local\}.$$

Thus, after five examples, only two hypotheses exist in the version space. They differ only on their prediction on an example that has *crime* true and *local* false. If the target concept can be represented as a conjunction, only an example with *crime* true and *local* false will change G or S . This version space can make predictions about all other examples.

The Bias Involved in Version-Space Learning

Recall (page 286) that a bias is necessary for any learning to generalize beyond the training data. There must have been a bias in Example 7.27 (page 329) because, after observing only 5 of the 16 possible assignments to the input variables, an agent was able to make predictions about examples it had not seen.

The bias involved in version-space learning is called a **language bias** or a **restriction bias** because the bias is obtained from restricting the allowable hypotheses. For example, a new example with *crime* false and *music* true will be classified as false (the user will not read the article), even though no such example has been seen. The restriction that the hypothesis must be a conjunction of literals is enough to predict its value.

This bias should be contrasted with the bias involved in decision-tree learning (page 298). The decision tree can represent any Boolean function. Decision-tree learning involves a **preference bias**, in that some Boolean functions are preferred over others; those with smaller decision trees are preferred over those with larger decision trees. A decision-tree learning algorithm that builds a single decision tree top-down also involves a **search bias** in that the decision tree returned depends on the search strategy used.

The candidate elimination algorithm is sometimes said to be an unbiased learning algorithm because the learning algorithm does not impose any bias beyond the language bias involved in choosing H . It is easy for the version space to collapse to the empty set – for example, if the user reads an article with *crime* false and *music* true. This means that the target concept is not in H . Version-space learning is not tolerant to noise; just one misclassified example can throw off the whole system.

The **bias-free** hypothesis space is where H is the set of all Boolean functions. In this case, G always contains one concept: the concept which says that all negative examples have been seen and every other example is positive. Similarly, S contains the single concept which says that all unseen examples are negative. The version space is incapable of concluding anything about examples it has not seen; thus, it cannot generalize. Without a language bias or a preference bias, no generalization and therefore no learning will occur.

7.7.2 Probably Approximately Correct Learning

So far, we have seen a number of different learning algorithms. This section covers some of the theoretical aspects of learning, developed in an area called **computational learning theory**.

Some relevant questions that we can ask about a theory of computational learning include the following:

- Is the learner guaranteed to converge to the correct hypothesis as the number of examples increases?
- How many examples are required to identify a concept?
- How much computation is required to identify a concept?

In general, the answer to the first question is “no,” unless it can be guaranteed that the examples always eventually rule out all but the correct hypothesis. Someone out to trick the learner could choose examples that do not help discriminate correct hypotheses from incorrect hypotheses. So if such a person cannot be ruled out, a learner cannot guarantee to find a consistent hypothesis. However, given randomly chosen examples, a learner that always chooses a consistent hypothesis can get arbitrarily close to the correct concept. This requires a notion of closeness and a specification of what is a randomly chosen example.

Consider a learning algorithm that chooses a hypothesis consistent with all of the training examples. Assume a probability distribution over possible examples and that the training examples and the test examples are chosen from the same distribution. The distribution does not have to be known. We will prove a result that holds for all distributions.

Define the **error of hypothesis** $h \in \mathcal{H}$ written $error(h)$, to be the probability of choosing an element i of I such that $h(i) \neq val(i, Y)$, where $h(i)$ is the predicted value of target variable Y on possible example i , and $val(i, Y)$ is the actual value of Y . Recall that I , the instance space, is the set of all possible examples. That is,

$$error(h) = P(h(i) \neq val(i, Y) | i \in I).$$

An agent typically does not know P or $val(i, Y)$ for all i and, thus, does not actually know the error of a particular hypothesis.

Given $c > 0$, hypothesis h is **approximately correct** if $error(h) \leq c$.

We make the following assumption.

Assumption 7.3. *The training and test examples are chosen independently from the same probability distribution as the population.*

It is still possible that the examples do not distinguish hypotheses that are far away from the concept – it is just very unlikely that they do not. A learner that chooses a hypothesis that is consistent with the training examples is **probably approximately correct** if, for an arbitrary number δ ($0 < \delta < 1$), the algorithm is not approximately correct in at most δ of the cases. That is, the hypothesis generated is approximately correct at least $1 - \delta$ of the time.

Under the preceding assumption, for arbitrary c and δ , we can guarantee that an algorithm that returns a consistent hypothesis will find a hypothesis with error less than c , in at least $1 - \delta$ of the cases. Moreover, this outcome does not depend on the probability distribution.

Suppose $c > 0$ and $\delta > 0$ are given. Partition the hypothesis space H into

$$H_0 = \{h \in H : \text{error}(h) \leq c\}$$

$$H_1 = \{h \in H : \text{error}(h) > c\}.$$

We want to guarantee that the learner does not choose an element of H_1 in more than δ of the cases.

Suppose $h \in H_1$, then

$$P(h \text{ is wrong for a single example}) \geq c$$

$$P(h \text{ is correct for a single example}) \leq 1 - c$$

$$P(h \text{ is correct for } m \text{ examples}) \leq (1 - c)^m$$

Thus,

$$\begin{aligned} P(H_1 \text{ contains a hypothesis that is correct for } m \text{ examples}) \\ &\leq |H_1| (1 - c) \\ &\leq |H| (1 - c)^m \\ &\leq |H| e^{-cm} \end{aligned}$$

using the inequality that $(1 - c) \leq e^{-c}$ if $0 \leq c \leq 1$.

Thus, if we ensure that $|H| e^{-cm} \leq \delta$, we guarantee that H does not contain a hypothesis that is correct for m examples in more than δ of the cases. Thus, H_0 contains all of the correct hypotheses in all but δ of the cases.

Solving for m gives

$$m \geq \frac{1}{c} \ln |H| + \ln \frac{1}{\delta}.$$

Thus, we can conclude the following proposition.

Proposition 7.4. *If a hypothesis is consistent with at least*

$$\frac{1}{c} \ln |H| + \ln \frac{1}{\delta}$$

training examples, it has error at most c , at least $1 - \delta$ of the time.

The number of examples required to guarantee this error bound is called the **sample complexity**. The number of examples required according to this proposition is a function of c , δ , and the size of the hypothesis space.

Example 7.28 Suppose the hypothesis space is the set of conjunctions of literals on n Boolean variables. In this case $|H| = 3^n$ because, for each conjunction, each variable is in one of three states: (1) it is unnegated in the conjunction, (2) it is negated, or (3) it does not appear; the “+1” is needed to represent

false, which is the conjunction of any atom and its negation. Thus, the sample complexity is $\frac{1}{\epsilon} n \ln 3 + \ln \frac{1}{\delta}$ examples, which is polynomial in n , $\frac{1}{\epsilon}$ and $\ln \frac{1}{\delta}$.

If we want to guarantee at most a 5% error 99% of the time and have 30 Boolean variables, then $\epsilon = 1/20$, $\delta = 1/100$, and $n = 30$. The bound says that we can guarantee this performance if we find a hypothesis that is consistent with $20 (30 \ln 3 + \ln 100) = 752$ examples. This is much less than the number of possible instances, which is $2^{30} = 1,073,741,824$, and the number of hypotheses, which is $3^{30} + 1 = 205,891,132,094,650$.

Example 7.29 If the hypothesis space H is the set of all Boolean functions on n variables, then $|H| = 2^{2^n}$; thus, we require $\frac{1}{\epsilon} 2^n \ln 2 + \ln \frac{1}{\delta}$ examples. The sample complexity is exponential in n .

If we want to guarantee at most a 5% error 99% of the time and have 30 Boolean variables, then $\epsilon = 1/20$, $\delta = 1/100$, and $n = 30$. The bound says that we can guarantee this performance if we find a hypothesis that is consistent with $20 \times (2^{30} \ln 2 + \ln 100) \approx 14,885,222,452$ examples.

Consider the third question raised at the start of this section, namely, how quickly a learner can find the probably approximately correct hypothesis. First, if the sample complexity is exponential in the size of some parameter (e.g., n above), the computational complexity must be exponential because an algorithm must at least consider each example. To show an algorithm with polynomial complexity, we must find a hypothesis space with polynomial sample complexity and show that the algorithm uses polynomial time for each example.

7.8 Bayesian Learning

Rather than choosing the most likely model or delineating the set of all models that are consistent with the training data, another approach is to compute the posterior probability of each model given the training examples.

The idea of **Bayesian learning** is to compute the posterior probability distribution of the target features of a new example conditioned on its input features and all of the training examples.

Suppose a new case has inputs $X=x$ and has target features, Y ; the aim is to compute $P(Y|X=x, \mathbf{e})$, where $\mathbf{e}$ is the set of training examples. This is the probability distribution of the target variables given the particular inputs and the examples. The role of a model is to be the assumed generator of the examples. If we let M be a set of disjoint and covering models, then reasoning by cases (page 224) and the chain rule give

$$\begin{aligned} P(Y|x \wedge \mathbf{e}) &= \sum_{m \in M} P(Y \wedge m | x \wedge \mathbf{e}) \\ &= \sum_{m \in M} P(Y|m \wedge x \wedge \mathbf{e}) \times P(m|x \wedge \mathbf{e}) \\ &= \sum_{m \in M} P(Y|m \wedge x) \times P(m|\mathbf{e}). \end{aligned}$$

The first two equalities are theorems from the definition of probability (page 223). The last equality makes two assumptions: the model includes all of the information about the examples that is necessary for a particular prediction [i.e., $P(Y \mid x, \mathbf{e}) = P(Y \mid x)$], and the model does not change depending on the inputs of the new example [i.e., $P(m \mid x, \mathbf{e}) = P(m \mid \mathbf{e})$]. This formula says that we average over the prediction of all of the models, where each model is weighted by its posterior probability given the examples.

$P(m \mid \mathbf{e})$ can be computed using Bayes' rule:

$$P(m \mid \mathbf{e}) = \frac{P(\mathbf{e} \mid m) \times P(m)}{P(\mathbf{e})}.$$

Thus, the weight of each model depends on how well it predicts the data (the likelihood) and its prior probability. The denominator, $P(\mathbf{e})$, is a normalizing constant to make sure the posterior probabilities of the models sum to 1. Computing $P(\mathbf{e})$ can be very difficult when there are many models.

A set of $e_1, \dots, e_k$ examples are **i.i.d. (independent and identically distributed)**, where the distribution is given by model m if, for all i and j , examples e_i and e_j are independent given m , which means $P(e_i, e_j \mid m) = P(e_i \mid m) \times P(e_j \mid m)$. We usually assume that the examples are i.i.d.

Suppose the set of training examples $\mathbf{e}$ is $e_1, \dots, e_k$. That is, $\mathbf{e}$ is the conjunction of the e_i , because all of the examples have been observed to be true. The assumption that the examples are i.i.d. implies

$$P(\mathbf{e} \mid m) = \prod_{i=1}^k P(e_i \mid m).$$

The set of models may include structurally different models in addition to models that differ in the values of the parameters. One of the techniques of Bayesian learning is to make the parameters of the model explicit and to determine the distribution over the parameters.

Example 7.30 Consider the simplest learning task under uncertainty. Suppose there is a single Boolean random variable, Y . One of two outcomes, a and $\neg a$, occurs for each example. We want to learn the probability distribution of Y given some examples.

There is a single parameter, φ , that determines the set of all models. Suppose that φ represents the probability of $Y = \text{true}$. We treat this parameter as a real-valued random variable on the interval $[0, 1]$. Thus, by definition of φ , $P(a \mid \varphi) = \varphi$ and $P(\neg a \mid \varphi) = 1 - \varphi$.

Suppose an agent has no prior information about the probability of Boolean variable Y and no knowledge beyond the training examples. This ignorance can be modeled by having the prior probability distribution of the variable φ as a uniform distribution over the interval $[0, 1]$. This is the probability density function labeled $n_0=0, n_1=0$ in Figure 7.15.

We can update the probability distribution of φ given some examples. Assume that the examples, obtained by running a number of independent experiments, are a particular sequence of outcomes that consists of n_0 cases where Y is false and n_1 cases where Y is true.

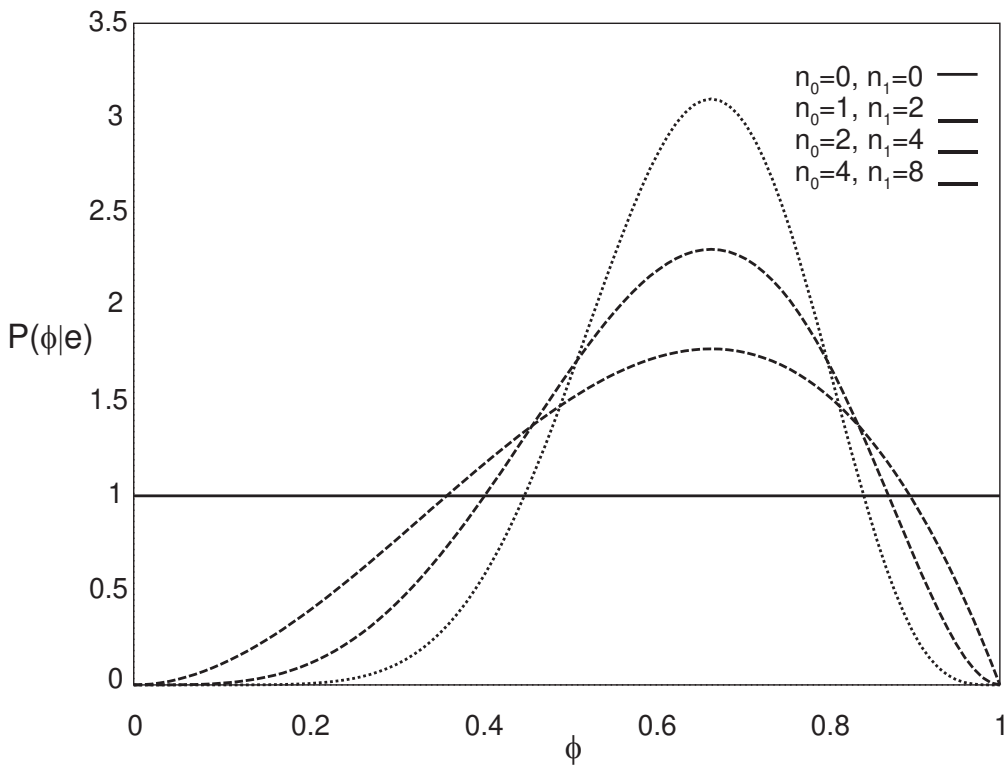


Figure 7.15: Beta distribution based on different samples

The posterior distribution for φ given the training examples can be derived by Bayes' rule. Let the examples $\mathbf{e}$ be the particular sequence of observation that resulted in n_1 occurrences of $Y=\text{true}$ and n_0 occurrences of $Y=\text{false}$. Bayes' rule gives us

$$P(\varphi|\mathbf{e}) = \frac{P(\mathbf{e}|\varphi) \times P(\varphi)}{P(\mathbf{e})}.$$

The denominator is a normalizing constant to make sure the area under the curve is 1.

Given that the examples are i.i.d.,

$$P(\mathbf{e}|\varphi) = \varphi^{n_1} \times (1 - \varphi)^{n_0}$$

because there are n_0 cases where $Y=\text{false}$, each with a probability of $1 - \varphi$, and n_1 cases where $Y=\text{true}$, each with a probability of φ .

One possible prior probability, $P(\varphi)$, is a uniform distribution on the interval $[0, 1]$. This would be reasonable when the agent has no prior information about the probability.

Figure 7.15 gives some posterior distributions of the variable φ based on different sample sizes, and given a uniform prior. The cases are $(n_0 = 1, n_1 = 2)$, $(n_0 = 2, n_1 = 4)$, and $(n_0 = 4, n_1 = 8)$. Each of these peak at the same place, namely at $\frac{2}{3}$. More training examples make the curve sharper.

The distribution of this example is known as the **beta distribution**; it is parametrized by two counts, a_0 and a_1 , and a probability p . Traditionally, the a_i parameters for the beta distribution are one more than the counts; thus, $a_i = n_i + 1$. The beta distribution is

$$\text{Beta}^{a_0, a_1}(p) = \frac{1}{K} p^{a_1-1} \times (1-p)^{a_0-1}$$

where K is a normalizing constant that ensures the integral over all values is 1. Thus, the uniform distribution on $[0, 1]$ is the beta distribution $\text{Beta}^{1,1}$.

The generalization of the beta distribution to more than two parameters is known as the Dirichlet distribution. The **Dirichlet distribution** with two sorts of parameters, the “counts” $a_1, \dots, a_k$, and the probability parameters $p_1, \dots, p_k$, is

$$\text{Dirichlet}^{a_1, \dots, a_k}(p_1, \dots, p_k) = \frac{1}{K} \prod_{j=1}^k p_j^{a_j-1}$$

where K is a normalizing constant that ensures the integral over all values is 1; p_i is the probability of the i th outcome (and so $0 \leq p_i \leq 1$) and a_i is one more than the count of the i th outcome. That is, $a_i = n_i + 1$. The Dirichlet distribution looks like Figure 7.15 along each dimension (i.e., as each p_j varies between 0 and 1).

For many cases, summing over all models weighted by their posterior distribution is difficult, because the models may be complicated (e.g., if they are decision trees or even belief networks). However, for the Dirichlet distribution, the expected value for outcome i (averaging over all p_j 's) is

$$\frac{a_i}{\sum_j a_j}.$$

The reason that the a_i parameters are one more than the counts is to make this formula simple. This fraction is well defined only when the a_j are all non-negative and not all are zero.

Example 7.31 Consider Example 7.30 (page 335), which determines the value of φ based on a sequence of observations made up of n_0 cases where Y is false and n_1 cases where Y is true. Consider the posterior distribution as shown in Figure 7.15. What is interesting about this is that, whereas the most likely posterior value of φ is $\frac{n_1}{n_0+n_1}$ the expected value (page 230) of this distribution is $\frac{n_1+1}{n_0+n_1+2}$.

Thus, the expected value of the $n_0=1, n_1=2$ curve is $\frac{3}{8}$ for the $n_0=2, n_1=4$ case the expected value is $\frac{5}{8}$, and for the $n_0=4, n_1=8$ case it is $\frac{9}{14}$. As the learner gets more training examples, this value approaches $\frac{n}{m}$.

This estimate is better than $\frac{n}{m}$ for a number of reasons. First, it tells us what to do if the learning agent has no examples: Use the uniform prior of $\frac{1}{2}$. This is the expected value of the $n=0, m=0$ case. Second, consider the case where

$n=0$ and $m=3$. The agent should not use $P(y)=0$, because this says that Y is impossible, and it certainly does not have evidence for this! The expected value of this curve with a uniform prior is $\frac{1}{5}$.

An agent does not have to start with a uniform prior; it can start with any prior distribution. If the agent starts with a prior that is a Dirichlet distribution, its posterior will be a Dirichlet distribution. The posterior distribution can be obtained by adding the observed counts to the a_i parameters of the prior distribution.

The i.i.d. assumption can be represented as a belief network, where each of the e_i are independent given model m . This independence assumption can be represented by the belief network shown on the left side of Figure 7.16. If m is made into a discrete variable, any of the inference methods of the previous chapter can be used for inference in this network. A standard reasoning technique in such a network is to condition on all of the observed e_i and to query the model variable or an unobserved e_i variable.

The problem with specifying a belief network for a learning problem is that the model grows with the number of observations. Such a network can be specified before any observations have been received by using a **plate model**. A plate model specifies what variables will be used in the model and what will be repeated in the observations. The right side of Figure 7.16 shows a plate model that represents the same information as the left side. The plate is drawn as a rectangle that contains some nodes, and an index (drawn on the bottom right of the plate). The nodes in the plate are indexed by the index. In the plate model, there are multiple copies of the variables in the plate, one for each value of the index. The intuition is that there is a pile of plates, one for each value of the index. The number of plates can be varied depending on the number of observations and what is queried. In this figure, all of the nodes in the plate share a common parent. The probability of each copy of a variable in a plate given the parents is the same for each index.

A plate model lets us specify more complex relationships between the variables. In a **hierarchical Bayesian model**, the parameters of the model can

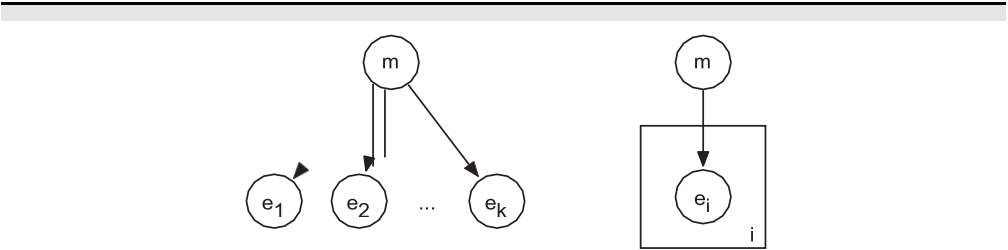


Figure 7.16: Belief network and plate models of Bayesian learning

depend on other parameters. Such a model is hierarchical in the sense that some parameters can depend on other parameters.

Example 7.32 Suppose a diagnostic assistant agent wants to model the probability that a particular patient in a hospital is sick with the flu before symptoms have been observed for this patient. This prior information about the patient can be combined with the observed symptoms of the patient. The agent wants to learn this probability, based on the statistics about other patients in the same hospital and about patients at different hospitals. This problem can range from the cases where a lot of data exists about the current hospital (in which case, presumably, that data should be used) to the case where there is no data about the particular hospital that the patient is in. A hierarchical Bayesian model can be used to combine the statistics about the particular hospital the patient is in with the statistics about the other hospitals.

Suppose that for patient X in hospital H there is a random variable S_{HX} that is true when the patient is sick with the flu. (Assume that the patient identification number and the hospital uniquely determine the patient.) There is a value ϕ_H for each hospital H that will be used for the prior probability of being sick with the flu for each patient in H . In a Bayesian model, ϕ_H is treated as a real-valued random variable with domain $[0, 1]$. S_{HX} depends on ϕ_H , with $P(S_{HX}|\phi_H) = \phi_H$. Assume that ϕ_H is distributed according to a beta distribution (page 337). We don't assume that ϕ_{h_i} and ϕ_{h_2} are independent of each other, but depend on **hyperparameters**. The hyperparameters can be the prior counts a_0 and a_1 . The parameters depend on the hyperparameters in terms of the conditional probability $P(\phi_{h_i}|a_0, a_1) = \text{Beta}^{a_0 a_1}(\phi_{h_i})$; a_0 and a_1 are real-valued random variables, which require some prior distribution.

The plate model and the corresponding belief network are shown in Figure 7.17. Part (a) shows the plate model, where there is a copy of the outside plate

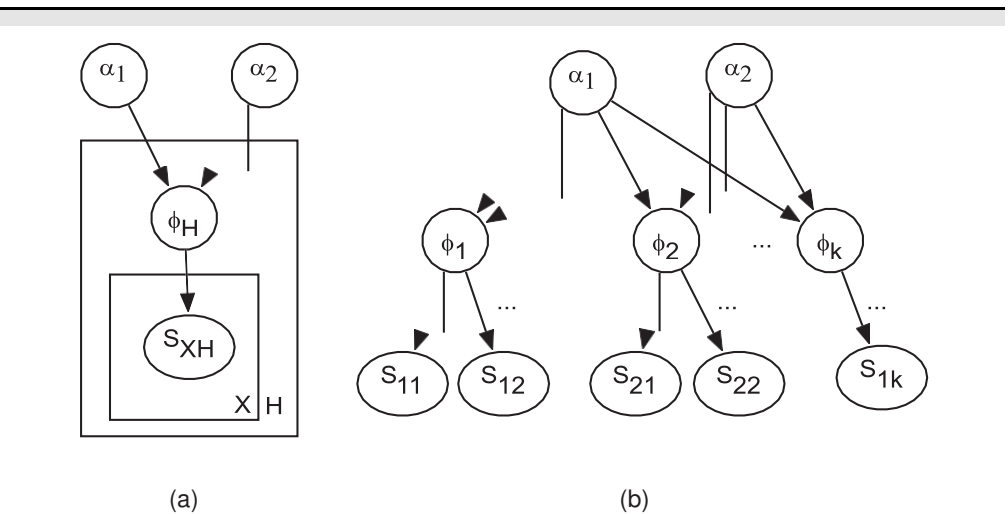


Figure 7.17: Hierarchical Bayesian model

for each hospital and a copy of the inside plate for each patient in the hospital. Part of the resulting belief network is shown in part (b). Observing some of the S_{HX} will affect the φ_H and so a_0 and a_1 , which will in turn affect the other φ_H variables and the unobserved S_{HX} variables.

Sophisticated methods exist to evaluate such networks. However, if the variables are made discrete, any of the methods of the previous chapter can be used.

In addition to using the posterior distribution of φ to derive the expected value, we can use it to answer other questions such as: What is the probability that the posterior probability of φ is in the range $[a, b]$? In other words, derive $P((\varphi \geq b) \mid \varphi \geq a)$. This is the problem that the Reverend Thomas Bayes solved more than 200 years ago [Bayes, 1763]. The solution he gave – although in much more cumbersome notation – was

$$\frac{\int_a^b p^n \times (1-p)^{m-n}}{\int_0^1 p^n \times (1-p)^{m-n}}.$$

This kind of knowledge is used in surveys when it may be reported that a survey is correct with an error of at most 5%, 19 times out of 20. It is also the same type of information that is used by probably approximately correct (PAC) learning (page 332), which guarantees an error at most ϵ at least $1 - \delta$ of the time. If an agent chooses the midpoint of the range $[a, b]$, namely $\frac{a+b}{2}$, as its hypothesis, it will have error less than or equal to $\frac{b-a}{2}$, just when the hypothesis is in $[a, b]$. The value $1 - \delta$ corresponds to $P(\varphi \geq a \mid \varphi \geq b)$. If $\epsilon = \frac{b-a}{2}$ and $\delta = 1 - P(\varphi \geq a \mid \varphi \geq b)$, choosing the midpoint will result in an error at most ϵ in $1 - \delta$ of the time. PAC learning gives worst-case results, whereas Bayesian learning gives the expected number. Typically, the Bayesian estimate is more accurate, but the PAC results give a guarantee of the error. The sample complexity (see Section 7.7.2) required for Bayesian learning is typically much less than that of PAC learning – many fewer examples are required to *expect to achieve* the desired accuracy than are needed to *guarantee* the desired accuracy.

7.9 Review

The following are the main points you should have learned from this chapter:

- Learning is the ability of an agent improve its behavior based on experience.
- Supervised learning is the problem that involves predicting the output of a new input, given a set of input-output pairs.
- Given some training examples, an agent builds a representation that can be used for new predictions.
- Linear classifiers, decision trees, and Bayesian classifiers are all simple representations that are the basis for more sophisticated models.

- An agent can choose the best hypothesis given the training examples, delineate all of the hypotheses that are consistent with the data, or compute the posterior probability of the hypotheses given the training examples.

7.10 References and Further Reading

For [good overviews of machine learning see Mitchell \[1997\]](#), Duda, Hart, and Stork [2001], [Bishop \[2008\]](#), and [Hastie, Tibshirani, and Friedman \[2009\]](#).

The collection of papers by [Shavlik and Dietterich \[1990\]](#) contains many classic learning papers. [Michie, Spiegelhalter, and Taylor \[1994\]](#) give empirical evaluation of many learning algorithms on many different problems. [Briscoe and Caelli \[1996\]](#) discuss many different machine learning algorithms. [Weiss and Kulikowski \[1991\]](#) overview techniques for classification learning. [Davis and Goadrich \[2006\]](#) discusses precision, recall, and ROC curves.

The approach to combining expert knowledge and data was proposed by [Spiegelhalter, Franklin, and Bull \[1990\]](#).

Decision-tree learning is discussed by [Quinlan \[1986\]](#). For an overview of a mature decision-tree learning tool see [Quinlan \[1993\]](#). The Gini index [Exercise 7.10 (page 345)] [is the splitting criteria used in CART \[Breiman, Friedman, Olshen, and Stone, 1984\]](#).

TAN networks are described by [Friedman, Greiger, and Goldszmidt \[1997\]](#). Latent tree models are described by [Zhang \[2004\]](#).

For [overviews of neural networks see Bishop \[1995\]](#), Hertz, Krogh, and Palmer [1991], and [Jordan and Bishop \[1996\]](#). Back-propagation is introduced in [Rumelhart, Hinton, and Williams \[1986\]](#). [Minsky and Papert \[1988\]](#) analyze the limitations of neural networks.

For reviews of ensemble learning see [Dietterich \[2002\]](#). Boosting is described in [Schapire \[2002\]](#) and [Meir and Rätsch \[2003\]](#).

For reviews [on case-based reasoning see Aamodt and Plaza \[1994\]](#), Kolodner and Leake [1996], and [Lopez De Mantaras, Mcsherry, Bridge, Leake, Smyth, Craw, Faltings, Maher, Cox, Forbus, Keane, Aamodt, and Watson \[2005\]](#). For a review of nearest-neighbor algorithms, see [Duda et al. \[2001\]](#) and [Dasarathy \[1991\]](#). The dimension-weighting learning nearest-neighbor algorithm is from [Lowe \[1995\]](#). [For a classical review of case-based reasoning, see Riesbeck and Schank \[1989\]](#), and for recent reviews see [Aha, Marling, and Watson \[2005\]](#).

Version spaces were defined by [Mitchell \[1977\]](#). PAC learning was introduced by [Valiant \[1984\]](#). [The analysis here is due to Haussler \[1988\]](#). Kearns and Vazirani [1994] give a good introduction to computational learning theory and PAC learning. For more details on version spaces and PAC learning, see [Mitchell \[1997\]](#).

For overviews [of Bayesian learning, see Jaynes \[2003\]](#), [Loredo \[1990\]](#), Howson and Urbach [2006], and [Cheeseman \[1990\]](#). See also books on Bayesian statistics [such as Gelman, Carlin, Stern, and Rubin \[2004\]](#) and [Bernardo and Smith \[1994\]](#). Bayesian learning of decision trees is described in [Buntine \[1992\]](#). [Grünwald \[2007\]](#) discusses the MDL principle.

For research results on machine learning, see the journals *Journal of Machine Learning Research* (JMLR), *Machine Learning*, the annual *International Conference on Machine Learning* (ICML), the *Proceedings of the Neural Information Processing Society* (NIPS), or general AI journals such as *Artificial Intelligence* and the *Journal of Artificial Intelligence Research*, and many specialized conferences and journals.

Solutions To Case Studies (continue)

Chapter 8

Planning with Certainty

He who every morning plans the transaction of the day and follows out that plan, carries a thread that will guide him through the maze of the most busy life. But where no plan is laid, where the disposal of time is surrendered merely to the chance of incidence, chaos will soon reign.

– Victor Hugo (1802–1885)

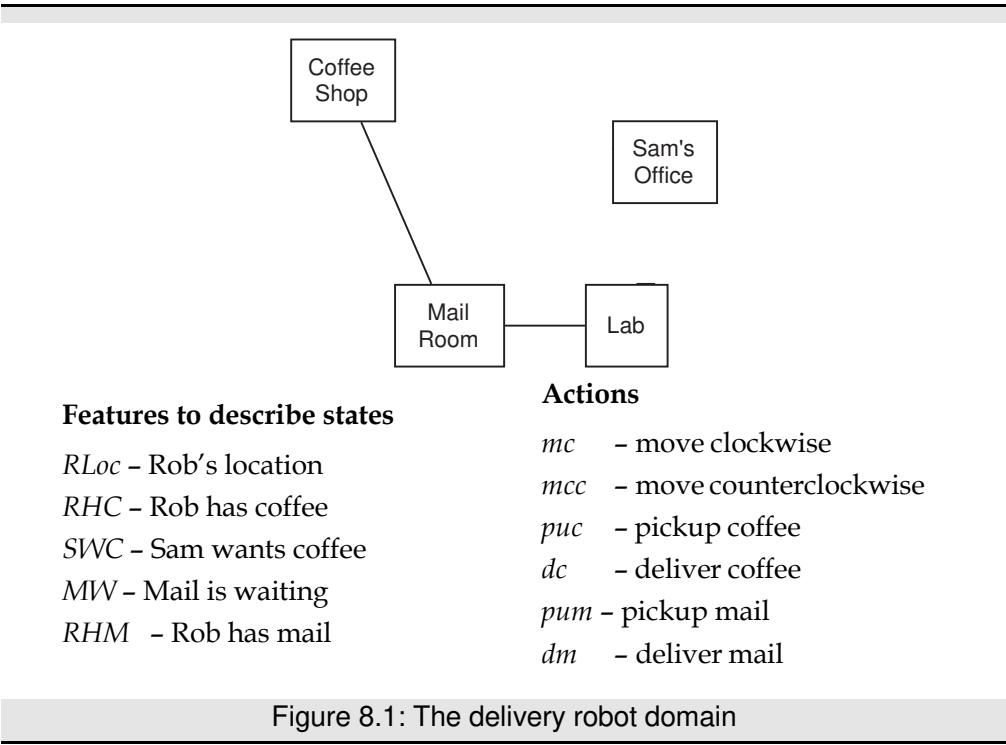
Planning is about how an agent achieves its goals. To achieve anything but the simplest goals, an agent must reason about its future. Because an agent does not usually achieve its goals in one step, what it should do at any time depends on what it will do in the future. What it will do in the future depends on the state it is in, which, in turn, depends on what it has done in the past. This chapter considers how an agent can represent its actions and their effects and use these

models to find a plan to achieve its goals.

In particular, this chapter considers the case where

- the agent's actions are deterministic; that is, the agent can predict the consequences of its actions.
- there are no exogenous events beyond the control of the agent that change the state of the world.
- the world is fully observable; thus, the agent can observe the current state of the world.
- time progresses discretely from one state to the next.
- goals are predicates of states that must be achieved or maintained.

Some of these assumptions are relaxed in the following chapters.



8.1 Representing States, Actions, and Goals

To reason about what to do, an agent must have goals, some model of the world, and a model of the consequences of its actions.

A deterministic **action** is a partial function from states to states. It is partial because not every action can be carried out in every state. For example, a robot cannot carry out the action to pick up a particular object if it is nowhere near the object. The **precondition** of an action specifies when the action can be carried out. The **effect** of an action specifies the resulting state.

Example 8.1 Consider a delivery robot world (page 30) with mail and coffee to deliver. Assume a simplified domain with four locations as shown in Figure 8.1. The robot, called Rob, can buy coffee at the coffee shop, pick up mail in the mail room, move, and deliver coffee and/or mail. Delivering the coffee to Sam's office will stop Sam from wanting coffee. There can be mail waiting at the mail room to be delivered to Sam's office. This domain is quite simple, yet it is rich enough to demonstrate many of the problems in representing actions and in planning.

The state can be described in terms of the following features:

- the robot's location (*RLoc*), which is one of the coffee shop (*cs*), Sam's office (*off*), the mail room (*mr*), or the laboratory (*lab*).
- whether the robot has coffee (*RHC*). Let *rhc* mean Rob has coffee and *rhc* mean Rob does not have coffee.

- whether Sam wants coffee (SWC). Let swc mean Sam wants coffee and $\overline{swc}$ mean Sam does not want coffee.
- whether mail is waiting at the mail room (MW). Let mw mean there is mail waiting and $\overline{mw}$ mean there is no mail waiting.
- whether the robot is carrying the mail (RHM). Let $rh\bar{m}$ mean Rob has mail, and $\bar{r}hm$ mean Rob does not have mail.

Suppose Rob has six actions:

- Rob can move clockwise (mc).
- Rob can move counterclockwise (mcc).
- Rob can pick up coffee if Rob is at the coffee shop. Let puc mean that Rob picks up coffee. The precondition of puc is $rh\bar{c} \wedge RLoc = cs$; that is, Rob can pick up coffee in any state where its location is cs , and it is not already holding coffee. The effect of this action is to make RHC true. It does not affect the other features.
- Rob can deliver coffee if Rob is carrying coffee and is at Sam’s office. Let dc mean that Rob delivers coffee. The precondition of dc is $rh\bar{c} \wedge RLoc = off$. The effect of this action is to make RHC true and make SWC false.
- Rob can pick up mail if Rob is at the mail room and there is mail waiting there. Let pum mean Rob picks up the mail.
- Rob can deliver mail if Rob is carrying mail and at Sam’s office. Let dm mean Rob delivers mail.

Assume that it is only possible for Rob to do one action at a time. We assume that a lower-level controller can implement these actions.

8.1.1 Explicit State-Space Representation

One possible representation of the effect and precondition of actions is to explicitly enumerate the states and, for each state, specify the actions that are possible in that state and, for each state–action pair, specify the state that results from carrying out the action in that state. This would require a table such as the following:

State	Action	Resulting State
s_7	act_{47}	s_{94}
s_7	act_{14}	s_{83}
s_{94}	act_5	s_{33}
...	...	...

The first tuple in this relation specifies that it is possible to carry out action act_{47} in state s_7 and, if it were to be carried out in state s_7 , the resulting state would be s_{94} .

Thus, this is the explicit representation of the actions in terms of a graph. This is called a **state-space graph**. This is the sort of graph that was used in Chapter 3.

Example 8.2 In Example 8.1 (page 350), the states are the quintuples specifying the robot's location, whether the robot has coffee, whether Sam wants coffee, whether mail is waiting, and whether the robot is carrying the mail. For example, the tuple

$$\cdot \quad \overline{lab}, \overline{rhc}, swc, \overline{m\bar{w}}, rhm \quad \Sigma$$

represents the state where Rob is at the Lab, does not have coffee, Sam wants coffee, there is no mail waiting, and Sam has mail.

$$\cdot \quad lab, rhc, swc, m\bar{w}, \overline{rhm} \quad \Sigma$$

represents the state where Rob is at the Lab, carrying coffee, Sam wants coffee, there is mail waiting, and Rob is not holding any mail.

In this example, there are $4 \times 2 \times 2 \times 2 = 64$ states. Intuitively, all of them are possible, even if you would not expect that some of them would be reached by an intelligent robot.

There are six actions, not all of which are applicable in each state.

The actions can be defined in terms of the state transitions:

State	Action	Resulting State
$\cdot \quad \overline{lab}, \overline{rhc}, swc, \overline{m\bar{w}}, rhm \quad \Sigma$	mc	$\cdot \quad mr, rhc, swc, \overline{m\bar{w}}, rhm \quad \Sigma$
$\cdot \quad \overline{lab}, \overline{rhc}, swc, m\bar{w}, rhm \quad \Sigma$	mcc	$\cdot \quad off, \overline{rhc}, swc, m\bar{w}, rhm \quad \Sigma$
$\cdot \quad \overline{off}, \overline{rhc}, swc, \overline{m\bar{w}}, rhm \quad \Sigma$	dm	$\cdot \quad off, \overline{rhc}, swc, \overline{m\bar{w}}, \overline{rhm} \quad \Sigma$
$\cdot \quad \overline{off}, \overline{rhc}, swc, m\bar{w}, rhm \quad \Sigma$	mcc	$\cdot \quad cs, \overline{rhc}, swc, m\bar{w}, rhm \quad \Sigma$
$\cdot \quad \overline{off}, \overline{rhc}, swc, \overline{m\bar{w}}, rhm \quad \Sigma$	mc	$\cdot \quad lab, \overline{rhc}, swc, \overline{m\bar{w}}, rhm \quad \Sigma$
...	...	...

This table shows the transitions for two of the states. The complete problem representation includes the transitions for the other 62 states.

This is not a good representation for three main reasons:

- There are usually too many states to represent, to acquire, and to reason with.
- Small changes to the model mean a large change to the representation. Modeling another feature means changing the whole representation. For example, to model the level of power in the robot, so that it can recharge itself in the Lab, every state has to change.
- There is also usually much more structure and regularity in the effects of actions. This structure can make the specification of the preconditions and the effects of actions, and reasoning about them, more compact and efficient.

An alternative is to model the effects of actions in terms of how the actions affect the features.

8.1.2 Feature-Based Representation of Actions

A feature-based representation of actions models

- which actions are possible in a state, in terms of the values of the features of the state, and
- how the feature values in the next state are affected by the feature values of the current state and the action.

The **precondition** of an action is a proposition that must be true before the action can be carried out. In terms of constraints, the robot is constrained to only be able to choose an action for which the precondition is true.

Example 8.3 In Example 8.1 (page 350), the action of Rob to pick up coffee (*puc*) has precondition *cs rhc*. That is, Rob must be at the coffee shop (*cs*), not carrying coffee (*rhc*). As a constraint, this means that *puc* is not available for any other location or when *rhc* is true.

The action move clockwise is always possible. Its precondition is *true*.

The **feature-based representation of actions** uses rules to specify the value of each variable in the state resulting from an action. The bodies of these rules can include the action carried out and the values of features in the previous state.

The rules have two forms:

- A **causal rule** specifies when a feature gets a new value.
- A **frame rule** specifies when a feature keeps its value.

It is useful to think of these as two separate cases: what makes the feature change its value, and what makes it keep its value.

Example 8.4 In Example 8.1 (page 350), Rob's location depends on its previous location and where it moved. Let $RLoc^t$ be the variable that specifies the location in the resulting state. The following rules specify the conditions under which Rob is at the coffee shop:

$$RLoc^t = cs \leftarrow RLoc = off \wedge Act = mcc.$$

$$RLoc^t = cs \leftarrow RLoc = mr \wedge Act = mc.$$

$$RLoc^t = cs \leftarrow RLoc = cs \wedge Act \neq mcc \wedge Act \neq mc.$$

The first two rules are causal rules and the last rule is a frame rule.

Whether the robot has coffee in the resulting state depends on whether it has coffee in the previous state and its action:

$$rhc^t \leftarrow rhc \wedge Act \neq dc.$$

$$rhc^t \leftarrow Act = puc.$$

The first of these is a frame rule that specifies that the robot having coffee persists unless the robot delivers the coffee. The rule implicitly implies that the robot cannot drop the coffee or lose it, or it cannot be stolen. The second is the causal rule specifying that picking up the coffee causes the robot to have coffee in the next time step.

Causal rules and frame rules do not specify when an action is possible. What is possible is defined by the precondition of the actions.

8.1.3 The STRIPS Representation

The previous representation was feature-centric in that, for each feature, there were rules that specified its value in the state resulting from an action. An alternative is an action-centric representation which, for each action, specifies the effect of the action. One such representation is the **STRIPS representation**. STRIPS, which stands for “STanford Research Institute Problem Solver,” was the planner used in Shakey, one of the first robots built using AI technology.

First, divide the features that describe the world into **primitive** and **derived** features. Definite clauses are used to determine the value of derived features from the values of the primitive features in any given state. The STRIPS representation is used to determine the values of primitive features in a state based on the previous state and the action taken by the agent.

The STRIPS representation is based on the idea that most things are not affected by a single action. For each action, STRIPS models when the action is possible and what primitive features are affected by the action. The effect of the action relies on the **STRIPS assumption**: All of the primitive features not mentioned in the description of the action stay unchanged.

The **STRIPS representation** for an action consists of

- the **precondition**, which is a set of assignments of values to features that must be true for the action to occur, and
- the **effect**, which is a set of resulting assignments of values to those primitive features that change as the result of the action.

Primitive feature V has value v after the action act if $V = v$ was on the effect list of act or if V was not mentioned in the effect list of act , and V had value v immediately before act . Non-primitive features can be derived from the values of the primitive features for each time.

When the variables are Boolean, it is sometimes useful to divide the effects into a **delete list**, which includes those variables made false, and an **add list**, which includes those variables made true.

Example 8.5 In Example 8.1 (page 350), the action of Rob to pick up coffee (pu_c) has the following STRIPS representation:

precondition [$cs, \overline{rhc}$]

effect [rhc]

That is, the robot must be at the coffee shop and not have coffee. After the action, rhc holds (i.e., $RHC = true$), and all other feature values are unaffected by this action.

Example 8.6 The action of delivering coffee (*dc*) can be defined by

precondition [*off*, *rhc*]

effect [$\overline{rhc}$, $\overline{swc}$]

The robot can deliver coffee when it is in the office and has coffee. It can deliver coffee whether Sam wants coffee or not. If Sam wanted coffee before the action, Sam no longer wants it after. Thus, the effects are to make $RHC = false$ and $SWC = false$.

The feature-based representation is more powerful than the STRIPS representation because it can represent anything representable in STRIPS. It can be more verbose because it requires explicit frame axioms, which are implicit in the STRIPS representation.

A STRIPS representation of a set of actions can be translated into the feature-based representation as follows. If the effects list of an action *act* is [$e_1, \dots, e_k$], the STRIPS representation is equivalent to the causal rules

$$e_i^t \leftarrow act.$$

for each e_i that is made true by the action and the frame rules

$$c^t \leftarrow c \wedge act.$$

for each condition *c* that does not involve a variable on the effects list. The precondition of each action in the representations is the same.

A **conditional effect** is an effect of an action that depends on the value of other features. The feature-based representation can specify conditional effects, whereas STRIPS cannot represent these directly.

Example 8.7 Consider representing the action *mc*. The effect of *mc* depends on the robot's location before *mc* was carried out.

The feature-based representation is as in Example 8.4 (page 353).

To represent this in the STRIPS representation, we construct multiple actions that differ in what is true initially. For example, *mc.cs* (move clockwise from coffee shop) has a precondition [$RLoc = cs$] and effect [$RLoc = off$].

8.1.4 Initial States and Goals

In a typical planning problem, where the world is fully observable and deterministic, the initial state is defined by specifying the value for each feature for the initial time.

There are two sorts of goals:

- An **achievement goal** is a proposition that must be true in the final state.
- A **maintenance goal** is a proposition that must be true in every state through which the agent passes. These are often **safety goals** – the goal of staying away from bad states.

There may be other kinds of goals such as transient goals (that must be achieved somewhere in the plan but do not have to hold at the end) or resource goals, such as wanting to minimize energy used or time traveled.

8.2 Forward Planning

A deterministic **plan** is a sequence of actions to achieve a **goal** from a given starting state. A deterministic **planner** is a problem solver that can produce a plan. The input to a planner is an initial world description, a specification of the actions available to the agent, and a goal description. The specification of the actions includes their preconditions and their effects.

One of the simplest planning strategies is to treat the planning problem as a path planning problem in the **state-space graph**. In a state-space graph, nodes are states, and arcs correspond to actions from one state to another. The arcs coming out of a state s correspond to all of the legal actions that can be carried out in that state. That is, for each state s , there is an arc for each action a whose precondition holds in state s , and where the resulting state does not violate a maintenance goal. A plan is a path from the initial state to a state that satisfies the achievement goal.

A **forward planner** searches the state-space graph from the initial state looking for a state that satisfies a goal description. It can use any of the search strategies described in Chapter 3.

Example 8.8 Figure 8.2 shows part of the search space starting from the state where Rob is at the coffee shop, Rob does not have coffee, Sam wants coffee, there is mail waiting, and Rob does not have mail. The search space is the same irrespective of the goal state.

Using a forward planner is not the same as making an explicit state-based representation of the actions (page 351), because the relevant part of the graph can be created dynamically from the representations of the actions.

A complete search strategy, such as A^* with multiple-path pruning or iterative deepening, is guaranteed to find a solution. The complexity of the search space is defined by the forward branching factor (page 75) of the graph. The branching factor is the set of all possible actions at any state, which may be quite large. For the simple robot delivery domain, the branching factor is 3 for the initial situation and is up to 4 for other situations. When the domain becomes bigger, the branching factor increases and so the search space explodes. This complexity may be reduced by finding good heuristics [see Exercise 8.6 (page 369)], but the heuristics have to be very good to overcome the combinatorial explosion.

A state can be represented as either

- (a) *a complete world description*, in terms of an assignment of a value to each primitive proposition or as a proposition that defines the state, or

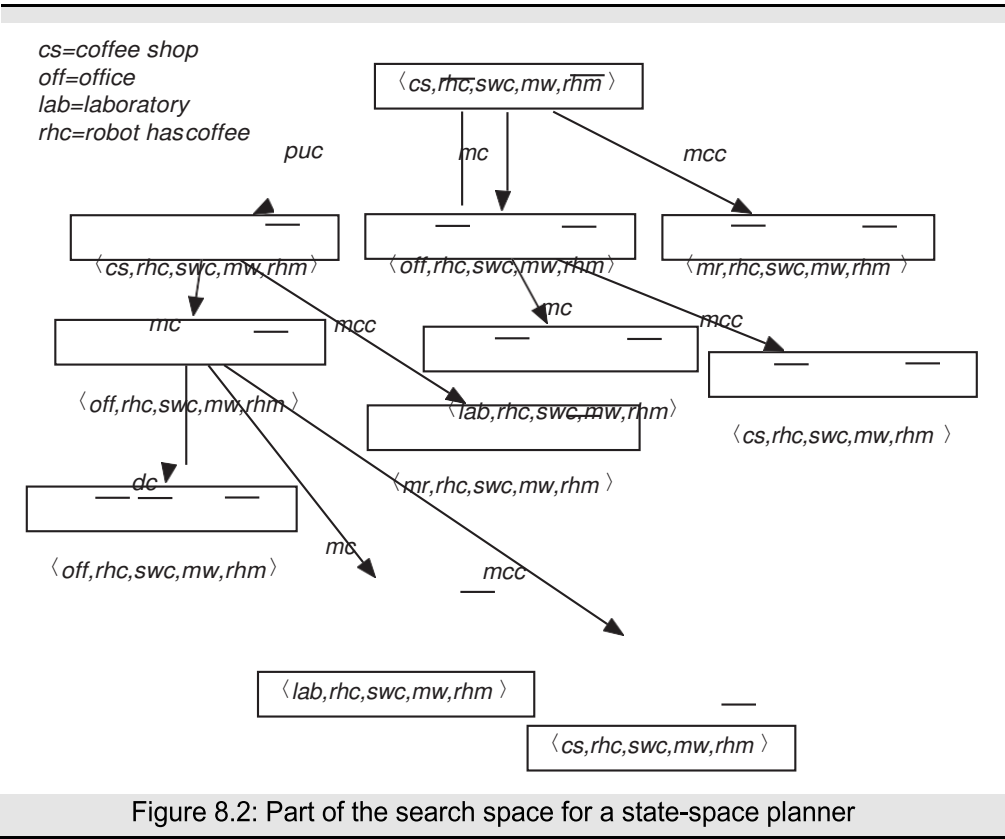


Figure 8.2: Part of the search space for a state-space planner

(b) a path from an initial state; that is, by the sequence of actions that were used to reach that state from the initial state. In this case, what holds in a state can be deduced from the axioms that specify the effects of actions.

The difference between representations (a) and (b) amounts to the difference between computing a whole new world description for each world created, or by calculating what holds in a world as necessary. Alternative (b) may save on space (particularly if there is a complex world description) and will allow faster creation of a new node, but it will be slower to determine what actually holds in any given world. Another difficulty with option (b) is that determining whether two states are the same (e.g., for loop detection or multiple-path pruning) is expensive.

We have presented state-space searching as a forward search method, but it is also possible to search backward from the set of states that satisfy the goal. Whereas the initial state is usually fully specified and so the frontier starts off containing a single state, the goal does not usually fully specify a state and so there would be many goal states that satisfy the goal. This would mean that the frontier is initially very large. Thus, backward search in the state space is often not practical.

8.3 Regression Planning

8.3 Regression Planning

It is often more efficient to search in a different search space – one where the nodes are not states but rather are goals to be achieved. Once the problem has

been transformed into a search problem, any of the algorithms of Chapter 3 can be used. We will only consider achievement goals and not maintenance goals; see Exercise 8.9 (page 369).

Regression planning is searching in the graph defined by the following:

- The nodes are goals that must be achieved. A goal is a set of assignments to (some of) the features.
- The arcs correspond to actions. In particular, an arc from node g to g^t , labeled with action act , means act is the last action that is carried out before goal g is achieved, and the node g^t is the goal that must be true immediately before act so that g is true immediately after act .
- The start node is the goal to be achieved. Here we assume it is a conjunction of assignments of values to features.
- The goal condition for the search, $goal(g)$, is true if all of the elements of g are true of the initial state.

Given a node that represents goal g , a neighbor of g exists for every action act such that

- act is **possible**: it is possible for act to be carried out and for g to be true immediately after act ; and
- act is **useful**: act achieves part of g .

The neighbor of g along the arc labeled with action act is the node g^t defined by the weakest precondition. The **weakest precondition** for goal g to hold after action act is a goal g^t such that

- g^t is true before act implies that g is true immediately after act .
- g^t is “weakest” in the sense that any proposition that satisfies the first condition must imply g^t . This precludes, for example, having unnecessary conditions conjoined onto a precondition.

A set of assignments of values to variables is **consistent** if it assigns at most one value to any variable. That is, it is inconsistent if it assigns two different values to any variable.

Suppose goal $g = X_1 = v_1, \dots, X_n = v_n$ is the node being considered.

Consider computing the neighbors of a node given the feature-based representation of actions. An action act is useful if there is a causal rule that achieves $X_i = v_i$ for some i , using action act . The neighbor of this node along the arc labeled with action act is the proposition

$$precondition(act) \wedge body(X_1 = v_1, act) \wedge \dots \wedge body(X_n = v_n, act)$$

where $body(X_i = v_i, act)$ is the set of assignments of variables in the body of a rule that specifies when $X_i = v_i$ is true immediately after act . There is no such neighbor if there is no corresponding rule for some i , or if the proposition is inconsistent (i.e., assigns different values to a variable). Note that,

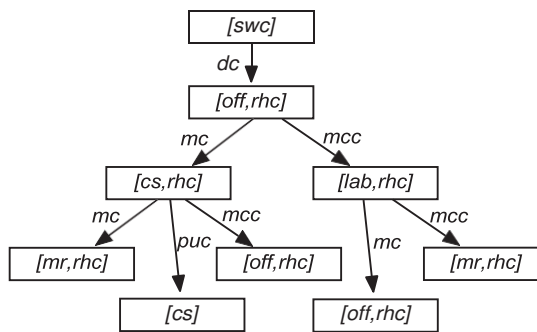


Figure 8.3: Part of the search space for a regression planner

if multiple rules are applicable for the same action, there will be multiple neighbors.

In terms of the STRIPS representation, act is useful for solving g if $X_i = v_i$ is an effect of action act , for some i . Action act is possible unless there is an effect $X_j = v_j$ of act and g contains $X_j = v_j^t$ where $v_j^t \neq v_j$. Immediately before act , the preconditions of act , as well as any $X_k = v_k$ not achieved by act , must hold.

Thus, the neighbor of the goal g on an arc labeled with act is

$$precondition(act) \cup (g \setminus effects(act))$$

as long as it is consistent.

Example 8.9 Suppose the goal is to achieve $\overline{swc}$. The start node is $[swc]$. If this is true in the initial state, the planner stops. If not, it chooses an action that achieves swc . In this case, there is only one: dc . The preconditions of dc are $off \wedge rhc$. Thus, there is one arc:

$$([\overline{swc}], [off, rhc]) \text{ labeled with } dc.$$

Consider the node $[off, rhc]$. There are two actions that can achieve off , namely mc from cs and mcc from lab . There is one action that can achieve rhc , namely puc . However, puc has as a precondition $cs \wedge rhc$, but cs and off are inconsistent (because they involve different assignments to the variable $RLoc$). Thus, puc is not a possible last action; it is not possible that, immediately after puc , the condition $[off, rhc]$ holds.

Figure 8.3 shows the first two levels of the search space (without multipath pruning or loop detection). Note that the search space is the same no matter what the initial state is. The starting state has two roles, first as a stopping criterion and second as a source of heuristics.

The following example shows how a regression planner can recognize what the last action of a plan must be.

Example 8.10 Suppose the goal was for Sam to not want coffee and for the robot to have coffee: $[\overline{swc}, rhc]$. The last action cannot be *dc* to achieve $\overline{swc}$, because this achieves *rhc*. The only last action must be *puc* to achieve *rhc*. Thus, the resulting goal is $[\overline{swc}, cs]$. Again, the last action before this goal cannot be to achieve $\overline{swc}$ because this has as a precondition *off*, which is inconsistent with *cs*. Therefore, the second-to-last action must be a move action to achieve *cs*.

A problem with the regression planner is that a goal may not be achievable. Deciding whether a set of goals is achievable is often difficult to infer from the definitions of the actions. For example, you may be required to know that an object cannot be at two different places at the same time; sometimes this is not explicitly represented and is only implicit in the effects of an action, and the fact that the object is only in one position initially. To perform consistency pruning, the regression planner can use domain knowledge to prune the search space.

Loop detection and multiple-path pruning may be incorporated into a regression planner. The regression planner does not have to visit exactly the same node to prune the search. If the goal represented by a node *n* implies a goal on the path to *n*, node *n* can be pruned. Similarly, for multiple-path pruning, see Exercise 8.11 (page 369).

A regression planner commits to a particular total ordering of actions, even if no particular reason exists for one ordering over another. This commitment to a total ordering tends to increase the complexity of the search space if the actions do not interact much. For example, it tests each permutation of a sequence of actions when it may be possible to show that no ordering succeeds.

8.4 Planning as a CSP

In forward planning, the search is constrained by the initial state and only uses the goal as a stopping criterion and as a source for heuristics. In regression planning, the search is constrained by the goal and only uses the start state as a stopping criterion and as a source for heuristics. It is possible to go forward and backward in the same planner by using the initial state to prune what is not reachable and the goal to prune what is not useful. This can be done by converting a planning problem to a constraint satisfaction problem (CSP) and using one of the CSP methods from Chapter 4.

For the CSP representation, it is also useful to describe the actions in terms of features – to have a factored representation of actions as well as a factored representation of states. The features representing actions are called **action features** and the features representing states are called **state features**.

Example 8.11 Another way to model the actions of Example 8.1 (page 350) is that, at each step, Rob gets to choose

- whether it will pick up coffee. Let *PUC* be a Boolean variable that is true when Rob picks up coffee.
- whether it will deliver coffee. Let *DelC* be a Boolean variable that is true when Rob delivers coffee.
- whether it will pick up mail. Let *PUM* be a Boolean variable that is true when Rob picks up mail.
- whether it will deliver mail. Let *DelM* be a Boolean variable that is true when Rob delivers mail.
- whether it moves. Let *Move* be a variable with domain $\{mc, mcc, nm\}$ that specifies whether Rob moves clockwise, moves counterclockwise, or does not move (*nm* means “not move”).

To construct a CSP from a planning problem, first choose a fixed **horizon**, which is the number of time steps over which to plan. Suppose this number is k . The CSP has the following variables:

- a variable for each state feature and each time from 0 to k . If there are n such features, there are $n(k + 1)$ such variables.
- a variable for each action feature for each time in the range 0 to $k - 1$. These are called action variables. The action at time t represents the action that takes the agent from the state at time t to the state at time $t + 1$.

There are a number of types of constraints:

- **State constraints** are constraints among variables at the same time step. These can include physical constraints on the state or can ensure that states that violate maintenance goals (page 355) are forbidden.
- **Precondition constraints** between state variables at time t and action variables at time t specify constraints on what actions are available from a state.
- **Effect constraints** among state variables at time t , action variables at time t , and state variables at time $t + 1$ constrain the values of the state variables at time $t + 1$ in terms of the actions and the previous state.
- **Action constraints** specify which actions cannot co-occur. These are sometimes called mutual exclusion or **mutex constraints**.
- **Initial-state constraints** are constraints on the initial state (at time 0). These constrain the initial state to be the current state of the agent. If there is a unique initial state, it can be represented as a set of domain constraints on the state variables at time 0.
- **Goal constraints** constrain the final state to be a state that satisfies the achievement goal. These are domain constraints if the goal is for certain variables to have particular values at the final step, but they can also be more general constraints – for example, if two variables must have the same value.

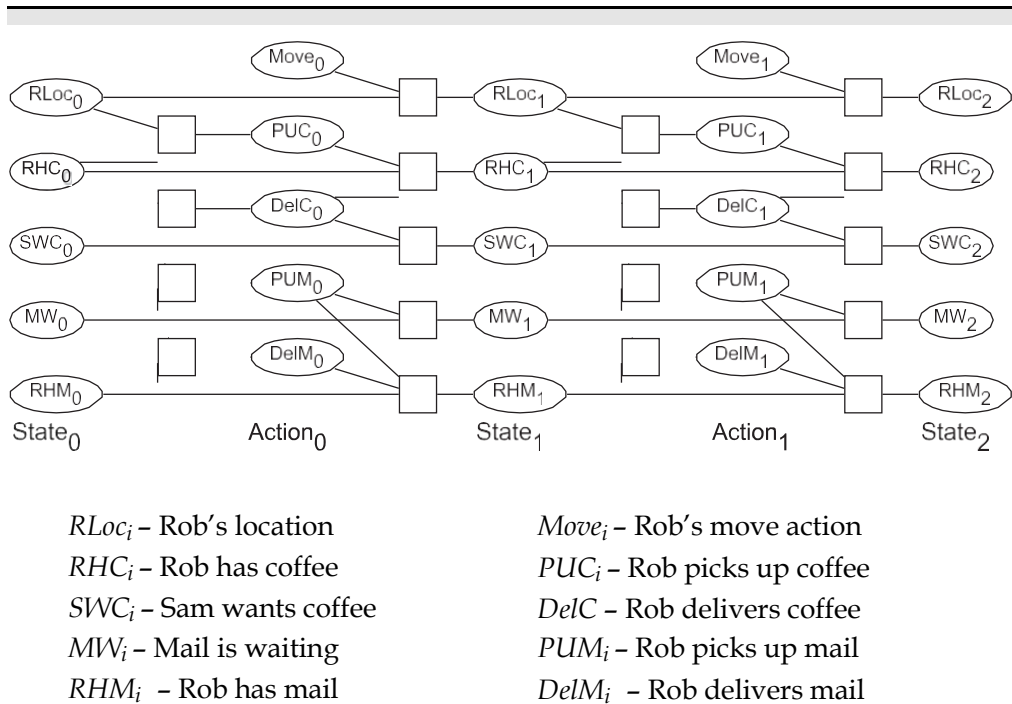


Figure 8.4: The delivery robot CSP planner for a planning horizon of 2

Example 8.12 Figure 8.4 shows a CSP representation of the delivery robot example, with a planning horizon of 2. There are three copies of the state variables: one at time 0, the initial state; one at time 1; and one at time 2, the final state. There are action variables for times 0 and 1.

There are no domain constraints in this example. You could make a constraint that says Rob cannot carry both the mail and coffee, or that Rob cannot carry mail when there is mail waiting, if these were true in the domain. They are not included here.

The constraints to the left of the actions are the precondition constraints, which specify what values the action variables can take (i.e., what actions are available for each of the action variables). The *Move_i* variables have no preconditions: all moves are available in all states. The *PUM_i* variable, which specifies whether Rob can pick up mail, depends on Rob's location at time *i* (i.e., the value of *RLoc_i*) and whether there is mail waiting at time *i* (*MW_i*). The negation of an action (e.g., *PUM_i = false*, written as $\overline{pum_i}$) is always available, assuming that the agent can choose not to perform an action. The action *PUM_i = true*, written as *pum_i*, is only available when the location is *mr* and *MW_i = true*. When the precondition of an action is a conjunction, it can be written as a set of constraints, because the constraints in a CSP are implicitly conjoined. If the precondition is a more complex proposition, it can be represented as a constraint involving more than two variables.

The constraints to the left of the state variables at times 1 and later indicate the values of the state variables as a function of the previous state and the

action. For example, the following constraint is among RHC_i , DC_i , PUC_i , and whether the robot has coffee in the subsequent state, RHC_{i+1} :

RHC_i	DC_i	PUC_i	RHC_{i+1}
true	true	true	true
true	true	false	false
true	false	true	true
true	false	false	true
false	true	true	true
false	true	false	false
false	false	true	true
false	false	false	false

This table represents the same constraint as the rules of Example 8.4 (page 353).

Example 8.13 Consider finding a plan to get Sam coffee, with a planning horizon of 2.

Initially, Sam wants coffee but the robot does not have coffee. This can be represented as two domain constraints: one on SWC_0 and one on RHC_0 . The goal is that Sam no longer wants coffee. This can be represented as the domain constraint $SWC_2 = false$.

Just running arc consistency on this network results in $RLoc_0 = cs$ (the robot has to start in the coffee shop), $PUC_0 = true$ (the robot has to pick up coffee initially), $Move_0 = mc$ (the robot has to move to the office), and $DC_1 = true$ (the robot has to deliver coffee at time 1).

The CSP representation assumes a fixed planning horizon (i.e., a fixed number of steps). To find a plan over any number of steps, the algorithm can be run for a horizon of $k = 0, 1, 2$, until a solution is found. For the stochastic local searching algorithm, it is possible to search multiple horizons at once, searching for all horizons, k from 0 to n , and allowing n to increase slowly. When solving the CSP using arc consistency and search, it may be possible to determine that trying a longer plan will not help. That is, by analyzing why no solution exists for a horizon of n steps, it may be possible to show that there can be no plan for any length greater than n . This will enable the planner to halt when there is no plan. See Exercise 8.12 (page 369).

8.5 Partial-Order Planning

The forward and regression planners enforce a total ordering on actions at all stages of the planning process. The CSP planner commits to the particular time that the action will be carried out. This means that those planners have to commit to an ordering of actions that cannot occur concurrently when adding them to a partial plan, even if there is no particular reason to put one action before another.

The idea of a **partial-order planner** is to have a partial ordering between actions and only commit to an ordering between actions when forced. This is

sometimes also called a **non-linear planner**, which is a misnomer because such planners often produce a linear plan.

A partial ordering is a less-than relation that is transitive and asymmetric. A **partial-order plan** is a set of actions together with a partial ordering, representing a “before” relation on actions, such that any total ordering of the actions, consistent with the partial ordering, will solve the goal from the initial state. Write $act_0 < act_1$ if action act_0 is before action act_1 in the partial order. This means that action act_0 must occur before action act_1 .

For uniformity, treat *start* as an action that achieves the relations that are true in the initial state, and treat *finish* as an action whose precondition is the goal to be solved. The pseudoaction *start* is before every other action, and *finish* is after every other action. The use of these as actions means that the algorithm does not require special cases for the initial situation and for the goals. When the preconditions of *finish* hold, the goal is solved.

An action, other than *start* or *finish*, will be in a partial-order plan to achieve a precondition of an action in the plan. Each precondition of an action in the plan is either true in the initial state, and so achieved by *start*, or there will be an action in the plan that achieves it.

We must ensure that the actions achieve the conditions they were assigned to achieve. Each precondition P of an action act_1 in a plan will have an action act_0 associated with it such that act_0 achieves precondition P for act_1 . The triple act_0, P, act_1 is a **causal link**. The partial order specifies that action act_0 occurs before action act_1 , which is written as $act_0 < act_1$. Any other action A that makes P false must either be before act_0 or after act_1 .

Informally, a partial-order planner works as follows: Begin with the actions *start* and *finish* and the partial order $start < finish$. The planner maintains an agenda that is a set of (P, A) pairs, where A is an action in the plan and P is an atom that is a precondition of A that must be achieved. Initially the agenda contains pairs $(G, finish)$, where G is an atom that must be true in the goal state. At

each stage in the planning process, a pair G, act_1 is selected from the agenda, where P is a precondition for action act_1 . Then an action, act_0 , is chosen to achieve

P . That action is either already in the plan – it could be the *start* action, for example – or it is a new action that is added to the plan. Action act_0 must happen before act_1 in the partial order. It adds a causal link that records that act_0 achieves P for action act_1 . Any action in the plan that deletes P must happen either before act_0 or after act_1 . If act_0 is a new action, its preconditions are added to the agenda, and the process continues until the agenda is empty. This is a non-deterministic procedure. The “choose” and the “either ... or ...” form choices that must be searched over. There are two choices that require search:

- which action is selected to achieve G and
- whether an action that deletes G happens before act_0 or after act_1 .

The algorithm *PartialOrderPlanner* is given in Figure 8.5.

```

1: non-deterministic procedure PartialOrderPlanner(Gs)
2:   Inputs
3:     Gs: set of atomic propositions to achieve
4:   Output
5:     linear plan to achieve Gs
6:   Local
7:     Agenda: set of  $\langle P, A \rangle$  pairs where  $P$  is atom and  $A$  an action
8:     Actions: set of actions in the current plan
9:     Constraints: set of temporal constraints on actions
10:    CausalLinks: set of  $(act_0, P, act_1)$  triples
11:    Agenda :=  $\{ \langle G, finish \rangle : G \in Gs \}$ 
12:    Actions :=  $\{ start, finish \}$ 
13:    Constraints :=  $\{ start < finish \}$ 
14:    CausalLinks :=  $\{ \}$ 
15:    repeat
16:      select and remove  $\langle G, act_1 \rangle$  from Agenda
17:      either
18:        choose  $act_0 \in Actions$  such that  $act_0$  achieves  $G$ 
19:      Or
20:        choose  $act_0 \notin Actions$  such that  $act_0$  achieves  $G$ 
21:        Actions := Actions  $\cup \{ act_0 \}$ 
22:        Constraints := add const( $start < act_0$ , Constraints)
23:        for each  $CL \in CausalLinks$  do
24:          Constraints := protect( $CL, act_0, Constraints$ )
25:        Agenda := Agenda  $\cup \{ \langle P, act_0 \rangle : P \text{ is a precondition of } act_0 \}$ 
26:        Constraints := add const( $act_0 < act_1$ , Constraints)
27:        CausalLinks := CausalLinks  $\cup \{ (act_0, G, act_1) \}$ 
28:        for each  $A \in Actions$  do
29:          Constraints := protect( $(act_0, G, act_1), A, Constraints$ )
30:    until Agenda =  $\{ \}$ 
31:    return total ordering of Actions consistent with Constraints

```

Figure 8.5: Partial-order planner

The function *add const*($act_0 < act_1$, *Constraints*) returns the constraints formed by adding the constraint $act_0 < act_1$ to *Constraints*, and it fails if $act_0 < act_1$ is incompatible with *Constraints*. There are many ways this function can be implemented. See Exercise 8.13.

The function *protect*($(act_0, G, act_1), A$) checks whether $A \neq act_0$ and $A \neq act_1$ and A deletes G . If so, either $A < act_0$ is added to the set of constraints or $act_1 < A$ is added to the set of constraints. This is a non-deterministic choice that is searched over.

Example 8.14 Consider the goal $\overline{swc} \wedge \overline{mw}$, where the initial state contains $RLoc = lab, swc, rhc, mw, rhm$.

Initially the agenda is

$(\overline{swc}, finish), (\overline{mw}, finish)$.

Suppose $(\overline{swc}, finish)$ is selected and removed from the agenda. One action exists that can achieve swc , namely deliver coffee, dc , with preconditions off and rhc . At the end of the **repeat** loop, *Agenda* contains

$(off, dc), (rhc, dc), (\overline{mw}, finish)$.

Constraints is $start < finish, start < dc, dc < finish$. There is one causal link, $dc, \overline{swc}, finish$. This causal link means that no action that undoes swc is allowed to happen after dc and before $finish$.

Suppose $(\overline{mw}, finish)$ is selected from the agenda. One action exists that can achieve this, pum , with preconditions mw and $RLoc = mr$. The causal link $pum, \overline{mw}, finish$ is added to the set of causal links; mw, pum and mr, pum are added to the agenda.

Suppose mw, pum is selected from the agenda. The action $start$ achieves mw , because mw is true initially. The causal link $start, mw, pum$ is added to the set of causal links. Nothing is added to the agenda.

At this stage, there is no ordering imposed between dc and pum .

Suppose off, dc is removed from the agenda. There are two actions that can achieve off : $mc\ cs$ with preconditions cs , and $mcc\ lab$ with preconditions lab . The algorithm searches over these choices. Suppose it chooses $mc\ cs$. Then the causal link $(mc\ cs, off, dc)$ is added.

The first violation of a causal link occurs when a move action is used to achieve mr, pum . This action violates the causal link $mc\ cs, off, dc$, and so must happen after dc (the robot goes to the mail room after delivering coffee) or before $mc\ cs$.

The preceding algorithm has glossed over one important detail. It is sometimes necessary to perform some action more than once in a plan. The preceding algorithm will not work in this case, because it will try to find a partial ordering with both instances of the action occurring at the same time. To fix this problem, the ordering should be between action instances, and not actions themselves. To implement this, assign an index to each instance of an action in the plan, and the ordering is on the action instance indexes and not the actions themselves. This is left as an exercise.

8.6 Review

The following are the main points you should have learned from this chapter:

- Planning is the process of choosing a sequence of actions to achieve a goal.
- An action is a function from a state to a state. A number of representations exploit structure in representation of states. In particular, the feature-based

representation of actions represents what must be true in the previous state for a feature to have a value in the next state. The STRIPS representation is an action-based representation that specifies the effects of actions.

- Different planning algorithms can be used to convert a planning problem into a search problem.

8.7 References and Further Reading

The STRIPS representation was developed by [Fikes and Nilsson \[1971\]](#).

There is much ongoing research into how to plan sequences of actions. [Yang \[1997\]](#) presents a textbook overview of planning. For a collection of classic papers, see [Allen, Hendler, and Tate \[1990\]](#).

Forward planning has been used successfully for planning in the blocks world, where some good heuristics have been identified by [Bacchus and Kanbanza \[1996\]](#). (See Exercise 8.6 (page 369).)

Regression planning was pioneered by [Waldinger \[1977\]](#). The use of weakest preconditions is based on the work of [Dijkstra \[1976\]](#), where it was used to define the semantics of imperative programming languages. This should not be too surprising because the commands of an imperative language are actions that change the state of the computer.

Planning as CSP is based on Graphplan [[Blum and Furst, 1997](#)] and Satplan [[Kautz and Selman, 1996](#)]. The treatment of planning as a CSP is also investigated by [Lopez and Bacchus \[2003\]](#) and [van Beek and Chen \[1999\]](#). [Bryce and Kambhampati \[2007\]](#) give a recent survey.

Partial-order planning was introduced in [Sacerdoti's \[1975\]](#) NOAH and followed up in [Tate's \[1977\]](#) NONLIN system, [Chapman's \[1987\]](#) TWEAK algorithm, and [McAllester and Rosenblitt's \[1991\]](#) systematic non-linear planning (SNLP) algorithm. See [Weld \[1994\]](#) for an overview of partial-order planning and see [Kambhampati, Knoblock, and Yang \[1995\]](#) for a comparison of the algorithms. The version presented here is basically SNLP (but see Exercise 8.15). See [Wilkins \[1988\]](#) for a discussion on practical issues in planning. See [Weld \[1999\]](#), [McDermott and Hendler \[1995\]](#), and [Nau \[2007\]](#) and associated papers for a recent overview.

Chapter 14

Relational Planning, Learning,

Figure 3.13: A grid-searching problem

and Probabilistic Reasoning

What is now required is to give the greatest possible development to mathematical logic, to allow to the full the importance of relations, and then to found upon this secure basis a new philosophical logic, which may hope to borrow some of the exactitude and certainty of its mathematical foundation. If this can be successfully accomplished, there is every reason to hope that the near future will be as great an epoch in pure philosophy as the immediate past has been in the principles of mathematics. Great triumphs inspire great hopes; and pure thought may achieve, within our generation, such results as will place our time, in this respect, on a level with the greatest age of Greece.

– Bertrand Russell [1917]

The representation scheme dimension (page 20) has, as its top level, reasoning in terms of individuals and relations. Reasoning in terms of relations allows for compact representations that can be built before the agent encounters particular individuals. When an agent is introduced to a new individual, it can infer relationships about that individual. This chapter outlines how, in three areas of AI, feature-based representations are expandable to (also) deal with individuals and relations. In each of these areas, the relational representation benefits from being able to be built before the individuals are known and, therefore, before the features are known. As Russell points out in the quote above, relational reasoning brings great advantages over propositional (and feature-based) representations.

14.1 Planning with Individuals and Relations

An agent's goals and its environment are often described in terms of individuals and relations. When the agent's knowledge base is built, and before the agent knows the objects it should reason about, it requires a representation that is independent of the individuals. Thus, it must go beyond feature-based representations. When the individuals become known, it is possible to ground out the representations in terms of features. Often, it is useful to reason in terms of the non-grounded representations.

With a relational representation, we can reify time. **Time** can be represented in terms of individuals that are points in time or temporal intervals. This section presents two relational representations that differ in how time is reified.

14.1.1 Situation Calculus

The idea behind **situation calculus** is that (reachable) states are definable in terms of the actions required to reach them. These reachable states are called situations. What is true in a situation can be defined in terms of relations with the situation as an argument. Situation calculus can be seen as a relational version of the feature-based representation of actions (page 353).

Here we only consider single agents, a fully observable environment, and deterministic actions.

Situation calculus is defined in terms of situations. A **situation** is either

- *init*, the initial situation, or
- $do(A, S)$, the situation resulting from doing action *A* in situation *S*, if it is possible to do action *A* in situation *S*.

Example 14.1 Consider the domain of Figure 3.1 (page 73). Suppose in the initial situation, *init*, the robot, Rob, is at location *o109* and there is a key *k1* at the mail room and a package at *storage*.

$$do(move(rob, o109, o103), init)$$

is the situation resulting from Rob moving from position *o109* in situation *init* to position *o103*. In this situation, Rob is at *o103*, the key *k1* is still at *mail*, and the package is at *storage*.

The situation

$$do(move(rob, o103, mail), \\ do(move(rob, o109, o103), \\ init))$$

is one in which the robot has moved from position *o109* to *o103* to *mail* and is currently at mail. Suppose Rob then picks up the key, *k1*. The resulting situation is

$$do(pickup(rob, k1), \\ do(move(rob, o103, mail),$$

$$do(move(rob, o109, o103), \\ init))).$$

In this situation, Rob is at position *mail* carrying the key *k1*.

A situation can be associated with a state. There are two main differences between situations and states:

- Multiple situations may refer to the same state if multiple sequences of actions lead to the same state. That is, equality between situations is not the same as equality between states.
- Not all states have corresponding situations. A state is **reachable** if a sequence of actions exists that can reach that state from the initial state. States that are not reachable do not have a corresponding situation.

Some $do(A, S)$ terms do not correspond to any state. However, sometimes an agent must reason about such a (potential) situation without knowing if A is possible in state S , or if S is possible.

Example 14.2 The term $do(unlock(rob, door1), init)$ does not denote a state at all, because it is not possible for Rob to unlock the door when Rob is not at the door and does not have the key.

A **static** relation is a relation for which the truth value does not depend on the situation; that is, its truth value is unchanging through time. A **dynamic** relation is a relation for which the truth value depends on the situation. To represent what is true in a situation, predicate symbols denoting dynamic relations have a situation argument so that the truth can depend on the situation. A predicate symbol with a situation argument is called a **fluent**.

Example 14.3 The relation $at(O, L, S)$ is true when object O is at location L in situation S . Thus, at is a fluent.

The atom

$$at(rob, o109, init)$$

is true if the robot *rob* is at position *o109* in the initial situation. The atom

$$at(rob, o103, do(move(rob, o109, o103), init))$$

is true if robot *rob* is at position *o103* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation. The atom

$$at(k1, mail, do(move(rob, o109, o103), init))$$

is true if *k1* is at position *mail* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation.

A dynamic relation is axiomatized by specifying the situations in which it is true. Typically, this is done inductively in terms of the structure of situations.

- Axioms with *init* as the situation parameter are used to specify what is true in the initial situation.
- A **primitive** relation is defined by specifying when it is true in situations of the form $do(A, S)$ in terms of what is true in situation S . That is, primitive relations are defined in terms of what is true at the previous situation.
- A **derived** relation is defined using clauses with a variable in the situation argument. The truth of a derived relation in a situation depends on what else is true in the same situation.
- Static relations are defined without reference to the situation.

Example 14.4 Suppose the delivery robot, Rob, is in the domain depicted in Figure 3.1 (page 73). Rob is at location *o109*, the parcel is in the storage room, and the key is in the mail room. The following axioms describe this initial situation:

$at(rob, o109, init).$
 $at(parcel, storage, init).$
 $at(k1, mail, init).$

The *adjacent* relation is a dynamic, derived relation defined as follows:

$adjacent(o109, o103, S).$
 $adjacent(o103, o109, S).$
 $adjacent(o109, storage, S).$
 $adjacent(storage, o109, S).$
 $adjacent(o109, o111, S).$
 $adjacent(o111, o109, S).$
 $adjacent(o103, mail, S).$
 $adjacent(mail, o103, S).$
 $adjacent(lab2, o109, S).$
 $adjacent(P_1, P_2, S) \leftarrow$
 $\quad between(Door, P_1, P_2) \wedge$
 $\quad unlocked(Door, S).$

Notice the free S variable; these clauses are true for all situations. We cannot omit the S because which rooms are adjacent depends on whether a door is unlocked. This can change from situation to situation.

The *between* relation is static and does not require a situation variable:

$between(door1, o103, lab2).$

We also distinguish whether or not an agent is being carried. If an object is not being carried, we say that the object is sitting at its location. We distinguish this case because an object being carried moves with the object carrying it. An object is at a location if it is sitting at that location or is being carried by an object at

that location. Thus, *at* is a derived relation:

$$\begin{aligned} at(Ob, P, S) \leftarrow & \\ & sitting\ at(Ob, P, S). \\ at(Ob, P, S) \leftarrow & \\ & carrying(Ob1, Ob, S) \wedge \\ & at(Ob1, P, S). \end{aligned}$$

Note that this definition allows for Rob to be carrying a bag, which, in turn, is carrying a book.

The **precondition** (page 350) of an action specifies when it is possible to carry out the action. The relation $poss(A, S)$ is true when action *A* is possible in situation *S*. This is typically a derived relation.

Example 14.5 An agent can always put down an object it is carrying:

$$\begin{aligned} poss(putdown(Ag, Obj), S) \leftarrow & \\ & carrying(Ag, Obj, S). \end{aligned}$$

For the *move* action, an autonomous agent can move from its current position to an adjacent position:

$$\begin{aligned} poss(move(Ag, P_1, P_2), S) \leftarrow & \\ & autonomous(Ag) \wedge \\ & adjacent(P_1, P_2, S) \wedge \\ & sitting\ at(Ag, P_1, S). \end{aligned}$$

The precondition for the unlock action is more complicated. The agent must be at the correct side of the door and carrying the appropriate key:

$$\begin{aligned} poss(unlock(Ag, Door), S) \leftarrow & \\ & autonomous(Ag) \wedge \\ & between(Door, P_1, P_2) \wedge \\ & at(Ag, P_1, S) \wedge \\ & opens(Key, Door) \wedge \\ & carrying(Ag, Key, S). \end{aligned}$$

We do not assume that the *between* relation is symmetric. Some doors can only open one way.

We define what is true in each situation recursively in terms of the previous situation and of what action occurred between the situations. As in the feature-based representation of actions (page 353), **causal rules** specify when a relation becomes true and **frame rules** specify when a relation remains true.

Example 14.6 The primitive *unlocked* relation can be defined by specifying how different actions can affect its being true. The door is unlocked in the situation resulting from an unlock action, as long as the unlock action was possible. This is represented using the following causal rule:

$$\text{unlocked}(\text{Door}, \text{do}(\text{unlock}(\text{Ag}, \text{Door}), S)) \leftarrow \\ \text{poss}(\text{unlock}(\text{Ag}, \text{Door}), S).$$

Suppose the only action to make the door locked is to lock the door. Thus, *unlocked* is true in a situation following an action if it was true before, if the action was not to lock the door, and if the action was possible:

$$\text{unlocked}(\text{Door}, \text{do}(A, S)) \leftarrow \\ \text{unlocked}(\text{Door}, S) \wedge \\ A \neq \text{lock}(\text{Door}) \wedge \\ \text{poss}(A, S).$$

This is a frame rule.

Example 14.7 The *carrying* predicate can be defined as follows.

An agent is carrying an object after picking up the object:

$$\text{carrying}(\text{Ag}, \text{Obj}, \text{do}(\text{pickup}(\text{Ag}, \text{Obj}), S)) \leftarrow \\ \text{poss}(\text{pickup}(\text{Ag}, \text{Obj}), S).$$

The only action that undoes the *carrying* predicate is the *putdown* action. Thus, *carrying* is true after an action if it was true before the action, and the action was not to put down the object. This is represented in the frame rule:

$$\text{carrying}(\text{Ag}, \text{Obj}, \text{do}(A, S)) \leftarrow \\ \text{carrying}(\text{Ag}, \text{Obj}, S) \wedge \\ \text{poss}(A, S) \wedge \\ A \neq \text{putdown}(\text{Ag}, \text{Obj}).$$

Example 14.8 The atom *sitting at*(*Obj*, *Pos*, *S*₁) is true in a situation *S*₁ resulting from object *Obj* moving to *Pos*, as long as the action was possible:

$$\text{sitting-at}(\text{Obj}, \text{Pos}, \text{do}(\text{move}(\text{Obj}, \text{Pos}_0, \text{Pos}), S)) \leftarrow \\ \text{poss}(\text{move}(\text{Obj}, \text{Pos}_0, \text{Pos}), S).$$

The other action that makes *sitting at* true is the *putdown* action. An object is sitting at the location where the agent who put it down was located:

$$\text{sitting-at}(\text{Obj}, \text{Pos}, \text{do}(\text{putdown}(\text{Ag}, \text{Obj}), S)) \leftarrow \\ \text{poss}(\text{putdown}(\text{Ag}, \text{Obj}), S) \wedge \\ \text{at}(\text{Ag}, \text{Pos}, S).$$

The only other time that *sitting at* is true in a (non-initial) situation is when it was true in the previous situation and it was not undone by an action. The

only actions that undo *sitting-at* is a *move* action or a *pickup* action. This can be specified by the following frame axiom:

$$\begin{aligned} \text{sitting-at}(\text{Obj}, \text{Pos}, \text{do}(A, S)) \leftarrow \\ \text{poss}(A, S) \wedge \\ \text{sitting-at}(\text{Obj}, \text{Pos}, S) \wedge \\ \forall \text{Pos}_1 A \neq \text{move}(\text{Obj}, \text{Pos}, \text{Pos}_1) \wedge \\ \forall \text{Ag } A \neq \text{pickup}(\text{Ag}, \text{Obj}). \end{aligned}$$

Note that the quantification in the body is not the standard quantification for rules. This can be represented using negation as failure (page 537):

$$\begin{aligned} \text{sitting-at}(\text{Obj}, \text{Pos}, \text{do}(A, S)) \leftarrow \\ \text{poss}(A, S) \wedge \\ \text{sitting-at}(\text{Obj}, \text{Pos}, S) \wedge \\ \sim \text{move-action}(A, \text{Obj}, \text{Pos}) \wedge \\ \sim \text{pickup-action}(A, \text{Obj}). \\ \text{move\_action}(\text{move}(\text{Obj}, \text{Pos}, \text{Pos}_1), \text{Obj}, \text{Pos}). \\ \text{pickup\_action}(\text{pickup}(\text{Ag}, \text{Obj}), \text{Obj}). \end{aligned}$$

These clauses are designed not to have a free variable in the scope of the negation.

Example 14.9 Situation calculus can represent more complicated actions than can be represented with simple addition and deletion of propositions in the state description.

Consider the *drop everything* action in which an agent drops everything it is carrying. In situation calculus, the following axiom can be added to the definition of *sitting-at* to say that everything the agent was carrying is now on the ground:

$$\begin{aligned} \text{sitting-at}(\text{Obj}, \text{Pos}, \text{do}(\text{drop-everything}(\text{Ag}), S)) \leftarrow \\ \text{poss}(\text{drop-everything}(\text{Ag}), S) \wedge \\ \text{at}(\text{Ag}, \text{Pos}, S) \wedge \\ \text{carrying}(\text{Ag}, \text{Obj}, S). \end{aligned}$$

A frame axiom for *carrying* specifies that an agent is not carrying an object after a *drop everything* action.

$$\begin{aligned} \text{carrying}(\text{Ag}, \text{Obj}, \text{do}(A, S)) \leftarrow \\ \text{poss}(A, S) \wedge \\ \text{carrying}(\text{Ag}, \text{Obj}, S) \wedge \\ A \neq \text{drop-everything}(\text{Ag}) \wedge \\ A \neq \text{putdown}(\text{Ag}, \text{Obj}). \end{aligned}$$

The *drop everything* action thus affects an unbounded number of objects.

Situation calculus is used for **planning** by asking for a situation in which a goal is true. Answer extraction (page 504) is used to find a situation in which the goal is true. This situation can be interpreted as a sequence of actions for the agent to perform.

Example 14.10 Suppose the goal is for the robot to have the key $k1$. The following query asks for a situation where this is true:

$$?carrying(rob, k1, S).$$

This query has the following answer:

$$S = do(pickup(rob, k1), \\ do(move(rob, o103, mail), \\ do(move(rob, o109, o103), \\ init))).$$

The preceding answer can be interpreted as a way for Rob to get the key: it moves from $o109$ to $o103$, then to $mail$, where it picks up the key.

The goal of delivering the parcel (which is, initially, in the lounge, lng) to $o111$ can be asked with the query

$$?at(parcel, o111, S).$$

This query has the following answer:

$$S = do(move(rob, o109, o111), \\ do(move(rob, lng, o109), \\ do(pickup(rob, parcel), \\ do(move(rob, o109, lng), init)))).$$

Therefore, Rob should go to the lounge, pick up the parcel, go back to $o109$, and then go to $o111$.

Using the top-down proof procedure (page 509) on the situation calculus definitions is very inefficient, because a frame axiom is almost always applicable. A complete proof procedure, such as iterative deepening, searches through all permutations of actions even if they are not relevant to the goal. The use of answer extraction does not negate the necessity for efficient planners, such as the ones in Chapter 8.

14.1.2 Event Calculus

The second representation, **event calculus**, models how the truth value of relations changes because of events occurring at certain times. Time can be modeled as either continuous or discrete.

Events are modeled as occurring at particular times. Event E occurring at time T is written as $event(E, T)$.

Events make some relations true and some no longer true:

- $initiates(E, R, T)$ is true if event E makes primitive relation R true at time T .
- $terminates(E, R, T)$ is true if event E makes primitive relation R no longer true at time T .

Time T is a parameter to *initiates* and *terminates* because the effect of an event can depend on what else is true at the time. For example, the effect of attempting to unlock a door depends on the position of the robot and whether it is carrying the appropriate key.

Relations are either true or false at any time. In event calculus, relations are reified, where $holds(R, T)$ means that relation R is true at time T . This is analogous to having T as the last argument to R in situation calculus. The use of the meta-predicate *holds* allows general rules that are true for all relations.

Derived relations are defined in terms of primitive relations and other derived relations for the same time.

Primitive relation R holds at time T if an event occurred before T that made R true, and there was no intervening event that made R no longer true. This can be specified as follows:

$$\begin{aligned}
 holds(R, T) \leftarrow & \\
 & event(E, T_0) \wedge \\
 & T_0 < T \wedge \\
 & initiates(E, R, T_0) \wedge \\
 & \sim clipped(R, T_0, T). \\
 clipped(R, T_0, T) \leftarrow & \\
 & event(E_1, T_1) \wedge \\
 & terminates(E_1, R, T_1) \wedge \\
 & T_0 < T_1 \wedge \\
 & T_1 < T.
 \end{aligned}$$

The atom $clipped(R, T_0, T)$ means there is an event between times T_0 and T that makes R no longer true; $T_0 < T_1$ is true if time T_0 is before time T_1 . Here $\sim$ indicates negation as failure (page 193), and so these clauses mean their completion.

Actions are represented in terms of what properties they initiate and terminate. As in situation calculus, the preconditions of actions are specified using the *poss* relation.

Example 14.11 The *pickup* action initiates a *carrying* relation, and it terminates a *sitting_at* relation as long as the preconditions for *pickup* are true:

$$\begin{aligned}
 & \text{initiates}(\text{pickup}(\text{Ag}, \text{Obj}), \text{carrying}(\text{Ag}, \text{Obj}), T) \leftarrow \\
 & \quad \text{poss}(\text{pickup}(\text{Ag}, \text{Obj}), T). \\
 & \text{terminates}(\text{pickup}(\text{Ag}, \text{Obj}), \text{sitting\_at}(\text{Obj}, \text{Pos}), T) \leftarrow \\
 & \quad \text{poss}(\text{pickup}(\text{Ag}, \text{Obj}), T). \\
 & \text{poss}(\text{pickup}(\text{Ag}, \text{Obj}), T) \leftarrow \\
 & \quad \text{autonomous}(\text{Ag}) \wedge \\
 & \quad \text{Ag} \neq \text{Obj} \wedge \\
 & \quad \text{holds}(\text{at}(\text{Ag}, \text{Pos}), T) \wedge \\
 & \quad \text{holds}(\text{sitting\_at}(\text{Obj}, \text{Pos}), T).
 \end{aligned}$$

This implies that if a *pickup* is attempted when the preconditions do not hold, nothing happens. It is also possible to write clauses that specify what happens under different circumstances, such as when a pickup is attempted for an object that is being held by something else.

Given particular action occurrences, and making the complete knowledge assumption that all intervening events are specified, the top-down proof procedure with negation as failure can be used to prove what is true. For planning, the agent can use abduction (page 199).

14.2 Learning with Individuals and Relations

Learning in terms of individuals and relations allows for learning where the structure of relationships is important for prediction. The learning methods presented in Chapter 7 assumed that the feature values are meaningful; something useful could be predicted about a case by using the values of features. However, when the examples are about individuals and relations, the value of a property may be a meaningless name of an individual. It is only by considering the attributes of the individual that the name denotes, and its relationship to other individuals, that an agent is able produce meaningful predictions. The learned knowledge can also be applied to individuals that did not appear in the training set.

Because much of this work has been done in the context of logic programming, it is often called **inductive logic programming**.

Example 14.12 Suppose a trading agent the following data set, in terms of the individual-property-value **triples** (page 552), about which resorts a person likes:

Individual	Property	Value
joe	likes	resort_14
joe	dislikes	resort_35
...	...	...
resort_14	type	resort
resort_14	near	beach_18
beach_18	type	beach
beach_18	covered_in	ws
ws	type	sand
ws	color	white
...	...	...

The agent wants to learn what Joe likes. What is important is not the value of the *likes* property, which is just a meaningless name, but the properties of the individual denoted by the name. Feature-based representations cannot do anything with this data set beyond learning that Joe likes *resort_14*, but not *resort_35*.

The sort of theory the agent should learn is that Joe likes resorts near sandy beaches. This theory can be expressed as a logic program:

$$\begin{aligned}
 \text{prop}(\text{joe}, \text{likes}, R) \leftarrow & \\
 & \text{prop}(R, \text{type}, \text{resort}) \wedge \\
 & \text{prop}(R, \text{near}, B) \wedge \\
 & \text{prop}(B, \text{type}, \text{beach}) \wedge \\
 & \text{prop}(B, \text{covered\_in}, S) \wedge \\
 & \text{prop}(S, \text{type}, \text{sand}).
 \end{aligned}$$

Logic programs provide the ability to be able to represent deterministic theories about individuals and relations. This rule can be applied to resorts and beaches that Joe has not visited.

The input to an inductive logic programming learner includes the following:

- A is a set of atoms whose definitions the agent is learning.
- E^+ is the set of ground instances of elements of A , called the **positive examples**, that are observed to be true.
- E^- is the set of ground instances of elements of A , called the **negative examples**, that are observed to be false.
- B , the **background knowledge**, is a set of clauses that define relations that can be used in the learned logic programs.
- H is a space of possible hypotheses. H is often represented implicitly as a set of operators that can generate the possible hypotheses. We assume that each hypothesis is a logic program.

Example 14.13 In Example 14.12 (page 606), suppose the agent wants to learn what Joe likes. In this case, the inputs are

- $A = \{prop(joe, likes, R)\}$
- $E^+ = \{prop(joe, likes, resort\ 14), \dots\}$. For the example that follows, we assume there are many such items that Joe likes.
- $E^- = \{prop(joe, likes, resort\ 35), \dots\}$. Note that these are written in the positive form; they have been observed to be false.
- $B = \{prop(resort\ 14, type, resort), prop(resort\ 14, near, beach\ 18), \dots\}$. This set contains all of the facts about the world that are not instances of A . The agent is not learning about these.
- H is a set of logic programs defining $prop(joe, likes, R)$. The heads of the clauses unify with $prop(joe, likes, R)$. H is too big to enumerate.

All of these, except for H , are given explicitly in the formulation of the problem.

The aim is to find a simplest hypothesis $h \in H$ such that

$$B \wedge h \models E^+ \text{ and}$$

$$B \wedge h \not\models E^-.$$

That is, the hypothesis implies the positive evidence and does not imply the negative evidence. It must be **consistent** with the negative evidence being false.

We, thus, want to find an element of the version space (page 328). This is similar to the definition of abduction (page 199), in which the knowledge base in abduction corresponds to the background knowledge. The hypothesis space of inductive logic programming is the set of logic programs. The second condition corresponds to consistency.

For the rest of this section, assume that there is a single target n -ary relation whose definition we want to learn, and for which we use the predicate symbol t . That is, $A = \{t(X_1, \dots, X_n)\}$. The hypothesis space consists of possible definitions for this relation using a logic program.

There are two main strategies used in inductive logic programming:

- The first strategy is to start with the simplest hypothesis and make it more complicated to fit the data. Because the logic program only states positive facts, the simplest hypothesis is the most general hypothesis (page 329), which is simply that $t(X_1, \dots, X_n)$ is always true. This hypothesis implies the positive examples, but it also implies the negative examples, if any exist. This strategy involves a general-to-specific search. It tries to find the simplest hypothesis that fits the data by searching the hypothesis space from the simplest hypothesis to more complex hypotheses, always implying the positive examples, until a hypothesis that does not imply the negative examples is found.
- The second strategy is to start with a hypothesis that fits the data and to make it simpler while still fitting the data. A hypothesis that fits the data is the set of positive examples. This strategy involves a specific-to-general search: start with the very specific hypothesis in which only the positive

examples are true, and then generalize the clauses, avoiding the negative cases.

We will expand on the general-to-specific search for definite clauses. The initial hypothesis contains a single clause:

$$\{t(X_1, \dots, X_n) \leftarrow\}.$$

A **specialization operator** takes a set G of clauses and returns a set S of clauses that specializes G . To specialize means that $S \models G$.

The following are three primitive specialization operators:

- Split a clause in G on condition c . Clause $a \leftarrow b$ in G is replaced by two clauses: $a \leftarrow b \wedge c$ and $a \leftarrow b \wedge \neg c$.
- Split a clause $a \leftarrow b$ in G on a variable X that appears in a or b . Clause $a \leftarrow b$ is replaced by the clauses

$$a \leftarrow b \wedge X = t_1.$$

...

$$a \leftarrow b \wedge X = t_k.$$

where the t_i are terms.

- Remove a clause that is not necessary to prove the positive examples.

The last operation changes the predictions of the clause set. Those cases no longer implied by the clauses are false.

These primitive specialization operators are used together to form the operators of H . The operators of H are combinations of primitive specialization operations designed so that progress can be evaluated using a **greedy** look-ahead. That is, the operators are defined so that one step is adequate to evaluate progress.

The first two primitive specialization operations should be carried out judiciously to ensure that the simplest hypothesis is found. An agent should carry out the splitting operations only if they make progress. Splitting is only useful when combined with clause removal. For example, adding an atom to the body of a clause is equivalent to splitting on the atom and then removing the clause containing the negation of the atom. A higher-level specialization operator may be to split on the atom $prop(X, type, T)$ for some variable X that appears in the clause, split on the values of T , and remove the resulting clauses that are not required to imply the positive examples. This operator makes progress on determining which types are useful.

Figure 14.1 (on the next page) shows a **local search** algorithm for the top-down induction of a logic program. It maintains a single hypothesis that it iteratively improves until finding a hypothesis that fits the data or until it fails to find such a hypothesis.

```

1: procedure TDInductiveLogicProgram( $t, B, E^+, E^-, R$ )
2:   Inputs
3:      $t$ : an atom whose definition is to be learned
4:      $B$ : background knowledge is a logic program
5:      $E^+$ : positive examples
6:      $E^-$ : negative examples
7:      $R$ : set of specialization operators
8:   Output
9:   logic program that classifies  $E^+$  positively and  $E^-$  negatively or  $\perp$  if
     no program can be found
10:  Local
11:     $H$  is a set of clauses
12:     $H \leftarrow \{t(X_1, \dots, X_n) \leftarrow\}$ 
13:    while there is  $e \in E^-$  such that  $B \cup H \models e$  do
14:      if there is  $r \in R$  such that  $B \cup r(H) \models E^+$  then
15:        Select  $r \in R$  such that  $B \cup r(H) \models E^+$ 
16:         $H \leftarrow r(H)$ 
17:      else
18:        return  $\perp$ 
19:  return  $H$ 

```

Figure 14.1: Top-down induction of a logic program

At each time step it chooses an operator to apply with the constraint that every hypothesis entails the positive examples. This algorithm glosses over two important details:

- which operators to consider and
- which operator to select.

The operators should be at a level so that they can be evaluated according to which one is making progress toward a good hypothesis. In a manner similar to decision-tree learning (page 301), the algorithm can perform a myopically optimal choice. That is, it chooses the operator that makes the most progress in minimizing the error. The error of a hypothesis can be the number of negative examples that are implied by the hypothesis.

Example 14.14 Consider Example 14.12 (page 606), in which the agent must learn about Joe's likes and dislikes.

The first hypothesis is that Joe likes everything:

$$\{prop(joe, likes, R) \leftarrow \},$$

which is inconsistent with the negative evidence.

It can split on properties that contain the individuals that Joe likes. The only way for the specialization to make progress – to prove fewer negative examples

while implying all positive examples – is to contain the variable R in the body. Thus, it must consider triples in which R is the subject or the object.

- It could consider splitting on the property *type*, splitting on the value of *type*, and keeping only those that are required to prove the positive examples. This results in the following clause (assuming the positive examples only include resorts):

$$\{prop(joe, likes, R) \leftarrow prop(R, type, resort)\}.$$

There can be other clauses if the positive examples include things other than resorts that Joe likes. If the negative examples include non-resorts, this split would be useful in reducing the error.

- It could consider splitting on other properties that can be used in proofs of the positive examples, such as *near*, resulting in

$$\{prop(joe, likes, R) \leftarrow prop(R, near, B)\}$$

if all of the positive examples are near something. If some of the negative examples are not near anything, this specialization could be useful in reducing the error.

It could then choose whichever of these splits makes the most progress. In the next step it could add the other one, other properties of R , or properties of B .

14.3 Probabilistic Relational Models

The belief network probability models of Chapter 6 were defined in terms of features. Many domains are best modeled in terms of individuals and relations. Agents must often build probabilistic models before they know what individuals are in the domain and, therefore, before they know what random variables exist. When the probabilities are being learned, the probabilities often do not depend on the individuals. Although it is possible to learn about an individual, an agent must also learn general knowledge that it can apply when it finds out about a new individual.

Example 14.15 Consider the problem of an intelligent tutoring system diagnosing students' arithmetic errors. From observing a student's performance on a number of examples, the tutor should try to determine whether or not the student understands the task and, if not, work out what the student is doing wrong so that appropriate remedies can be applied.

Consider the case of diagnosing two-digit addition of the form

$$\begin{array}{r} x_1 \ x_0 \\ + \ y_1 \ y_0 \\ \hline z_2 \ z_1 \ z_0 \end{array}$$

The student is given the values for the x 's and the y 's and provides values for the z 's.

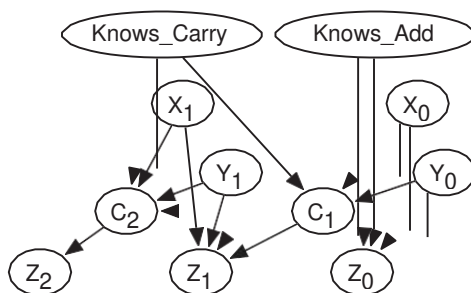


Figure 14.2: Belief network for two-digit addition

Students' answers depend on whether they know basic addition and whether they know how to carry. A belief network for this example is shown in Figure 14.2. The carry into digit i , given by the variable C_i , depends on the x -value, the y -value, and the carry for the previous digit, and on whether the student knows how to carry. The z -value for digit i , given by the variable Z_i , depends on the x -value, the y -value, and the carry for digit i , in addition to whether the student knows basic addition.

By observing the value of the x 's and the y 's in the problem and the value of z 's given by the student, the posterior probability that the student knows addition and knows how to carry can be inferred. One feature of the belief network model is that it allows students to make random errors; even though they know arithmetic, they can still get the wrong answer.

The problem with this representation is that it is inflexible. A flexible representation would allow for the addition of multiple digits, multiple problems, multiple students, and multiple times. Multiple digits require the replication of the network for the digits. Multiple times allow for modeling how students' knowledge and their answers change over time, even if the problems do not change over time.

If the conditional probabilities were stored as tables, the size of these tables would be enormous. For example, if the X_i , Y_i , and Z_i variables each have a domain size of 11 (the digits 0 to 9 or the blank), and the C_i and $Knows\_Add$ variables are binary, a tabular representation of

$$P(Z_1 | X_1, Y_1, C_1, Knows\_Add)$$

would have a size greater than 4,000. There is much more structure in the conditional probability than is expressed in the tabular representation. The representation that follows allows for both relational probabilistic models and compact descriptions of conditional probabilities.

A **probabilistic relational model (PRM)** or a **relational probability model** is a model in which the probabilities are specified on the relations, independently of the actual individuals. Different individuals share the probability parameters.

A **parametrized random variable** is either a logical atom or a term (page 494). That is, it is of the form $p(t_1, \dots, t_n)$, where each t_i is a logical variable or a constant. The parametrized random variable is said to be parametrized by the logical variables that appear in it. A ground instance of a parametrized random variable is obtained by substituting constants for the logical variables in the parametrized random variable. The ground instances of a parametrized random variable correspond to random variables. The random variables are Boolean if p is a predicate symbol. If p is a function symbol, the domain of the random variables is the range of p .

We use the convention that logical variables are in upper case. Random variables correspond to ground atoms and ground terms, so, for this section, these are written in lower case.

Example 14.16 For a PRM of the multidigit arithmetic problem of the previous example, there is a separate x -variable for each digit D and for each problem P , represented by the parametrized random variable $x(D, P)$. Thus, for example, $x(1, prob17)$ may be a random variable representing the x -value of the first digit of problem 17. Similarly there is a parametrized random variable, $y(D, P)$, which represents a random variable for each digit D and problem P .

There is a variable for each student S and time T that represents whether S knows how to add properly at time T . The parametrized random variable $knows\_add(S, T)$ represents whether student S knows addition at time T . The random variable $knows\_add(fred, mar23)$ is true if Fred knows addition on March 23. Similarly, there is a parametrized random variable $knows\_carry(S, T)$.

There is a different z -value and a different carry for each digit, problem, student, and time. These values are represented by the parametrized random variables $z(D, P, S, T)$ and $carry(D, P, S, T)$. So, $z(1, prob17, fred, mar23)$ is a random variable representing the answer Fred gave on March 23 for digit 1 of problem 17. Function z has range $\emptyset, \dots, 9, blank$, so the random variables all have this set as their domain.

A **plate model** consists of

- a directed graph in which the nodes are parameterized random variables,
- a population of individuals for each logical variable, and
- a conditional probability of each node given its parents.

A plate model means its grounding – the belief network in which nodes are all ground instances of the parametrized random variables (each logical variable replaced by an individual in its population). The conditional probabilities of the belief network are the corresponding instances of the plate model.

Example 14.17 Figure 7.17 (page 339) gives a hierarchical Bayesian model as (a) a plate model and (b) its grounding. The plate representation, on the left of the figure, has four parametrized random variables: $s(X, H)$, $\varphi(H)$, a_1 , and a_2 . The population for X is the set of all patients. The population for H is the set of all hospitals. The grounding of the network is shown on the right of the

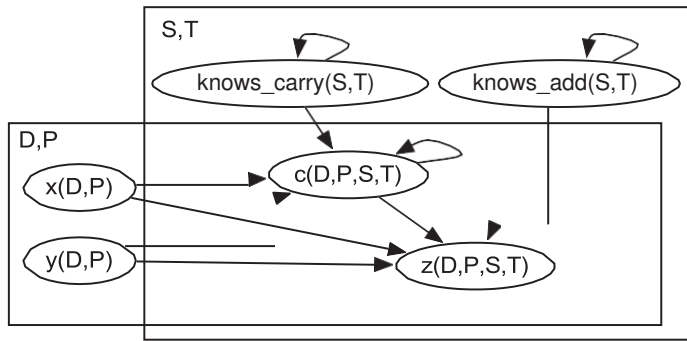


Figure 14.3: Belief network with plates for multidigit addition

figure. There is a separate random variable $\varphi(h)$ for each hospital h , and a separate random variable $s(x, h)$ for each patient x and hospital h . There is a conditional probability of each $s(x, h)$ given the corresponding $\varphi(h)$, a conditional probability of each $\varphi(h)$ given a_1 and a_2 , and prior probabilities of a_1 and a_2 . The $s(x, h)$ variables in the grounding all have the same conditional probability given parent. The variables $s(x_1, h)$ and $s(x_2, h)$, for different patients x_1 and x_2 , are independent of each other given $\varphi(h)$. The variables $s(x_1, h_1)$ and $s(x_2, h_2)$, for different hospitals h_1 and h_2 , are independent of each other given a_1 and a_2 .

Example 14.18 A plate model with the parametrized random variables of Example 14.16 is shown in Figure 14.3. The rectangles correspond to plates. For the plate labeled with D, P , an instance of each variable exists for each digit D and person P . One way to view this is that the instances come out of the page, like a stack of plates. Similarly, for the plate labeled S, T , there is a copy of the variables for each student S and each time T . For the variables in the intersection of the plates, there is a random variable for each digit D , person P , student S , and time T . Note that the notation here is redundant; the plates provide the same information as the explicit parameters; the parameters for c have to be D, P, S, T because these are the plates it is in.

The plate representation denotes the same independence as a belief network (page 240); each node is independent of its non-descendants given its parents. In this case, the diagram describes its grounding. Thus, for particular values $d \in \text{dom}(D)$, $p \in \text{dom}(P)$, $s \in \text{dom}(S)$, and $t \in \text{dom}(T)$, $z(d, p, s, t)$ is a random variable, with parents $x(d, p)$, $y(d, p)$, $c(d, p, s, t)$ and $\text{knows\_add}(s, t)$. There is a loop in the plate model on the $c(D, P, S, T)$ parametrized random variable because the carry for one digit depends on the carry for the previous digit for the same problem, student, and time. Similarly, whether students know how to carry at some time depends on whether they knew how to carry at the previous time. However, the conditional probability tables are such that the ground network is acyclic.

There is a conditional probability of each parametrized random variable, given its parents. This conditional probability is shared among its ground instances.

Unfortunately, the plate representation is not adequate when the dependency occurs among different instances of the same relation. In the preceding example, $c(D, P, S, T)$ depends, in part, on $c(D-1, P, S, T)$, that is, on the carry from the previous digit (and there is some other case for the first digit). A more complex example is to determine the probability that two authors are collaborators, which depends on whether they have written a paper in common. To represent such examples, it is useful to be able to specify how the logical variables interact, as is done in logic programs.

One representation that combines the ideas of belief networks, plates, and logic programs is the **independent choice logic (ICL)**. The ICL consists of a set of independent choices, a logic program that gives the consequences of the choices, and probability distributions over the choices. In more detail, the ICL is defined as follows:

An **alternative** is a set of atoms (page 494) all sharing the same logical variables. A **choice space** is a set of alternatives such that none of the atoms in the alternatives unify with each other. An ICL theory contains

- a choice space C . Let C^t be the set of ground instances of the alternatives. Thus, C^t is a set of sets of ground atoms.
- an acyclic logic program (that can include negation as failure), in which the head of the clauses doesn't unify with an element of an alternative in the choice space.
- a probability distribution over each alternative.

The atoms in the logic program and the choice space can contain constants, variables, and function symbols.

A **selector function** selects a single element from each alternative in C^t . There exists a **possible world** for each selector function. The logic program specifies what is true in each possible world. Atom g is true in a possible world if it follows from the atoms selected by the selector function added to the logic program. The probability of proposition g is given by a measure over sets of possible worlds, where the atoms in different ground instances of the alternatives are probabilistically independent. The instances of an alternative share the same probabilities, and the probabilities of different instances are multiplied.

The acyclicity of the logic program means that only a finite number of alternatives exist with atoms involved in proofs of any ground g . So, although there may be infinitely many possible worlds, for any g , the probability measure can be defined on the sets of possible worlds that are describable in terms of a finite number of alternatives. A description of these worlds can be found by making the atoms in the alternatives assumable (page 187), with different atoms in the same alternative inconsistent, and by using abduction (page 199) to find descriptions of the sets of worlds in which g is true.

An ICL theory can be seen as a causal model (page 206) in which the causal mechanism is specified as a logic program and the background variables, corresponding to the alternatives, have independent probability distributions over them. It may seem that this logic, with only unconditionally independent atoms and a deterministic logic program, is too weak to represent the sort of knowledge required. However, even without logical variables, the independent choice logic can represent anything that can be represented in a belief network.

Example 14.19 Consider representing the belief network of Example 6.10 (page 236) in the ICL. The same technique works for any belief network.

Fire and tampering have no parents, so they can be represented directly as alternatives:

$\{fire, nofire\},$
 $\{tampering, notampering\}.$

The probability distribution over the first alternative is $P(fire) = 0.01$, $P(nofire) = 0.99$. Similarly, $P(tampering) = 0.02$, $P(notampering) = 0.89$.

The dependence of *Smoke* on *Fire* can be represented using two alternatives:

$\{smokeWhenFire, nosmokeWhenFire\},$
 $\{smokeWhenNoFire, nosmokeWhenNoFire\},$

with $P(smokeWhenFire) = 0.9$ and $P(smokeWhenNoFire) = 0.01$. Two rules can be used to specify when there is smoke:

$smoke \leftarrow fire \wedge smokeWhenFire.$
 $smoke \leftarrow \sim fire \wedge smokeWhenNoFire.$

To represent how *Alarm* depends on *Fire* and *Tampering*, there are four alternatives:

$\{alarmWhenTamperingFire, noalarmWhenTamperingFire\},$
 $\{alarmWhenNoTamperingFire, noalarmWhenNoTamperingFire\},$
 $\{alarmWhenTamperingNoFire, noalarmWhenTamperingNoFire\},$
 $\{alarmWhenNoTamperingNoFire, noalarmWhenNoTamperingNoFire\},$

where $P(alarmWhenTamperingFire) = 0.5$, $P(alarmWhenNoTamperingFire) = 0.99$, and similarly for the other atoms using the probabilities from Example 6.10 (page 236). There are also rules specifying when *alarm* is true, depending on tampering and fire:

$alarm \leftarrow tampering \wedge fire \wedge alarmWhenTamperingFire.$
 $alarm \leftarrow \sim tampering \wedge fire \wedge alarmWhenNoTamperingFire.$
 $alarm \leftarrow tampering \wedge \sim fire \wedge alarmWhenTamperingNoFire.$
 $alarm \leftarrow \sim tampering \wedge \sim fire \wedge alarmWhenNoTamperingNoFire.$

Other random variables are represented analogously, using the same number of alternatives as there are assignments of values to the parents of a node.

An ICL representation of a conditional probability can be seen as a rule form of a decision tree (page 298) with probabilities at the leaves. There is a rule and an alternative for each branch. Non-binary alternatives are useful when non-binary variables are involved.

The independent choice logic may not seem very intuitive for representing standard belief networks, but it can make complicated relational models much simpler, as in the following example.

Example 14.20 Consider the parametrized version of the multidigit addition of Example 14.16 (page 613). The plates correspond to logical variables.

There are three cases for the value of $z(D, P, S, T)$. The first is when the student knows addition at this time, and the student did not make a mistake. In this case, they get the correct answer:

$$\begin{aligned} z(D, P, S, T, V) \leftarrow & \\ & x(D, P, Vx) \wedge \\ & y(D, P, Vy) \wedge \\ & carry(D, P, S, T, Vc) \wedge \\ & knowsAddition(S, T) \wedge \\ & \sim mistake(D, P, S, T) \wedge \\ & V \text{ is } (Vx + Vy + Vc) \text{ div } 10. \end{aligned}$$

Here we write the atom $z(D, P, S, T, V)$ to mean that the parametrized random variable $z(D, P, S, T)$ has value V . This is a standard logical rule and has the same meaning as a definite clause.

There is an alternative for whether or not the student happened to make a mistake in this instance:

$$\forall P \forall S \forall T \{ noMistake(D, P, S, T), mistake(D, P, S, T) \},$$

where the probability of $mistake(D, P, S, T)$ is 0.05, assuming students make an error in 5% of the cases even when they know how to do arithmetic.

The second case is when the student knows addition at this time but makes a mistake. In this case, we assume that the students are equally likely to pick each of the digits:

$$\begin{aligned} z(D, P, S, T, V) \leftarrow & \\ & knowsAddition(S, T) \wedge \\ & mistake(D, P, S, T) \wedge \\ & selectDig(D, P, S, T, V). \end{aligned}$$

There is an alternative that specifies which digit the student chose:

$$\forall P \forall S \forall T \{ selectDig(D, P, S, T, V) \mid V \in \{0..9\} \}.$$

Suppose that, for each v , the probability of $selectDig(D, P, S, T, v)$ is 0.1.

The final case is when the student does not know addition. In this case, the student selects a digit at random:

$$\begin{aligned} z(D, P, S, T) = V \leftarrow \\ \sim \text{knowsAddition}(S, T) \wedge \\ \text{selectDig}(D, P, S, T) = V. \end{aligned}$$

These three rules cover all of the rules for z ; it is much simpler than the table of size greater than 4,000 that was required for the tabular representation and it also allows for arbitrary digits, problems, students, and times. Different digits and problems give different values for $x(D, P)$, and different students and times have different values for whether they know addition.

The rules for *carry* are similar. The main difference is that the carry in the body of the rule depends on the previous digit.

Whether a student knows addition at any time depends on whether they know addition at the previous time. Presumably, the student's knowledge also depends on what actions occur (what the student and the teacher do). Because the ICL allows standard logic programs (with "noise"), either of the representations for modeling change introduced at the start of this chapter can be used.

AILog, as used in the previous chapters, also implements ICL.

14.4 Review

The following are the main points you should have learned from this chapter:

- Relational representations are used when an agent requires models to be given or learned before it knows what individuals it will encounter.
- Many of the representations in earlier chapters can be made relational.
- Situation calculus represents time in terms of the action of an agent, using the *init* constant and the *do* function.
- Event calculus allows for continuous and discrete time and axiomatizes what follows from the occurrence of events.
- Inductive logic programming can be used to learn relational models, even when the values of features are meaningless names.
- The independent choice logic allows for the specification of probabilistic models before the individuals are known.

14.5 References and Further Reading

Situation calculus was proposed by McCarthy and Hayes [1969]. The form of the frame axioms presented here can be traced back to Kowalski [1979], Schuber [1990], and Reiter [1991]. Reiter [2001] presents a comprehensive overview of situation calculus; see also Brachman and Levesque [2004]. There have been many other suggestions about how to solve the **frame problem**, which is the problem of concisely specifying what does not change during an action. Event

Relational, Identity, and Existence Uncertainty

The models presented in Section 14.3 are concerned with **relational uncertainty**; uncertainty about whether a relation is true of some individuals. For example, we could have a probabilistic model of *likes*(X, Y) for the case where $X = Y$, which models when different people like each other. This could depend on external characteristics of X and Y . We could also have a probabilistic model for *friends*(X, X), which models whether someone likes themselves. This could depend on internal characteristics of themselves. Such models would be useful for a tutoring system to determine whether two people should work together, or whether a student's errors are due to the difficulty of the material or the student's personality.

Given individuals *sam* and *chris*, we can use the first of these models for *likes*(*sam*, *chris*) only if we know that Sam and Chris are different people. The problem of **identity uncertainty** concerns uncertainty of whether, or not, two terms denote the same individual. This is a problem for medical systems, where it is important to determine whether the person who is interacting with the system now is the same person as one who visited yesterday. This problem is particularly difficult if the patient is non-communicative or wants to deceive the system, for example to get drugs. In the medical community, this is the problem of **record linkage**, as the problem is to determine which medical records are for the same people.

In all of the above cases, the individuals are known to exist; to give names to Chris and Sam presupposes that they exist. Given a description, the problem of determining if there exists an individual that fits the description is the problem of **existence uncertainty**. Existence uncertainty is problematic because there may be no individuals who fit the description or there may be multiple individuals. We cannot give properties to non-existent individuals, because individuals that do not exist do not have properties. If we want to give a name to an individual exists, we need to be concerned about which individual we are referring to if there are multiple individuals that exist. One particular case of existence uncertainty is **number uncertainty**, about the number of individuals that exist. For example, a purchasing agent may be uncertain about the number of people who would be interested in a package tour, and whether to offer the tour depends on the number of people who may be interested.

Reasoning about existence uncertainty can be very tricky if there are complex roles involved, and the problem is to determine if there are individuals to fill the roles. Consider a purchasing agent who must find an apartment for Sam and her son Chris. Whether Sam wants an apartment probabilistically depends, in part, on the size of her room and the color of Chris' room. However, individual apartments do not come labeled with Sam's room and Chris' room, and there may not exist a room for each of them. Given a model of an apartment Sam would want, it isn't obvious how to even condition on the observations.

calculus was proposed by [Kowalski and Sergot \[1986\]](#). [Shanahan \[1997\]](#) provides an excellent introduction to the issues involved in representing change and to the frame problem in particular.

For overviews of inductive logic programming see [Muggleton and De Raedt \[1994\]](#), [Muggleton \[1995\]](#), and [Quinlan and Cameron-Jones \[1995\]](#).

Independent choice logic was proposed by [Poole \[1993, 1997\]](#). [De Raedt, Frasconi, Kersting, and Muggleton \[2008\]](#) and [Getoor and Taskar \[2007\]](#) are collections of papers that provide overviews on probabilistic relational models and how they can be learned.

PART 2 AI ROBOTICS

Contents:

Overview

Chapter 1: From Teleoperation to Autonomy

Chapter 2: The Hierarchical Paradigm

Chapter 3: Biological Foundations of the Reactive Paradigm

Chapter 4: The Reactive Paradigm

Chapter 5: Designing a Reactive Implementation

Chapter 6: Common Sensing Technique for Reactive Robots

Chapter 7: The Hybrid Deliberative/Reactive Paradigm

Chapter 8: Multiple Mobile Robots

Overview

The eight chapters in this part are devoted to describing what is *AI robotics* and the three major paradigms for achieving it. These paradigms characterize the ways in which intelligence is organized in robots. This part of the book also covers architectures that provide exemplars of how to transfer the principles of the paradigm into a coherent, reusable implementation on a single robot or teams of robots.

What Are Robots?

One of the first questions most people have about robotics is “what is a robot?” followed immediately by “what can they do?”

In popular culture, the term “robot” generally connotes some anthropomorphic (human-like) appearance; consider robot “arms” for welding. The tendency to think about robots as having a human-like appearance may stem from the origins of the term “robot.” The word “robot” came into the popular consciousness on January 25, 1921, in Prague with the first performance of Karel Capek’s play, *R.U.R.* (Rossum’s Universal Robots).³⁷ In *R.U.R.*, an unseen inventor, Rossum, has created a race of workers made from a vat of biological parts, smart enough to replace a human in any job (hence “universal”). Capek

described the workers as robots, a term derived from the Czech

word “robota” which is loosely translated as menial laborer. Robot workers implied that the artificial creatures were strictly meant to be servants to free “real” people from any type of labor, but were too lowly to merit respect. This attitude towards robots has disastrous consequences, and the moral of the rather socialist story is that work defines a person.

The shift from robots as human-like servants constructed from biological parts to human-like servants made up of mechanical parts was probably due to science fiction. Three classic films, *Metropolis* (1926), *The Day the Earth Stood Still* (1951), and *Forbidden Planet* (1956), cemented the connotation that robots were mechanical in origin, ignoring the biological origins in Capek’s play. Meanwhile, computers were becoming commonplace in industry and accounting, gaining a perception of being literal minded. Industrial automation confirmed this suspicion as robot arms were installed which would go through the motions of assembling parts, even if there were no parts. Eventually, the term robot took on nuances of factory automation: mindlessness and good only for well-defined repetitious types of work. The notion of anthropomorphic, mechanical, and literal-minded robots complemented the viewpoint taken in many of the short stories in Isaac Asimov’s perennial favorite collection, *I, Robot*.¹⁵ Many (but not all) of these stories involve either a “robotpsychologist,” Dr. Susan Calvin, or two erstwhile trouble shooters, Powell and Donovan, diagnosing robots who behaved logically but did the wrong thing.

The shift from human-like mechanical creatures to whatever shape gets the job done is due to reality. While robots are mechanical, they don’t have to be anthropomorphic or even animal-like. Consider robot vacuum cleaners; they look like vacuum cleaners, not janitors. And the HelpMate Robotics, Inc., robot which delivers hospital meals to patients to permit nurses more time with patients, looks like a cart, not a nurse.

It should be clear from Fig. I.1 that appearance does not form a useful definition of a robot. Therefore, the definition that will be used in this book is *an intelligent robot is a mechanical creature which can function autonomously*. “Intelligent” implies that the robot does not do things in a mindless, repetitive way; it is the opposite of the connotation from factory automation. The “mechanical creature” portion of the definition is an acknowledgment of the fact that our scientific technology uses mechanical building blocks, not biological components (although with recent advances in cloning, this may change). It also emphasizes that a

robot is not the same as a computer. A robot may use a computer as a building block, equivalent to a nervous system or brain, but the robot is able to interact with its world: move around, change

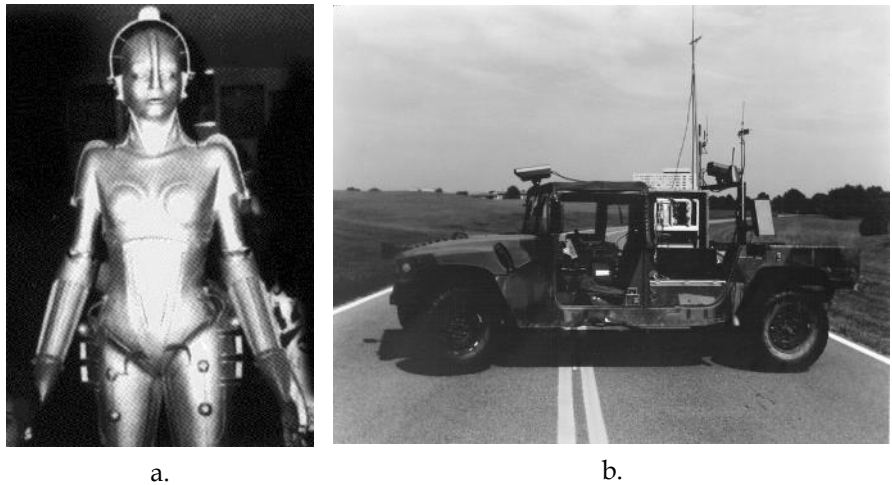


Figure I.1 Two views of robots: a) the humanoid robot from the 1926 movie *Metropolis* (image courtesy Fr. Doug Quinn and the *Metropolis* Home Page), and b) a HMMWV military vehicle capable of driving on roads and open terrains. (Photograph courtesy of the National Institute for Standards and Technology.)

it, etc. A computer doesn't move around under its own power. "Function autonomously" indicates that the robot can operate, self-contained, under all reasonable conditions without requiring recourse to a human operator. Autonomy means that a robot can adapt to changes in its environment (the lights get turned off) or itself (a part breaks) and continue to reach its goal.

Perhaps the best example of an intelligent mechanical creature which can function autonomously is the Terminator from the 1984 movie of the same name. Even after losing one camera (eye) and having all external cover- ings (skin, flesh) burned off, it continued to pursue its target (Sarah Connor). Extreme adaptability and autonomy in an extremely scary robot! A more practical (and real) example is *Marvin*, the mail cart robot, for the Baltimore FBI office, described in a Nov. 9, 1996, article in the *Denver Post*. Marvin is able to accomplish its goal of stopping and delivering mail while adapting to people getting in its way at

Part I
unpredictable times and locations.

What are Robotic Paradigms?

PARADIGM A *paradigm* is a philosophy or set of assumptions and/or techniques which characterize an approach to a class of problems. It is both a way of looking at the world and an implied set of tools for solving problems. No one paradigm is right; rather, some problems seem better suited for different approaches. For example, consider calculus problems. There are problems that could be solved by differentiating in cartesian ($X; Y; Z$) coordinates, but are much easier to solve if polar coordinates ($r; \theta$) are used. In the domain of calculus problems, Cartesian and polar coordinates represent two different paradigms for viewing and manipulating a problem. Both produce the correct answer, but one takes less work for certain problems.

ROBOTIC PARADIGMS Applying the right paradigm makes problem solving easier. Therefore, knowing the paradigms of AI robotics is one key to being able to successfully program a robot for a particular application. It is also interesting from a historical perspective to work through the different paradigms, and to examine the issues that spawned the shift from one paradigm to another.

ROBOT PARADIGM There are currently three paradigms for organizing intelligence in robots: hierarchical, reactive, and hybrid deliberative/reactive. The paradigms are described in two ways.

PRIMITIVES 1. **By the relationship between the three commonly accepted primitives of robotics: SENSE, PLAN, ACT.** The functions of a robot can be divided into three very general categories. If a function is taking in information from the robot's sensors and producing an output useful by other functions, then that function falls in the **SENSE** category. If the function is taking in information (either from sensors or its own knowledge about how the world works) and producing one or more tasks for the robot to perform (go down the hall, turn left, proceed 3 meters and stop), that function is in the **PLAN** category. Functions which produce output commands to motor actuators fall into **ACT** (turn 98°, clockwise, with a turning velocity of 0.2mps). Fig. 1.2 attempts to define these three primitives in terms of inputs and outputs; this figure will appear throughout the chapters in Part I.

2. **By the way sensory data is processed and distributed through the system.** How much a person or robot or animal is influenced by what it senses. So it is often difficult to adequately describe a paradigm with just

Part I a box labeled **SENSE**. In some paradigms, sensor information is restricted⁹
to being used in a specific, or dedicated, way for each function of a robot;

ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	Sensed information or directives	Actuator commands

Figure I.2 Robot primitives defined in terms of inputs and outputs.

SENSING
ORGANIZATION IN
ROBOT PARADIGMS

in that case processing is *local* to each function. Other paradigms expect all sensor information to be first processed into one *global* world model and then subsets of the model distributed to other functions as needed.

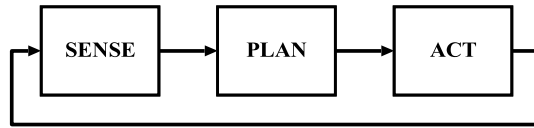
Overview of the Three Paradigms

HIERARCHICAL

PARADIGM

In order to set the stage for learning details, it may be helpful to begin with a general overview of the robot paradigms. Fig. I.3 shows the differences between the three paradigms in terms of the **SENSE**, **PLAN**, **ACT** primitives. The *Hierarchical Paradigm* is the oldest paradigm, and was prevalent from 1967–1990. Under it, the robot operates in a top-down fashion, heavy on planning (see Fig. I.3). This was based on an introspective view of how people think. “I see a door, I decide to head toward it, and I plot a course around the chairs.” (Unfortunately, as many cognitive psychologists now know, introspection is not always a good way of getting an accurate assessment of a thought process. We now suspect no one actually plans how they get out of a room; they have default schemas or behaviors.) Under the Hierarchical Paradigm, the robot senses the world, plans the next action, and then acts (**SENSE**, **PLAN**, **ACT**). Then it senses the world, plans, acts. At each step, the robot explicitly plans the next move. The other distinguishing feature of the Hierarchical paradigm is that all the sensing data tends to be gathered into one global world model, a

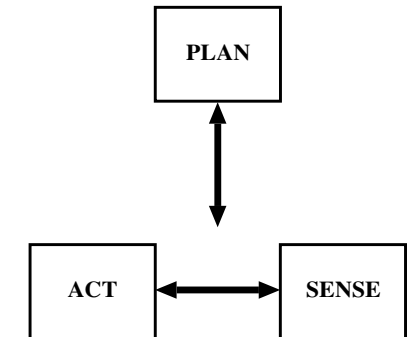
Partial single representation that the planner can use and can be routed to the actions. Constructing generic global world models¹¹



a.



b.



c.

Figure I.3 Three paradigms: a.) Hierarchical, b.) Reactive, and c.) Hybrid deliberative/reactive.

turns out to be very hard and brittle due to the *frame problem* and the need for a *closed world assumption*.

Fig. I.4 shows how the Hierarchical Paradigm can be thought of as a transitive, or Z-like, flow of events through the primitives given in Fig. I.4. Unfortunately, the flow of events ignored biological evidence that sensed information can be directly coupled to an action, which is why the sensed information input is blacked out.

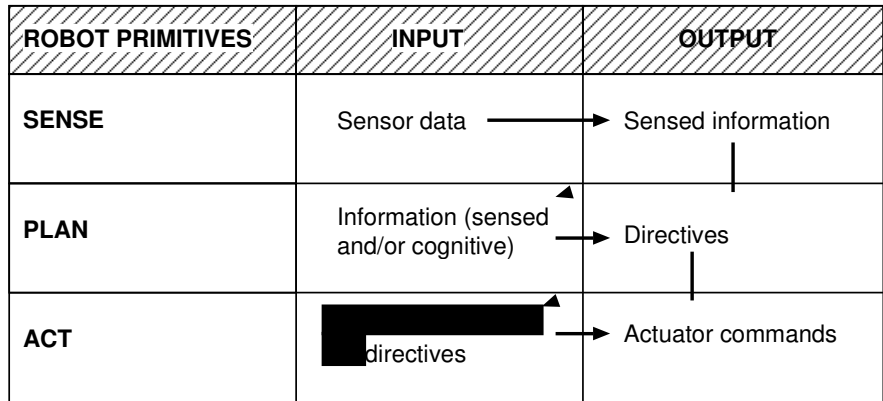


Figure I.4 Another view of the Hierarchical Paradigm.

REACTIVE PARADIGM

The *Reactive Paradigm* was a reaction to the Hierarchical Paradigm, and led to exciting advances in robotics. It was heavily used in robotics starting in 1988 and continuing through 1992. It is still used, but since 1992 there has been a tendency toward hybrid architectures. The Reactive Paradigm was made possible by two trends. One was a popular movement among AI researchers to investigate biology and cognitive psychology in order to examine living exemplars of intelligence. Another was the rapidly decreasing cost of computer hardware coupled with the increase in computing power. As a result, researchers could emulate frog and insect behavior with robots costing less than \$500 versus the \$100,000s Shakey, the first mobile robot, cost.

The Reactive Paradigm threw out planning all together (see Figs. I.3b and I.5). It is a **SENSE-ACT (S-A)** type of organization. Whereas the Hierarchical Paradigm assumes that the input to a **ACT** will always be the result of a **PLAN**, the Reactive Paradigm assumes that the input to an **ACT** will always be the direct output of a sensor, **SENSE**.

If the sensor is directly connected to the action, why isn't a robot running under the Reactive Paradigm limited to doing just one thing? The robot has multiple instances of **SENSE-ACT** couplings, discussed in Ch. 4. These couplings are concurrent processes, called behaviors, which take local sensing data and compute the best action to take independently of what the other processes are doing. One behavior can direct the robot to "move forward 5 meters" (**ACT** on drive motors) to reach a goal (**SENSE** the goal), while another behavior can say "turn 90

" (ACT on steer
motors) to avoid
a collision

Part I

14

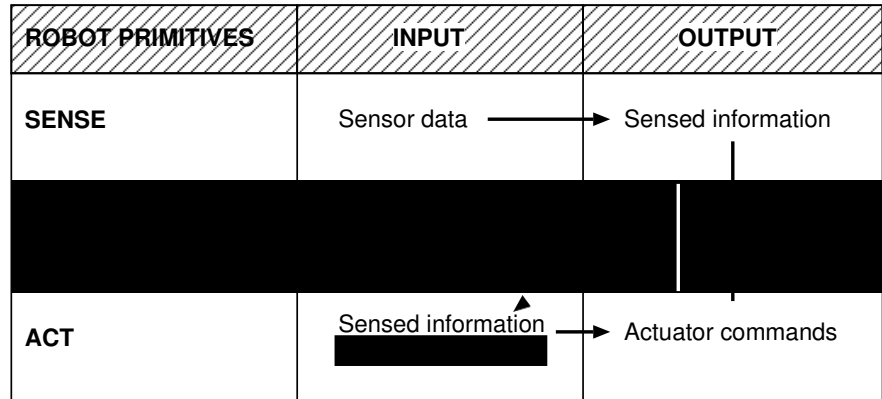


Figure I.5 The reactive paradigm.

with an object dead ahead (SENSE obstacles). The robot will do a combination of both behaviors, swerving off course temporarily at a 45° angle to avoid the collision. Note that neither behavior directed the robot to ACT with a 45° turn; the final ACT emerged from the combination of the two behaviors. While the Reactive Paradigm produced exciting results and clever robot insect demonstrations, it quickly became clear that throwing away planning was too extreme for general purpose robots. In some regards, the Reactive Paradigm reflected the work of Harvard psychologist B. F. Skinner in stimulus-response training with animals. It explained how some animals accomplished tasks, but was a dead end in explaining the entire range of human intelligence.

HYBRID DELIBERATIVE/REACTIVE PARADIGM

But the Reactive Paradigm has many desirable properties, especially the fast execution time that came from eliminating any planning. As a result, the Reactive Paradigm serves as the basis for the *Hybrid Deliberative/Reactive Paradigm*, shown in Fig.I.3c. The Hybrid Paradigm emerged in the 1990's and continues to be the current area of research. Under the Hybrid Paradigm, the robot first plans (deliberates) how to best decompose a task into subtasks (also called "mission planning") and then what are the suitable behaviors to accomplish each subtask, etc. Then the behaviors start executing as per the Reactive Paradigm. This type of organization is PLAN, SENSE-ACT (P, S-A), where the comma indicates that planning is done at one step, then sensing and acting are done together. Sensing organization in the Hybrid Paradigm is also a

mixture of
Hierarchical and
Reactive styles.
Sensor data gets
routed to each
behavior that
needs that sensor,
but is also
available to the
planner

Part I

16

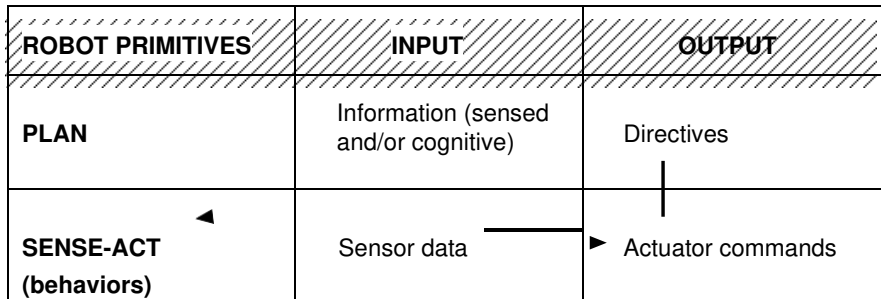


Figure I.6 The hybrid deliberative/reactive paradigm.

for construction of a task-oriented global world model. The planner may also “eavesdrop” on the sensing done by each behavior (i.e., the behavior identifies obstacles that could then be put into a map of the world by the planner). Each function performs computations at its own rate; deliberative planning, which is generally computationally expensive may update every 5 seconds, while the reactive behaviors often execute at 1/60 second. Many robots run at 80 centimeters per second.

Architectures

Determining that a particular paradigm is well suited for an application is certainly the first step in constructing the AI component of a robot. But that step is quickly followed with the need to use the tools associated with that paradigm. In order to visualize how to apply these paradigms to real-world applications, it is helpful to examine representative architectures. These architectures provide templates for an implementation, as well as examples of what each paradigm really means.

What is an architecture? Arkin offers several definitions in his book, *Behavior-Based Robots*.¹⁰ Two of the definitions he cites from other researchers capture how the term will be used in this book. Following Mataric,⁸⁹ an architecture provides a principled way of organizing a control system. However, in addition to providing structure, it imposes constraints on the way the control problem can be solved. Following Dean and Wellman,⁴³ an architecture describes a set of architectural components and how they interact. This book is interested in the components common in robot architectures; these are the basic building

blocks for programming a robot. It also is interested in the principles and rules of thumb for connecting these components together.

To see the importance of an architecture, consider building a house or a car. There is no “right” design for a house, although most houses share the same components (kitchens, bathrooms, walls, floors, doors, etc.). Likewise with designing robots, there can be multiple ways of organizing the components, even if all the designs follow the same paradigm. This is similar to cars designed by different manufacturers. All internal combustion engine types of cars have the same basic components, but the cars look different (BMW’s and Jaguars look quite different than Hondas and Fords). The internal combustion (IC) engine car is a paradigm (as contrasted to the paradigm of an electric car). Within the IC engine car community, the car manufacturers each have their own architecture. The car manufacturers may make slight modifications to the architecture for sedans, convertibles, sport-utility vehicles, etc., to throw out unnecessary options, but each style of car is a particular instance of the architecture. The point is: by studying representative robot architectures and the instances where they were used for a robot application, we can learn the different ways that the components and tools associated with a paradigm can be used to build an artificially intelligent robot.

Since a major objective in robotics is to learn how to build them, an important skill to develop is evaluating whether or not a previously developed architecture (or large chunks of it) will suit the current application. This skill will save both time spent on re-inventing the wheel and avoid subtle problems that other people have encountered and solved. Evaluation requires a set of criteria. The set that will be used in this book is adapted from *Behavior-Based Robotics*:¹⁰

MODULARITY

NICHE TARGETABILITY

PORTABILITY

ROBUSTNESS

1. **Support for modularity:** does it show good software engineering principles?
2. **Niche targetability:** how well does it work for the intended application?
3. **Ease of portability to other domains:** how well would it work for other applications or other robots?
4. **Robustness:** where is the system vulnerable, and how does it try to reduce that vulnerability?

Note that niche targetability and ease of portability are often at odds with each other. Most of the architectures described in this book were intended to be generic, therefore emphasizing portability. The generic structures, however, often introduce undesirable computational and

storage overhead,
so in practice the
designer must
make trade-offs.

Part I

20

Layout of the Section

This section is divided into eight chapters, one to define robotics and the other seven to intertwine both the theory and practice associated with each paradigm. Ch. 2 describes the Hierarchical Paradigm and two representative architectures. Ch. 3 sets the stage for understanding the Reactive Paradigm by reviewing the key concepts from biology and ethology that served to motivate the shift from Hierarchical to Reactive systems. Ch. 4 describes the Reactive Paradigm and the architectures that originally popularized this approach. It also offers definitions of primitive robot behaviors. Ch. 5 provides guidelines and case studies on designing robot behaviors. It also introduces issues in coordinating and controlling multiple behaviors and the common techniques for resolving these issues. At this point, the reader should be almost able to design and implement a reactive robot system, either in simulation or on a real robot. However, the success of a reactive system depends on the sensing. Ch. 6 discusses simple sonar and computer vision processing techniques that are commonly used in inexpensive robots. Ch. 7 describes the Hybrid Deliberative-Reactive Paradigm, concentrating on architectural trends. Up until this point, the emphasis is towards programming a single robot. Ch. 8 concludes the section by discussing how the principles of the three paradigms have been transferred to teams of robots.

End Note

Robot paradigm primitives.

While the SENSE, PLAN, ACT primitives are generally accepted, some researchers are suggesting that a fourth primitive be added, LEARN. There are no formal architectures at this time which include this, so a true paradigm shift has not yet occurred.

1

From Teleoperation To Autonomy

Chapter Objectives:

Define *intelligent robot*.

Be able to describe at least two differences between AI and Engineering approaches to robotics.

Be able to describe the difference between *telepresence* and *semi-autonomous control*.

Have some feel for the history and societal impact of robotics.

1.1 Overview

This book concentrates on the role of artificial intelligence for robots. At first, that may appear redundant; aren't robots intelligent? The short answer is "no," most robots currently in industry are not intelligent by any definition. This chapter attempts to distinguish an intelligent robot from a non-intelligent robot.

The chapter begins with an overview of artificial intelligence and the social implications of robotics. This is followed with a brief historical perspective on the evolution of robots towards intelligence, as shown in Fig. 1.1. One way of viewing robots is that early on in the 1960's there was a fork in the evolutionary path. Robots for manufacturing took a fork that has focused on engineering robot arms for manufacturing applications. The key to success in industry was precision and repeatability on the assembly line for mass production, in effect, industrial engineers wanted to automate the workplace. Once a robot arm was programmed, it should be able to operate for weeks and months with only minor maintenance. As a result, the emphasis was

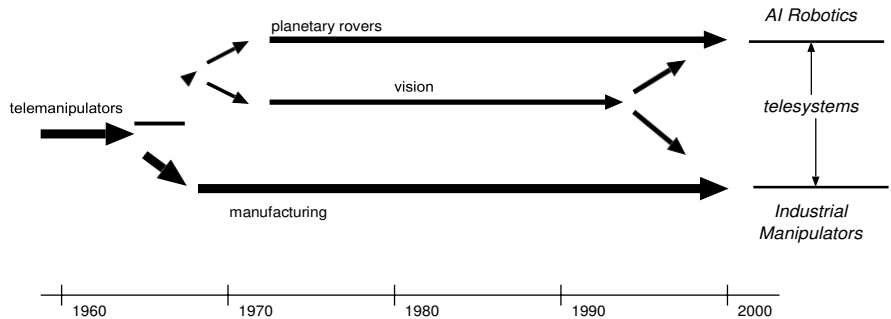


Figure 1.1 A timeline showing forks in development of robots.

placed on the mechanical aspects of the robot to ensure precision and repeatability and methods to make sure the robot could move precisely and repeatably, quickly enough to make a profit. Because assembly lines were engineered to mass produce a certain product, the robot didn't have to be able to notice any problems. The standards for mass production would make it more economical to devise mechanisms that would ensure parts would be in the correct place. A robot for automation could essentially be blind and senseless.

Robotics for the space program took a different fork, concentrating instead on highly specialized, one-of-a-kind planetary rovers. Unlike a highly automated manufacturing plant, a planetary rover operating on the dark side of the moon (no radio communication) might run into unexpected situations. Consider that on Apollo 17, astronaut and geologist Harrison Schmitt found an orange rock on the moon; an orange rock was totally unexpected. Ideally, a robot would be able to notice something unusual, stop what it was doing (as long as it didn't endanger itself) and investigate. Since it couldn't be pre-programmed to handle all possible contingencies, it had to be able to notice its environment and handle any problems that might occur. At a minimum, a planetary rover had to have some source of sensory inputs, some way of interpreting those inputs, and a way of modifying its actions to respond to a changing world. And the need to sense and adapt to a partially unknown environment is the need for intelligence.

The fork toward AI robots has not reached a termination point of truly autonomous, intelligent robots. In fact, as will be seen in Ch. 2 and 4, it wasn't until the late 1980's that any visible progress toward that end was

Part I made. So what happened when someone had an application for a robot¹⁵
which needed

real-time adaptability before 1990? In general, the lack of machine intelligence was compensated by the development of mechanisms which allow a human to control all, or parts, of the robot remotely. These mechanisms are generally referred to under the umbrella term: teleoperation. Teleoperation can be viewed as the “stuff” in the middle of the two forks. In practice, intelligent robots such as the Mars Sojourner are controlled with some form of teleoperation. This chapter will cover the flavors of teleoperation, given their importance as a stepping stone towards truly intelligent robots.

The chapter concludes by visiting the issues in AI, and argues that AI is imperative for many robotic applications. Teleoperation is simply not sufficient or desirable as a long term solution. However, it has served as a reasonable patch.

It is interesting to note that the two forks, manufacturing and AI, currently appear to be merging. Manufacturing is now shifting to a “mass customization” phase, where companies which can economically make short runs of special order goods are thriving. The pressure is on for industrial robots, more correctly referred to as industrial manipulators, to be rapidly reprogrammed and more forgiving if a part isn’t placed exactly as expected in its workspace. As a result, AI techniques are migrating to industrial manipulators.

1.2 How Can a Machine Be Intelligent?

ARTIFICIAL *The science of making machines act intelligently* is usually referred to as *artifi-*

INTELLIGENCE *cial intelligence*, or AI for short. Artificial Intelligence has no commonly accepted definitions. One of the first textbooks on AI defined it as “the study of ideas that enable computers to be intelligent,”¹⁴³ which seemed to beg the question. A later textbook was more specific, “AI is the attempt to get the computer to do things that, for the moment, people are better at.”¹²⁰ This definition is interesting because it implies that once a task is performed successfully by a computer, then the technique that made it possible is no longer AI, but something mundane. That definition is fairly important to a person researching AI methods for robots, because it explains why certain topics suddenly seem to disappear from the AI literature: it was perceived as being solved! Perhaps the most amusing of

~~Part~~ AI definitions was the slogan for the now defunct computer company,¹⁷
Thinking Machines, Inc., "... making machines that will be proud of us."

The term AI is controversial, and has sparked ongoing philosophical debates on whether a machine can ever be intelligent. As Roger Penrose notes in his book, *The Emperor's New Mind*: "Nevertheless, it would be fair to say that, although many clever things have indeed been done, the simulation of anything that could pass for genuine intelligence is yet a long way off."¹¹⁵ Engineers often dismiss AI as wild speculation. As a result of such vehement criticisms, many researchers often label their work as "intelligent systems" or "knowledge-based systems" in an attempt to avoid the controversy surrounding the term "AI."

A single, precise definition of AI is not necessary to study AI robotics. AI robotics is the application of AI techniques to robots. More specifically, AI robotics is the consideration of issues traditionally covered by AI for application to robotics: learning, planning, reasoning, problem solving, knowledge representation, and computer vision. An article in the May 5, 1997 issue of *Newsweek*, "Actually, Chess is Easy," discusses why robot applications are more demanding for AI than playing chess. Indeed, the concepts of the reactive paradigm, covered in Chapter 4, influenced major advances in traditional, non-robotic areas of AI, especially planning. So by studying AI robotics, a reader interested in AI is getting exposure to the general issues in AI.

1.3 What Can Robots Be Used For?

Now that a working definition of a robot and artificial intelligence has been established, an attempt can be made to answer the question: what can intelligent robots be used for? The short answer is that robots can be used for just about any application that can be thought of. The long answer is that robots are well suited for applications where 1) a human is at significant risk (nuclear, space, military), 2) the economics or menial nature of the application result in inefficient use of human workers (service industry, agriculture), and 3) for humanitarian uses where there is great risk (demining an area of land mines, urban search and rescue). Or as the well-worn joke among roboticists goes, robots are good for *the 3 D's*: jobs that are dirty, dull, or dangerous.

THE 3 D's

Historically, the military and industry invested in robotics in order to build nuclear weapons and power plants; now, the emphasis is on using robots for environmental remediation and restoration of irradiated and

polluted sites.
Many of the
same
technologies
developed for
the nuclear
industry for pro-
cessing
radioactive ore is
now being
adapted for the
pharmaceutical
indus-

try; processing immune suppressant drugs may expose workers to highly toxic chemicals.

Another example of a task that poses significant risk to a human is space exploration. People can be protected in space from the hard vacuum, solar radiation, etc., but only at great economic expense. Furthermore, space suits are so bulky that they severely limit an astronaut's ability to perform simple tasks, such as unscrewing and removing an electronics panel on a satellite. Worse yet, having people in space necessitates more people in space. Solar radiation embrittlement of metals suggests that astronauts building a large space station would have to spend as much time repairing previously built portions as adding new components. Even more people would have to be sent into space, requiring a larger structure. the problem escalates. A study by Dr. Jon Erickson's research group at NASA Johnson Space Center argued that a manned mission to Mars was not feasible without robot drones capable of constantly working outside of the vehicle to repair problems introduced by deadly solar radiation.⁵¹ (Interestingly enough, a team of three robots which did just this were featured in the 1971 film, *Silent Running*, as well as by a young R2D2 in *The Phantom Menace*.)

Nuclear physics and space exploration are activities which are often far removed from everyday life, and applications where robots figure more prominently in the future than in current times.

The most obvious use of robots is manufacturing, where repetitious activities in unpleasant surroundings make human workers inefficient or expensive to retain. For example, robot "arms" have been used for welding cars on assembly lines. One reason that welding is now largely robotic is that it is an unpleasant job for a human (hot, sweaty, tedious work) with a low tolerance for inaccuracy. Other applications for robots share similar motivation: to automate menial, unpleasant tasks – usually in the service industry. One such activity is janitorial work, especially maintaining public rest rooms, which has a high turnover in personnel regardless of payscale. The janitorial problem is so severe in some areas of the US, that the Postal Service offered contracts to companies to research and develop robots capable of autonomously cleaning a bathroom (the bathroom could be designed to accommodate a robot).

Agriculture is another area where robots have been explored as an economical alternative to hard to get menial labor. Utah State University has been working with automated harvesters, using GPS (global positioning satellite system) to traverse the field while adapting the speed of

harvesting to the rate of food being picked, much like a well-adapted insect. The De-

partment of Mechanical and Material Engineering at the University of Western Australia developed a robot called *Shear Majic* capable of shearing a live sheep. People available for sheep shearing has declined, along with profit margins, increasing the pressure on the sheep industry to develop economic alternatives. Possibly the most creative use of robots for agriculture is a mobile automatic milker developed in the Netherlands and in Italy.^{68;32} Rather than have a person attach the milker to a dairy cow, the roboticized milker arm identifies the teats as the cow walks into her stall, targets them, moves about to position itself, and finally reaches up and attaches itself.

Finally, one of the most compelling uses of robots is for humanitarian purposes. Recently, robots have been proposed to help with detecting unexploded ordinance (land mines) and with urban search and rescue (finding survivors after a terrorist bombing of a building or an earthquake). Humanitarian land demining is a challenging task. It is relatively easy to demine an area with bulldozer, but that destroys the fields and improvements made by the civilians and hurts the economy. Various types of robots are being tested in the field, including aerial and ground vehicles.⁷³

1.3.1 Social implications of robotics

While many applications for artificially intelligent robots will actively reduce risk to a human life, many applications appear to compete with a human's livelihood. Don't robots put people out of work? One of the pervasive themes in society has been the impact of science and technology on the dignity of people. Charlie Chaplin's silent movie, *Modern Times*, presented the world with visual images of how manufacturing-oriented styles of management reduces humans to machines, just "cogs in the wheel."

Robots appear to amplify the tension between productivity and the role of the individual. Indeed, the scientist in *Metropolis* points out to the corporate ruler of the city that now that they have robots, they don't need workers anymore. People who object to robots, or technology in general, are of-

LUDDITES

ten called *Luddites*, after Ned Ludd, who is often credited with leading a short-lived revolution of workers against mills in Britain. Prior to the industrial revolution in Britain, wool was woven by individuals in their homes or collectives as a cottage industry. Mechanization of the weaving

process changed the jobs associated with weaving, the status of being a weaver (it was a skill), and required people to work in a centralized location (like having your telecommuting job terminated). Weavers attempted to organize and destroyed looms and mill owners' properties in reaction. After escalating vi-

olence in 1812, legislation was passed to end worker violence and protect the mills. The rebelling workers were persecuted. While the Luddite movement may have been motivated by a quality-of-life debate, the term is often applied to anyone who objects to technology, or “progress,” for any reason. The connotation is that Luddites have an irrational fear of technological progress. The impact of robots is unclear, both what is the real story and how people interact with robots. The HelpMate Robotics, Inc. robots and janitorial robots appear to be competing with humans, but are filling a niche where it is hard to get human workers at any price. Cleaning office buildings is menial and boring, plus the hours are bad. One janitorial company has now invested in mobile robots through a Denver-based company, Continental Divide Robotics, citing a 90% yearly turnover in staff, even with profit sharing after two years. The Robotics Industries Association, a trade group, produces annual reports outlining the need for robotics, yet possibly the biggest robot money makers are in the entertainment and toy industries.

The cultural implications of robotics cannot be ignored. While the sheep shearing robots in Australia were successful and were ready to be commercialized for significant economic gains, the sheep industry reportedly rejected the robots. One story goes that the sheep ranchers would not accept a robot shearer unless it had a 0% fatality rate (it’s apparently fairly easy to nick an artery on a squirming sheep). But human shearers accidentally kill several sheep, while the robots had a demonstrably better rate. The use of machines raises an ethical question: is it acceptable for an animal to die at the hands of a machine rather than a person? What if a robot was performing a piece of intricate surgery on a human?

1.4 A Brief History of Robotics

Robotics has its roots in a variety of sources, including the way machines are controlled and the need to perform tasks that put human workers at risk.

In 1942, the United States embarked on a top secret project, called the Manhattan Project, to build a nuclear bomb. The theory for the nuclear bomb had existed for a number of years in academic circles. Many military leaders of both sides of World War II believed the winner would

be the side who could build the first nuclear device: the Allied Powers led by USA or the Axis, led by Nazi Germany.

One of the first problems that the scientists and engineers encountered was handling and processing radioactive materials, including uranium and



Figure 1.2 A Model 8 Telem manipulator. The upper portion of the device is placed in the ceiling, and the portion on the right extends into the hot cell. (Photograph courtesy Central Research Laboratories.)

plutonium, in large quantities. Although the immensity of the dangers of working with nuclear materials was not well understood at the time, all the personnel involved knew there were health risks. One of the first solutions was the *glove box*. Nuclear material was placed in a glass box. A person stood (or sat) behind a leaded glass shield and stuck their hands into thick rubberized gloves. This allowed the worker to see what they were doing and to perform almost any task that they could do without gloves.

But this was not an acceptable solution for highly radioactive materials, and mechanisms to physically remove and completely isolate the nuclear materials from humans had to be developed. One such mechanism was a force reflecting *telem manipulator*, a sophisticated mechanical linkage which translated motions on one end of the mechanism to motions at the other end. A popular telem manipulator is shown in Fig. 1.2.

A nuclear worker would insert their hands into (or around) the telem manipulator, and move it around while watching a display of what the other end of the arm was doing in a containment cell. Telem manipulators are similar in principle to the power gloves now used in computer games, but much harder to use. The mechanical technology of the time did not

allow a perfect mapping of hand and arm movements to the robot arm. Often the opera-

tor had to make non-intuitive and awkward motions with their arms to get the robot arm to perform a critical manipulation—very much like working in front of a mirror. Likewise, the telemanipulators had challenges in providing force feedback so the operator could feel how hard the gripper was holding an object. The lack of naturalness in controlling the arm (now referred to as a poor Human-Machine Interface) meant that even simple tasks for an un-encumbered human could take much longer. Operators might take years of practice to reach the point where they could do a task with a telemanipulator as quickly as they could do it directly.

After World War II, many other countries became interested in producing a nuclear weapon and in exploiting nuclear energy as a replacement for fossil fuels in power plants. The USA and Soviet Union also entered into a nuclear arms race. The need to mass-produce nuclear weapons and to support peaceful uses of nuclear energy kept pressure on engineers to design robot arms which would be easier to control than telemanipulators. Machines that looked more like and acted like robots began to emerge, largely due to advances in control theory. After WWII, pioneering work by Norbert Wiener allowed engineers to accurately control mechanical and electrical devices using cybernetics.

1.4.1 Industrial manipulators

Successes with at least partially automating the nuclear industry also meant the technology was available for other applications, especially general manufacturing. Robot arms began being introduced to industries in 1956 by Unimation (although it wouldn't be until 1972 before the company made a profit).³⁷ The two most common types of robot technology that have evolved for industrial use are robot arms, called industrial manipulators, and mobile carts, called automated guided vehicles (AGVs).

INDUSTRIAL

An *industrial manipulator*, to paraphrase the Robot Institute of America's

MANIPULATOR

definition, is a reprogrammable and multi-functional mechanism that is designed to move materials, parts, tools, or specialized devices. The emphasis in industrial manipulator design is being able to program them to be able to perform a task repeatedly with a high degree of accuracy and speed. In order to be multi-functional, many manipulators have

1.4 A Brief History of Robotics 25
multiple degrees of freedom, as shown in Fig. 1.4. The MOVEMASTER arm has five degrees of freedom, because it has five joints, each of which is capable of a single rotational degree of freedom. A human arm has three joints (shoulder, el-



Figure 1.3 An RT3300 industrial manipulator. (Photograph courtesy of Seiko Instruments.)

bow, and wrist), two of which are complex (shoulder and wrist), yielding six degrees of freedom.

Control theory is extremely important in industrial manipulators. Rapidly moving around a large tool like a welding gun introduces interesting problems, like when to start decelerating so the gun will stop in the correct location without overshooting and colliding with the part to be welded. Also, oscillatory motion, in general, is undesirable. Another interesting problem is the joint configuration. If a robot arm has a wrist, elbow and shoulder joints like a human, there are redundant degrees of freedom. Redundant degrees of freedom means there are multiple ways of moving the joints that will accomplish the same motion. Which one is better, more efficient, less stressful on the mechanisms?

It is interesting to note that most manipulator control was assumed to be *ballistic control*, or *open loop control*. In ballistic control, the position trajectory

and velocity profile is computed once, then the arm carries it out. There are no “in-flight” corrections, just like a ballistic missile doesn’t make any course corrections. In order to accomplish a precise task with ballistic control, everything about the device and how it works has to be modeled and figured

into the computation. The opposite of ballistic control is *closed-loop control*,

BALLISTIC CONTROL

OPEN LOOP CONTROL

CLOSED-LOOP

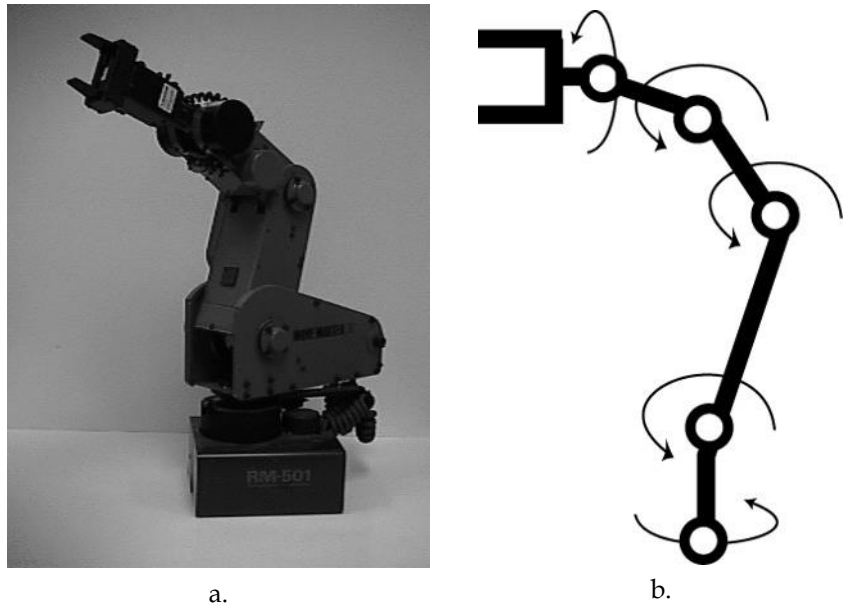


Figure 1.4 A MOVEMASTER robot: a.) the robot arm and b.) the associated joints.

FEEDBACK

and a new trajectory and profile is computed and executed, then modified on the next update, and so on. Closed-loop control requires external sensors to provide the error signal, or *feedback*.

In general, if the structural properties of the robot and its cargo are known, these questions can be answered and a program can be developed. In practice, the control theory is complex. The dynamics (how the mechanism moves and deforms) and kinematics (how the components of the mechanism are connected) of the system have to be computed for each joint of the robot, then those motions can be propagated to the next joint iteratively. This requires a computationally consuming change of coordinate systems from one joint to the next. To move the gripper in Fig 1.4 requires four changes of coordinates to go from the base of the arm to the gripper. The coordinate transformations often have singularities, causing the equations to perform divide by zeros. It can take a programmer weeks to reprogram a manipulator.

One simplifying solution is to make the robot rigid at the desired velocities, reducing the dynamics. This eliminates having to compute the terms for overshooting and oscillating. However, a robot is made rigid

by making it

TEACH PENDANT

heavier. The end result is that it is not uncommon for a 2 ton robot to be able to handle only a 200 pound payload. Another simplifying solution is to avoid the computations in the dynamics and kinematics and instead have the programmer use a teach pendant. Using a *teach pendant* (which often looks like a joystick or computer game console), the programmer guides the robot through the desired set of motions. The robot remembers these motions and creates a program from them. Teach pendants do not mitigate the danger of working around a 2 ton piece of equipment. Many programmers have to direct the robot to perform delicate tasks, and have to get physically close to the robot in order to see what the robot should do next. This puts the programmer at risk of being hit by the robot should it hit a singularity point in its joint configuration or if the programmer makes a mistake in directing a motion. You don't want to have your head next to a 2 ton robot arm if it suddenly spins around!

AUTOMATIC GUIDED

Automatic guided vehicles, or AGVs, are intended to be the most flexible con-

VEHICLES

veyor system possible: a conveyor which doesn't need a continuous belt or roller table. Ideally an AGV would be able to pick up a bin of parts or manufactured items and deliver them as needed. For example, an AGV might receive a bin containing an assembled engine. It could then deliver it automatically across the shop floor to the car assembly area which needed an engine. As it returned, it might be diverted by the central computer and instructed to pick up a defective part and take it to another area of the shop for reworking.

However, navigation (as will be seen in Part II) is complex. The AGV has to know where it is, plan a path from its current location to its goal destination, and to avoid colliding with people, other AGVs, and maintenance workers and tools cluttering the factory floor. This proved too difficult to do, especially for factories with uneven lighting (which interferes with vision) and lots of metal (which interferes with radio controllers and on-board radar and sonar). Various solutions converged on creating a trail for the AGV to follow. One method is to bury a magnetic wire in the floor for the AGV to sense. Unfortunately, changing the path of an AGV required ripping up the concrete floor. This didn't help with the flexibility needs of modern manufacturing. Another method is to put down a strip of photochemical tape for the vehicle to follow. The strip is unfortunately vulnerable, both to wear and to vandalism by unhappy workers. Regardless of the guidance method, in the end the simplest way to thwart an AGV was to something on its path.

1.4 A Brief History of Robotics 31
If the AGV did not have range sensors, then it would be unable to detect an expensive piece of equipment or a person put deliberately in its path.
A

few costly collisions would usually lead to the AGV's removal. If the AGV did have range sensors, it would stop for anything. A well placed lunch box could hold the AGV for hours until a manager happened to notice what was going on. Even better from a disgruntled worker's perspective, many AGVs would make a loud noise to indicate the path was blocked. Imagine having to constantly remove lunch boxes from the path of a dumb machine making unpleasant siren noises.

From the first, robots in the workplace triggered a backlash. Many of the human workers felt threatened by a potential loss of jobs, even though the jobs being mechanized were often menial or dangerous. This was particularly true of manufacturing facilities which were unionized. One engineer reported that on the first day it was used in a hospital, a HelpMate Robotics cart was discovered pushed down the stairs. Future models were modified to have some mechanisms to prevent malicious acts.

BLACK FACTORY

Despite the emerging Luddite effect, industrial engineers in each of the economic powers began working for a *black factory* in the 1980's. A black factory is a factory that has no lights turned on because there are no workers. Computers and robots were expected to allow complete automation of manufacturing processes, and courses in "Computer-Integrated Manufacturing Systems" became popular in engineering schools.

But two unanticipated trends undermined industrial robots in a way that the Luddite movement could not. First, industrial engineers did not have experience designing manufacturing plants with robots. Often industrial manipulators were applied to the wrong application. One of the most embarrassing examples was the IBM Lexington printer plant. The plant was built with a high degree of automation, and the designers wrote numerous articles on the exotic robot technology they had cleverly designed. Unfortunately, IBM had grossly over-estimated the market for printers and the plant sat mostly idle at a loss. While the plant's failure wasn't the fault of robotics, per se, it did cause many manufacturers to have a negative view of automation in general. The second trend was the changing world economy. Customers were demanding "mass customization." Manufacturers who could make short runs of a product tailored to each customer on a large scale were the ones making the money. (Mass customization is also referred to as "agile manufacturing.") However, the lack of adaptability and difficulties in programming industrial robot arms and changing the paths of AGVs interfered with

rapid retooling. The lack of adaptability, combined with concerns over worker safety and the Luddite effect, served to discourage companies from investing in robots through most of the 1990's.

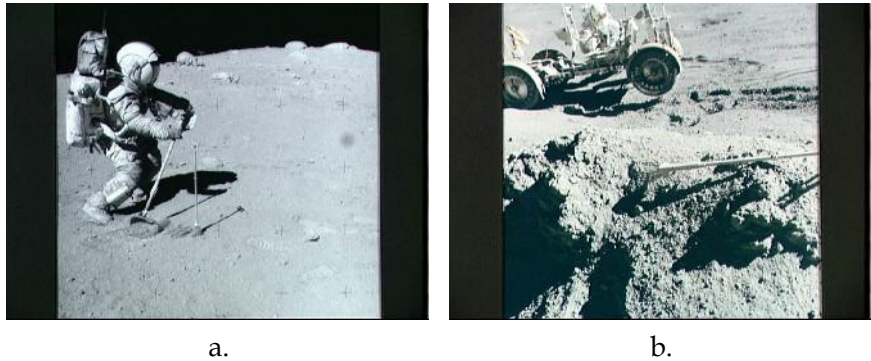


Figure 1.5 Motivation for intelligent planetary rovers: a.) Astronaut John Young awkwardly collecting lunar samples on Apollo 16, and b.) Astronaut Jim Irwin stopping the lunar rover as it slides down a hill on Apollo 15. (Photographs courtesy of the National Aeronautics and Space Administration.)

1.4.2 Space robotics and the AI approach

While the rise of industrial manipulators and the engineering approach to robotics can in some measure be traced to the nuclear arms race, the rise of the AI approach can be said to start with the space race. On May 25, 1961, spurred by the success of the Soviet Union's Sputnik space programs, President John F. Kennedy announced that United States would put a man on the moon by 1970. Walking on the moon was just one aspect of space exploration. There were concerns about the Soviets setting up military bases on the Moon and Mars and economic exploitation of planetary resources.

Clearly there was going to be a time lag of almost a decade before humans from the USA would go to the Moon. And even then, it would most likely be with experimental spacecraft, posing a risk to the human astronauts. Even without the risk to humans, the bulk of spacesuits would make even trivial tasks difficult for astronauts to perform. Fig. 1.5a shows astronaut John Young on Apollo 16 collecting samples with a lunar rake. The photo shows the awkward way the astronaut had to bend his body and arms to complete the task.

Planetary rovers were a possible solution, either to replace an astronaut or assist him or her. Unfortunately, rover technology in the 1960's was limited. Because of the time delays, a human would be unable to safely control a rover over the notoriously poor radio links of the time, even if

1.4 *A Brief History of Robotics*
the rover went very

35

slow. Therefore, it would be desirable to have a robot that was autonomous. One option would be to have mobile robots land on a planetary conduct preliminary explorations, conduct tests, etc., and radio back the results. These automated planetary rovers would ideally have a high degree of autonomy, much like a trained dog. The robot would receive commands from Earth to explore a particular region. It would navigate around boulders and not fall into canyons, and traverse steep slopes without rolling over. The robot might even be smart enough to regulate its own energy supply, for example, by making sure it was sheltered during the planetary nights and to stop what it was doing and position itself for recharging its solar batteries. A human might even be able to speak to it in a normal way to give it commands.

Getting a mobile robot to the level of a trained dog immediately presented new issues. Just by moving around, a mobile robot could change the world- for instance, by causing a rock slide. Fig. 1.5b shows astronaut Jim Irwin rescuing the lunar rover during an extra-vehicular activity (EVA) on Apollo 15 as it begins to slide downhill. Consider that if an astronaut has difficulty finding a safe parking spot on the moon, how much more challenging it would be for an autonomous rover. Furthermore, an autonomous rover would have no one to rescue it, should it make a mistake.

Consider the impact of uncertain or incomplete information on a rover that didn't have intelligence. If the robot was moving based on a map taken from a telescope or an overhead command module, the map could still contain errors or at the wrong resolution to see certain dangers. In order to navigate successfully, the robot has to compute its path with the new data or risk colliding with a rock or falling into a hole. What if the robot did something broke totally unexpected or all the assumptions about the planet were wrong? In theory, the robot should be able to diagnose the problem and attempt to continue to make progress on its task. What seemed at first like an interim solution to putting humans in space quickly became more complicated.

Clearly, developing a planetary rover and other robots for space was going to require a concentrated, long-term effort. Agencies in the USA such as NASA *Jet Propulsion Laboratory (JPL)* in Pasadena, California, were given the task of developing the robotic technology that would be needed to prepare the way for astronauts in space. They were in a position to take advantage of the outcome of the *Dartmouth Conference*. The Dartmouth Conference was a gathering hosted by the Defense

Advanced Research Projects Agency (DARPA) in 1955 of prominent scientists working with computers or on the theory for computers. DARPA was interested in hearing what the potential

uses for computers were. One outcome of the conference was the term “artificial intelligence”; the attending scientists believed that computers might become powerful enough to understand human speech and duplicate human reasoning. This in turn suggested that computers might mimic the capabilities of animals and humans sufficiently for a planetary rover to survive for long periods with only simple instructions from Earth.

As an indirect result of the need for robotics converging with the possibility of artificial intelligence, the space program became one of the earliest proponents of developing AI for robotics. NASA also introduced the notion that AI robots would of course be mobile, rather than strapped to a factory floor, and would have to integrate all forms of AI (understanding speech, planning, reasoning, representing the world, learning) into one program—a daunting task which has not yet been reached.

1.5 Teleoperation

TELEOPERATION

Teleoperation is when a human operator controls a robot from a distance (*tele* means “remote”). The connotation of teleoperation is that the distance is too great for the operator to see what the robot is doing, so radio controlled toy cars are not considered teleoperation systems. The operator and robot have some type of master-slave relationship. In most cases, the human operator sits at a workstation and directs a robot through some sort of interface, as seen in Fig. 1.6.

LOCAL REMOTE

The control interface could be a joystick, virtual reality gear, or any number of innovative interfaces. The human operator, or teleoperator, is often referred to as the *local* (due to being at the local workstation) and the robot as the *remote* (since it is operating at a remote location from the teleoperator). The local must have some type of display and control mechanisms, while the remote must have sensors, effectors, power, and in the case of mobile robots, mobility.¹⁴¹ The teleoperator cannot look at what the remote is doing directly, either because the robot is physically remote (e.g., on Mars) or the local has to be shielded (e.g., in a nuclear or pharmaceutical processing plant hot cell). Therefore, the *sensors* which acquire information about the remote location, the *display* technology for allowing the operator to see the sensor data, and the *communication link* between the local and remote are critical components of a telesystem.¹⁴¹

SENSORS

DISPLAY

COMMUNICATION LINK

Teleoperation is a popular solution for controlling remotes because AI technology is nowhere near human levels of competence, especially in terms of

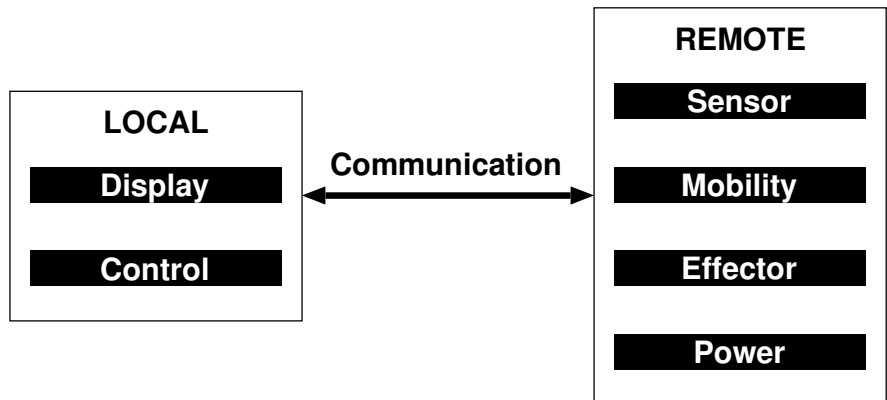
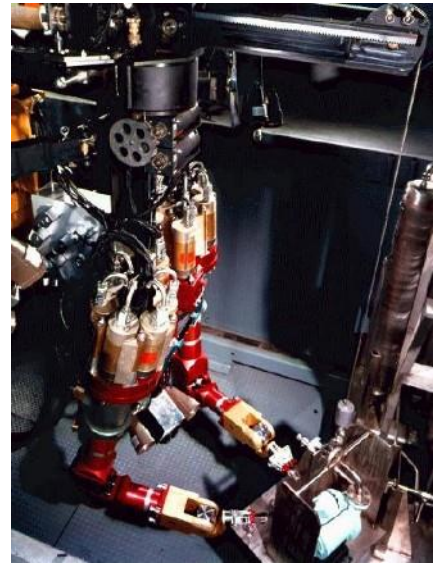


Figure 1.6 Organization of a telesystem. (Photographs courtesy of Oak Ridge National Laboratory.)

perception and decision making. One example of teleoperation is the exploration of underwater sites such as the Titanic. Having a human control a robot is advantageous because a human can isolate an object of interest, even partially obscured by mud in murky water as described by W. R. Uttal.¹⁴¹ Humans can also perform dextrous manipulation (e.g., screwing a nut on a bolt), which is very difficult to program a manipulator to do.



Figure 1.7 Sojourner Mars rover. (Photograph courtesy of the National Aeronautics and Space Administration.)

Another example is the Sojourner robot (shown in Fig. 1.7) which explored Mars from July 5 to September 27, 1997, until it ceased to reply to radio commands. Since there was little data before Sojourner on what Mars is like, it is hard to develop sensors and algorithms which can detect important attributes or even control algorithms to move the robot. It is important that any unusual rocks or rock formations (like the orange rock Dr. Schmitt found on the Moon during Apollo 17) be detected. Humans are particularly adept at perception, especially seeing patterns and anomalies in pictures. Current AI perceptual abilities fall far short of human abilities. Humans are also adept at problem solving. When the Mars Pathfinder craft landed on Mars, the air bags that had cushioned the landing did not deflate properly. When the petals of the lander opened, an airbag was in the way of Sojourner. The solution? The ground controllers sent commands to retract the petals and open them again. That type of problem solving is extremely difficult for the current capabilities of AI.

But teleoperation is not an ideal solution for all situations. Many tasks are repetitive and boring. For example, consider using a joystick to drive a radio-controlled car; after a few hours, it tends to get harder and harder to pay attention. Now imagine trying to control the car while only looking through a small camera mounted in front. The task becomes

1.5 Teleoperation
much harder

31

because of the limited field of view; essentially there is no peripheral vision. Also, the camera may not be transmitting new images very fast because the communication link has a limited bandwidth, so the view is jerky. Most people quickly experience *cognitive fatigue*; their attention wanders and they may even experience headaches and other physical symptoms of stress. Even if the visual display is excellent, the teleoperator may get *simulator sickness* due to the discordance between the visual system saying the operator is moving and the inner ear saying the operator is stationary.¹⁴¹

COGNITIVE FATIGUE
SIMULATOR SICKNESS

Another disadvantage of teleoperation is that it can be inefficient to use for applications that have a *large time delay*.¹²⁸ A large time delay can result in the teleoperator giving a remote a command, unaware that it will place the remote in jeopardy. Or, an unanticipated event such as a rock fall might occur and destroy the robot before the teleoperator can see the event and command the robot to flee. A rule of thumb, or *heuristic*, is that the time it takes to do

TELEOPERATION

a task with traditional teleoperation grows linearly with the transmission delay. A teleoperation task which took 1 minute for a teleoperator to guide a remote to do on the Earth might take 2.5 minutes to do on the Moon, and 140 minutes on Mars.¹⁴² Fortunately, researchers have made some progress with

HEURISTIC

predictive displays, which immediately display what the simulation result of the command would be.

PREDICTIVE DISPLAYS

The impact of time delays is not limited to planetary rovers. A recent example of an application of teleoperation are unmanned aerial vehicles (UAV) used by the United States to verify treaties by flying overhead and taking videos of the ground below. Advanced prototypes of these vehicles can fly autonomously, but take-offs and landings are difficult for on-board computer control. In this case of the Darkstar UAV (shown in Fig. 1.8), human operators were available to assume teleoperation control of the vehicle should it encounter problems during take-off. Unfortunately, the contingency plan did not factor in the 7 second delay introduced by using a satellite as the communications link. Darkstar no. 1 did indeed experience problems on take-off, but the teleoperator could not get commands to it fast enough before it crashed. As a result, it earned the unofficial nickname "Darkspot."

Another practical drawback to teleoperation is that there is at least one person per robot, possibly more. The Predator unmanned aerial vehicle has been used by the United States for verification of the Dayton Accords

1.5 Teleoperation

in Bosnia. One Predator requires at least one teleoperator to fly the vehicle and another teleoperator to command the sensor payload to look at particular areas. Other UAVs have teams composed of up to four teleoperators plus a fifth team member who specializes in takeoffs and landings. These teleop-

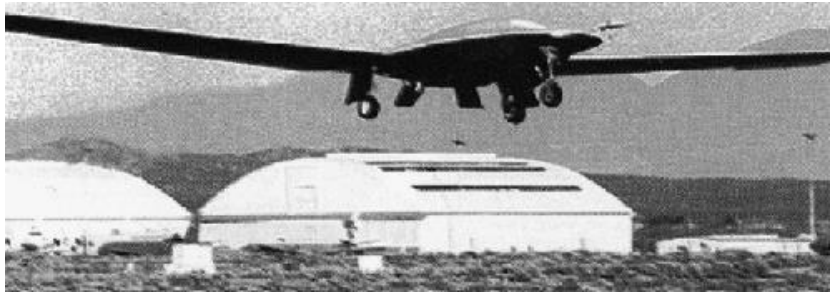


Figure 1.8 Dark Star unmanned aerial vehicle. (Photograph courtesy of DefenseLink, Office of the Assistant Secretary of Defense-Public Affairs.)

erators may have over a year of training before they can fly the vehicle. In the case of UAVs, teleoperation permits a dangerous, important task to be completed, but with a high cost in manpower.

TASK According to Wampler,¹⁴² teleoperation is best suited for applications

CHARACTERISTICS where:

1. The tasks are unstructured and not repetitive.
2. The task workspace cannot be engineered to permit the use of industrial manipulators.
3. Key portions of the task intermittently require dextrous manipulation, especially hand-eye coordination.
4. Key portions of the task require object recognition, situational awareness, or other advanced perception.
5. The needs of the display technology do not exceed the limitations of the communication link (bandwidth, time delays).
6. The availability of trained personnel is not an issue.

1.5.1 Telepresence

An early attempt at reducing cognitive fatigue was to add more cameras with faster update rates to widen the field of view and make it more consistent with how a human prefers to look at the world. This may not

for many applications because of limited bandwidth. Video telephones, picture phones, or video-conferencing over the Internet with their jerky, asynchronous updates are usually examples of annoying limited bandwidth. In these instances, the physical restrictions on how much and how fast information can be transmitted result in image updates much slower than the rates human brains expect. The result of limited bandwidth is jerky motion and increased cognitive fatigue. So adding more cameras only exacerbates the problem by adding more information that must be transmitted over limited bandwidth.

TELEPRESENCE

VIRTUAL REALITY

One area of current research in teleoperation is the use of *telepresence* to reduce cognitive fatigue and simulator sickness by making the human-robot interface more natural. Telepresence aims for what is popularly called *virtual reality*, where the operator has complete sensor feedback and feels as if she were the robot. If the operator turns to look in a certain direction, the view from the robot is there. If the operator pushes on a joystick for the robot to move forward and the wheels are slipping, the operator would hear and feel the motors straining while seeing that there was no visual change. This provides a more natural interface to the human, but it is very expensive in terms of equipment and requires very high bandwidth rates. It also still requires one person per robot. This is better than traditional teleoperation, but a long way from having one teleoperator control multiple robots.

1.5.2 Semi-autonomous control

SEMI-AUTONOMOUS

Another line of research in teleoperation is *semi-autonomous control*, often

CONTROL
SUPERVISORY CONTROL

called *supervisory control*, where the remote is given an instruction or portion of a task that it can safely do on its own. There are two flavors of semi-autonomous control: continuous assistance, or *shared control*, and *control trading*.

SHARED CONTROL

In continuous assistance systems, the teleoperator and remote share control. The teleoperator can either delegate a task for the robot to do or can do it via direct control. If the teleoperator delegates the task to the robot, the human must still monitor to make sure that nothing goes wrong. This is particularly useful for teleoperating robot arms in space. The operator can relax (relatively) while the robot arm moves into the specified position near a panel, staying on alert in case something goes wrong. Then the operator can take over and perform the actions which

require hand-eye coordination.

Shared control helps the operator avoid cognitive fatigue by delegating boring, repetitive control actions to the robot. It also exploits the ability of a

1.5 Teleoperation

human to perform delicate operations. However, it still requires a high communication bandwidth.

CONTROL TRADING

An alternative approach is *control trading*, where the human initiates an action for the robot to complete autonomously. The human only interacts with the robot to give it a new command or to interrupt it and change its orders. The overall scheme is very much like a parent giving a 10-year old child a task to do. The parent knows what the child is able to do autonomously (e.g., clean their room). They have a common definition (clean room means go to the bedroom, make the bed, and empty the wastebaskets). The parent doesn't care about the details of how the child cleans the room (e.g., whether the wastebasket is emptied before the bed is made or vice versa). Control trading assumes that the robot is capable of autonomously accomplishing certain tasks without sharing control. The advantage is that, in theory, the local operator can give a robot a task to do, then turn attention to another robot and delegate a task to it, etc. A single operator could control multiple robots because they would not require even casual monitoring while they were performing a task. Supervisory control also reduces the demand on bandwidth and problems with communication delays. Data such as video images need to be transferred only when the local is configuring the remote for a new task, not all the time. Likewise, since the operator is not involved in directly controlling the robot, a 2.5 minute delay in communication is irrelevant; the robot either wrecked itself or it didn't. Unfortunately, control trading assumes that robots have actions that they can perform robustly even in unexpected situations; this may or may not be true. Which brings us back to the need for artificial intelligence.

Sojourner exhibited both flavors of supervisory control. It was primarily programmed for traded control, where the geologists could click on a rock and Sojourner would autonomously navigate close to it, avoiding rocks, etc. However, some JPL employees noted that the geologists tended to prefer to use shared control, watching every movement. A difficulty with most forms of shared control is that it is assumed that the human is smarter than the robot. This may be true, but the remote may have better sensor viewpoints and reaction times.

1.6 The Seven Areas of AI

Now that some possible uses and shortcomings of robots have been

^{1.5 Teleoperation}
covered, it is motivating to consider what are the areas of artificial intelligence and

39

how they could be used to overcome these problems. The Handbook of Artificial Intelligence ⁶⁴ divides up the field into seven main areas: *knowledge representation, understanding natural language, learning, planning and problem solving, inference, search, and vision*.

KNOWLEDGE

1. **Knowledge representation.** An important, but often overlooked, issue is

REPRESENTATION

how does the robot represent its world, its task, and itself. Suppose a robot is scanning a pile of rubble for a human. What kind of data structure and algorithms would it take to represent what a human looks like? One way is to construct a structural model: a person is composed of an oval head, a cylindrical torso, smaller cylindrical arms with bilateral symmetry, etc. Of course, what happens if only a portion of the human is visible?

UNDERSTANDING

2. **Understanding natural language.** Natural language is deceptively chal-

NATURAL LANGUAGE

lenging, apart from the issue of recognizing words which is now being done by commercial products such as Via Voice and Naturally Speaking. It is not just a matter of looking up words, which is the subject of the following apocryphal story about AI. The story goes that after Sputnik went up, the US government needed to catch up with the Soviet scientists. However, translating Russian scientific articles was time consuming and not many US citizens could read technical reports in Russian. Therefore, the US decided to use these newfangled computers to create translation programs. The day came when the new program was ready for its first test. It was given the proverb: the spirit is willing, but the flesh is weak. The reported output: the vodka is strong, but the meat is rotten.

LEARNING

3. **Learning.** Imagine a robot that could be programmed by just watching a human, or by just trying the task repeatedly itself.

PLANNING, PROBLEM

4. **Planning and problem solving.** Intelligence is associated with the ability

SOLVING

to plan actions needed to accomplish a goal and solve problems with those plans or when they don't work. One of the earliest childhood fables, the Three Pigs and the Big, Bad Wolf, involves two unintelligent pigs who don't plan ahead and an intelligent pig who is able to solve the problem of why his brothers' houses have failed, as well as plan an unpleasant demise for the wolf.

1.5 *Teleoperation*
5. **Inference.** Inference is generating an answer when there isn't complete information. Consider a planetary rover looking at a dark region on the ground. Its range finder is broken and all it has left is its camera and a fine AI system. Assume that depth information can't be extracted from

the camera. Is the dark region a canyon? Is it a shadow? The rover will need to use inference to either actively or passively disambiguate what the dark region is (e.g., kick a rock at the dark area versus reason that there is nothing nearby that could create that shadow).

SEARCH 6. **Search.** Search doesn't necessarily mean searching a large physical space for an object. In AI terms, search means efficiently examining a knowledge representation of a problem (called a "search space") to find the answer. Deep Blue, the computer that beat the World Chess master Gary Kasparov, won by searching through almost all possible combinations of moves to find the best move to make. The legal moves in chess given the current state of the board formed the search space.

VISION 7. **Vision.** Vision is possibly the most valuable sense humans have. Studies by Harvard psychologist Steven Kosslyn suggest that much of problem solving abilities stem from the ability to visually simulate the effects of actions in our head. As such, AI researchers have pursued creating vision systems both to improve robotic actions and to supplement other work in general machine intelligence.

Finally, there is a temptation to assume that the history of AI Robotics is the story of how advances in AI have improved robotics. But that is not the case. In many regards, robotics has played a pivotal role in advancing AI. Breakthroughs in methods for planning (operations research types of problems) came after the paradigm shift to reactivity in robotics in the late 1980's showed how unpredictable changes in the environment could actually be exploited to simplify programming. Many of the search engines on the world wide web use techniques developed for robotics. These programs are called *software agents*: autonomous programs which can interact with and adapt to their world just like an animal or a smart robot. The term *web-bot* directly reflects on the robotic heritage of these AI systems. Even animation is being changed by advances in AI robotics. According to a keynote address given by Danny Hillis at the 1997 Autonomous Agents conference, animators for Disney's *Hunchback of Notre Dame* programmed each cartoon character in the crowd scenes as if it were a simulation of a robot, and used methods that will be discussed in Ch. 4.

SOFTWARE AGENTS

WEB-BOT

1.7 Summary

AI robotics is a distinct field, both historically and in scope, from industrial robotics. Industrial robots have concentrated on control theory issues, particularly solving the dynamics and kinematics of a robot. This is concerned with having the stationary robot perform precise motions, repetitively in a structured factory environment. AI robotics has concentrated on how a mobile robot should handle unpredictable events in an unstructured world. The design of an AI robot should consider how the robot will represent knowledge about the world, whether it needs to understand natural language, can it learn tasks, what kind of planning and problem solving will it have to do, how much inference is expected, how can it rapidly search its database and knowledge for answers, and what mechanisms will it use for perceiving the world.

Teleoperation arose as an intermediate solution to tasks that required automation but for which robots could not be adequately programmed to handle. Teleoperation methods typically are cognitive fatiguing, require high communication bandwidths and short communication delays, and require one or more teleoperators per remote. Telepresence techniques attempt to create a more natural interface for the human to control the robot and interpret what it is doing and seeing, but at a high communication cost. Supervisory control attempts to delegate portions of the task to the remote, either to do autonomously (traded control) or with reduced, but continuous, human interaction (shared control).

2

The Hierarchical Paradigm

Chapter Objectives:

1.5 *Teleoperation* Describe the Hierarchical Paradigm in terms of the three robot paradigms and its organization of sensing. ³⁸

Name and evaluate one representative Hierarchical architecture in terms of: support for modularity, niche targetability, ease of portability to other domains, robustness.

Solve a simple navigation problem using Strips, given a world model, operators, difference table, and difference evaluator. The state of the world model should be shown after each step.

Understand the meaning of the following terms as applied to robotics: precondition, closed world assumption, open world, frame problem.

Describe the mission planner, navigator, pilot organization of the Nested Hierarchical Controller.

List two advantages and disadvantages of the Hierarchical Paradigm.

2.1 Overview

The Hierarchical Paradigm is historically the oldest method of organizing intelligence in mainstream robotics. It dominated robot implementations from 1967, with the inception of the first AI robot, Shakey (Fig. 2.1) at SRI, up until the late 1980's when the Reactive Paradigm burst onto the scene.

This chapter begins with a description of the Hierarchical Paradigm in terms of the SENSE, PLAN, ACT primitives and by its sensing representation. The chapter then covers the use of Strips in Shakey to reason and plan

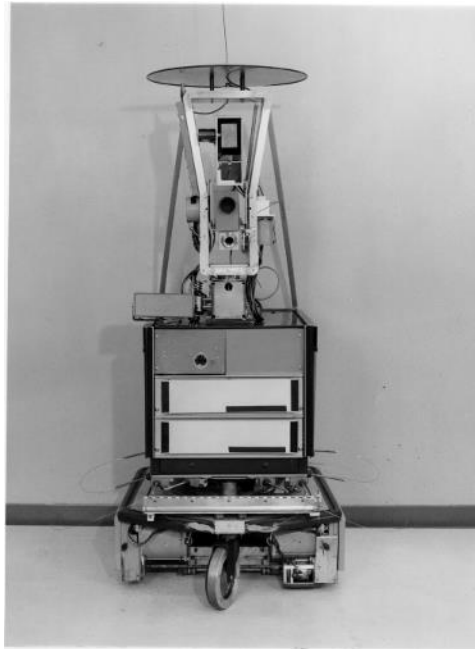


Figure 2.1 Shakey, the first AI robot. It was built by SRI for DARPA 1967–70. (Photograph courtesy of SRI.)

a path. Strips will serve to motivate the reader as to the computer challenges inherent in even as simple a task as walking across a room. However, Strips is not an architecture, per se, just an interesting technique which emerged from trying to build an architecture. Two representative architectures are presented, NHC and RCS, that serve as examples of robot architectures popular at the time. The chapter concludes with programming considerations.

2.2 Attributes of the Hierarchical Paradigm

As noted in Part I, a robotic paradigm is defined by the relationship between the three primitives (SENSE, PLAN, ACT) and by the way sensory data is processed and distributed through the system.

The Hierarchical Paradigm is sequential and orderly, as shown in Figs. 2.2 and 2.3. First the robot senses the world and constructs a global world map. Then “eyes” closed, the robot plans all the directives needed to

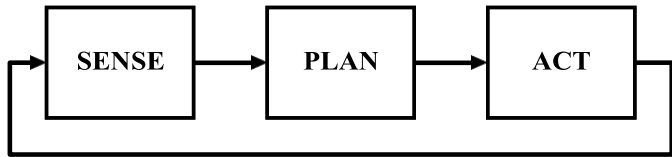


Figure 2.2 S,P,A organization of Hierarchical Paradigm.

ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	directives	Actuator commands

Figure 2.3 Alternative description of how the 3 primitives interact in the Hierarchical Paradigm.

goal. Finally, the robot acts to carry out the first directive. After the robot has carried out the **SENSE-PLAN-ACT** sequence, it begins the cycle again: eyes open, the robot senses the consequence of its action, replans the directives (even though the directives may not have changed), and acts.

As shown in Fig. 2.3, sensing in the Hierarchical Paradigm is monolithic: all the sensor observations are fused into one global data structure, which the planner accesses. The global data structure is generally referred to as a *world model*. The term world model is very broad; “world” means both the outside world, and whatever meaning the robot ascribes to it. In the Hierarchical Paradigm, the world model typically contains

WORLD MODEL

A PRIORI

1. an *a priori* (previously acquired) representation of the environment the robot is operating in (e.g., a map of the building),
2. sensing information (e.g., “I am in a hallway, based on where I’ve traveled, I must be in the northwest hallway”), plus
3. any additional cognitive knowledge that might be needed to accomplish a task (e.g., all packages received in the mail need to be delivered to Room

Creating a single representation which can store all of this information can be very challenging. Part of the reason for the “sub-turtle” velocity was the lack of computing power during the 1960’s. However, as roboticists in the 1980’s began to study biological intelligence, a consensus arose that even with increased computing power, the hierarchical, logic-based approach was unsatisfactory for navigational tasks which require a rapid response time to an open world.

2.2.1 Strips

ALGORITHM

Shakey, the first AI mobile robot, needed a generalized algorithm for planning how to accomplish goals. (An *algorithm* is a procedure which is correct and terminates.) For example, it would be useful to have the same program allow a human to type in that the robot is in Office 311 and should go to Office 313 or that the robot is in 313 and should the red box.

GENERAL PROBLEM

The method finally selected was a variant of the *General Problem Solver*

SOLVER (GPS)

STRIPS

MEANS-ENDS ANALYSIS

method, called *Strips*. Strips uses an approach called *means-ends analysis*, where if the robot can’t accomplish the task or reach the goal in one “move- ment”, it picks a action which will reduce the difference between what state it was in now (e.g., where it was) versus the goal state (e.g., where it wanted to be). This is inspired by cognitive behavior in humans; if you can’t see how to solve a problem, you try to solve a portion of the problem to see if it gets you closer to the complete solution.

GOAL STATE

INITIAL STATE

Consider trying to program a robot to figure out how to get to the Stanford AI Lab (SAIL). Unless the robot is at SAIL (represented in Strips as a variable goal state), some sort of transportation will have to be arranged. Suppose the robot is in Tampa, Florida (initial state). The robot may represent the decision process of how to get to a location as function called an operator which would consider the Euclidean distance (a variable named difference) between the goal state and initial state. The difference between locations could be computed for comparison purposes, or evaluation, by the square of the hypotenuse (difference evaluator). For

OPERATOR

DIFFERENCE

DIFFERENCE

EVALUATOR

example using an arbitrary frame of reference that put Tampa at the center of the world with made-up distances to Stanford:

initial state: Tampa, Florida (0,0)

goal state: Stanford, California (1000,2828)

DIFFERENCE TABLE

This could lead to a data structure called a difference table of how to make decisions:

difference	operator
d 200	fly
100<d < 200	ride train
d 100	drive
d< 1	walk

Different modes of transportation are appropriate for different distances. A mode of transportation, fly, ride_train, drive, walk, in the table is really a function in the robot's program. It is also called an operator, because it reduces the value stored in difference as to the distance from being in the initial state of Tampa and wanting to be at the goal state. A robot following this difference table would begin by flying as close as it could to SAIL.

But suppose the robot flew into the San Francisco airport. It'd be within 100 miles of SAIL, so the robot appears to have made an intelligent decision. But now the robot has a new difference to reduce. It examines the difference table with a new value of difference. The table says the robot should drive. Drive what? A car? Ooops: if the robot did have a personal car, it would be back in Tampa. The robot needs to be able to distinguish between driving its car and driving a rental car. This is done by listing the preconditions that have to be true in order to execute that particular operator. The preconditions are a column in the difference table, where a single operator can have multiple preconditions. In practice, the list of preconditions is quite lengthy, but for the purposes of this example, only drive_rental, drive_personal will be shown with preconditions.

PRECONDITIONS

difference	operator	preconditions
d 200	fly	
100<d<200	ride_train	
d 100	drive_rental	at airport
	drive_personal	at home
d<1	walk	

The difference table is now able to handle the issue of deciding to drive a rental car. But this introduces a new issue: how does the robot know where it is at? This is done by monitoring the state of the robot and its world. If it took an airplane from Tampa to the San Francisco airport, its

state has changed.

Its initial
state is now at
the San Francisco
airport, and no

2.2 Attributes of the Hierarchical Paradigm

49

ADD-LIST
DELETE-LIST

longer Tampa. Therefore, whenever the robot executes an operator, there is almost always something that has to be added to the robot's knowledge of the world state (which is entered into a `add-list`) and something that has to be deleted (`delete-list`). These two lists are stored in the difference table so that when the robot picks an operator based on the difference and operator, it can easily apply the appropriate modifications to the world. The difference table below is expanded to show the add and delete lists.

difference	operator	pre-conditions	add-list	delete-list
d 200	fly		at Y at airport	at X
100<d<200	train		at Y at station	at X
d 100	drive_rental	at airport		
	drive_personal	at home		
d<1	walk			

Of course, the above difference table is fairly incomplete. Driving a rental car should have a precondition that there is a rental car available. (And that the robot have a waiver from the state highway patrol to drive as an experimental vehicle and a satisfactory method of payment.) The number of facts and preconditions that have to be explicitly represented seem to be growing explosively. Which is Very Bad from a programming standpoint.

The main point is that the difference table appears to be a good data structure for representing what a robot needs in planning a trip. It should be apparent that a recursive function can be written which literally examines each entry in the table for the first operator that reduces the difference. The resulting list of operators is actually the plan: a list of the steps (operators) that the robot has to perform in order to reach a goal. The robot actually constructs the plan before handing it off to another program to execute.

At this point in time, it isn't likely that a robot will get on a plane and then drive. So perhaps the criticisms of Strips is because the example used too complicated a task to be realistic. Let's see if Strips is more streamlined with a simple task of getting from one room to another.

2.2.2 More realistic Strips example

51

CONSTRUCTING A
WORLD MODEL

The first step in creating a Strips planner is to construct a Strips-based representation of the world, or world model. Everything in the world that is

AXIOMS
PREDICATES

relevant to the problem is represented by facts, or *axioms*, in predicate logic. *Predicates* are functions that evaluate to TRUE or FALSE. By years of AI programming convention, predicates are usually written in uppercase.

Consider the problem of a robot named IT in a room, R1, who needs to go to another room, R2, in the house shown in Fig. 2.4. In order to solve this problem using Strips, the robot has to be given a way of representing the world, which will in turn influence the difference table, a difference evaluator, and how the add and delete lists are written. The world model in the previous example was never formally defined.

A world model is generally built up from static facts (represented as predicates) from a set of candidates, and things in the world, like the robot. The robot's name is in all capitals because it exists (and therefore is TRUE). Lowercase identifiers indicate that the thing is a variable, that a real thing hasn't been assigned to that placeholder yet.

Suppose the robot was limited to knowing only whether a movable object was in a room, next to a door or another movable object, and whether a door was open or closed and what rooms it connected. In a programming sense, there would be only three types of things in the world: `movable_object` (such as the robot, boxes it should pick up), `room`, and `door`. The robot's knowledge could be represented by the following predicates:

<code>INROOM(x, r)</code>	where x is an object of type <code>movable_object</code> , r is type <code>room</code>
<code>NEXTTO(x, t)</code>	where x is a <code>movable_object</code> , t is type <code>door</code> or <code>movable_object</code>
<code>STATUS(d, s)</code>	where d is type <code>door</code> , s is an enumerated type: <code>OPEN</code> or <code>CLOSED</code>
<code>CONNECTS(d, rx, ry)</code>	where d is type <code>door</code> , rx, ry are the <code>room</code>

With the above predicates, the world model for the initial state of the world in Fig. 2.4 would be represented by:

```
initial state:
INROOM(IT, R1)
INROOM(B1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
```

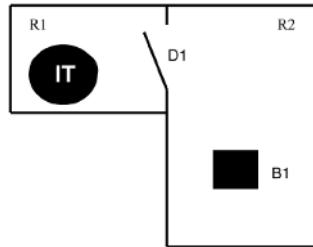



Figure 2.4 An example for Strips of two rooms joined by an open door.

This world model captures that a specific `movable_object` named `IT` is in a room named `R1`, and `B1` is in another room labeled `R2`. A door `D1` connects a room named `R1` to `R2` and it connects `R2` to `R1`. (Two different `CONNECTS` predicates are used to represent that a robot can go through the door from either door.) A door called `D1` has the enumerated value of being `OPEN`. The `NEXTTO` predicate wasn't used, because it wasn't true and there would be nothing to bind the variables to.

Under this style of representation, the world model for the `goal state` would be:

```
goal state:
INROOM(IT,R2)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
```

CONSTRUCTING THE
DIFFERENCE TABLE

Once the world model is established, it is possible to construct the difference table. The partial difference table is:

operator	preconditions	add-list	delete-list
OP1:	INROOM(IT, rk)	NEXTTO(IT, dx)	
GOTODOOR(IT, dx)	CONNECT(dx, rk, rm)		
OP2:	CONNECT(dx, rk, rm)	INROOM(IT, rm)	INROOM(IT, rk)
GOTHRUDOOR(IT, dx)	NEXTTO(IT, dx) STATUS(dx, OPEN) INROOM(IT, rk)		

This difference table says the robot is programmed for only two operations: go to a door, and go through a door. The GOTODOOR operator can be applied only if the following two preconditions are true:

INROOM(IT, rk) The robot is in a room, which will be assigned to the identifier rk.

CONNECT(dx, rk, rm) There is a door, which will be assigned to the identifier dx, which connects rk to some other room called rm.

The label IT is used to constrain the predicates. Notice that only the variables dx and rk get bound when GOTODOOR is called. rm can be anything. If GOTODOOR is executed, the robot is now next to the door called dx. Nothing gets deleted from the world state because the robot is still in room rk, the door dx still connects the two rooms rk and rm. The only thing that has changed is that the robot is now in a noteworthy position in the room: next to the door.

The difference table specifies that the GOTHRUDOOR operator will only work if the robot is in the room next to the door, the door is open, and the door connects the room the robot is in to another room. In this case, predicates must be added and deleted from the world model when the operator executes. When the robot is in room rk and goes through the door, it is now in room rm (which must be added to the world model) and is no longer in room rk (which must be deleted).

So far, the world model and difference table should seem reasonable, although tedious to construct. But constructing a difference table is pointless without an evaluation function for differences. (Notice that there wasn't a column for the difference in the above table.) The difference evaluator in the travel example was Euclidean distance. In this example, the evaluator is predicate calculus, where the initial state is logically subtracted from the goal state. The logical difference between the initial state goal state is simply:

:INROOM(IT, R2) or INROOM(IT, R2)=FALSE

REDUCING Reducing differences is a bit like a jigsaw puzzle where Strips tries different substitutions to see if a particular operator will reduce the difference. In order to reduce the difference, Strips looks in the difference table, starting at

DIFFERENCES the top, under the add-list column for a match. It looks in the add-list rather than a separate differences column because the add-list expresses what the result of the operator is. If Strips finds an operator that produces the goal state, then that operator eliminates the existing difference between the initial and goal states.

The add-list in OP2: GOTHRUDOOR has a match on form. If $rm=R2$, then the result of OP2 would be INROOM(IT, R2). This would eliminate the difference, so OP2 is a candidate operator.

Before the OP2 can be applied, Strips must check the preconditions. To do this, rm must be replaced with R2 in every predicate in the preconditions. OP2 has two preconditions, only CONNECTS(dx, rk, rm) is affected. It becomes CONNECTS(dx, rk, R2). Until dx and rk are bound, the predicate doesn't have a true or false value. Essentially dx, rk are wildcards, CONNECTS(*, *, R2). To fill in values for these variables, Strips looks at the current state of the world model to find a match. The predicate in the current state of the world model CONNECTS(D1, R1, R2) matches CONNECTS(*, *, R2). D1 is now bound to dx and R1 is bound to rk.

Now Strips propagates the bindings to the next precondition on the list: NEXTTO(IT, dx). NEXTTO(IT, D1) is FALSE because the predicate is not in the current world state. NEXTTO(IT, D1) is referred to as a *failed*

FAILED

PRECONDITIONS *precondition*. An informal interpretation is that GOTHRUDOOR(IT, D1) will get the robot to the goal state, but before it can do that, IT has to be next to D1.

RECURSION TO

Rather than give up, STRIPS *recurses* (uses the programming technique of

RESOLVE DIFFERENCES

recursion) to repeat the entire procedure. It marks the original goal state as G0, pushes it on a stack, then it creates a new sub-goal state, G1.

The difference between NEXTTO(IT, D1) and the current world state is:

:NEXTTO(IT, D1)

Strips once again searches through the add-list in the difference table to find an operator that would negate this. Indeed, OP1: GOTODOOR(IT, dx) has a match in the add-list of NEXTTO(IT, dx).

2.2 Attributes of the Hierarchical Paradigm
Strips has to start over with reassigning values to the identifiers because⁵⁸
the program has entered a new programming scope, so $dx=D1$.

Again, Strips examines the preconditions. This time $rk=R1$ and $rm=R2$ can be matched with `CONNECTS(dx, rk, rm)`, and all preconditions are satisfied (that is, they evaluate to true). Strips puts the operator `OP1` on the plan stack and applies the operator to the world model, changing the state. (Note that this is the equivalent of a “mental operation”; the robot doesn’t actually physically go to the door, it just changes the state to imagine what would happen if it did.)

To recall, the initial state of the world model was:

```
initial state:
INROOM(IT, R1)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
```

Applying the operator `OP1` means making the changes on the add-list and delete-list. There is only a predicate on the add-list and none on the delete- list. After adding `NEXTTO(IT, D1)`, the state of the world is:

```
state after OP1:
INROOM(IT, R1)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
NEXTTO(IT, D1)
```

Strips then returns control back to the previous call. It resumes where it left off in evaluating the preconditions for `OP2` with $dx=D1$, $rm=R2$ and $rk=R1$, only now the world model has changed. Both `STATUS(D1, OPEN)` and `INROOM(IT, R1)` are true, so all the preconditions for `OP2` are satisfied. Strips puts `OP2` on its plan stack and changes the world model by applying the add-list and delete-list predicates. This results in what the state of the world will be when the plan executes:

```
state after OP2:
INROOM(IT, R2)
INROOM(B1,R2)
```



```

CONNECTS (D1, R1, R2)
CONNECTS (D1, R2, R1)
STATUS (D1, OPEN)
NEXTTO (IT, D1)

```

Strips exits and the plan for the robot to physically execute (in reverse order on the stack) is: GOTODOOR (IT, D1) , GOTHURDOOR (IT, D1) .

2.2.3 Strips summary

Strips works recursively; if it can't reach a goal directly, it identifies the problem (a failed precondition), then makes the failed precondition a subgoal. Once the subgoal is reached, Strips puts the operator for reaching the subgoal on a list, then backs up (pops the stack) and resumes trying to reach the previous goal. Strips plans rather than execute: it creates a list of operators to apply; it does not apply the operator as it goes. Strips implementations requires the designer to set up a:

world model representation

difference table with operators, preconditions, add, and delete lists

difference evaluator

The steps in executing Strips are:

1. Compute the difference between the goal state and the initial state using the difference evaluation function. If there is no difference, terminate.
2. If there is a difference, reduce the difference by selecting the first operator from the difference table whose add-list has a predicate which negates the difference.
3. Next, examine the preconditions to see if a set of bindings for the variables can be obtained which are all true. If not, take the first false precondition, make it the new goal and store the original goal by pushing it on a stack. Recursively reduce that difference by repeating step 2 and 3.
4. When all preconditions for an operator match, push the operator onto the plan stack and update a copy of the world model. Then return to the operator with the failed precondition so it can apply its operator or recurse on another failed precondition.

2.3 Closed World Assumption and the Frame Problem

CLOSED WORLD	Strips sensitized the robotics community to two pervasive issues: the <i>closed</i>
ASSUMPTION	<i>world assumption</i> and the <i>frame problem</i> . As defined earlier, the closed world
FRAME PROBLEM	assumption says that the world model contains everything the robot needs to know: there can be no surprises. If the closed world assumption is violated, the robot may not be able to function correctly. But, on the other hand, it is very easy to forget to put all the necessary details into the world model. As a result, the success of the robot depends on how well the human programmer can think of everything.
	But even assuming that the programmer did come up with all the cases, the resulting world model is likely to be huge. Consider how big and cumbersome the world model was just for moving between 2 rooms. And there were no obstacles! People began to realize that the number of facts (or axioms) that the program would have to sort through for each pass through the difference table was going to become intractable for any realistic application. The problem of representing a real-world situation in a way that was computationally tractable became known as the frame problem. The opposite of the closed world assumption is known as the <i>open world assumption</i> .
OPEN WORLD	
ASSUMPTION	When roboticists say that “a robot must function in the open world,” they are saying the closed world assumption cannot be applied to that particular domain.
	The above example, although trivial, shows how tedious Strips is (though computers are good at tedious algorithms). In particular, the need to formally represent the world and then maintain every change about it is non-intuitive. It also illustrates the advantage of a closed-world assumption: imagine how difficult it would be to modify the planning algorithm if the world model could suddenly change. The algorithm could get lost between recursions. The example should also bring home the meaning of the frame problem: imagine what happens to the size of the world model if a third room is added with boxes for the robot to move to and pick up! And this is only for a world of rooms and boxes. Clearly the axioms which frame the world will become too numerous for any realistic domain.
	One early solution was ABStrips which tried to divide the problem into multiple layers of abstraction, i.e., solve the problem on a coarse level

2.2 Attributes of the Hierarchical Paradigm 62
first. That had its drawbacks, and soon many people who had started out in robotics found themselves working on an area of AI called planning. The two fields became distinct, and by the 1980's, the planning and robotics researchers had separate conferences and publications. Many roboticists dur-

ing the 1970's and 1980's worked on either computer vision related issues, trying to get the robots to be able to better sense the world, or on path plan- ning, computing the most efficient route around obstacles, etc. to a goal lo- cation.

24 Representative Architectures

As mentioned in Part I an architecture is a method of implementing a para- digm, of embodying the principles in some concrete way. Ideally, an archi- tecture is generic; like a good object-oriented program design, it should have many reusable pieces for other robot platforms and tasks.

Possibly the two best known architectures of the Hierarchical period are the Nested Hierarchical Controller (NHC) developed by Meystel⁹³ and the NIST Realtime Control System (RCS) originally developed by Albus,¹ with its teleoperation version for JPL called NASREM.

2.4.1 Nested Hierarchical Controller

As shown in Fig. 2.5, the Nested Hierarchical Controller architecture has components that are easily identified as either **SENSE**, **PLAN**, or **ACT**. The robot begins by gathering observations from its sensors and combining those observations to form the World Model data structure through the **SENSE** activity. The World Model may also contain *a priori* knowledge about the world, for example, maps of a building, rules saying to stay away from the foyer during the start and finish of business hours, etc. After the World Model has been created or updated, then the robot can **PLAN** what actions it should take. Planning for navigation has a local procedure consisting of three steps executed by the Mission Planner, Navigator, and Pilot. Each of these modules has access to the World Model in order to compute their por- tion of planning. The last step in planning is for the Pilot module to generate specific actions for the robot to do (e.g., Turn left, turn right, move straight at a velocity of 0.6 meters per second). These actions are translated into actua- tor control signals (e.g., Velocity profile for a smooth turn) by the Low-Level Controller. Together, the Low-Level Controller and actuators form the **ACT** portion of the architecture.

The major contribution of NHC was its clever decomposition of planning into 3 different functions or subsystems aimed at supporting navigation: the

MISSION PLANNER
NAVIGATOR
PILOT

2.4 Representative Architectures

Mission Planner, the Navigator, and the Pilot. As shown in Fig. 2.6, the Mission Planner either receives a mission from a human or generates a mission

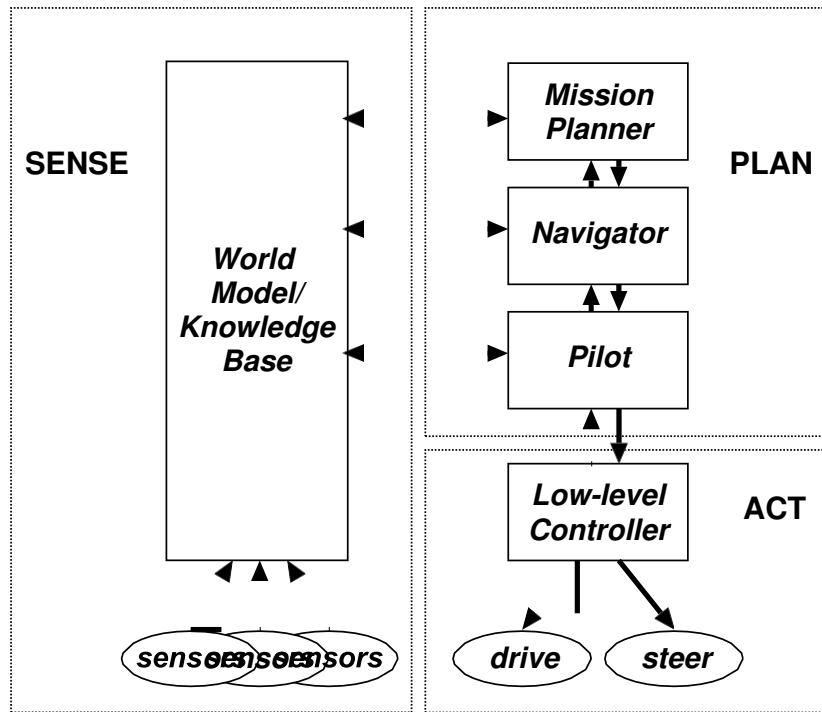


Figure 2.5 Nested Hierarchical Controller.

for itself, for example: pick up the box in the next room. The Mission Planner is responsible for operationalizing, or translating, this mission into terms that other functions can understand: box=B1; rm=ROOM311. The Mission Planner then accesses a map of the building and locates where the robot is and where the goal is. The Navigator takes this information and generates a path from the current location to the goal. It generates a set of waypoints, or straight lines for the robot to follow. The path is passed to the Pilot. The Pilot takes the first straight line or path segment and determines what actions the robot has to do to follow the path segment. For instance, the robot may need to turn around to face the way point before it can start driving forward.

What happens if the Pilot gives directions for a long path segment (say 50 meters) or if a person suddenly walks in front of the robot? Unlike Shakey, under NHC, the robot is not necessarily walking around with its eyes closed. After the Pilot gives the Low-Level Controller commands

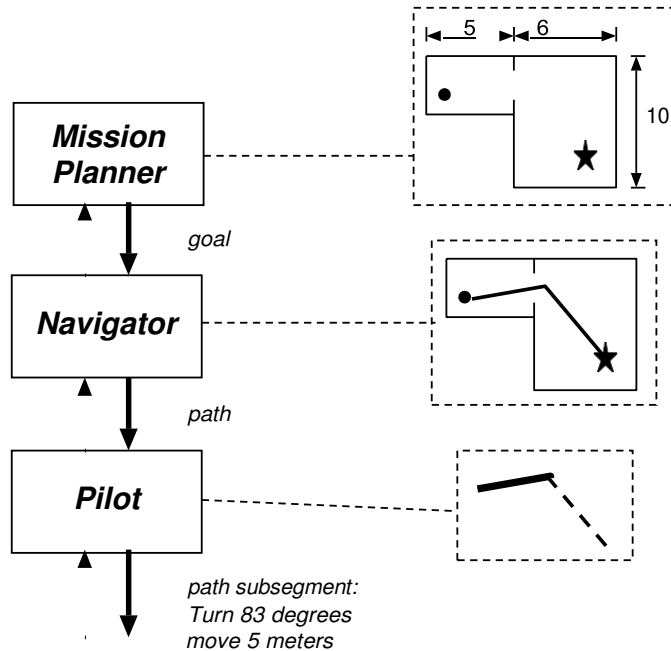


Figure 2.6 Examination of planning components in the NHC architecture.

sends actuator signals, the robot polls its sensors again. The World Model is updated. However, the entire planning cycle does not repeat. Since the robot has a plan, it doesn't need to rerun the Mission Planner or the Navigator. Instead, the Pilot checks the World Model to see if the robot has drifted off the path subsegment (in which case it generates a new control signal), has reached the waypoint, or if an obstacle has appeared. If the robot has reached its waypoint, the Pilot informs the Navigator. If the waypoint isn't the goal, then there is another path subsegment for the robot to follow, and so the Navigator passes the new subsegment to the Pilot. If the waypoint is the goal, then the Navigator informs the Mission Planner that the robot has reached the goal. The Mission Planner may then issue a new goal, e.g., Return to the starting place. If the robot has encountered an obstacle to its path, the Pilot passes control back to the Navigator. The Navigator must compute a new path, and subsegments, based on the updated World Model. Then it gives the updated path subsegment to the Pilot to carry out.

NHC has several advantages. It differs from Strips in that it interleaves planning and acting. The robot comes up with a plan, starts executing it, then changes it if the world is different than it expected. Notice that the decomposition is inherently hierarchical in intelligence and scope. The Mission Planner is “smarter” than the Navigator, who is smarter than the Pilot. The Mission Planner is responsible for a higher level of abstraction than the Navigator, etc. We will see that other architectures, both in the Hierarchical and Hybrid paradigms, will make use of the NHC organization.

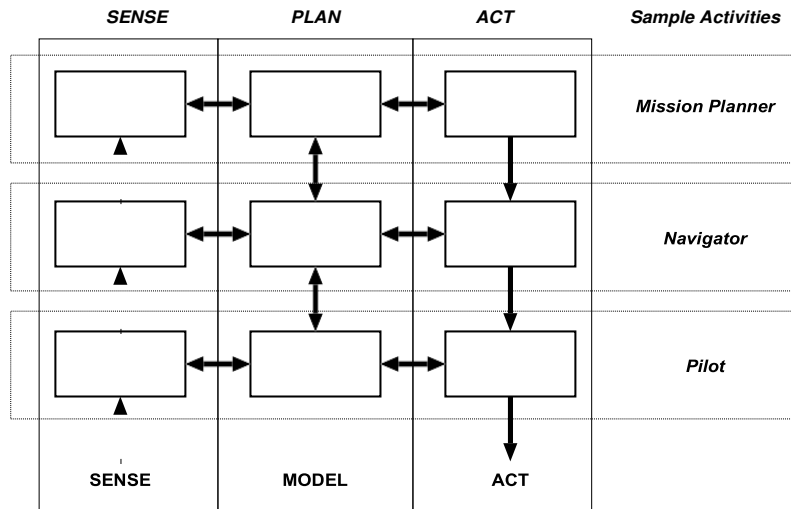
One disadvantage of the NHC decomposition of the planning function is that it is appropriate only for navigation tasks. The division of responsibilities seems less helpful, or clear, for tasks such as picking up a box, rather than just moving over to it. The role of a Pilot in controlling end-effectors is not clear. At the time of its initial development, NHC was never implemented and tested on a real mobile robot; hardware costs during the Hierarchical period forced most roboticists to work in simulation.

2.4.2 NIST RCS

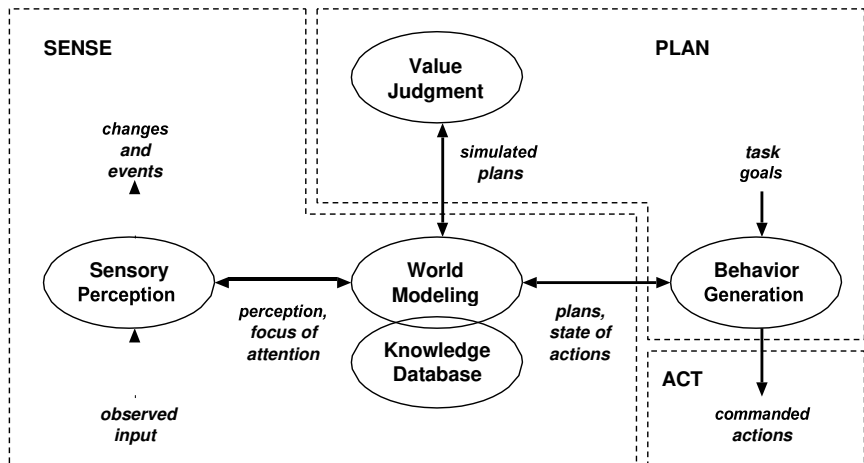
Jim Albus at the National Bureau of Standards (later renamed the National Institute of Standards and Technology or NIST) anticipated the need for intelligent industrial manipulators, even as engineering and AI researchers were splitting into two groups. He saw that one of the major obstacles in applying AI to manufacturing robots was that there were no common terms, no common set of design standards. This made industry and equipment manufacturers leery of AI, for fear of buying an expensive robot that would not be compatible with robots purchased in the future. He developed a very detailed architecture called the Real-time Control System (RCS) Architecture to serve as a guide for manufacturers who wanted to add more intelligence to their robots. RCS used NHC in its design, as shown in Fig. 2.7.

SENSE activities are grouped into a set of modules under the heading of sensory perception. The output of the sensors is passed off to the world modeling module which constructs a global map using information in its associated knowledge database about the sensors and any domain knowledge (e.g., the robot is operating underwater). This organization is similar to NHC. The main difference is that the sensory perception

module introduces a useful preprocessing step between the sensor and the fusion into a world model. As will be seen in Ch. 6, sensor preprocessing is often referred to as *feature extraction*.



a.



b.

Figure 2.7 Layout of RCS: a.) hierarchical layering of sense-model-act, and b.) functional decomposition.

The Value Judgment module provides most of the functionality associated with the PLAN activity: it plans, then simulates the plans to ensure they will work. Then, as with Shakey, the Planner hands off the plan to another module, Behavior Generation, which converts the plans into actions that the robot can actually perform (ACT). Notice that the Behavior Generation module is similar to the Pilot in NHC, but there appears to be less focus on navigation tasks. The term “behavior” will be used by Reactive and Hybrid Deliberative/Reactive architectures. (This use of “behavior” in RCS is a bit of retrofit, as Albus and his colleagues at NIST have attempted to incorporate new advances. The integration of all sensing into a global world model for planning and acting keeps RCS a Hierarchical architecture.) There is another module, operator interface, which is not shown which allows a human to “observe” and debug what a program constructed with the architecture is doing.

The standard was adapted by many government agencies, such as NASA and the US Bureau of Mines, who were contracting with universities and companies to build robot prototypes. RCS serves as a blueprint for saying: “here’s the types of sensors I want, and they’ll be fused by this module into a global map, etc.” The architecture was considered too detailed and restrictive when it was initially developed by most AI researchers, who continued development of new architectures and paradigms on their own. Fig. 2.8 shows three of the diverse mobile robots that have used RCS.

NASREM

A close inspection of the NHC and RCS architectures suggests that they are well suited for semi-autonomous control. The human operator could provide the world model (via eyes and brain), decide the mission, decompose it into a plan, and then into actions. The lower level controller (robot) would carry out the actions. As robotics advanced, the robot could replace more functions and “move up” the autonomy hierarchy. For example, taking over the pilot’s responsibilities; the human could instruct the robot to stay on the road until the first left turn. As AI advanced, the human would only have to serve as the Mission Planner: “go to the White House.” And so on. Albus noted this and worked with JPL to develop a version of RCS for teleoperating a robot arm in space. This is called the *NASREM* architecture and is still in use today.

2.4.3 Evaluation of hierarchical architectures

2.4 Representative Architectures
Recall from Part I that there are four criteria for evaluating an architecture: support for modularity, niche targetability, ease of portability to other domains, and robustness. NHC and RCS both provide some guidelines in how

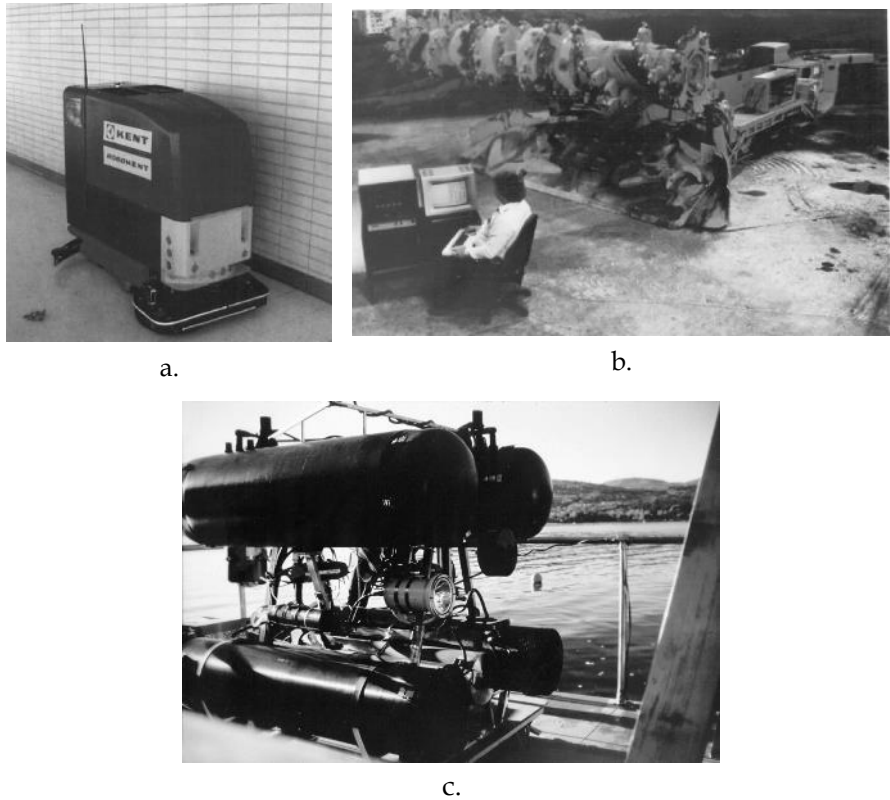


Figure 2.8 Three of the diverse mobile robots that have used RCS: a.) a commercial floor cleaning robot, b.) a mining robot, and c.) a submersible or underwater robot. (Photographs courtesy of the National Institute of Standards and Technology.)

to decompose a robot program into intuitive modules. The NHC decomposition of mission planner, navigator, and pilot was focused strictly on navigation, while RCS appears broader. Both have been used recently for successful vehicle guidance, with RCS being used to control mining equipment, submarines, and cars. So both have reasonable niche targetability. The ease of portability to other domains is unclear. The architectures are expressed at a very broad level, akin to “a house should have bedrooms, bathrooms, and a kitchen.” Architectures which are more specific, “there should be one bathroom for every two bedrooms,” and which have associated techniques simplify portability. It is hard to see how the code written for a mining machine

could be reused for a submarine, especially since RCS is not object-oriented. In terms of robustness, RCS does attempt to provide some explicit mechanisms. In particular, it assumes the Value Judgment module simulates a plan to confirm that it should be successful when deployed. The use of simulation is common for operating equipment in a well-known environment where every piece of equipment is known. The most notable example is a nuclear processing cell. With such detailed information, it is fairly straightforward (although computationally expensive) to simulate whether a particular course for a robot would collide with equipment and cause a spill. This is a very limited form of robustness. The disadvantage is the time delay caused by the robot mentally rehearsing its actions prior to executing them. Simulation may not be appropriate for all actions; if a piece of the ceiling is falling on the robot, it needs to get out of the way immediately or risk coming up with the best place to move too late to avoid being crushed.

25 Advantages and Disadvantages

Robots built in the time period before 1990 typically had a Hierarchical style of software organization. They were generally developed for a specific application rather than to serve as a generic architecture for future applications. The robots are interesting because they illustrate the diversity and scope of applications being considered for mobile robots as far back as 15 or 20 years ago.

The primary advantage of the Hierarchical Paradigm was that it provides an ordering of the relationship between sensing, planning, and acting. The primary disadvantage was planning. Every update cycle, the robot had to update a global world model and then do some type of planning. The sensing and planning algorithms of the day were extremely slow (and many still are), so this introduced a significant bottleneck. Notice also that sensing and acting are always disconnected. This effectively eliminated any stimulus-response types of actions ("a rock is crashing down on me, I should move *anywhere*") that are seen in nature.

The dependence on a global world model is related to the frame problem. In Strips, in order to do something as simple as opening a door, the robot had to reason over all sorts of details that were irrelevant (like other rooms, other doors). NHC and RCS represent attempts to divide up

the world model into pieces best suited for the type of actions; for example, consider the roles of the Mission Planner, Navigator, and Pilot. Unfortunately, these decomposi-

tions appear to be dependent on a particular application. As a result, robotics gained a reputation as being more of an art than a science.

Another issue that was never really handled by architectures in the Hierarchical Paradigm was uncertainty. Uncertainty comes in many forms, such as semantic (how close does NEXTTO mean anyway?), sensor noise, and actuator errors. Another important aspect of uncertainty is action completion: did the robot actually accomplish the action? One robotics researcher said that their manipulator was only able to pick up a cup 60% of the attempts; therefore they had to write a program to check to see if it was holding a cup and then restart the action if it wasn't. Because Shakey essentially closed its eyes during planning and acting, it was vulnerable to uncertainty in action completion.

26 Programming Considerations

It is interesting to note that the use of predicate logic and recursion by Strips favors languages like Lisp and PROLOG. These languages were developed by AI researchers specifically for expressing logical operations. These languages do not necessarily have good real-time control properties like C or C++. However, during the 1960's the dominant scientific and engineering language was FORTRAN IV which did not support recursion. Therefore, researchers in AI robotics often chose the lesser of two evils and programmed in Lisp. The use of special AI languages for robotics may have aided the split between the engineering and AI approaches to robotics, as well as slowed down the infusion of ideas from the two communities. It certainly discouraged non-AI researchers from becoming involved in AI robotics.

The Hierarchical Paradigm tends to encourage monolithic programming, rather than object-oriented styles. Although the NHC decomposes the planning portion of intelligence, the decomposition is strictly functional. In particular, NHC and RCS don't provide much guidance on how to build modular, reusable components.

27 Summary

The Hierarchical Paradigm uses a SENSE then PLAN then ACT (S,P,A). It organizes sensing into a global data structure usually called a world model that may have an associated knowledge base to contain *a priori*

maps or knowledge relevant to a task. Global data structures often flag that an ar-

chitecture will suffer from the frame problem. The Hierarchical Paradigm was introduced in the first mobile robot, Shakey. Strips is an important planning technique that came out of the Shakey project at SRI, which focused on the PLAN primitive in robotics. Concepts and terms which emerged that continue to play an important role in defining robotics are: preconditions, the closed and open world assumptions, and the frame problem. Hierarchical systems have largely fallen out of favor except for the NIST Realtime Control Architecture. The decline in popularity is due in part to its focus on strong niche targetability at the expense of true modularity and portability. However, as will be seen in the following chapters, insights from biology and cognitive science have led to paradigms with more intuitive appeal. One often overlooked property of most hierarchical architectures is that they tend to support the evolution of intelligence from semi-autonomous control to fully autonomous control.

3 *Biological Foundations of the Reactive Paradigm*

Chapter Objectives:

Describe the three levels in a *computational theory*.

Explain in one or two sentences each of the following terms: *reflexes, taxes, fixed-action patterns, schema, affordance*.

Be able to write pseudo-code of an animal's behaviors in terms of *innate releasing mechanisms*, identifying the releasers for the behavior.

Given a description of an animal's sensing abilities, its task, and environment, identify an *affordance* for each behavior.

Given a description of an animal's sensing abilities, its task, and environment, define a set of behaviors using schema theory to

3.1 Overview

Progress in robotics in the 1970's was slow. The most influential robot was the Stanford Cart developed by Hans Moravec, which used stereo vision to see and avoid obstacles protruding from the ground. In the late 1970's and early 80's, Michael Arbib began to investigate models of animal intelligence from the biological and cognitive sciences in the hopes of gaining insight into what was missing in robotics. While many roboticists had been fascinated by animals, and many, including Moravec, had used some artifact of animal behavior for motivation, no one approached the field with the seriousness and dedication of Arbib.

At nearly the same time, a slender volume by Valentino Braitenberg, called *Vehicles: Experiments in Synthetic Psychology*,²⁵ appeared. It was a series of gedanken or entirely hypothetical thought experiments, speculating as to how machine intelligence could evolve. Braitenberg started with simple thermal sensor-motor actuator pair (Vehicle 1) that could move faster in warm areas and slower in cold areas. The next, more complex vehicle had two thermal sensor-motor pairs, one on each side of the vehicle. As a result of the differential drive effect, Vehicle 2 could turn around to go back to cold areas. Throughout the book, each vehicle added more complexity. This layering was intuitive and seemed to mimic the principles of evolution in primates. *Vehicles* became a cult tract among roboticists, especially in Europe.

Soon a new generation of AI researchers answered the siren's call of biological intelligence. They began exploring the biological sciences in search of new organizing principles and methods of obtaining intelligence. As will be seen in the next chapter, this would lead to the Reactive Paradigm. This chapter attempts to set the stage for the Reactive Paradigm by recapping influential studies and discoveries, and attempting to cast them in light of how they can contribute to robotic intelligence.

The chapter first covers animal behaviors as the fundamental primitive for sensing and acting. Next, it covers the work of Lorenz and Tinbergen in defining how concurrent simple animal behaviors interact to produce complex behaviors through Innate Releasing Mechanisms (IRMs). A key aspect of an animal behavior is that perception is needed to support the behavior. The previous chapter on the Hierarchical Paradigm showed how early roboticists attempted to fuse all sensing into a global world map, supplemented with a knowledge base. This chapter covers how the work of cognitive psychologists Ulrich Neisser¹⁰⁹ and J.J. Gibson⁵⁹ provides a foundation for thinking about robotic perception. Gibson refuted the necessity of global world models, a direct contradiction to the way perception was handled in the hierarchical paradigm. Gibson's use of *affordances*, also called *direct perception*, is an important key to the success of the Reactive Paradigm. Later work by Neisser attempts to define when global models are appropriate and when an affordance is more elegant.

ETHOLOGY
COGNITIVE

Many readers find the coverage on *ethology* (study of animal behavior) and *cognitive psychology* (study of how humans think and represent knowledge) to be interesting, but too remote from robotics. In order to address this con-

PSYCHOLOGY

^{3.1 Overview}
cern, the chapter discusses specific principles and how they can be⁶⁹
applied to robotics. It also raises issues in transferring animal models of
behavior to robots. Finally, the chapter covers *schema theory*, an attempt
in cognitive

psychology to formalize aspects of behavior. Schema theory has been used successfully by Arbib to represent both animal and robot behavior. It is implicitly object-oriented and so will serve as the foundation of discussions throughout the remainder of this book.

3.1.1 Why explore the biological sciences?

Why should roboticists explore biology, ethology, cognitive psychology and other biological sciences? There is a tendency for people to argue against considering biological intelligence with the analogy that airplanes don't flap their wings. The counter-argument to that statement is that almost everything else about a plane's aerodynamics duplicates a bird's wing: almost all the movable surfaces on the wing of a plane perform the same functions as parts of a bird's wing. The advances in aeronautics came as the Wright Brothers and others extracted aerodynamic principles. Once the principles of flight were established, mechanical systems could be designed which adhered to these principles and performed the same functions but not necessarily in the same way as biological systems. The "planes don't flap their wings" argument turns out to be even less convincing for computer systems: animals make use of innate capabilities, robots rely on compiled programs.

Many AI roboticists often turn to the biological sciences for a variety of reasons. Animals and man provide existence proofs of different aspects of intelligence. It often helps a researcher to know that an animal can do a particular task, even if it isn't known how the animal does it, because the researcher at least knows it is possible. For example, the issue of how to combine information from multiple sensors (sensor fusion) has been an open question for years. At one point, papers were being published that robots shouldn't even try to do sensor fusion, on the grounds that sensor fusion was a phenomenon that sounded reasonable but had no basis in fact. Additional research showed that animals (including man) do perform sensor fusion, although with surprisingly different mechanisms than most researchers had considered.

The principles of animal intelligence are extremely important to roboticists. Animals live in an *open world*, and roboticists would like to overcome

OPEN WORLD

ASSUMPTION
CLOSED WORLD
ASSUMPTION

the *closed world assumption* that presented so many problems with Shakey.

FRAME PROBLEM

Many “simple” animals such as insects, fish, and frogs exhibit intelligent behavior yet have virtually no brain. Therefore, they must be doing something that avoids the frame problem.

3.1.2 Agency and computational theory

Even though it seems reasonable to explore biological and cognitive sciences for insights in intelligence, how can we compare such different systems: carbon and silicon “life” forms? One powerful means of conceptualizing the different systems is to think of an abstract intelligent system. Consider some-

AGENT

thing we’ll call an *agent*. The agent is self-contained and independent. It has its own “brains” and can interact with the world to make changes or to sense what is happening. It has self-awareness. Under this definition, a person is an agent. Likewise, a dog or a cat or a frog is an agent. More importantly, an intelligent robot would be an agent, even certain kinds of web search engines which continue to look for new items of interest to appear, even after the user has logged off. Agency is a concept in artificial intelligence that allows researchers to discuss the properties of intelligence without discussing the details of how the intelligence got in the particular agent. In OOP terms, “agent” is the superclass and the classes of “person” and “robot” are derived from it.

COMPUTATIONAL

Of course, just referring to animals, robots, and intelligent software packages as “agents” doesn’t make the correspondences between intelligence any clearer. One helpful way of seeing correspondences is to decide the level at which these entities have something in common. The set of levels of commonality lead to what is often called a *computational theory*⁸⁸ after David

THEORY

Marr. Marr was a neurophysiologist who tried to recast biological vision processes into new techniques for computer vision. The levels in a computational theory can be greatly simplified as:

Level 1: Existence proof of what can/should be done. Suppose a roboticist is interested in building a robot to search for survivors trapped in a building after an earthquake. The roboticist might consider animals which seek out humans. As anyone who has been camping knows, mosquitoes are very good at finding people. Mosquitoes provide an existence proof that it is possible for a computationally simple agent to find a human being using heat. At Level 1, agents can share a commonality of purpose or functionality.

Level 2: Decomposition of “what” into inputs, outputs, and transformations. This level can be thought of as creating a flow chart of “black boxes.” Each box represents a transformation of an input into an output.

3.1 *Overview* Returning to the example of a mosquito, the roboticist might realize from biology that the mosquito finds humans by homing on the heat of a hu-

73

man (or any warm blooded animal). If the mosquito senses a hot area, it flies toward it. The roboticist can model this process as: $\text{input} = \text{thermal image}, \text{output} = \text{steering command}$. The “black box” is how the mosquito transforms the input into the output. One good guess might be to take the centroid of the thermal image (the centroid weighted by the heat in each area of the image) and steer to that. If the hot patch moves, the thermal image will change with the next sensory update, and a new steering command will be generated. This might not be exactly how the mosquito actually steers, but it presents an idea of how a robot could duplicate the functionality. Also notice that by focusing on the process rather than the implementation, a roboticist doesn’t have to worry about mosquitoes flying, while a search and rescue robot might have wheels. At Level 2, agents can exhibit common processes.

Level 3: How to implement the process. This level of the computational theory focuses on describing how each transformation, or black box, is implemented. For example, in a mosquito, the steering commands might be implemented with a special type of neural network, while in a robot, it might be implemented with an algorithm which computes the angle between the centroid of heat and where the robot is currently pointing. Likewise, a researcher interested in thermal sensing might examine the mosquito to see how it is able to detect temperature differences in such a small package; electro-mechanical thermal sensors weigh close to a pound! At Level 3, agents may have little or no commonality in their implementation.

It should be clear that Levels 1 and 2 are abstract enough to apply to any agent. It is only at Level 3 that the differences between a robotic agent and a biological agent really emerge. Some roboticists actively attempt to emulate biology, reproducing the physiology and neural mechanisms. (Most roboticists are familiar with biology and ethology, but don’t try to make exact duplicates of nature.) Fig. 3.1 shows work at Case Western Reserve’s Bio-Bot Laboratory under the direction of Roger Quinn, reproducing a cockroach on a neural level.

In general, it may not be possible, or even desirable, to duplicate how a biological agent does something. Most roboticists do not strive to precisely replicate animal intelligence, even though many build creatures which resemble animals, as seen by the insect-like Genghis in Fig. 3.2.

But by focusing on Level 2 of a computational theory of intelligence, roboticists can gain insights into how to organize intelligence.

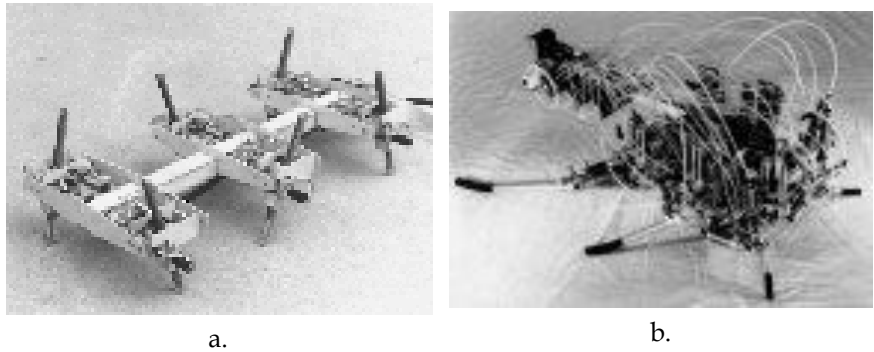


Figure 3.1 Robots built at the Bio-Bot Laboratory at Case Western Reserve University which imitate cockroaches at Level 3: a.) Robot I, an earlier version, and b.) Robot III. (Photographs courtesy of Roger Quinn.)

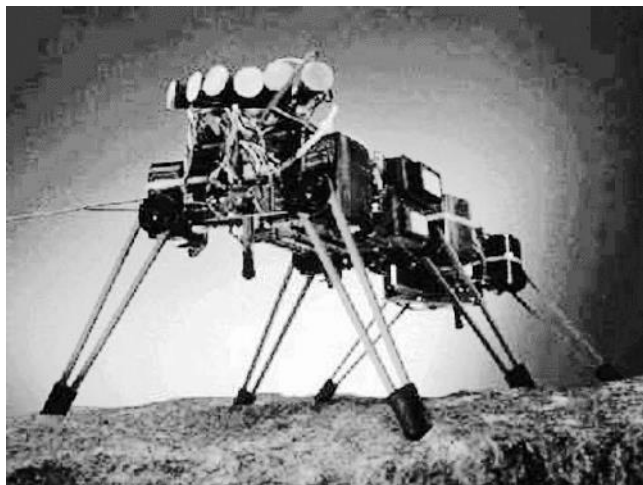


Figure 3.2 Genghis, a legged robot built by Colin Angle, IS Robotics, which imitates an insect at Levels 1 and 2. (Photograph courtesy of the National Aeronautics and Space Administration.)

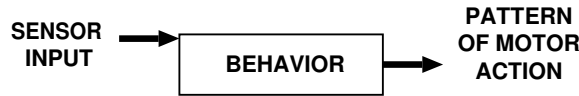


Figure 3.3 Graphical definition of a behavior.

3.2 What Are Animal Behaviors?

BEHAVIOR

Scientists believe the fundamental building block of natural intelligence is a behavior. A *behavior* is a mapping of sensory inputs to a pattern of motor actions which then are used to achieve a task. For example, if a horse sees a predator, it flattens its ears, lowers its head, and paws the ground. In this case, the sensory input of a predator triggers a recognizable pattern of a defense behavior. The defensive motions make up a pattern because the actions and sequence is always the same, regardless of details which vary each episode (e.g., how many times the horse paws the ground). See Fig. 3.3.

Scientists who study animal behaviors are called *ethologists*. They often spend years in the field studying a species to identify its behaviors. Often the pattern of motor actions is easy to ascertain; the challenging part is to discover the sensory inputs for the behavior and why the behavior furthers the species survival.

REFLEXIVE BEHAVIOR

STIMULUS-RESPONSE

Behaviors can be divided into three broad categories.¹⁰ *Reflexive behaviors* are *stimulus-response (S-R)*, such as when your knee is tapped, it jerks upward. Essentially, reflexive behaviors are “hardwired”; neural circuits ensure that the stimulus is directly connected to the response in order to produce the fastest response time. *Reactive behaviors* are learned, and then consolidated to where they can be executed without conscious thought. Any behavior that involves what is referred to in sports as “muscle memory” is usually a *reactive behavior* (e.g., riding a bike, skiing). *Reactive behaviors* can also be changed by conscious thought; a bicyclist riding over a very narrow bridge might “pay attention” to all the movements. *Conscious behaviors* are *deliberative* (assembling a robot kit, stringing together previously developed behaviors, etc.).

REACTIVE BEHAVIOR

CONSCIOUS BEHAVIOR

The categorization is worthy of note for several reasons. First, the Reactive Paradigm will make extensive use of reflexive behaviors, to the point that some architectures only call a robot behavior a behavior if it is

S-R. Second, the categorization can help a designer determine what type of behavior to use, leading to insights about the appropriate implementation. Third, the

use of the word “reactive” in ethology is at odds with the way the word is used in robotics. In ethology, reactive behavior means learned behaviors or a skill; in robotics, it connotes a reflexive behavior. If the reader is unaware of these differences, it may be hard to read either the ethological or AI literature without being confused.

3.2.1 Reflexive behaviors

Reflexive types of behaviors are particularly interesting, since they imply no need for any type of cognition: if you sense it, you do it. For a robot, this would be a hardwired response, eliminating computation and guaranteed to be fast. Indeed, many kit or hobby robots work off of reflexes, represented by circuits.

Reflexive behaviors can be further divided into three categories:¹⁰

REFLEXES	1. <i>reflexes</i> : where the response lasts only as long as the stimulus, and the response is proportional to the intensity of the stimulus.
TAXES	2. <i>taxes</i> : where the response is to move to a particular orientation. Baby turtles exhibit <i>tropotaxis</i> ; they are hatched at night and move to the brightest light. Until recently the brightest light would be the ocean reflecting the moon, but the intrusion of man has changed that. Owners of beach front property in Florida now have to turn off their outdoor lights during hatching season to avoid the lights being a source for tropotaxis. Baby turtles hatch at night, hidden from shore birds who normally eat them. It had been a mystery as to how baby turtles knew which way was the ocean when they hatched. The story goes that a volunteer left a flashlight on the sand while setting up an experiment intended to show that the baby turtles used magnetic fields to orient themselves. The magnetic field theory was abandoned after the volunteers noticed the baby turtles heading for the flashlight! Ants exhibit a particular taxis known as <i>chemotaxis</i> ; they follow trails of pheromones.
FIXED-ACTION	3. <i>fixed-action patterns</i> : where the response continues for a longer duration
PATTERNS	than the stimulus. This is helpful for fleeing predators. It is important to keep in mind that a taxis can be any orientation relative to a stimulus, not just moving towards.

The above categories are not mutually exclusive. For example, an animal going over rocks or through a forest with trees to block its view

(fixed-action patterns) in orienting itself to the last sensed location of a food source (taxis) when it loses sight of it.

IDIOTHETIC
ALLOTHETIC

The tight coupling of action and perception can often be quantified by mathematical expressions. An example of this is orienting in angelfish. In order to swim upright, an angelfish uses an internal (*idiothetic*) sense of gravity combined with its vision sense (*allothetic*) to see the external percept of the horizon line of the water to swim upright. If the fish is put in a tank with prisms that make the horizon line appear at an angle, the angelfish will swim cockeyed. On closer inspection, the angle that the angelfish swims at is the vector sum of the vector parallel to gravity with the vector perpendicular to the perceived horizon line! The ability to quantify animal behavior suggests that computer programs can be written which do likewise.

3.3 Coordination and Control of Behaviors

KONRAD LORENZ

Konrad Lorenz and Niko Tinbergen were the founding fathers of ethology.

NIKO TINBERGEN

Each man independently became fascinated not only with individual behaviors of animals, but how animals acquired behaviors and selected or coordinated sets of behaviors. Their work provides some insight into four different ways an animal might acquire and organize behaviors. Lorenz and Tinbergen's work also helps with a computational theory Level 2 understanding of how to make a process out of behaviors.

The four ways to acquire a behavior are:

INNATE

1. to be born with a behavior (*innate*). An example is the feeding behavior in baby arctic terns. Arctic terns, as the name implies, live in the Arctic where the terrain is largely shades of black and white. However, the Arctic tern has a bright reddish beak. When babies are hatched and are hungry, they peck at the beak of their parents. The pecking triggers a regurgitation reflex in the parent, who literally coughs up food for the babies to eat. It turns out that the babies do not recognize their parents, per se. Instead, they are born with a behavior that says: if hungry, peck at the largest red blob you see. Notice that the only red blobs in the field of vision should be the beaks of adult Arctic terns. The largest blob should be the nearest parent (the closer objects are, the bigger they appear). This is a simple, effective, and computationally inexpensive strategy.

SEQUENCE OF INNATE

2. to be born with a *sequence of innate behaviors*. The animal is born with a

A female digger wasp mates with a male, then builds a nest. Once it sees the nest, the female lays eggs. The sequence is logical, but the important point is the role of stimulus in triggering the next step. The nest isn't built until the female mates; that is a change in internal state. The eggs aren't laid until the nest is built; the nest is a visual stimulus releasing the next step. Notice that the wasp doesn't have to "know" or understand the sequence. Each step is triggered by the combination of internal state and the environment. This is very similar to Finite State Machines in computer science programming, and will be discussed later in Ch. 5.

INNATE WITH MEMORY

3. to be born with behaviors that need some initialization (*innate with memory*). An animal can be born with innate behaviors that need customizing based on the situation the animal is born in. An example of this is bees. Bees are born in hives. The location of a hive is something that isn't innate; a baby bee has to learn what its hive looks like and how to navigate to and from it. It is believed that the curious behavior exhibited by baby bees (which is innate) allows them to learn this critical information. A new bee will fly out of the hive for a short distance, then turn around and come back. This will get repeated, with the bee going a bit farther along the straight line each time. After a time, the bee will repeat the behavior but at an angle from the opening to the hive. Eventually, the bee will have circumnavigated the hive. Why? Well, the conjecture is that the bee is learning what the hive looks like from all possible approach angles. Furthermore, the bee can associate a view of the hive with a motor command ("fly left and down") to get the bee to the opening. The behavior of zooming around the hive is innate; what is learned about the appearance of the hive and where the opening is requires memory.

LEARN

4. to *learn* a set of behaviors. Behaviors are not necessarily innate. In mammals and especially primates, babies must spend a great deal of time learning. An example of learned behaviors is hunting in lions. Lion cubs are not born with any hunting behaviors. If they are not taught by their mothers over a period of years, they show no ability to fend for themselves. At first it might seem strange that something as fundamental as hunting for food would be learned, not innate. However, consider the complexity of hunting for food. Hunting is composed of many sub-behaviors, such as searching for food, stalking, chasing, and so on. Hunting may also require teamwork with other members of the pride. It requires great sensitivity to the type of the animal being hunted and the terrain. Imagine trying to write a program to cover all the possibilities!

While the learned behaviors are very complex, they can still be represented by innate releasing mechanisms. It is just that the releasers and actions are learned; the animal creates the program itself.

Note that the number of categories suggests that a roboticist will have a spectrum of choices as to how a robot can acquire one or more behaviors: from being pre-programmed with behaviors (innate) to somehow learning them (learned). It also suggests that behaviors can be innate but require memory. The lesson here is that while S-R types of behaviors are simple to pre-program or hardwire, robot designers certainly shouldn't exclude the use of memory. But as will be seen in Chapter 4, this is a common constraint placed on many robot systems. This is especially true in a popular style of hobby robot building called BEAM robotics (biology, electronics, aesthetics, and mechanics), espoused by Mark Tilden. Numerous BEAM robot web sites guide adherents through construction of circuits which duplicate memory-less innate reflexes and taxes.

INTERNAL STATE

An important lesson that can be extracted from Lorenz and Tinbergen's work is that the internal state and/or motivation of an agent may play a role

MOTIVATION in releasing a behavior. Being hungry is a stimulus, equivalent to the pain introduced by a sharp object in the robot's environment. Another way of looking at it is that motivation serves as a stimulus for behavior. Motivations can stem from survival conditions (like being hungry) or more abstract goals (e.g., need to check the mail). One of the most exciting insights is that behaviors can be sequenced to create complex behaviors. Something as complicated as mating and building a nest can be decomposed into primitives or certainly more simple behaviors. This has an appeal to the software engineering side of robotics.

3.3.1 Innate releasing mechanisms

Lorenz and Tinbergen attempted to clarify their work in how behaviors are coordinated and controlled by giving it a special name *innate releasing mech-*

INNATE RELEASING

MECHANISMS

anisms (IRM). An IRM presupposes that there is a specific stimulus (either internal or external) which releases, or triggers, the stereotypical pattern of

RELEASER

action. The IRM activates the behavior. A *releaser* is a latch or a Boolean

variable that has to be set. One way to think of IRMs is as a process of behaviors. In a computational theory of intelligence using IRMs, the basic black boxes of the process would be behaviors. Recall that behaviors take sensory input and produce motor actions. But IRMs go further and specify when a behavior-

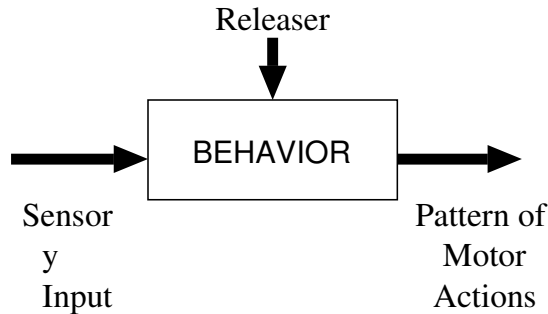


Figure 3.4 Innate Releasing Mechanism as a process with behaviors.

ior gets turned on and off. The releaser acts as a control signal to activate a behavior. If a behavior is not released, it does not respond to sensory inputs and does not produce motor outputs. For example, if a baby arctic tern isn't hungry, it doesn't peck at red, even if there is a red beak nearby.

Another way to think of IRMs is as a simple computer program. Imagine the agent running a C program with a continuous `while` loop. Each execution through the loop would cause the agent to move for one second, then the loop would repeat.

```

enum          Releaser={PRESENT, NOT_PRESENT};
Releaser      predator;
while (TRUE)
{
    predator = sensePredators();
    if (predator == PRESENT)
        flee();
}
  
```

In this example, the agent does only two things: sense the world and then flees if it senses a predator. Only one behavior is possible: `flee`. `flee` is released by the presence of a predator. A predator is of type `Releaser` and has only two possible values: it is either present or it is not. If the agent does not sense the releaser for the behavior, the agent does nothing. There is no “default” behavior.

This example also shows filtering of perception. In the above example, the agent only looks for predators with a dedicated detection function,

3.3 Coordination and Control of Behaviors
sense-Predators(y). The dedicated predator detection function⁸²
could be a spe- cialized sense (e.g., retina is sensitive to the frequency of
motions associated

with predator movement) or a group of neurons which do the equivalent of a computer algorithm.

COMPOUND RELEASERS

Another important point about IRMs is that the releaser can be a *compound of releasers*. Furthermore, the releaser can be a combination of either external (from the environment) or internal (motivation). If the releaser in the compound isn't satisfied, the behavior isn't triggered. The pseudo-code below shows a compound releaser.

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        food;
while (TRUE)
{
    food = senseFood();
    hungry = checkState();
    if (food == PRESENT && hungry==PRESENT)
        feed();
}
```

IMPLICIT CHAINING

The next example below shows what happens in a sequence of behaviors, where the agent eats, then nurses its young, then sleeps, and repeats the sequence. The behaviors are implicitly chained together by their releasers. Once the initial releaser is encountered, the first behavior occurs. It executes for one second (one "movement" interval), then control passes to the next statement. If the behavior isn't finished, the releasers remain unchanged and no other behavior is triggered. The program then loops to the top and the original behavior executes again. When the original behavior has completed, the internal state of the animal may have changed or the state of the environment may have been changed as a result of the action. When the motivation and environment match the stimulus for the releaser, the second behavior is triggered, and so on.

```
enum            Releaser={PRESENT, NOT_PRESENT};
Releaser        food, hungry, nursed;
while (TRUE) {
    food = sense();
    hungry = checkStateHunger();
    child = checkStateChild();
    if (hungry==PRESENT)
        searchForFood(); //sets food = PRESENT when done
    if (hungry==PRESENT && food==PRESENT)
        feed(); // sets hungry = NOT_PRESENT when done
```

```

    if (hungry== NOT_PRESENT && parent==PRESENT)
        nurse(); // set nursed = PRESENT when done
    if (nursed ==PRESENT)
        sleep();
}

```

The example also reinforces the nature of behaviors. If the agent sleeps and wakes up, but isn't hungry, what will it do? According to the releasers created above, the agent will just sit there until it gets hungry.

In the previous example, the agent's behaviors allowed it to feed and enable the survival of its young, but the set of behaviors did not include fleeing or fighting predators. Fleeing from predators could be added to the program as follows:

```

enum          Releaser={PRESENT, NOT_PRESENT};
Releaser      food, hungry, nursed, predator;
while (TRUE) {
    predator = sensePredator();
    if (predator==PRESENT)
        flee();
    food = senseFood();
    hungry = checkStateHunger();
    parent = checkStateParent();
    if (hungry==PRESENT)
        searchForFood();
    if (hungry==PRESENT && food==PRESENT)
        feed();
    if (hungry== NOT_PRESENT && parent==PRESENT)
        nurse();
    if (nursed ==PRESENT)
        sleep();
}

```

Notice that this arrangement allowed the agent to flee the predator regardless of where it was in the sequence of feeding, nursing, and sleeping because predator is checked for first. But fleeing is temporary, because it did not change the agent's internal state (except possibly to make it more hungry which will show up on the next iteration). The code may cause the agent to flee for one second, then feed for one second.

One way around this is to *inhibit*, or turn off, any other behavior until fleeing is completed. This could be done with an *if-else* statement:

```

while (TRUE) {
    predator = sensePredator();
    if (predator==PRESENT)
        flee();
    else {
        food = senseFood();
        hungry = checkStateHunger();
        ...
    }
}

```

The addition of the `if-else` prevents other, less important behaviors from executing. It doesn't solve the problem with the predator releaser disappearing as the agent runs away. If the agent turns and the predator is out of view (say, behind the agent), the value of `predator` will go to `NOT_PRESENT` in the next update. The agent will go back to foraging, feeding, nursing, or sleeping. Fleeing should be a fixed-pattern action behavior which persists for some period of time, `T`. The fixed-pattern action effect can be accomplished with:

```

#define T LONG_TIME
while (TRUE) {
    predator = sensePredator();
    if (predator==PRESENT)
        for(time = T; time > 0; time--)
            flee();
    else {
        food = senseFood();
        ...
    }
}

```

The C code examples were implemented as an implicit sequence, where the order of execution depended on the value of the releasers. An implementation of the same behaviors with an explicit sequence would be:

```

Releaser          food, hungry, nursed, predator;
while (TRUE) {

    predator = sensePredator();
    if (predator==PRESENT)

```

```

        flee();
        food = senseFood();
        hungry = checkStateHunger();
        parent = checkStateParent();
        if (hungry==PRESENT)
            searchForFood();
        feed();
        nurse();
        sleep();
    }

```

The explicit sequence at first may be more appealing. It is less cluttered and the compound releasers are hidden. But this implementation is not equivalent. It assumes that instead of the loop executing every second and the behaviors acting incrementally, each behavior takes control and runs to completion. Note that the agent cannot react to a predator until it has finished the sequence of behaviors. Calls to the fleeing behavior could be inserted between each behavior or fleeing could be processed on an interrupt basis. But every “fix” makes the program less general purpose and harder to add and maintain.

The main point here is: *simple behaviors operating independently can lead to what an outside observer would view as a complex sequence of actions.*

3.3.2 Concurrent behaviors

An important point from the examples with the IRMs is that behaviors can, and often do, execute concurrently and independently. What appears to be a fixed sequence may be the result of a normal series of events. However, some behaviors may violate or ignore the implicit sequence when the environment presents conflicting stimuli. In the case of the parent agent, fleeing a predator was mutually exclusive of the feeding, nursing, and sleeping behaviors.

Interesting things can happen if two (or more) behaviors are released that usually are not executed at the same time. It appears that the strange interactions fall into the following categories:

EQUILIBRIUM

Equilibrium (the behaviors seem to balance each other out): Consider feeding versus fleeing in a squirrel when the food is just close enough to a person on a park bench. A squirrel will often appear to be visibly undecided as to whether to go for the food or to stay away.

DOMINANCE	<i>Dominance of one (winner take all):</i> you're hungry and sleepy. You do one or the other, not both simultaneously.
CANCELLATION	<i>Cancellation (the behaviors cancel each other out):</i> Male sticklebacks (fish) when their territories overlap get caught between the need to defend their territory and to attack the other fish. So the males make another nest! Apparently the stimuli cancels out, leaving only the stimulus normally associated with nest building.

Unfortunately, it doesn't appear to be well understood when these different mechanisms for conflicting behaviors are employed. Clearly, there's no one method. But it does emphasize that a roboticist who works with behaviors should pay close attention to how the behaviors will interact. This will give rise to the differences in architectures in the Reactive and Hybrid Paradigms, discussed in later chapters.

3.4 Perception in Behaviors

While Lorenz and Tinbergen's work provides some insights into behaviors, it's clear that behaviors depend on perception. Ulrich Neisser, who literally created the term "cognitive psychology" in his book, *Cognition and Reality*, argued that perception cannot be separated from action.¹⁰⁹ As will be seen in this section, J.J. Gibson, a very controversial cognitive psychologist, spent his career advocating an ecological approach to perception. The ecological approach is the opposite of the top-down, model-based reasoning about the environment approach favored by psychologists, including Neisser. Interestingly enough, Neisser took a position at Cornell where J.J. Gibson was, and they became close colleagues. Since then, Neisser has spent significant time and thought trying to reconcile the two views based on studies; this has led to his identification of two perceptual systems.

3.4.1 Action-perception cycle

ACTION-PERCEPTION	The <i>action-perception cycle</i> illustrates that perception is fundamental to any intelligent agent. A simple interpretation of the cycle is: When an agent acts, it interacts with its environment because it is situated in that environment; it is an integral part of the environment. So as it acts, it changes things or

3.4 Perception in Behaviors ⁸⁴
how it perceives it (e.g., move to a new viewpoint, trigger a rock slide, etc.). Therefore the agent's perception of the world is modified. This new perception is

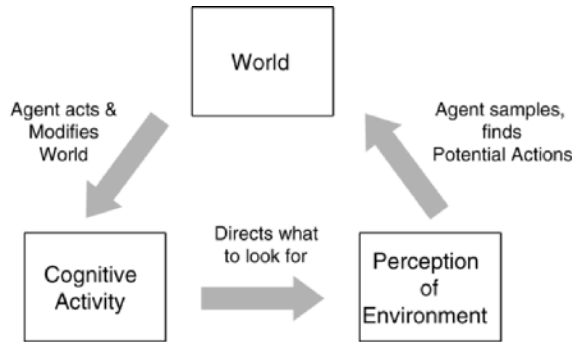


Figure 3.5 Action-Perception Cycle. ⁷

then used for a variety of functions, including both cognitive activities like planning for what to do next as well as reacting. The term cognitive activity includes the concepts of feedback and feedforward control, where the agent senses an error in what it attempted to do and what actually happened. An equally basic cognitive activity is determining what to sense next. That activity can be something as straightforward as activating processes to look for releasers, or as complex as looking for a particular face in a crowd.

Regardless of whether there is an explicit conscious processing of the senses or the extraction of a stimulus or releaser, the agent is now directed in terms of what it is going to perceive on the next update(s). This is a type of selective attention or focus-of-attention. As it perceives, the agent perceptually samples the world. If the agent actually acts in a way to gather more perception before continuing with its primary action, then that is sometimes referred to as active perception. Part of the sampling process is to determine the potential for action. Lorenz and Tinbergen might think of this as the agent having a set of releasers for a task, and now is observing whether they are present in the world. If the perception supports an action, the agent acts. The action modifies the environment, but it also modifies the agent's assessment of the situation. In the simplest case, this could be an error signal to be used for control or a more abstract difference such as at the level of those used in STRIPS/MEA.

In some regards, the action-perception cycle appears to bear a superficial resemblance to the Hierarchical Paradigm of SENSE, PLAN, ACT. However, note that 1) there is no box which contains ACT, and 2)

the cycle does not require the equivalent of planning to occur at each update. Action is implicit

in an agent; the interesting aspect of the cycle is where perception and cognition come in. The agent may have to act to acquire more perception or to accomplish a task. Also, the agent may or may not need to “plan” an action on each update.

3.4.2 Two functions of perception

Perception in behavior serves two functions. First, as we saw with IRMs, it serves to *release* a behavior. However, releasing a behavior isn’t necessarily the same as the second function: perceiving the information needed to accomplish the behavior. For example, consider an animal in a forest fire. The fire activates the fleeing. But the fleeing behavior needs to extract information about open spaces to run through obstacles in order to *guide* the behavior. A frightened deer might bolt right past a hunter without apparently noticing.

In both roles as a releaser and as a guide for behavior, perception filters the incoming stimulus for the task at hand. This is often referred to as action-oriented perception by roboticists, when they wish to distinguish their perceptual approach from the more hierarchical global models style of perception. Many animals have evolved specialized detectors which simplify perception for their behaviors. Some frogs which sit in water all day with just half their eyes poking up have a split retina: the lower half is good for seeing in water, the upper half in air.

3.4.3 Gibson: Ecological approach

The central tenet of Gibson’s approach is that “... the world is its own best representation.” Gibson’s work is especially interesting because it complements the role of perception in IRM and is consistent with the action-perception cycle. Gibson postulated (and proved) the existence of affordances.

Affordances are perceivable potentialities of the environment for an action. For example, to a baby arctic tern, the color red is perceivable and represents the potential for feeding. So an affordance can be a more formal way of defining the external stimulus in IRM. But like IRMs, an affordance is only a potential— it doesn’t count until all the other conditions are satisfied (the baby tern is hungry). An affordance can also be the percept that guides the behavior. The presence of red to a hungry baby arctic tern releases the feeding behavior. But the feeding behavior consists of

pecking at the red object. So in this case, red is also the percept being used to guide the action, as well as release it.

3.4 Perception in Behaviors

Gibson referred to his work as an “ecological approach” because he believed that perception evolved to support actions, and that it is silly to try to discuss perception independently of an agent’s environment, and its survival behaviors. For example, a certain species of bees prefers one special type of poppy. But for a long time, the scientists couldn’t figure out how the bees recognized that type of poppy because as color goes, it was indistinguishable from another type of poppy that grows in the same area. Smell? Magnetism? Neither. They looked at the poppy under UV and IR light. In the non-visible bands that type of poppy stood out from other poppy species. And indeed, the scientists were able to locate retinal components sensitive to that bandwidth. The bee and poppy had co-evolved, where the poppy’s color evolved to a unique bandwidth while at the same time the bee’s retina was becoming specialized at detecting that color. With a retina “tuned” for the poppy, the bee didn’t have to do any reasoning about whether there was a poppy in view, and, if so, was it the right species of poppy. If that color was present, the poppy was there.

Fishermen have exploited affordances since the beginning of time. A fishing lure attempts to emphasize those aspects of a fish’s desired food, presenting the strongest stimulus possible: if the fish is hungry, the stimulus of the lure will trigger feeding. As seen in Fig. 3.6, fishing lures often look to a human almost nothing like the bait they imitate.

DIRECT PERCEPTION

What makes Gibson so interesting to roboticists is that an affordance is directly perceivable. Direct perception means that the sensing process doesn’t require memory, inference, or interpretation. This means minimal computation, which usually translates to very rapid execution times (near instantaneous) on a computer or robot.

But can an agent actually perceive anything meaningful without some memory, inference, or interpretation? Well, certainly baby arctic terns don’t need memory or inference to get food from a parent. And they’re definitely not interpreting red in the sense of: “oh, there’s a red blob. It’s a small oval, which is the right shape for Mom, but that other one is a square, so it must be a graduate ethology student trying to trick me.” For baby arctic terns, it’s simply: red = food, bigger red = better.

Does this work for humans? Consider walking down the hall and somebody throws something at you. You will most likely duck. You also probably ducked without recognizing the object, although later you may determine it was only a foam ball. The response happens too fast for any reasoning: “Oh look, something is moving towards me. It must be a ball.

Balls are usually hard. I should duck.” Instead, you probably used a phenomena so basic that

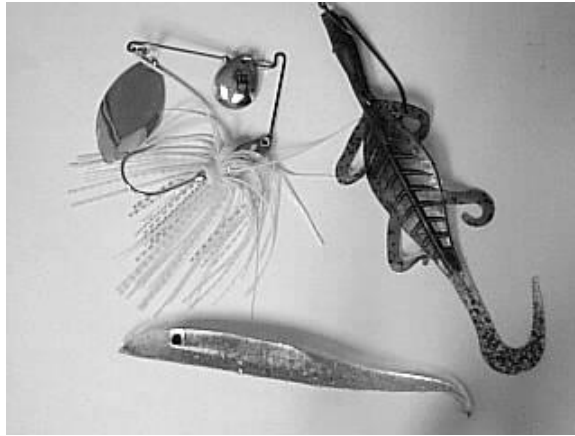


Figure 3.6 A collection of artificial bait, possibly the first example of humans exploiting affordances. Notice that the lures exaggerate one or more attributes of what a fish might eat.

OPTIC FLOW

TIME TO CONTACT

you haven't noticed it, called *optic flow*. Optic flow is a neural mechanism for determining motion. Animals can determine time to contact quite easily with it. You probably are somewhat familiar with optic flow from driving in a car. When driving or riding in a car, objects in front seem to be in clear focus but the side of the road is a little blurry from the speed. The point in space that the car is moving to is the focus of expansion. From that point outward, there is a blurring effect. The more blurring on the sides, the faster the car is going. (They use this all the time in science fiction movies to simulate faster-than-light travel.) That pattern of blurring is known as a flow field (because it can be represented by vectors, like a gravitational or magnetic field). It is straightforward, neurally, to extract the time to contact, represented in the cognitive literature by .

Gannets and pole vaulters both use optic flow to make last-minute, precise movements as reflexes. Gannets are large birds which dive from high altitudes after fish. Because the birds dive from hundreds of feet up in the air, they have to use their wings as control surfaces to direct their dive at the targeted fish. But they are plummeting so fast that if they hit the water with their wings open, the hollow bones will shatter. Gannets fold their wings just before hitting the water. Optic flow allows the time to

contact, , to be a
stim- ulus: when
the time to contact
dwindles below a
threshold, fold
those wings!

3.4 Perception in Behaviors

92

Pole vaulters also make minute adjustments in where they plant their pole as they approach the hurdle. This is quite challenging given that the vaulter is running at top speed. It appears that pole vaulters use optic flow rather than reason (slowly) about where the best place is for the pole. (Pole vaulting isn't the only instance where humans use optic flow, just one that has been well-documented.)

In most applications, a fast computer program can extract an affordance. However, this is not the case (so far) with optic flow. Neural mechanisms in the retina have evolved to make the computation very rapid. It turns out that computer vision researchers have been struggling for years to duplicate the generation of an optical flow field from a camera image. Only recently have we seen any algorithms which ran in real-time on regular computers.⁴⁸ The point is that affordances and specialized detectors can be quite challenging to duplicate in computers.

Affordances are not limited to vision. A common affordance is knowing when a container is almost filled to the top. Think about filling a jug with water or the fuel tank of a car. Without being able to see the cavity, a person knows when the tank is almost filled by the change in sound. That change in sound is directly perceivable; the person doesn't need to know anything about the size or shape of the volume being filled or even what the liquid is. One particularly fascinating application of affordances to robotics, which also serves to illustrate what an affordance is, is the research of Louise Stark and Kevin Bowyer.¹³⁵ A seemingly unsurmountable problem in computer vision has been to

STRUCTURAL MODELS

have a computer recognize an object from a picture. Literally, the computer should say, "that's a chair" if the picture is of a chair.

The traditional way of approaching the problem has been to use *structural models*. A structural model attempts to describe an object in terms of physical components: "A chair has four legs, a seat, and a back." But not all chairs fit the same structural model. A typing chair has only one leg, with supports at the bottom. Hanging baskets don't have legs at all. A bench seat doesn't have a back. So clearly the structural approach has problems: instead of one structural representation, the computer has to have access to many different models. Structural models also lack flexibility. If the robot is presented with a new kind of chair (say someone has designed a chair to look like your toilet or an upside down trash can), the robot would not be able to recognize it without someone explicitly constructing another structural model.

Stark and Bowyer explored an alternative to the structural approach

called GRUFF. GRUFF identifies chairs by *function* rather than form. Under Gibsonian perception, a chair should be a chair because it affords sitting, or serves

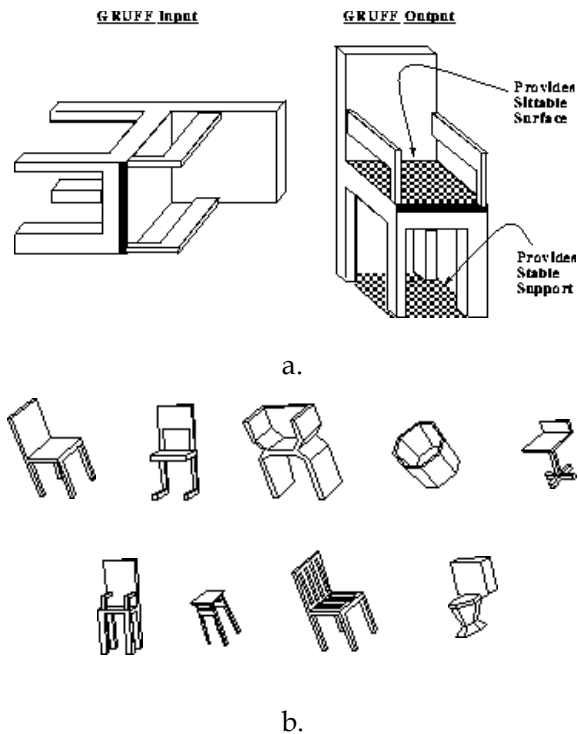


Figure 3.7 The GRUFF system: a.) input, and b.) different types of chairs recognized by GRUFF. (Figures courtesy of Louise Stark.)

the function of sittability. And that affordance of sittability should be some- thing that can be extracted from an image:

Without memory (the agent doesn't need to memorize all the chairs in the world).

Without inference (the robot doesn't need to reason: "if it has 4 legs, and a seat and a back, then it's a chair; we're in an area which should have lots of chairs, so this makes it more likely it's a chair").

Without an interpretation of the image (the robot doesn't need to reason: "there's an arm rest, and a cushion, ... "). A computer should just be able to look at a picture and say if something in that picture is sittable or not.

Stark and Bowyer represented sittability as a reasonably level and continuous surface which is at least the size of a person's butt and at about the height of their knees. (Everything else like seat backs just serve to specify the kind of chair.) Stark and Bowyer wrote a computer program which accepted CAD/CAM drawings from students who tried to come up with non-intuitive things that could serve as chairs (like toilets, hanging basket chairs, trash cans). The computer program was able to correctly identify sittable surfaces that even the students missed.

It should be noted that Stark and Bowyer are hesitant to make claims about what this says about Gibsonian perception. The computer vision algorithm can be accused of some inference and interpretation ("that's the seat, that's the right height"). But on the other hand, that level of inference and interpretation is significantly different than that involved in trying to determine the structure of the legs, etc. And the relationship between seat size and height could be represented in a special neural net that could be *released* whenever the robot or animal got tired and wanted to sit down. The robot would start noticing that it could sit on a ledge or a big rock if a chair or bench wasn't around.

3.4.4 Neisser: Two perceptual systems

At this point, the idea of affordances should seem reasonable. A chair is a chair because it affords sittability. But what happens when someone sits in *your* chair? It would appear that humans have some mechanism for recognizing specific instances of objects. Recognition definitely involves memory ("my car is a blue Ford Explorer and I parked it in slot 56 this morning"). Other tasks, like the kind of sleuthing Sherlock Holmes does, may require inference and interpretation. (Imagine trying to duplicate Sherlock Holmes in a computer. It's quite different than mimicking a hungry baby arctic tern.) So while affordances certainly are a powerful way of describing perception in animals, it is clearly not the only way animals perceive. Neisser postulated that there are two perceptual systems in the brain (and cites neurophysiological data):¹¹⁰

DIRECT PERCEPTION

1. *direct perception*. This is the "Gibsonian," or ecological, track of the brain, and consists of structures low in the brain which evolved earlier on and accounts for affordances.

RECOGNITION

2. *recognition*. This is more recent perceptual track in the brain, which ties in

with the
problem
solving and
other cognitive
activities. This
part accounts

3.4 Perception in Behaviors

97

for the use of internal models to distinguish “your coffee cup” from “my coffee cup.” This is where top-down, model-based perception occurs.

On a more practical note, Neisser’s dichotomy suggests that the first decision in designing a behavior is to determine whether a behavior can be accomplished with an affordance or requires recognition. If it can be accomplished with an affordance, then there may be a simple and straightforward way to program it in a robot; otherwise, we will most certainly have to employ a more sophisticated (and slower) perceptual algorithm.

3.5 Schema Theory

Schema theory provides a helpful way of casting some of the insights from above into an object-oriented programming format.⁶ Psychologists have used schema theory since the early 1900’s. It was first brought to the serious attention of AI roboticists by Michael Arbib while at the University of Massachusetts, and later used extensively by Arkin and Murphy for mobile robotics, Lyons and Iberall for manipulation,⁷⁵ and Draper et al. for vision.⁴⁶

SCHEMA

SCHEMA CLASS

Schemas were conceived of by psychologists as a way of expressing the basic unit of activity. A *schema* consists both of the knowledge of how to act and/or perceive (knowledge, data structures, models) as well as the computational process by which it is used to accomplish the activity (the algorithm). The idea of a schema maps nicely onto a class in object-oriented programming (OOP). A *schema class* in C++ or Java would contain both data (knowledge, models, releasers) and methods (algorithms for perceiving and acting), as shown below.

Schema :

Data
Methods

SCHEMA INSTANTIATION (SI)

A schema is a generic template for doing some activity, like riding a bi-cycle. It is a template because a person can ride different bicycles without starting the learning process all over. Since a schema is parameterized like a class, parameters (type of bicycle, height of the bicycle seat, position of the handlebars) can be supplied to the object at the time of instantiation (when an object is created from the class). As with object-oriented programming, the creation of a specific schema is called a *schema instantiation (SI)*.

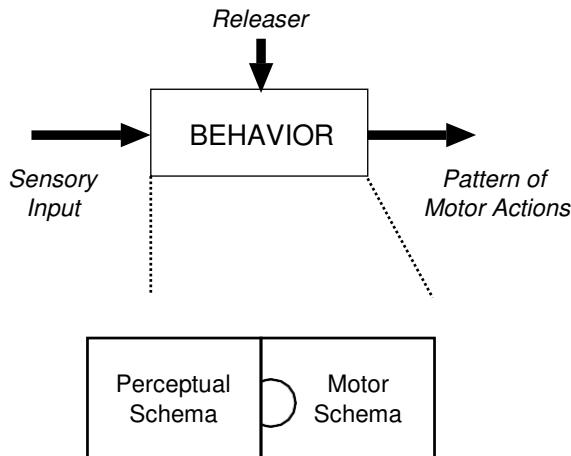


Figure 3.8 Behaviors decomposed into perceptual and motor schemas.

The schema instantiation is the object which is constructed with whatever parameters are needed to tailor it to the situation. For example, there could be a `move_to_food` schema where the agent always heads in a straight line to the food. Notice that the “always heads in a straight line” is a template of activity, and a reusable algorithm for motion control. However, it is just a method; until the `move_to_food` schema is instantiated, there is no specific goal to head for, e.g., the candy bar on the table. The same schema could be instantiated for moving to a sandwich.

3.5.1 Behaviors and schema theory

In the Arbibian application of schema theory towards a computational theory of intelligence, a *behavior* is a schema which is composed of a motor

COMPOSITION OF A
BEHAVIOR
MOTOR SCHEMA
PERCEPTUAL SCHEMA

schema and a perceptual schema. The *motor schema* represents the template for the physical activity, the *perceptual schema* embodies the sensing. The motor schema and perceptual schema are like pieces of a puzzle; both pieces must be together in place before there is a behavior. This idea is shown below in Fig. 3.8.

Essentially, the perceptual and motor schema concept fits in with ethology and cognitive psychology as follows:

3.5 *Schema Theory* 94
A behavior takes sensory inputs and produces motor actions as an output.

A behavior can be represented as a schema, which is essentially an object-oriented programming construct.

A behavior is activated by releasers.

The transformation of sensory inputs into motor action outputs can be divided into two sub-processes: a perceptual schema and a motor schema.

In OOP terms, the motor schema and perceptual schema classes are derived from the schema class. A primitive behavior just has one motor and one perceptual schema.

Behavior::Schema

Data	
Methods	perceptual_schema() motor_schema()

Recall from IRMs, more sophisticated behaviors may be constructed by sequencing behaviors. In the case of a sequence of behaviors, the overall behavior could be represented in one of two ways. One way is to consider the behavior to be composed of several primitive behaviors, with the releasing logic to serve as the knowledge as to when to activate each primitive behaviors. This is probably the easiest way to express a “meta” behavior.

A meta-behavior composed of three behaviors can be thought of as:

Behavior::Schema

Data	releaser1 releaser2 releaser3 IRM logic()
Methods	behavior1() behavior2() behavior3()

However, in more advanced applications, the agent may have a choice of either perceptual or motor schemas to tailor its behavior. For example, a person usually uses vision (the default perceptual schema) to navigate out of a room (motor schema). But if the power is off, the person can use touch (an alternate perceptual schema) to feel her way out of a dark room. In this case, the schema-specific knowledge is knowing which

to use for different environmental conditions. Schema theory is expressive enough to represent basic concepts like IRMs, plus it supports building new behaviors out of primitive components. This will be discussed in more detail in later chapters.

This alternative way of creating a behavior by choosing between alternative perceptual and motor schemas can be thought of as:

Behavior::Schema

Data	environmental state
Methods	choose_PS(environmental_state) perceptual_schema_1() perceptual_schema_2() motor_schema()

Arbib and colleagues did work constructing computer models of visually guided behaviors in frogs and toads. They used schema theory to represent the toad's behavior in computational terms, and called their model *rana computatrix* (rana is the classification for toads and frogs). The model explained Ingle's observations as to what occasionally happens when a toad sees two flies at once.³³ Toads and frogs can be characterized as responding visually to either small, moving objects and large, moving objects. Small, moving objects release the feeding behavior, where the toad orients itself towards the object (taxis) and then snaps at it. (If the object turns out not to be a fly, the toad can spit it out.) Large moving objects release the fleeing behavior, causing the toad to hop away. The feeding behavior can be modeled as a behavioral schema, or template, shown in Fig. 3.9.

When the toad sees a fly, an instance of the behavior is instantiated; the toad turns toward that object and snaps at it. Arbib's group went one level further on the computational theory.⁷ They implemented the taxis

behavior as a vector field: *rana computatrix* would literally feel an attractive force along the direction of the fly. This direction and intensity (magnitude) was represented as a vector. The direction indicated where rana had to turn and the magnitude indicated the strength of snapping.

This is shown in Fig. 3.10. What is particularly interesting is that the *rana computatrix* program predicts what Ingle saw in real toads and frogs when they are presented with two flies simultaneously. In this case, each fly releases a separate instance of the feeding behavior. Each behavior produces the vector that the toad needs to turn to in order to snap at that

fly, without
knowing that the
other be-

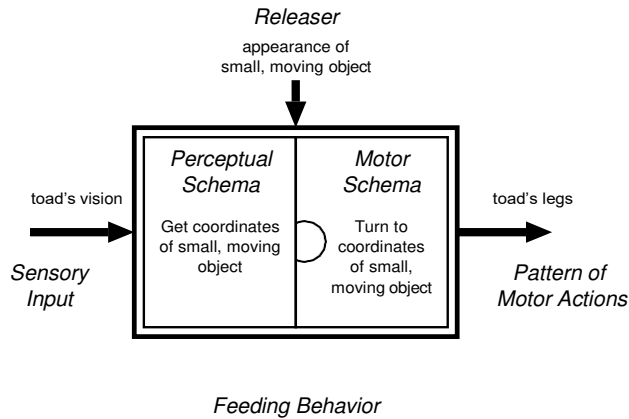


Figure 3.9 Toad's feeding behavior represented as a behavior with schema theory.

havior exists. According to the vector field implementation of the schema model, the toad now receives two vectors, instead of one. What to do? Well, rana computatrix summed the two vectors, resulting in a third vector in between the original two! The toad snaps at neither fly, but in the middle. The unexpected interaction of the two independent instances probably isn't that much of a disadvantage for a toad, because if there are two flies in such close proximity, eventually one of them will come back into range.

This example illustrates many important lessons for robotics. First, it validates the idea of a computational theory, where functionality in an animal and a computer can be equivalent. The concept of behaviors is Level 1 of the computational theory, schema theory (especially the perceptual and motor schemas) expresses Level 2, and Level 3 is the vector field implementation of the motor action. It shows the property of emergent behavior, where the agent appears to do something fairly complex, but is really just the result of interaction between simple modules. The example also shows how behaviors correspond to object-orienting programming principles.

Another desirable aspect of schema theory is that it supports reflex behaviors. Recall that in reflex behaviors, the strength of the response is proportional to the strength of the stimulus. In schema theory, the perceptual schema is permitted to pass both the percept and a gain to the

3.5 *Schema Theory*. The motor schema can use the gain to compute a magnitude on the output action. This is an example of how a particular schema can be tailored for a behavior.

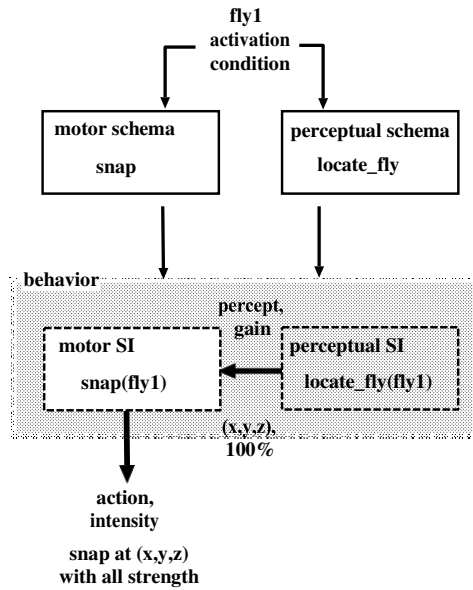


Figure 3.10 Schema theory of a frog snapping at a fly.

Schema theory does not specify how the output from concurrent behaviors is combined; that is a Level 3, or implementation, issue. Previous examples in this chapter have shown that in some circumstances the output is combined or summed, in others the behaviors would normally occur in a sequence and not overlap, and sometimes there would be a winner-take-all effect. The winner-take-all effect is a type of inhibition, where one behavior inhibits the instantiation of another behavior.

INHIBITION

Arbib and colleagues also modeled an instance of *inhibition* in frogs and toads.⁷ Returning to the example of feeding and fleeing, one possible way to model this behavior is with two behaviors. The *feeding* behavior would consist of a motor schema for moving toward an object, with a perceptual schema for finding small, moving objects. The *fleeing* behavior would be similar only with a motor schema for moving away from the perception of large moving objects. Lesion studies with frogs showed something different. The *feeding* behavior actually consists of moving toward *any* moving object. So the perceptual schema is more general than anticipated. The frog would try to eat anything, including

predators. The
perceptual
schema in the
fleeing
behavior detects
large moving
objects. It flees
from them, but

3.5 Schema Theory

102

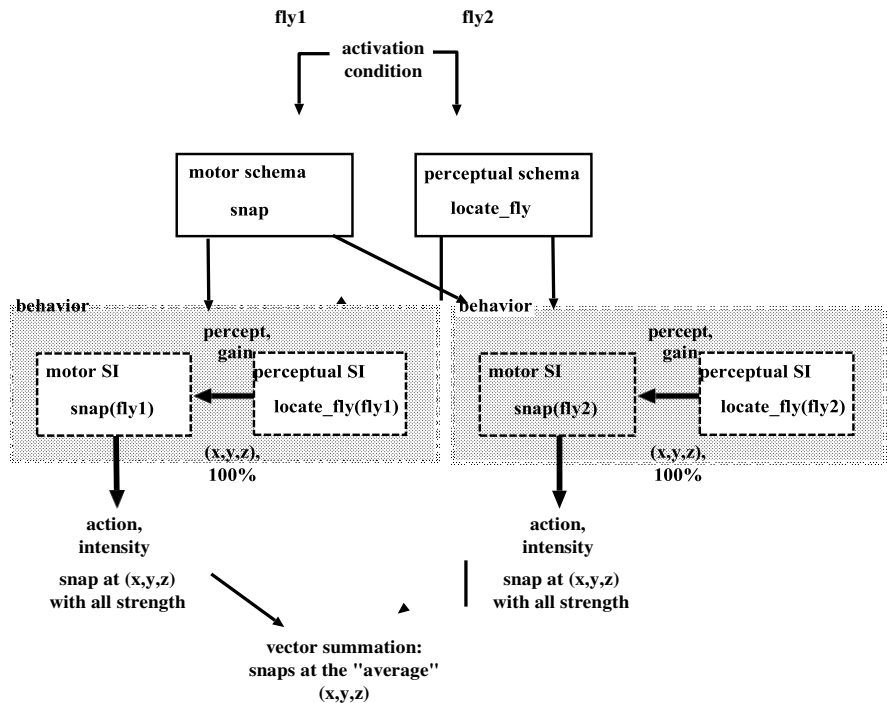


Figure 3.11 Schema theory of a frog snapping at a fly when presented with two flies equidistant.

it also inhibits the perceptual schema for feeding. As a result, the inhibition keeps the frog from trying to both flee from predators and eat them.

36 Principles and Issues in Transferring Insights to Robots

To summarize, some general principles of natural intelligence which may be useful in programming robots:

PRINCIPLES FOR
PROGRAMMING

Programs should decompose complex actions into independent behaviors, which tightly couple sensing and acting. Behaviors are inherently parallel and distributed.

In order to simplify control and coordination of behaviors, an agent

should rely on
straightforward,
boolean
activation
mechanisms
(e.g. IRM).

3.5 Schema Theory

104

In order to simplify sensing, perception should filter sensing and consider only what is relevant to the behavior (action-oriented perception).

Direct perception (affordances) reduces the computational complexity of sensing, and permits actions to occur without memory, inference, or interpretation.

Behaviors are independent, but the output from one 1) may be combined with another to produce a resultant output, or 2) may serve to inhibit another (competing-cooperating).

UNRESOLVED ISSUES

Unfortunately, studying natural intelligence does not give a complete picture of how intelligence works. In particular there are several unresolved issues:

How to resolve conflicts between concurrent behaviors? Robots will be required to perform concurrent tasks; for example, a rescue robot sent in to evacuate a building will have to navigate hallways while looking for rooms to examine for people, as well as look for signs of a spreading fire. Should the designer specify dominant behaviors? Combine? Let conflicting behaviors cancel and have alternative behavior triggered? Indeed, one of the biggest divisions in robot architectures is how they handle concurrent behaviors.

When are explicit knowledge representations and memory necessary? Direct perception is wonderful in theory, but can a designer be sure that an affordance has not been missed?

How to set up and/or learn new sequences of behaviors? Learning appears to be a fundamental component of generating complex behaviors in advanced animals. However, the ethological and cognitive literature is unsure of the mechanisms for learning.

It is also important to remember that natural intelligence does not map perfectly onto the needs and realities of programming robots. One major advantage that animal intelligence has over robotic intelligence is evolution. Animals evolved in a way that leads to survival of the species. But robots are expensive and only a small number are built at any given time. Therefore, individual robots must “survive,” not species. This puts tremendous pressure on robot designers to get a design right the first time. The lack of evolutionary pressures over long periods of time

makes robots extremely vulnerable to design errors introduced by a poor understanding of the robot's ecology.

3.5 *Schema Theory*

Ch. 5 will provide a case study of a robot which was programmed to follow white lines in a path-following competition by using the affordance of white. It was distracted off course by the white shoes of a judge. Fortunately that design flaw was compensated for when the robot got back on course by reacting to a row of white dandelions in seed.

Robots introduce other challenges not so critical in animals. One of the most problematic attributes of the Reactive Paradigm, Ch. 4, is that roboticists have no real mechanism for completely predicting emergent behaviors. Since a psychologist can't predict with perfect certainty what a human will do under a stressful situation, it seems reasonable that a roboticist using principles of human intelligence wouldn't be able to predict what a robot would do either. However, robotics end-users (military, NASA, nuclear industry) have been reluctant to accept robots without a guarantee of what it will do in critical situations.

3.7 Summary

BEHAVIOR

A behavior is the fundamental element of biological intelligence, and will serve as the fundamental component of intelligence in most robot systems. A *behavior* is defined as a mapping of sensory inputs to a pattern of motor actions which then are used to achieve a task. Innate Releasing Mechanisms are one model of how intelligence is organized. IRMs model intelligence at Level 2 of a computational theory, describing the process but not the implementation. In IRM, releasers activate a behavior. A releaser may be either an internal state (motivation) and/or an environmental stimulus. Unfortunately, IRMs do not make the interactions between concurrent, or potentially concurrent, behaviors easy to identify or diagram.

Perception in behaviors serves two roles, either as a releaser for a behavior or as the percept which guides the behavior. The same percept can be used both as a releaser and a guide; for example, a fish can respond to a lure and follow it. In addition to the way in which perception is used, there appear to be two pathways for processing perception. The direct perception pathway uses affordances: perceivable potentialities for action inherent in the environment. Affordances are particularly attractive to roboticists because they can be extracted without inference, memory, or intermediate representations. The recognition pathway makes use of memory and global representations to identify and label

specific things in
the world.

3.5 Schema Theory

10:

Important principles
which can be
extracted from
natural intelligence
are:

Agents programs should decompose complex actions into independent behaviors (or objects), which tightly couple sensing and acting. Behaviors are inherently parallel and distributed.

In order to simplify control and coordination of behaviors, an agent should rely on straightforward, boolean activation mechanisms (e.g. IRM).

In order to simplify sensing, perception should filter sensing and consider only what is relevant to the behavior (action-oriented perception).

Direct perception (affordances) reduces the computational complexity of sensing, and permits actions to occur without memory, inference, or interpretation.

Behaviors are independent, but the output from one 1) may be combined with another to produce a resultant output, or 2) may serve to inhibit another (competing-cooperating).

Schema theory is an object-oriented way of representing and thinking about behaviors. The important attributes of schema theory for behaviors are:

Schema theory is used to represent behaviors in both animals and computers, and is sufficient to describe intelligence at the first two levels of a computational theory.

A behavioral schema is composed of at least one motor schema and at least one perceptual schema, plus local, behavior-specific knowledge about how to coordinate multiple component schemas.

More than one behavior schema can be instantiated at a time, but the schemas act independently.

A behavior schema can have multiple instantiations which act independently, and are combined.

Behaviors or schemas can be combined, sequenced, or inhibit one another.

3

The Reactive Paradigm

Chapter Objectives:

Define what the reactive paradigm is in terms of i) the three primitives SENSE, PLAN, and ACT, and ii) sensing organization.

List the characteristics of a reactive robotic system, and discuss the connotations surrounding the reactive paradigm.

Describe the two dominant methods for combining behaviors in a reactive architecture: *subsumption* and *potential field summation*.

Evaluate subsumption and potential fields architectures in terms of: *support for modularity, niche targetability, ease of portability to other domains, robustness*.

Be able to program a behavior using a potential field methodology.

Be able to construct a new potential field from primitive potential fields, and sum potential fields to generate an emergent behavior.

4.1 Overview

This chapter will concentrate on an overview of the reactive paradigm and two representative architectures. The Reactive Paradigm emerged in the late 1980's. The Reactive Paradigm is important to study for at least two reasons. First, robotic systems in limited task domains are still being constructed using reactive architectures. Second, the Reactive Paradigm will form the basis for the Hybrid Reactive-Deliberative Paradigm; everything covered here will be used (and expanded on) by the systems in Ch. 7.

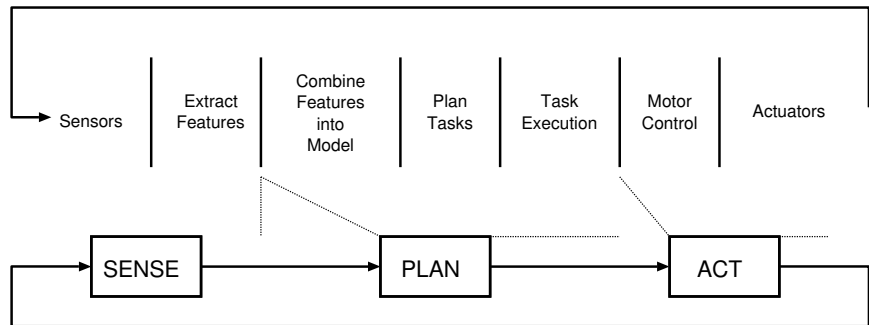


Figure 4.1 Horizontal decomposition of tasks into the S,P,A organization of the Hierarchical Paradigm.

The Reactive Paradigm grew out of dissatisfaction with the hierarchical paradigm and with an influx of ideas from ethology. Although various reactive systems may or may not strictly adhere to principles of biological intelligence, they generally mimic some aspect of biology. The dissatisfaction with the Hierarchical Paradigm was best summarized by Rodney Brooks,²⁷ who characterized those systems as having a *horizontal decomposition* as shown in

HORIZONTAL

DECOMPOSITION

Fig. 4.1.

Instead, an examination of the ethological literature suggests that intelligence is layered in a *vertical decomposition*, shown in Fig. 4.2. Under a ver-

VERTICAL

DECOMPOSITION

tical decomposition, an agent starts with primitive survival behaviors and evolves new layers of behaviors which either reuse the lower, older behaviors, inhibit the older behaviors, or create parallel tracks of more advanced behaviors. The parallel tracks can be thought of layers, stacked vertically. Each layer has access to sensors and actuators independently of any other layers. If anything happens to an advanced behavior, the lower level behaviors would still operate. This return to a lower level mimics degradation of autonomous functions in the brain. Functions in the brain stem (such as breathing) continue independently of higher order functions (such as counting, face recognition, task planning), allowing a person who has brain damage from a car wreck to still breathe, etc.

Work by Arkin, Brooks, and Payton focused on defining behaviors and on mechanisms for correctly handling situations when multiple

behaviors are active simultaneously. Brooks took an approach now known as subsumption and built insect-like robots with behaviors captured in hardware circuitry.

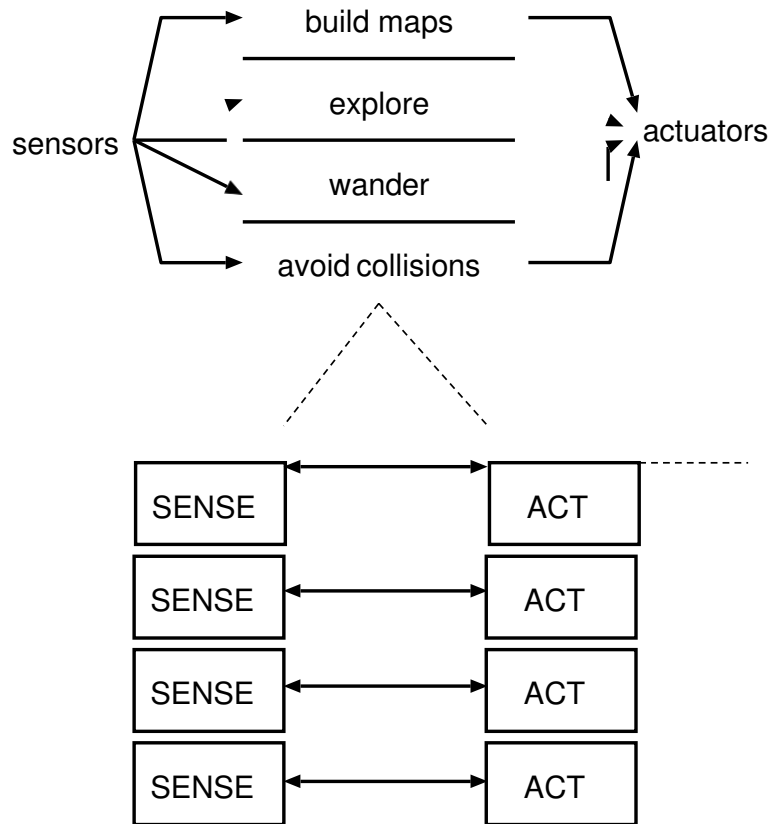


Figure 4.2 Vertical decomposition of tasks into an S-A organization, associated with the Reactive Paradigm.

Arkin and Payton used a potential fields methodology, favoring software implementations. Both approaches are equivalent. The Reactive Paradigm was initially met with stiff resistance from traditional customers of robotics, particularly the military and nuclear regulatory agencies. These users of robotic technologies were uncomfortable with the imprecise way in which discrete behaviors combine to form a rich emergent behavior. In particular, reactive behaviors are not amenable to mathematical proofs showing they are sufficient and correct for a task. In the end, the rapid execution times associated with the reflexive behaviors led to its acceptance among users, just as researchers shifted to the Hybrid paradigm in order to fully explore layering of intelligence.

The major theme of this chapter is that all reactive systems are composed of behaviors, though the meaning of a behavior may be slightly different in each reactive architecture. Behaviors can execute concurrently and/or sequentially. The two representative architectures, subsumption and potential fields, are compared and contrasted using the same task as an example. This chapter will concentrate on how architecture handles concurrent behaviors to produce an emergent behavior, deferring sequencing to the next chapter.

4.2 Attributes of Reactive Paradigm

BEHAVIORS

The fundamental attribute of the reactive paradigm is that all actions are accomplished through behaviors. As in ethological systems, *behaviors are a direct mapping of sensory inputs to a pattern of motor actions that are then used to achieve a task*. From a mathematical perspective, behaviors are simply a transfer function, transforming sensory inputs into actuator commands. For the purposes of this book, a behavior will be treated as a schema, and will consist of at least one motor schema and one perceptual schema. The motor schema contains the algorithm for generating the pattern of action in a physical actuator and the perceptual schema contains the algorithm for extracting the percept and its strength. Keep in mind that few reactive robot architectures describe their behaviors in terms of schemas. But in practice, most behavioral implementations have recognizable motor and perceptual routines, even though they are rarely referred to as schemas.

SENSE-ACT

The Reactive Paradigm literally threw away the PLAN component of the SENSE, PLAN, ACT triad, as shown in Fig. 4.3. The SENSE and ACT components are tightly coupled into behaviors, and all robotic activities emerge as the result of these behaviors operating either in sequence or concurrently. The S-A organization does not specify how the behaviors are coordinated

ORGANIZATION

and controlled; this is an important topic addressed by architectures.

BEHAVIOR-SPECIFIC

Sensing in the Reactive Paradigm is local to each behavior, or behavior-

(LOCAL) SENSING specific. Each behavior has its own dedicated sensing. In many cases, this is implemented as one sensor and perceptual schema per behavior. But in other cases, more than one behavior can take the same output from a sensor and process it differently (via the behavior's perceptual schema).

4.2 Attributes of Reaction Paradigm
One behavior literally does not know what another behavior is doing or perceiving. Fig. 4.4 graphically shows the sensing style of the Reactive Paradigm.

Note that this is fundamentally opposite of the global world model used in the hierarchical paradigm. Sensing is immediately available to the be-

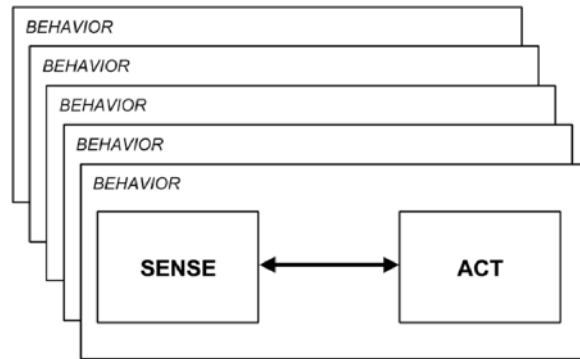


Figure 4.3 S-A organization of the Reactive Paradigm into multiple, concurrent behaviors.

havior’s perceptual schema, which can do as little or as much processing as needed to extract the relevant percept. If a computationally inexpensive affordance is used, then the sensing portion of the behavior is nearly instantaneous and action is very rapid.

As can be seen from the previous chapter on the biological foundations of the reactive paradigm, behaviors favor the use of affordances. In fact, Brooks was fond of saying (loudly) at conferences, “we don’t need no stinking representations.” It should be noted that often the perceptual schema portion of the behavior has to use a behavior-specific representation or data structure to substitute for specialized detectors capable of extracting affordances. For example, extracting a red region in an image is non-trivial with a computer compared with an animal seeing red. The point is that while a computer program may have to have data structures in order to duplicate a simple neural function, the behavior does not rely on any central representation built up from all sensors.

In early implementations of the reactive paradigm, the idea of “one sensor, one behavior” worked well. For more advanced behaviors, it became useful to fuse the output of multiple sensors within one perceptual schema to have increased precision or a better measure of the strength of the stimulus. This type of sensor fusion is permitted within the reactive paradigm as long as the fusion is local to the behavior. Sensor

4.2 *Attributes of Reactive Paradigm*
fusion will be detailed in Ch. 6.

109

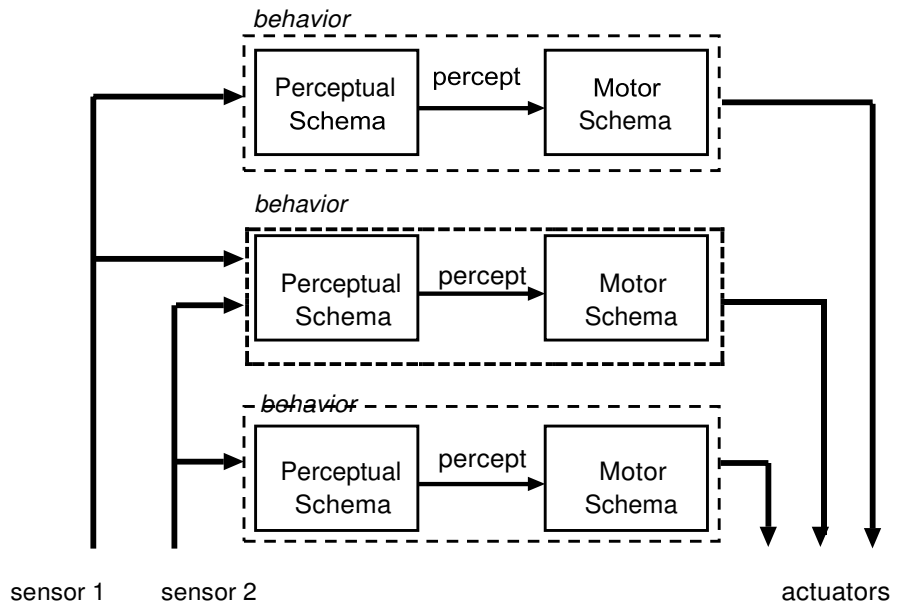


Figure 4.4 Behavior-specific sensing organization in the Reactive Paradigm: sensing is local, sensors can be shared, and sensors can be fused locally by a behavior.

4.2.1 Characteristics and connotations of reactive behaviors

As seen earlier, a reactive robotic system decomposes functionality into behaviors, which tightly couple perception to action without the use of intervening abstract (global) representations. This is a broad, vague definition. Over the years, the reactive paradigm has acquired several connotations and characteristics from the way practitioners have used the paradigm.

The primary connotation of a reactive robotic system is that it executes rapidly. The tight coupling of sensing and acting permits robots to operate in real-time, moving at speeds of 1-2 cm per second. Behaviors can be implemented directly in hardware as circuits, or with low computational complexity algorithms ($O(n)$). This means that behaviors execute quickly regardless of the processor. Behaviors execute not only fast in their own right, they are particularly fast when compared to the execution times of Shakey and the Stanford Cart. A secondary connotation is that reactive robotic systems have no memory, limiting reactive behaviors to what biologists would call pure stimulus-response reflexes. In practice, many

fixed-action pattern type of response, where the behavior persists for a short period of time without the direct presence of the stimulus. The main point is that behaviors are controlled by what is happening in the world, duplicating the spirit of innate releasing mechanisms, rather than by the program storing and remembering what the robot did last. The examples in the chapter emphasize this point.

The five characteristics of almost all architectures that follow the Reactive Paradigm are:

SITUATED AGENT

1. *Robots are situated agents operating in an ecological niche.* As seen earlier in Part I, *situated agent* means that the robot is an integral part of the world. A robot has its own goals and intentions. When a robot acts, it changes the world, and receives immediate feedback about the world through sensing. What the robot senses affects its goals and how it attempts to meet them, generating a new cycle of actions. Notice that situatedness is defined by Neisser's Action-Perception Cycle. Likewise, the goals of a robot, the world it operates in, and how it can perceive the world form the ecological niche of the robot. To emphasize this, many robotic researchers say they are working on *ecological robotics*.

ECOLOGICAL ROBOTICS

2. *Behaviors serve as the basic building blocks for robotic actions, and the overall behavior of the robot is emergent.* Behaviors are independent, computational entities and operate concurrently. The overall behavior is emergent: there is no explicit "controller" module which determines what will be done, or functions which call other functions. There may be a coordinated control program in the schema of a behavior, but there is no external controller of all behaviors for a task. As with animals, the "intelligence" of the robot is in the eye of the beholder, rather than in a specific section of code. Since the overall behavior of a reactive robot emerges from the way its individual behaviors interact, the major differences between reactive architectures is usually the specific mechanism for interaction. Recall from Chapter 3 that these mechanisms include combination, suppression, and cancellation.

EGO-CENTRIC

3. *Only local, behavior-specific sensing is permitted.* The use of explicit abstract representational knowledge in perceptual processing, even though it is behavior-specific, is avoided. Any sensing which does require representation is expressed in *ego-centric* (robot-centric) coordinates. For example, consider obstacle avoidance. An ego-centric representation means that it does not matter that an obstacle is in the world at coordinates (x,y,z) , only

where it is relative to the robot. Sensor data, with the exception of GPS, is inherently ego-centric (e.g., a range finder returns a distance to the nearest object from the transducer), so this eliminates unnecessary processing to create a world model, then extract the position of obstacles relative to the robot.

4. *These systems inherently follow good software design principles.* The modularity of these behaviors supports the decomposition of a task into component behaviors. The behaviors are tested independently, and behaviors may be assembled from primitive behaviors.
5. *Animal models of behavior are often cited as a basis for these systems or a particular behavior.* Unlike in the early days of AI robotics, where there was a conscious effort to not mimic biological intelligence, it is very acceptable under the reactive paradigm to use animals as a motivation for a collection of behaviors.

4.2.2 Advantages of programming by behavior

Constructing a robotic system under the Reactive Paradigm is often referred to as programming by behavior, since the fundamental component of any implementation is a behavior. Programming by behavior has a number of advantages, most of them consistent with good software engineering principles. Behaviors are inherently modular and easy to test in isolation from the system (i.e., they support unit testing). Behaviors also support incremental expansion of the capabilities of a robot. A robot becomes more intelligent by having more behaviors. The behavioral decomposition results in an implementation that works in real-time and is usually computationally inexpensive. Although we'll see that sometimes duplicating specialized detectors (like optic flow) is slow. If the behaviors are implemented poorly, then a reactive implementation can be slow. But generally, the reaction speeds of a reactive robot are equivalent to stimulus-response times in animals.

Behaviors support good software engineering principles through decomposition, modularity and incremental testing. If programmed with as high a degree of independence (also called *low coupling*) as possible, and *high co-*

LOW COUPLING

HIGH COHESION

hesion, the designer can build up libraries of easy to understand, maintain, and reuse modules that minimize side effects. Low coupling means that the modules can function independently of each other with minimal

4.2 Attributes of Reactive Paradigm
connections or interfaces, promoting easy reuse. Cohesion means that the data and operations contained by a module relate only to the purpose of that module.¹¹¹

Higher cohesion is associated with modules that do one thing well, like the `SQRT` function in C. The examples in Sec. 4.3 and 4.4 attempt to illustrate the choices a designer has in engineering the behavioral software of a robot.

4.2.3 Representative architectures

In order to implement a reactive system, the designer must identify the set of behaviors necessary for the task. The behaviors can either be new or use existing behaviors. The overall action of the robot emerges from multiple, concurrent behaviors. Therefore a reactive architecture must provide mechanisms for 1) triggering behaviors and 2) for determining what happens when multiple behaviors are active at the same time. Another distinguishing feature between reactive architectures is how they define a behavior and any special use of terminology. Keep in mind that the definitions presented in Sec. 4.2 are a generalization of the trends in reactive systems, and do not necessarily have counterparts in all architectures.

There are many architectures which fit in the Reactive Paradigm. The two best known and most formalized are the subsumption and potential field methodologies. Subsumption refers to how behaviors are combined. Potential Field Methodologies require behaviors to be implemented as potential fields, and the behaviors are combined by summation of the fields. A third

RULE ENCODING

style of reactive architecture which is popular in Europe and Japan is *rule encoding*, where the motor schema component of behaviors and the combination mechanism are implemented as logical rules. The rules for combining behaviors are often ad hoc, and so will not be covered in this book. Other methods for combining behaviors exist, including fuzzy methods and winner-take-all voting, but these tend to be implementation details rather than an over-arching architecture.

4.3 Subsumption Architecture

Rodney Brooks' subsumption architecture is the most influential of the purely Reactive Paradigm systems. Part of the influence stems from the publicity surrounding the very naturalistic robots built with subsumption. As seen in Fig. 4.5, these robots actually looked like shoe-box sized insects, with six legs and antennae. In many implementations,

~~4.3 Subsumption Architecture~~¹¹⁴
the behaviors are embedded directly in the hardware or on small microprocessors, allowing the robots to have all on-board computing (this was unheard of in the processor-impoverished mid-1980's). Furthermore, the robots were the first to be able

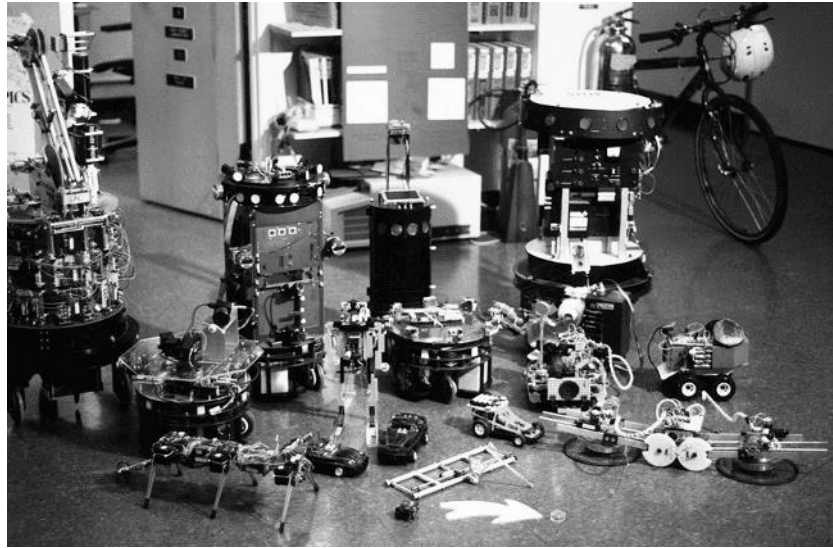


Figure 4.5 “Veteran” robots of the MIT AI Laboratory using the subsumption architecture. (Photograph courtesy of the MIT Artificial Intelligence Laboratory.)

to walk, avoid collisions, and climb over obstacles without the “move-think- move-think” pauses of Shakey.

The term “behavior” in the subsumption architecture has a less precise meaning than in other architectures. A behavior is a network of sensing and acting modules which accomplish a task. The modules are augmented finite state machines AFSM, or finite state machines which have registers, timers, and other enhancements to permit them to be interfaced with other modules. An AFSM is equivalent to the interface between the schemas and the coordinated control strategy in a behavioral schema. In terms of schema theory, a subsumption behavior is actually a collection of one or more schemas into an abstract behavior.

Behaviors are released in a stimulus-response way, without an external program explicitly coordinating and controlling them. Four interesting aspects of subsumption in terms of releasing and control are:

LAYERS OF

COMPETENCE

1. Modules are grouped into *layers of competence*. The layers reflect a hierarchy of intelligence, or competence. Lower layers encapsulate basic survival functions such as avoiding collisions, while higher

more goal-directed actions such as mapping. Each of the layers can be viewed as an abstract behavior for a particular task.

LAYERS CAN SUBSUME

2. Modules in a higher layer can override, or subsume, the output from be-

LOWER LAYERS

haviors in the next lower layer. The behavioral layers operate concurrently and independently, so there needs to be a mechanism to handle potential conflicts. The solution in subsumption is a type of winner-take-all, where the winner is always the higher layer.

NO INTERNAL STATE

3. The use of internal state is avoided. Internal state in this case means any type of local, persistent representation which represents the state of the world, or a model. Because the robot is a situated agent, most of its information should come directly from the world. If the robot depends on an internal representation, what it believes may begin to dangerously diverge from reality. Some internal state is needed for releasing behaviors like being scared or hungry, but good behavioral designs minimize this.

TASKABLE

4. A task is accomplished by activating the appropriate layer, which then activates the lower layers below it, and so on. However, in practice, subsumption style systems are not easily *taskable*, that is, they can't be ordered to do another task without being reprogrammed.

4.3.1 Example

These aspects are best illustrated by an example, extensively modified from Brooks' original paper²⁷ in order to be consistent with schema theory terminology and to facilitate comparison with a potential fields methodology. A robot capable of moving forward while not colliding with anything could be

LEVEL 0: AVOID

represented with a single layer, Level 0. In this example, the robot has multiple sonars (or other range sensors), each pointing in a different direction, and two actuators, one for driving forward and one for turning.

POLAR PLOT

Following Fig. 4.6, the SONAR module reads the sonar ranges, does any filtering of noise, and produces a *polar plot*. A polar plot represents the range readings in polar coordinates, $(r; \theta)$, surrounding the robot. As shown in Fig. 4.7, the polar plot can be "unwound."

If the range reading for the sonar facing dead ahead is below a certain threshold, the COLLIDE module declares a collision and sends the halt signal to the FORWARD drive actuator. If the robot was moving forward,

it now stops. Meanwhile, the FEELFORCE module is receiving the same polar plot. It treats each sonar reading as a repulsive force, which can be represented

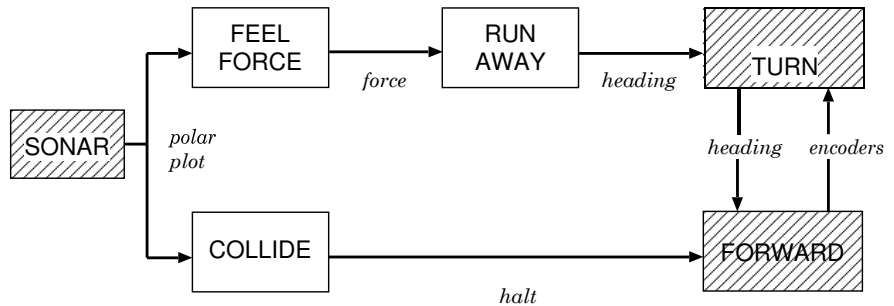


Figure 4.6 Level 0 in the subsumption architecture.

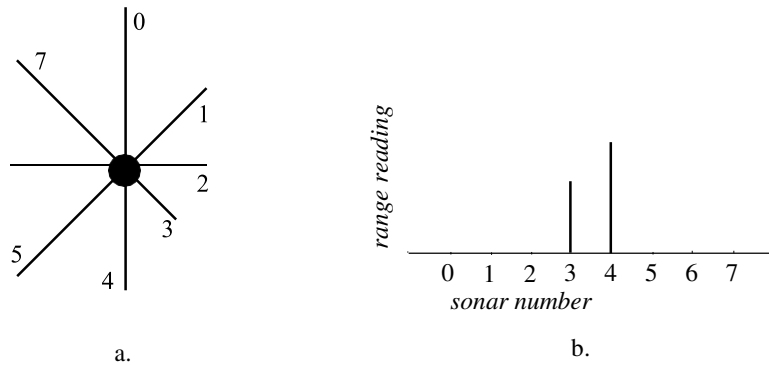


Figure 4.7 Polar plot of eight sonar range readings: a.) “robo-centric” view of range readings along acoustic axes, and b.) unrolled into a plot.

as a vector. Recall that a vector is a mathematical construct that consists of a magnitude and a direction. **FEELFORCE** can be thought of as summing the vectors from each of the sonar readings. This results in a new vector. The repulsive vector is then passed to the **TURN** module. The **TURN** module splits off the direction to turn and passes that to the steering actuators. **TURN** also passes the vector to the **FORWARD** module, which uses the magnitude of the vector to determine the magnitude of the next forward motion (how far or how fast). So the robot turns and moves a short distance away from the obstacle.

The observable behavior is that the robot will sit still if it is in an unoccupied space, until an obstacle comes near it. If the obstacle is on one side of

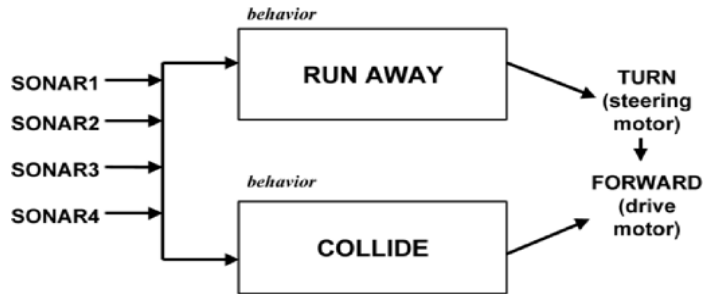


Figure 4.8 Level 0 recast as primitive behaviors.

the robot, the robot will turn 180° the other way and move forward; essentially, it runs away. This allows a person to herd the robot around. The robot can react to an obstacle if the obstacle (or robot) is motionless or moving; the response is computed at each sensor update.

However, if part of the obstacle, or another obstacle, is dead ahead (some- one tries to herd the robot into a wall), the robot will stop, then apply the results of RUNAWAY. So it will stop, turn and begin to move forward again. Stopping prevents the robot from side-swiping the obstacle while it is turn- ing and moving forward. Level 0 shows how a fairly complex set of actions can emerge from very simple modules.

It is helpful to recast the subsumption architecture in the terms used in this book, as shown in Fig. 4.8. Note how this looks like the vertical decompo- sition in Fig. 4.2: the sensor data flows through the concurrent behaviors to the actuators, and the independent behaviors cause the robot to do the right thing. The SONAR module would be considered a global interface to the sen- sors, while the TURN and FORWARD modules would be considered part of the actuators (an interface). For the purposes of this book, a behavior must consist of a perceptual schema and a motor schema. Perceptual schemas are connected to a sensor, while motor schemas are connected to actuators. For Level 0, the perceptual schemas would be contained in the FEELFORCE and COLLIDE modules. The motor schemas are RUNAWAY and COLLIDE mod- ules. COLLIDE combines both perceptual processing (extracts the vector for the sonar

facing directly ahead) and the pattern of action (halt if there is a

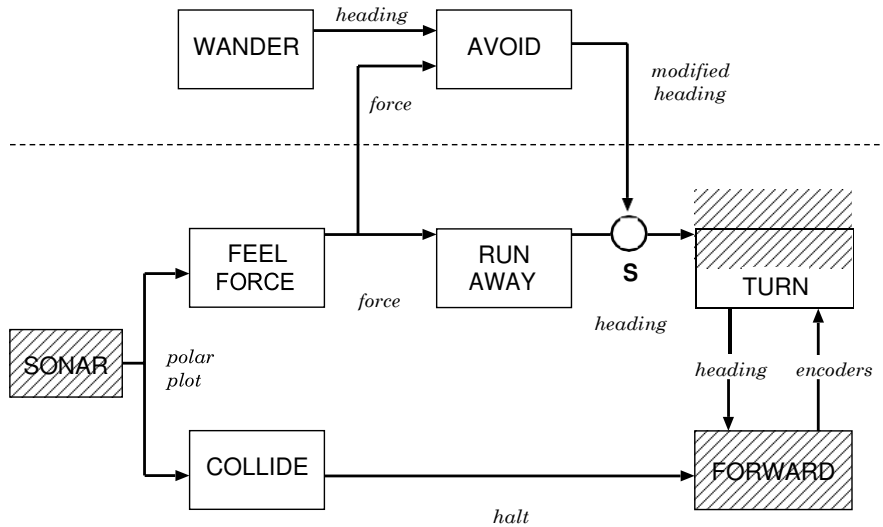


Figure 4.9 Level 1: wander.

reading). The primitive behaviors reflect the two paths through the layer. One might be called the runaway behavior and the other the collide behavior. Together, the two behaviors create a rich obstacle avoidance behavior, or a layer of competence.

It should also be noticed that the behaviors used direct perception, or affordances. The presence of a range reading indicated there was an obstacle; the robot did not have to know what the obstacle was.

Consider building a robot which actually wandered around instead of sitting motionless, but was still able to avoid obstacles. Under subsumption, a second layer of competence (Level 1) would be added, shown in Fig. 4.9. In this case, Level 1 consists of a WANDER module which computes a random heading every n seconds. The random heading can be thought of as a vector. It needs to pass this heading to the TURN and FORWARD modules. But it can't be passed to the TURN module directly. That would sacrifice obstacle avoidance, because TURN only accepts one input. One solution is to add another module in Level 1, AVOID, which combines the FEEL FORCE vector with the WANDER vector. Adding a new avoid module offers an opportunity to create a more sophisticated response to obstacles. AVOID combines the direction

LEVEL 1: WANDER

of the force of avoidance with the desired heading. This results in the actual heading being mostly in the right direction rather than having the robot turn

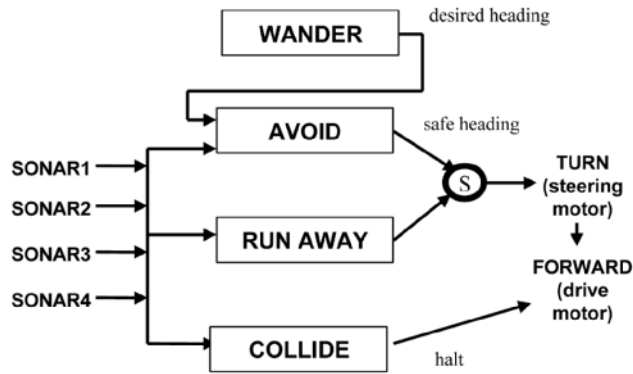


Figure 4.10 Level 1 recast as primitive behaviors.

around and lose forward progress. (Notice also that the AVOID module was able to “eavesdrop” on components of the next lower layer.) The heading output from AVOID has the same representation as the output of RUNAWAY, so TURN can accept from either source.

The issue now appears to be when to accept the heading vector from which layer. Subsumption makes it simple: the output from the higher level subsumes the output from the lower level. Subsumption is done in one of two ways:

INHIBITION

1. *inhibition*. In inhibition, the output of the subsuming module is connected to the output of another module. If the output of the subsuming module is “on” or has any value, the output of the subsumed module is blocked or turned “off.” Inhibition acts like a faucet, turning an output stream on and off.

SUPPRESSION

2. *suppression*. In suppression, the output of the subsuming module is connected to the input of another module. If the output of the subsuming module is on, it *replaces* the normal input to the subsumed module. Suppression acts like a switch, swapping one input stream for another.

In this case, the AVOID module suppresses (marked in the diagram with a S) the output from RUNAWAY. RUNAWAY is still executing, but its output doesn’t go anywhere. Instead, the output from AVOID goes to TURN.

The use of layers and subsumption allows new layers to be built on top of less competent layers, without modifying the lower layers. This is good software engineering, facilitating modularity and simplifying testing. It also adds some robustness in that if something should disable the Level 1 behaviors, Level 0 might remain intact. The robot would at least be able to preserve its self-defense mechanism of fleeing from approaching obstacles.

Fig. 4.10 shows Level 1 recast as behaviors. Note that `FEELFORCE` was used by both `RUNAWAY` and `AVOID`. `FEELFORCE` is the perceptual component (or schema) of both behaviors, with the `AVOID` and `RUNAWAY` modules being the motor component (or schema). As is often the case, behaviors are usually named after the observable action. This means that the behavior (which consists of perception and action) and the action component have the same name. The figure does not show that the `AVOID` and `RUNAWAY` behaviors share the same `FEELFORCE` perceptual schema. As will be seen in the next chapter, the object-oriented properties of schema theory facilitate the reuse and sharing of perceptual and motor components.

LEVEL 2: FOLLOW

Now consider adding a third layer to permit the robot to move down corridors, as shown in Fig. 4.11. (The third layer in Brooks' original paper is "explore," because he was considering a mapping task.) The `LOOK` module examines the sonar polar plot and identifies a corridor. (Note that this is another example of behaviors sharing the same sensor data but using it locally for different purposes.) Because identifying a corridor is more computationally expensive than just extracting range data, `LOOK` may take longer to run than behaviors at lower levels. `LOOK` passes the vector representing the direction to the middle of the corridor to the `STAYINMIDDLE` module. `STAYINMIDDLE` subsumes the `WANDER` module and delivers its output to the `AVOID` module which can then swerve around obstacles.

CORRIDORS

But how does the robot get back on course if the `LOOK` module has not computed a new direction? In this case, the `INTEGRATE` module has been observing the robot's actual motions from shaft encoders in the actuators. This gives an estimate of how far off course the robot has traveled since the last update by `LOOK`. `STAYINMIDDLE` can use the dead reckoning data with the intended course to compute the new course vector. It serves to fill in the gaps in mismatches between update rates of the different modules. Notice that `LOOK` and `STAYINMIDDLE` are quite sophisticated from a software perspective.

INTEGRATE is an example of a module which is supplying a dangerous internal state: it is actually substituting for feedback from the real world. If for some reason, the LOOK module never updates, then the robot could op-

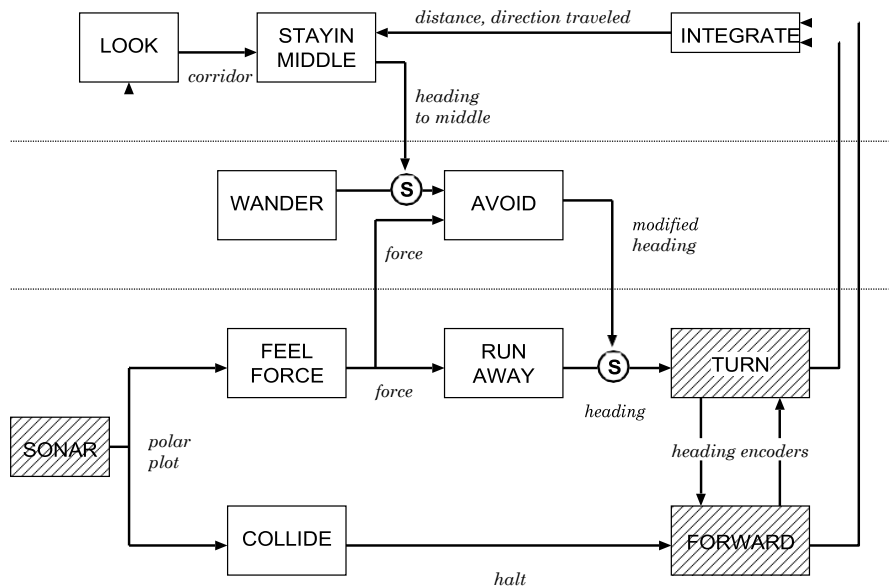


Figure 4.11 Level 2: follow corridors.

erate without any sensor data forever. Or at least until it crashed! Therefore, subsumption style systems include time constants on suppression and inhibition. If the suppression from `STAY IN MIDDLE` ran for longer than n seconds without a new update, the suppression would cease. The robot would then begin to wander, and hopefully whatever problem (like the corridor being totally blocked) that had led to the loss of signal would fix itself.

Of course, a new problem is how does the robot know that it hasn't started going down the hallway it just came up? Answer: it doesn't. The design assumes that that a corridor will always be present in the robot's ecological niche. If it's not, the robot does not behave as intended. This is an example of the connotation that reactive systems are "memory-less."

4.3.2 Subsumption summary

To summarize subsumption:

Subsumption has a loose definition of behavior as a tight coupling of

sens- ing and acting. Although it is not a schema-theoretic architecture, it can

be described in those terms. It groups schema-like modules into layers of competence, or abstract behaviors.

Higher layers may subsume and inhibit behaviors in lower layers, but behaviors in lower layers are never rewritten or replaced. From a programming standpoint, this may seem strange. However, it mimics biological evolution. Recall that the fleeing behavior in frogs (Ch. 3) was actually the result of two behaviors, one which always moved toward moving objects and the other which actually suppressed that behavior when the object was large.

The design of layers and component behaviors for a subsumption implementation, as with all behavioral design, is hard; it is more of an art than a science. This is also true for all reactive architectures.

There is nothing resembling a STRIPS-like plan in subsumption. Instead, behaviors are released by the presence of stimulus in the environment.

Subsumption solves the frame problem by eliminating the need to model the world. It also doesn't have to worry about the open world being non-monotonic and having some sort of truth maintenance mechanism, because the behaviors do not remember the past. There may be some perceptual persistence leading to a fixed-action pattern type of behavior (e.g., corridor following), but there is no mechanism which monitors for changes in the environment. The behaviors simply respond to whatever stimulus is in the environment.

Perception is largely direct, using affordances. The releaser for a behavior is almost always the percept for guiding the motor schema.

Perception is ego-centric and distributed. In the wander (layer 2) example, the sonar polar plot was relative to the robot. A new polar plot was created with each update of the sensors. The polar plot was also available to any process which needed it (shared global memory), allowing user modules to be distributed. Output from perceptual schemas can be shared with other layers.

4.4 Potential Fields Methodologies

Another style of reactive architecture is based on potential fields. The

specific architectures that use some type of potential fields are too numerous to

VECTORS

describe here, so instead a generalization will be presented. Potential field styles of behaviors always use *vectors* to represent behaviors and *vector sum-*

VECTOR SUMMATION

mation to combine vectors from different behaviors to produce an emergent behavior.

4.4.1 Visualizing potential fields

The first tenet of a potential fields architecture is that the motor action of a behavior must be represented as a potential field. A potential field is an array, or field, of vectors. As described earlier, a vector is a mathematical construct which consists of a magnitude and a direction. Vectors are often used to represent a force of some sort. They are typically drawn as an arrow, where the length of the arrow is the magnitude of the force and the angle of the arrow is the direction. Vectors are usually represented with a boldface capital letter, for example, **V**. A vector can also be written as a tuple $(m; d)$, where m stands for magnitude and d for direction. By convention the magnitude is a real number between 0.0 and 1, but the magnitude can be any real number.

ARRAY REPRESENTING

The array represents a region of space. In most robotic applications, the

A FIELD space is in two dimensions, representing a bird's eye view of the world just like a map. The map can be divided into squares, creating a (x,y) grid. Each element of the array represents a square of space. Perceivable objects in the world exert a force field on the surrounding space. The force field is analogous to a magnetic or gravitation field. The robot can be thought of as a particle that has entered the field exuded by an object or environment. The vector in each element represents the force, both the direction to turn and the magnitude or velocity to head in that direction, a robot would feel if it were at that particular spot. Potential fields are continuous because it doesn't matter how small the element is; at each point in space, there is an associated vector.

Fig. 4.12 shows how an obstacle would exert a field on the robot and make it run away. If the robot is close to the obstacle, say within 5 meters, it is inside the potential field and will feel a force that makes it want to face directly away from the obstacle (if it isn't already) and move away. If the robot is not within range of the obstacle, it just sits there because there is no force on it. Notice that the field represents what the robot should do (the motor schema) based on if the robot perceives an obstacle (the

4.4 Potential Fields Methodologies 125
perceptual schema). The field isn't concerned with how the robot came to be so close to the obstacle; the robot feels the same force if it were happening to move within range or if it was just sitting there and someone put their hand next to the robot.

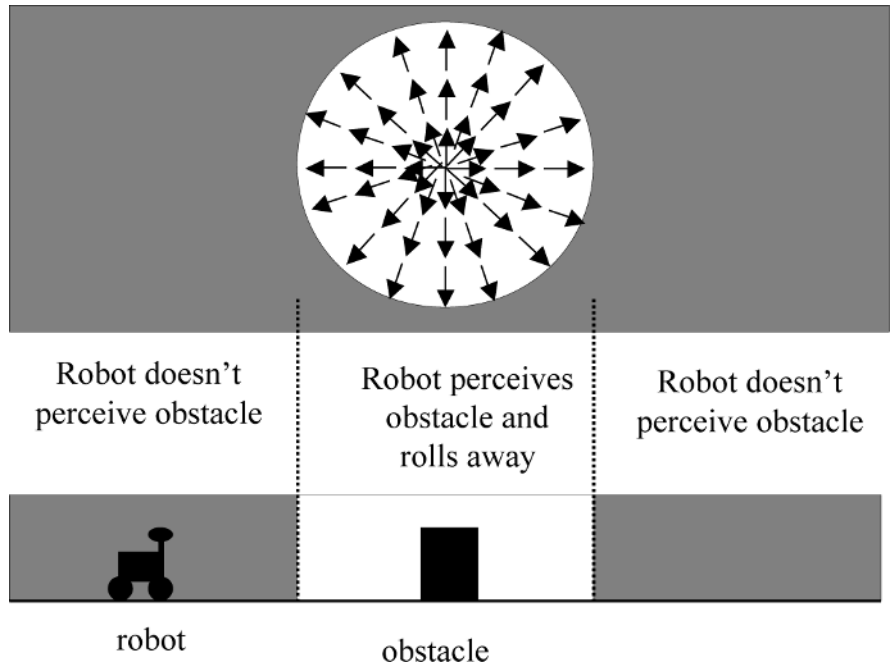


Figure 4.12 Example of an obstacle exerting a repulsive potential field over the radius of 1 meter.

One way of thinking about potential fields is to imagine a force field acting on the robot. Another way is to think of them as a potential energy surface in three dimensions (gravity is often represented this way) and the robot as a marble. In that case, the vector indicates the direction the robot would “roll” on the surface. Hills in the surface cause the robot to roll away or around (vectors would be pointing away from the “peak” of the hill), and valleys would cause the robot to roll downward (vectors pointing toward the bottom).

There are five basic potential fields, or primitives, which can be combined to build more complex fields: *uniform*, *perpendicular*, *attractive*, *repulsive*, and *tangential*. Fig. 4.13 shows a uniform field. In a uniform field, the robot would feel the same force no matter where it was. No matter where it got set down and at what orientation, it would feel a need to turn to align itself to the direction the arrow points and to move in that direction at a velocity propor-

FIVE PRIMITIVE FIELDS

UNIFORM FIELD

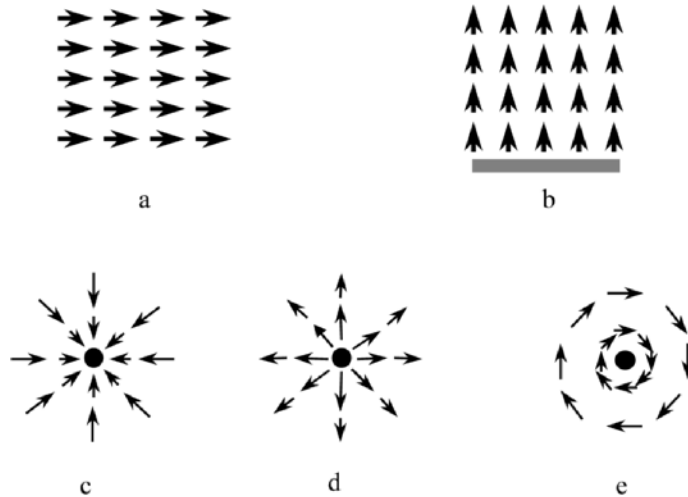


Figure 4.13 Five primitive potential fields: a.) uniform, b.) perpendicular, c.) attraction, d.) repulsion, and e.) tangential.

tional to the length of the arrow. A uniform field is often used to capture the behavior of “go in direction n .”

PERPENDICULAR FIELD

Fig. 4.13b shows a perpendicular field, where the robot is oriented perpendicular to some object or wall or border. The field shown is directed away from the gray wall, but a perpendicular field can be pointed towards an object as well.

ATTRACTIVE FIELD

Fig. 4.13c illustrates an attractive field. The circle at the center of the field represents an object that is exerting an attraction on the robot. Wherever the robot is, the robot will “feel” a force relative to the object. Attractive fields are useful for representing a taxis or tropism, where the agent is literally attracted to light or food or a goal. The opposite of an attractive field is a repulsive field, shown in Fig. 4.13d. Repulsive fields are commonly associated with obstacles, or things the agent should avoid. The closer the robot is to the object, the stronger the repulsive force 180° away from it.

TANGENTIAL FIELD

The final primitive field is the tangential field in Fig. 4.13e. The field is a tangent around the object (think of a tangent vector as being

perpendicular to

radial lines extending outward from the object). Tangential fields can “spin” either clockwise or counterclockwise; Fig. 4.13 shows a clockwise spin. They are useful for directing a robot to go around an obstacle, or having a robot investigate something.

4.4.2 Magnitude profiles

MAGNITUDE PROFILE

Notice that in Fig. 4.13, the length of the arrows gets smaller closer to the object. The way the magnitude of vectors in the field change is called the *magnitude profile*. (The term “magnitude profile” is used here because the term “velocity profile” is used by control engineers to describe how a robot’s motors actually accelerate and decelerate to produce a particular movement without jerking.)

Consider the repulsive field in Fig. 4.12. Mathematically, the field can be represented with polar coordinates and the center of the field being the origin (0,0):

$$(4.1) \quad \begin{aligned} V_{direction} &= \\ V_{magnitude} &= c \end{aligned}$$

In that case, the magnitude was a constant value, c : the length of the arrows was the same. This can be visualized with a plot of the magnitude shown in Fig. 4.14a.

This profile says that the robot will run away (the direction it will run is) at the same velocity, no matter how close it is to the object, as long as it is in the range of the obstacle. As soon as the robot gets out of range of the obstacle, the velocity drops to 0.0, stopping the robot. The field is essentially binary: the robot is either running away at a constant speed or stopped. In practice there is a problem with a constant magnitude. It leads to jerky motion on the perimeter of the range of the field. This is illustrated when a robot is heading in a particular direction, then encounters an obstacle. It runs away, leaving the field almost immediately, and turns back to its original path, encounters the field again, and so on.

REFLEXIVITY

Magnitude profiles solve the problem of a constant magnitude. They also make it possible for a robot designer to represent reflexivity (that a response should be proportional to the strength of a stimulus) and to

create interesting responses. Now consider the profile in Fig. 4.13c. It can be described as how

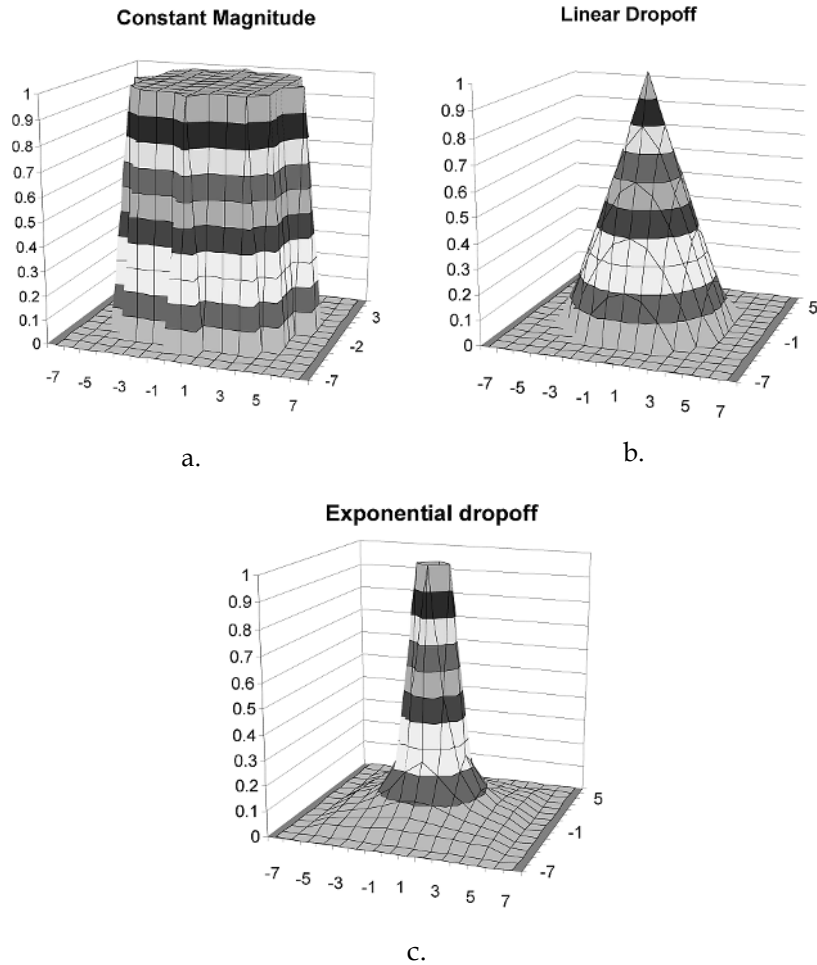


Figure 4.14 Plots of magnitude profiles for a field of radius 5 units: a.) constant magnitude, b.) linear drop off with a slope of -1, and c.) exponential drop off.

LINEAR DROP OFF

an observer would see a robot behave in that field: if the robot is far away from the object, it will turn and move quickly towards it, then slow up to keep from overshooting and hitting the object. Mathematically, this is called a *linear drop off*, since the rate at which the magnitude of the vectors drops off can be plotted as a straight line. The formula for a straight line is $y = mx + b$, where x is the distance and y is magnitude. b biases where the line starts, and m is the slope ($m = \frac{y}{x}$). Any value of m and b is acceptable. If it is not specified, $m = 1$ or -1 (a 45° slope up or down) and $b = 0$ in linear functions.

EXPONENTIAL DROP

The linear profile in Fig. 4.14b matches the desired behavior of the designer: to have the robot react more, the closer it is. But it shares the problem of the constant magnitude profile in the sharp transition to 0.0 velocity. Therefore, another profile might be used to capture the need for a strong reaction but with more of a taper. One such profile is a *exponential drop off*

^{OFF} function, where the drop off is proportional to the square of the distance: for every unit of distance away from the object, the force on the robot drops in half. The exponential profile is shown in Fig. 4.14c.

As can be seen from the previous examples, almost any magnitude profile is acceptable. The motivation for using magnitude profiles is to fine-tune the behavior. It is important to note that the robot only computes the vectors acting on it at its current location. The figures display the entire field for all possible locations of the robot. The question then arises as to why do the figures show an entire field over space? First, it aids visualizing what the robot will do overall, not just at one particular time step. Second, since fields are continuous representations, it simplifies confirming that the field is correct and makes any abrupt transitions readily apparent.

4.4.3 Potential fields and perception

In the previous examples, the force of the potential field at any given point was a function of both the relative distance between the robot and an object and the magnitude profile. The strength of a potential field can be a function of the stimulus, regardless of distance. As an example recall from Ch. 3 the feeding behavior of baby arctic terns where the feeding behavior is guided by the stimulus “red.” This can be modeled by an attractive field. The bigger and redder an object in the baby’s field

4.4 Potential Fields Methodologies 133
of view, the stronger the attraction, suggesting that a magnitude profile using an increasing exponential function. Another important point that has already been mentioned is that potential fields are ego-centric because robot perception is ego-centric.

4.4.4 Programming a single potential field

Potential fields are actually easy to program, especially since the fields are ego-centric to the robot. The visualization of the entire field may appear to indicate that the robot and the objects are in a fixed, absolute coordinate system, but they are not. The robot computes the effect of the potential field, usually as a straight line, at every update, with no memory of where it was previously or where the robot has moved. This should become clear through the following examples.

A primitive potential field is usually represented by a single function. The vector impacting the robot is computed each update. Consider the case of a robot with a single range sensor facing forward. The designer has decided that a repulsive field with a linear drop off is appropriate. The formula is:

$$(4.2) \quad \begin{aligned} V_{direction} &= 180 \\ V_{magnitude} &= \begin{cases} \frac{(D-d)}{D} & \text{for } d \leq D \\ 0 & \text{for } d > D \end{cases} \end{aligned}$$

where D is the maximum range of the field's effect, or the maximum distance at which the robot can detect the obstacle. (D isn't always the detection range. It can be the range at which the robot should respond to a stimulus. For example, many sonars can detect obstacles 20 feet away, producing an almost infinitesimal response in emergent behavior but requiring the runtime overhead of a function call. In practice, a roboticist might set a D of 2 meters.) Notice that the formula produces a result where $0.0 \leq V_{magnitude} \leq 1.0$.

Below is a C code fragment that captures the repulsive field.

```
typedef struct {
    double magnitude;
    double direction;
} vector;

vector repulsive(double d, double D)
{
    if (d <= D) {
        outputVector.direction = -180; //turn around!
```



```

Potential Fields Methodologies
outputvector.magnitude = (D-d)/D; //linear dropoff
}
else {

```

```

        outputVector.direction=0.0
        outputVector.magnitude=0.0
    }
    return outputVector;
}

```

At this point, it is easy to illustrate how a potential field can be used by a behavior, *runaway*, for the robot with a single sensor. The *runaway* behavior will use the `repulsive()` function as the motor schema, and a function `readSonar()` as the perceptual schema. The output of the behavior is a vector. *runaway* is called by the robot on every update cycle.

```

vector runaway( ){
    double reading;
    reading=readSonar();//perceptual schema
    vector=repulsive (reading, MAX_DISTANCE); //motor schema
    return Voutput;
}
while (robot==ON)
{
    Vrunaway=runaway(reading); // motor schema
    turn(Vrunaway.direction);
    forward(Vrunaway.magnitude*MAX_VELOCITY);
}

```

4.4.5 Combination of fields and behaviors

As stated earlier in the chapter, the first attribute of a true potential fields methodology is that it requires all behaviors to be implemented as potential fields. The second attribute is that it combines behaviors not by one sub-suming others, but by vector summation. A robot will generally have forces acting on it from multiple behaviors, all acting concurrently. This section provides two examples of how multiple behaviors arise and how they are implemented and combined.

The first example is simple navigation, where a robot is heading for a goal (specified as "10.3m in direction ") and encounters an obstacle. The

motor

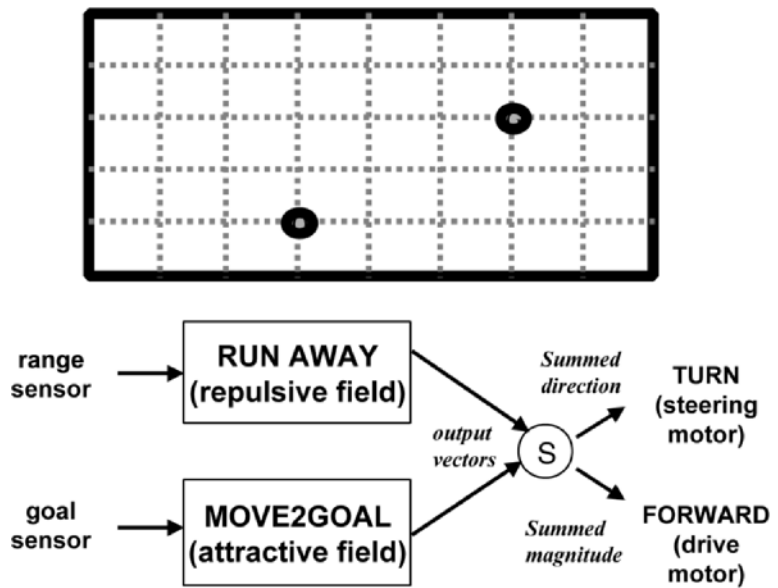


Figure 4.15 A bird's eye view of a world with a goal and obstacle, and the two active behaviors for the robot who will inhabit this world.

schema of the `move2goal` behavior is represented with an attractive potential field, which uses the shaft encoders on the robot to tell if it has reached the goal position. The `runaway` behavior is a repulsive field and uses a range sensor to detect if something is in front of it. Fig. 4.15 shows a bird's eye view of an area, and shows a visualization of the potential field. The `move2goal` behavior in Fig. 4.16b exerts an attractive field over the entire space; where ever the robot is, it will feel a force from the goal. The `runaway` behavior in Fig. 4.15 exerts a repulsive field in a radius around the obstacle (technically the repulsive field extends over all of the space as does the `move2goal`, but the magnitude of the repulsion is 0.0 beyond the radius). The combined field is shown in Fig. 4.16c.

Now consider the emergent behavior of the robot in the field if it starts in the lower right corner, shown in Fig. 4.17. At time t_0 , the robot senses the world. It can only perceive the goal and cannot perceive the obstacle, so the only vector it feels is attraction (`runaway` returns a vector with

magnitude

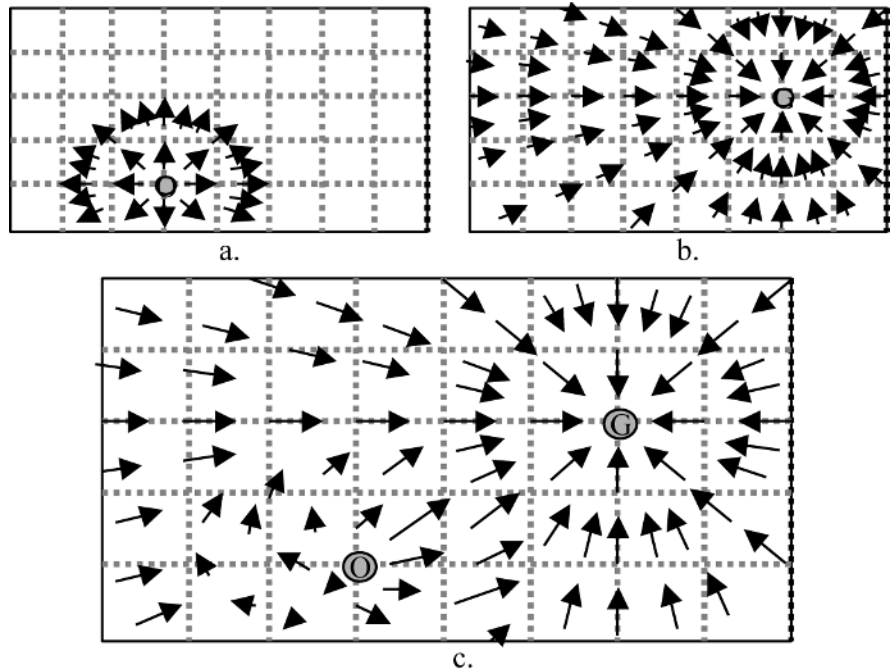


Figure 4.16 Potential fields from the world in Fig. 4.15: a.) repulsive from the obstacle, b.) attractive from the goal, and c.) combined.

of 0.0). It moves on a straight line for the goal. At t_2 , it updates its sensors and now perceives both the goal and the obstacle. Both behaviors contribute a vector; the vectors are summed and the robot now moves off course. At t_3 , the robot has almost moved beyond the obstacle and the goal is exerting the stronger force. At t_4 , it resumes course and reaches the goal.

The example illustrates other points about potential fields methods: the impact of update rates, holonomicity, and local minima. Notice that the distance (length of the arrows) between updates is different. That is due to the changes in the magnitude of the output vector, which controls the robot velocity. If the robot has a “shorter” vector, it travels more slowly and, therefore, covers less distance in the same amount of time. It can also “overshoot” as seen between t_3 and t_4 where the robot actually goes farther without turning and has to turn back to go to the goal. As a result, the path is jagged with sharp lines. The resulting path will be

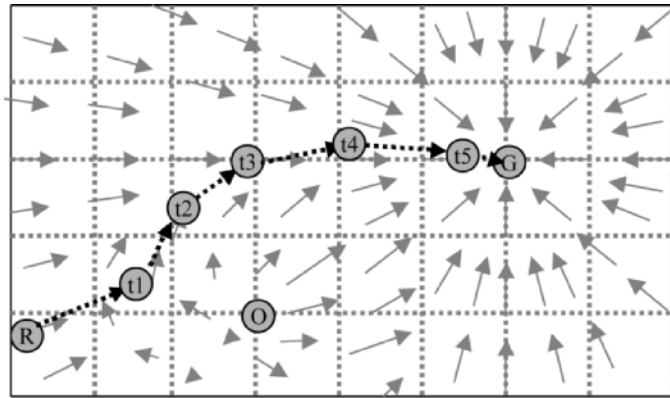


Figure 4.17 Path taken by the robot.

update interval. Another aspect of the update rate is that the robot can over- shoot the goal, especially if it is using shaft encoders (the goal is 10.3 meters from where the robot started). Sometimes designers use attractive fields with a magnitude that drops off as the robot approaches, slowing it down so the that the robot can tell with it has reached the goal. (Programmers usually put a tolerance around the goal location, for example instead of 10.3m, the goal is $10.3\text{m} \pm 0.5\text{m}$.)

Potential fields treat the robot as if it were a particle that could change velocity and direction instantaneously. This isn't true for real robots. Research robots such as Kheperas (shown in Fig. 4.18) can turn in any direction in place, but they have to be stopped and there is a measurable amount of error due to the contact between the wheels and the surface. Many robots have Ackerman, or automobile, steering, and anyone who has tried to parallel park an automobile knows that a car can go only in certain directions.

A third problem is that the fields may sum to 0.0. Returning to Fig. 4.16, draw a line between the Goal and the Obstacle. Along that line behind the Obstacle, the vectors have only a head (direction of the arrow) and no body (length of the arrow). This means that the magnitude is 0.0 and that if the robot reaches that spot, it will stop and not move again. This is called the local minima problem, because the potential field has a minima, or valley, that traps the robot. Solutions to the local minima

problem will be described at the end of the chapter.

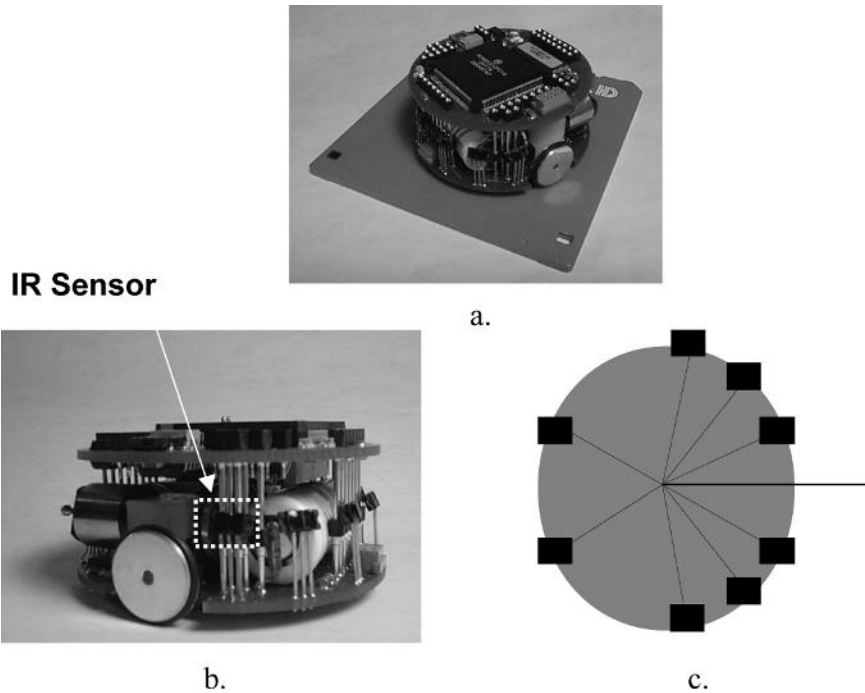


Figure 4.18 Khepera miniature robot: a.) a khepera on a floppy disk, b.) IR sensors (small black squares) along the “waistline,” and c.) orientation of the IR sensors.

4.4.6 Example using one behavior per sensor

As another example of how powerful this idea of vector summation is, it is useful to consider how obstacle avoidance runaway is commonly implemented on real robots. Fig. 4.18 shows a layout of the IR range sensors on a Khepera robot. Since the sensors are permanently mounted on the platform, their angle i relative to the front is known. If a sensor receives a range reading, something is in front of that sensor. Under a repulsive field RUN- AWAY, the output vector will be 180 opposite i . The IR sensor isn't capable of telling that the obstacle may be a little off the sensor axis, so a reading is treated as if an obstacle was straight in front of that sensor and perpendicular to it.

If this sensor were the only sensor on the robot, the RUNAWAY behavior is very straightforward. But what if, as in the case of a Khepera, the robot has

multiple range sensors? Bigger obstacles will be detected by multiple sensors at the same time. The common way is to have a RUNAWAY behavior for each sensor. This called multiple instantiations of the same behavior. Below is a code fragment showing multiple instantiations; all that had to be done is add a for loop to poll each sensor. This takes advantage of two properties of vector addition: it is associative ($a+b+c+d$ can be performed as $((a + b) + c) + d$), and it is commutative (doesn't matter what order the vectors are summed).

```
while (robot==ON) {
    vector.mag=vector.dir=0.0; //initialize to 0
    for (i=0; i<=numberIR; i++) {
        vectorCurrent=Runaway(i); // accept a sensor number
        vectorOutput = VectorSum(tempVector,vectorCurrent);
    }
    turn(vector.direction);
    forward(vector.magnitude*MAX-VELOCITY);
}
```

As seen in Fig. 4.19, the robot is able to get out of the cave-like trap called a *box canyon* without building a model of the wall. Each instance contributes a vector, some of which have a X or Y component that cancels out.

BOX CANYON

From an ethological perspective, the above program is elegant because it is equivalent to behavioral instantiations in animals. Recall from Ch. 3 the model of *rana computrix* and its real-life toad counterpart where each eye sees and responds to a fly independently of the other eye. In this case, the program is treating the robot as if it had 8 independent eyes!

From a robotics standpoint, the example illustrates two important points. First, the direct coupling of sensing to action works. Second, behavioral programming is consistent with good software engineering practices. The RUNAWAY function exhibits *functional cohesion*, where the function does one

FUNCTIONAL

thing well and every statement in the function has something directly to do with the function's purpose.¹²² Functional cohesion is desirable, because it means the function is unlikely to introduce side effects into the main program

COHESION

DATA COUPLING

or be dependent on another function. The overall organization shows

data coupling, where each function call takes a simple argument.¹²² Data coupling is good, because it means all the functions are independent; for example, the program can be easily changed to accommodate more IR sensors.

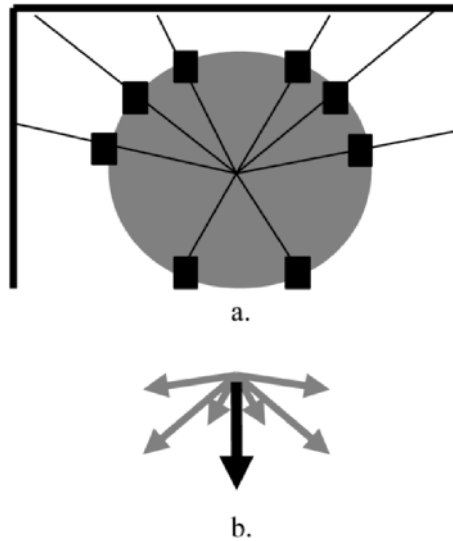


Figure 4.19 Khepera in a box canyon a.) range readings and b.) vectors from each instance of Runaway (gray) and the summed output vector.

PROCEDURAL

COHESION

The alternative to multiple instantiation is to have the perceptual schema for RUNAWAY process all 8 range readings. One approach is to sum all 8 vectors internally. (As an exercise show that the resulting vector is the same.) This is not as elegant from a software engineering perspective because the code is now specific to the robot (the function is said to have *procedural cohesion*),¹²² and can be used only with a robot that has eight range sensors at those locations. Another approach, which produces a different emergent behavior, is to have the perceptual schema return the direction and distance of the single largest range reading. This makes the behavior more selective.

4.4.7 Pfields compared with subsumption

How can simple behaviors form a more complicated emergent behavior? The same way more complicated potential fields are constructed from

4.4 Potential Fields Methodologies
primitive fields. combining multiple instantiations of primitive¹⁴⁰
behaviors. This can be

seen by revisiting the example behaviors used to describe the subsumption architecture. In the case of Level 0 in subsumption, if there are no obstacles within range, the robot feels no repulsive force and is motionless. If an obstacle comes within range and is detected by more than one sonar, each of the sonar readings create a vector, pointing the robot in the opposite direction. In the subsumption example, it could be imagined that these vectors were summed in the `RUNAWAY` module as shown in Fig. 4.20. In a potential fields system, each sonar reading would release an instance of the `RUNAWAYpf` behavior (the “pf” will be used to make it clear which runaway is being referred to). The `RUNAWAYpf` behavior uses a repulsive potential field. The output vectors would then be summed, and then the resultant vector would be used to guide the turn and forward motors.

```
while (robot==ON)
{
    vector.magnitude=vector.direction=0;
    for (i=0; i<=numberSonars; i++) {
        reading=readSonar();           //perceptual schema
        currentVector=runaway(reading); // motor schema
        vector = vectorSum(vector, currentVector);
    }
    turn(vector.direction);
    forward(vector.magnitude*MAX-VELOCITY);
}
```

The `COLLIDE` module in subsumption does not map over to a behavior in a potential fields methodology. Recall that the purpose of `COLLIDE` is to stop the robot if it touches an obstacle; in effect, if the `RUNAWAY` behavior has failed. This fits the definition of a behavior: it has a sensory input (range to obstacle = 0) and a recognizable pattern of motor activity (stop). But it doesn’t produce a potential field, unless a uniform field of vectors with 0 magnitude is permissible. If it were treated as a behavior, the vector it contributes to would be summed with any other vectors contributed by other behaviors. But a vector with 0 magnitude is the identity function for vector addition, so a `COLLISION` vector would have no impact. Instead, collisions are often treated as “panic” situations, triggering an emergency response outside the potential field framework.

Some of the subtle differences between potential fields and subsumption appear when the case of Level 2 is considered. The same functionality can be accomplished by adding only a single instance of the

WANDER behavior, as shown in Fig. 4.21. As before, the behavior generates a new direction

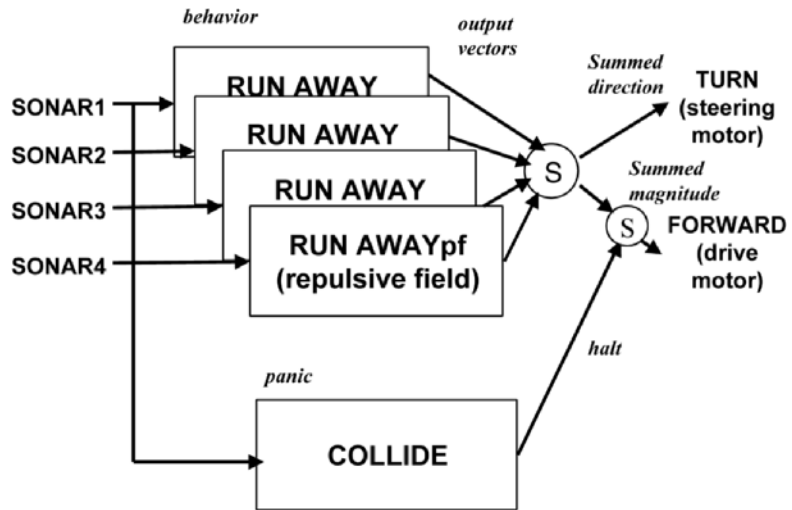


Figure 4.20 Level 0 redone as Potential Fields Methodology.

to move every n seconds. This would be represented by a uniform field where the robot felt the same attraction to go a certain direction, regardless of location, for n seconds. However, by combining the output of WANDER with the output vectors from RUNAWAYpf, the need for a new AVOID behavior is eliminated. The WANDER vector is summed with the repulsive vectors, and as a result, the robot moves both away from the obstacles and towards the desired direction. This is shown in Fig. 4.22. The primary differences in this example are that potential fields explicitly encapsulate sensing and acting into primitive behaviors, and it did not have to subsume any lower behaviors. As with subsumption, the robot became more intelligent when the WANDERpf behavior was added to the RUNAWAYpf behavior.

Now consider how Level 3, corridor following, would be implemented in a potential field system. This further illustrates the conceptual differences between the two approaches. The robot would have two concurrent behaviors: RUNAWAYpf and follow-corridor. RUNAWAYpf would remain the same as before, but WANDER would be discarded. In the parlance of potential fields, the task of following a corridor requires only two behaviors, while the task of wandering

4.4 Potential Fields Methodologies
requires two different behaviors.

144

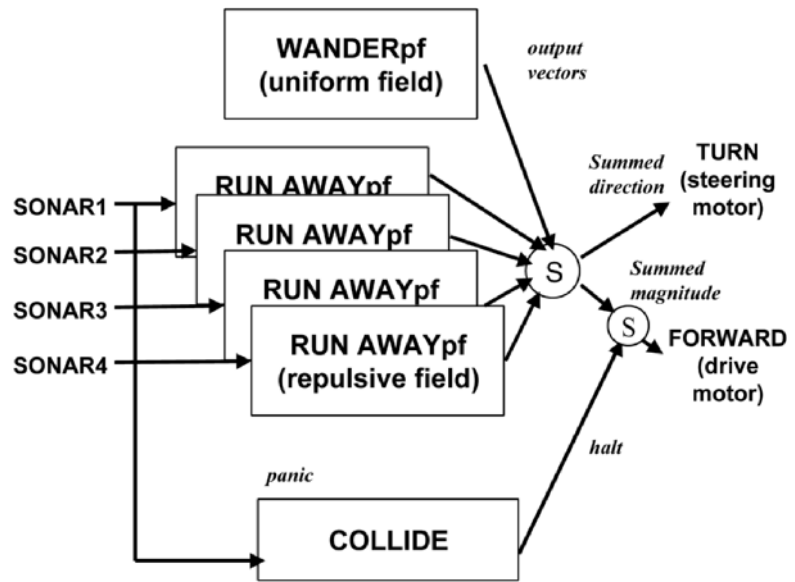


Figure 4.21 Level 1 redone with Potential Fields Methodology.

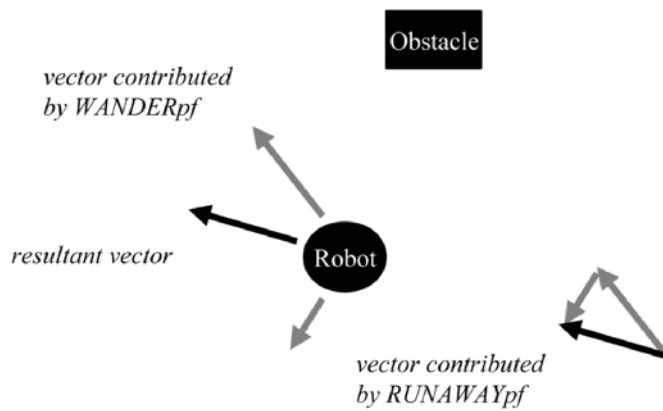


Figure 4.22 Example resultant vector of WANDERpf and RUNAWAYpf.

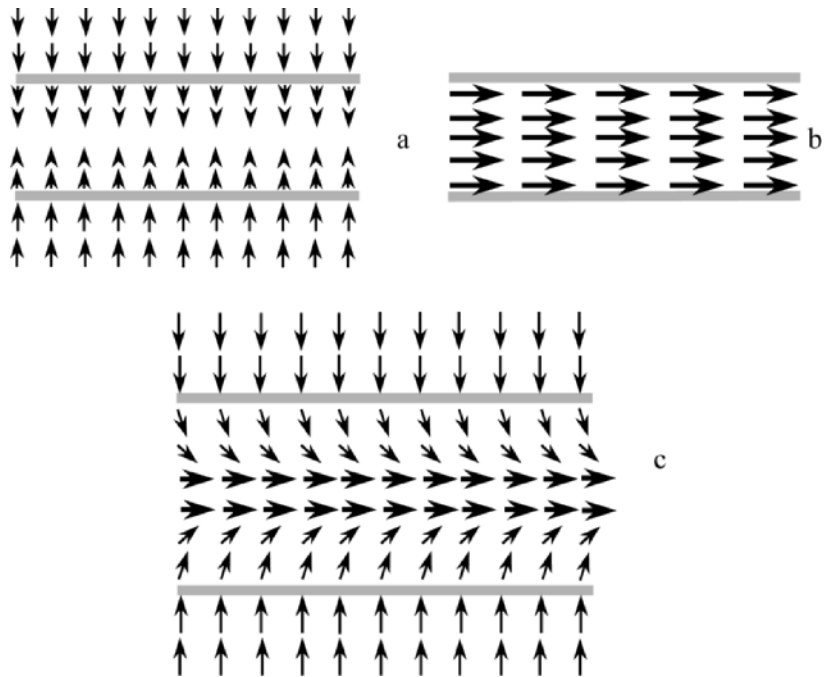


Figure 4.23 The a.) perpendicular and b.) uniform fields combining into c.) a follow-corridor field.

The follow-corridor behavior is interesting, because it requires a more complex potential field. As shown in Fig. 4.23, it would be desirable for the robot to stay in the middle of the corridor. This can be accomplished using two potential fields: a uniform field perpendicular to the left boundary and pointing to the middle, and a uniform field perpendicular to the right boundary and pointing to the middle. Notice that both fields have a linear decrease in magnitude as the field nears the center of the corridor. In practice, this taper prevents the robot from see-sawing in the middle.

Also notice that the two uniform fields are not sufficient because they do not permit the robot to move forward; the robot would move to the middle of the corridor and just sit there. Therefore, a third uniform field is added which is parallel to the corridor. All three fields combined yield a smooth field which sharply pushes the robot back to the middle of the corridor as a function of its proximity to a wall. In the meantime, the robot is constantly making forward progress. The figure below shows the

fields involved. Re-

member, in this example, the robot is not projecting past or future boundaries of the corridor; the visualization of the field makes it appear that way.

The `follow-corridor` behavior is using the same sonar data as `avoid`; therefore, walls will produce a repulsive field, which would generally push the robot onto the middle of the corridor. Why not just use a single uniform parallel field for `follow-corridor`? First, behaviors are independent. If there is a corridor following behavior, it must be able to follow halls without depending on side effects from other behaviors. Second, the polar symmetry of the repulsive fields may cause see-sawing, so there is a practical advantage to having a separate behavior.

The use of behavior-specific domain knowledge (supplied at instantiation time as an optional initialization parameter) can further improve the robot's overall behavior. If the robot knows the width of the hall *a priori*, `follow-corridor` can suppress instances of `avoid` which are using obstacles it decides form the boundary of the wall. Then it will only avoid obstacles that are in the hall. If there are no obstacles, `follow-corridor` will produce a smooth trajectory. If the obstacles are next to a wall, the `follow-corridor` will treat the profile of the obstacle as a wall and move closer to the center.

The motor schemas for a behavior may be sequenced. One example of this is the `docking` behavior.¹² Docking is when a robot moves to a specific location and orientation relative to a docking station. This is useful for robots performing materials handling in industry. In order to accept a piece of material to carry, the robot has to be close enough to the correct side of the end of a conveyor and facing the right way. Because docking requires a specific position and orientation, it can't be done with an attraction motor schema. That field would have the robot make a bee-line for the dock, even if it was coming from behind; the robot would stop at the back in the wrong position and orientation. Instead, a *selective attraction* field is appropriate. Here the robot only "feels" the attractive force when it is within a certain angular range of the docking station, as shown in Fig. 4.24.

Unfortunately selective attraction does not cover the case of when the robot approaches from behind or to the side. How does the robot move to an area where the selective attraction field can take effect? One way to do this is to have tangential field, which makes the robot orbit the dock until it gets into the selective attraction area. The combination of the two motor schema produces a very smooth field which funnels the robot into

SEQUENCING AND
PARAMETERIZING
POTENTIAL FIELDS

SELECTIVE

ATTRACTION

the correct position and orientation, as shown in Fig. 4.25.

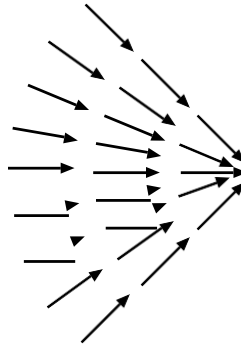


Figure 4.24 Selective attraction field, width of 45 .

An interesting aspect of the docking behavior is that the robot is operating in close proximity to the dock. The dock will also release an instance of avoid, which would prevent the robot from getting near the desired position. In this case, the docking behavior would lower the magnitude (gain) of the output vector from an avoid instantiated in the dock area. Essentially, this partially inhibits the avoid behavior in selected regions. Also, the magnitude or gain can define the correct distance: the robot stops where the selective attraction of the dock balances with the repulsion.

The selective and tangential fields are not sufficient in practice, because of the limitations of perception. If the robot can't see the dock, it can't deploy the fields. But an industrial robot might know the relative direction of a dock, much as a bee recalls the direction to its hive. Therefore an attractive force attracts the robot to the vicinity of the dock, and then when the robot sees the dock, it begins the funnel effect into the correct position and orientation, even in the presence of obstacles, as seen in Fig. 4.26.

At least three perceptual schemas are needed for the docking behavior. One is needed to extract the relative direction of the dock for the regular attraction. Another is a perceptual schema capable of recognizing the dock in general, even from behind or the sides, in order to support the tangential field. The third perceptual schema is needed for the selective attention field; it has to be able to respond to the front of the dock and extract the robot's relative distance and orientation.

The docking behavior is now defined as having three perceptual

schemas and three motor schemas (they could be grouped into 3 primitive behaviors). A schema-theoretic representation indicates that the behavior has some co-

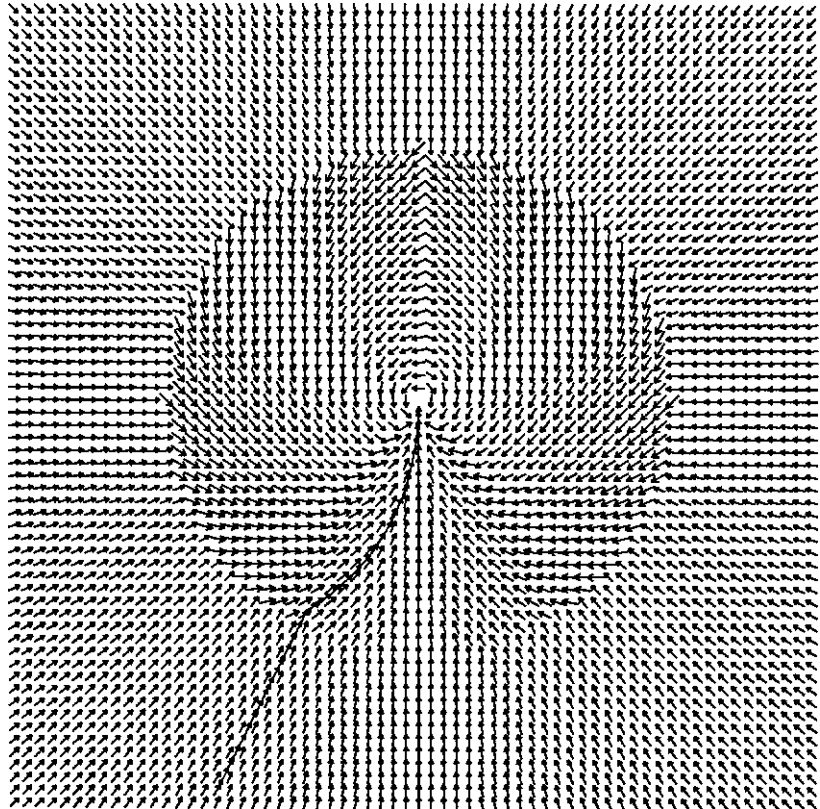


Figure 4.25 Docking potential field showing path of robot entering from slightly off course.

ordinated control program to coordinate and control these schemas. In the case of the docking behavior, a type of finite state machine is a reasonable choice for coordinating the sequence of perceptual and motor schemas. It provides a formal means of representing the sequence, and also reminds the designer to consider state transitions out of the ordinary. For instance, the robot could be moving toward the dock under the tangential plus selective attention fields when a person walks in front. This would occlude the view

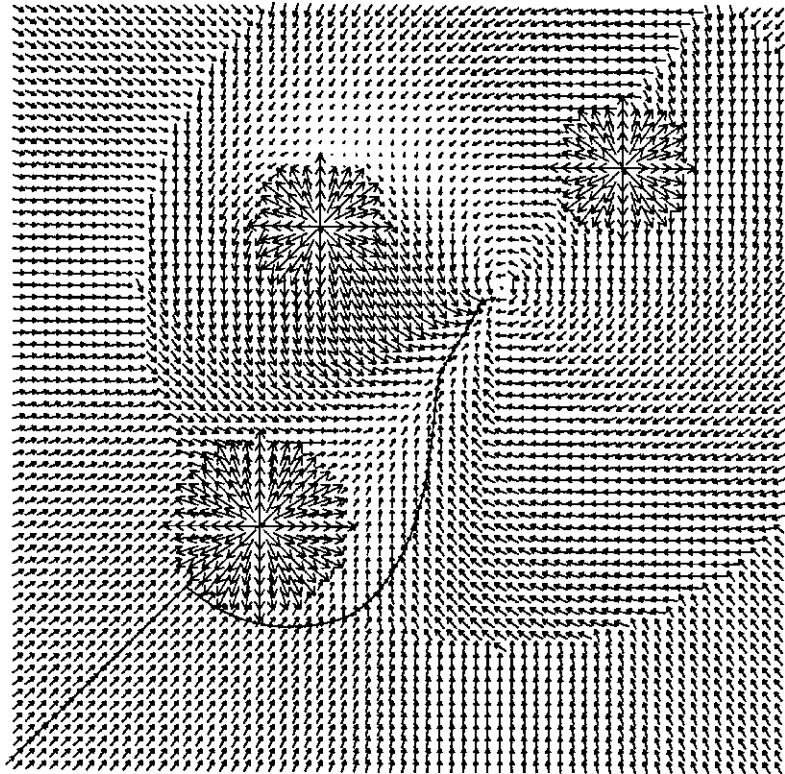


Figure 4.26 Visualization of the docking behavior with obstacles.

of the dock. The general attraction motor schema would then be reactivate, along with the associated perceptual schema to estimate the direction of the dock. This vector would allow the robot to avoid the human in a direction favorable as well as turn back towards the dock and try to re-acquire it.

The docking behavior also illustrates how the sensing capabilities of the robot impact the parameters of the motor schemas. Note that the angular size of the selective attention field would be determined by the angles at which the third perceptual schema could identify the dock. Likewise the radius of the tangential and selective attraction fields is determined by the distance at which the robot can perceive the dock.

4.4.8 Advantages and disadvantages

Potential field styles of architectures have many advantages. The potential field is a continuous representation that is easy to visualize over a large region of space. As a result, it is easier for the designer to visualize the robot's overall behavior. It is also easy to combine fields, and languages such as C++ support making behavioral libraries. The potential fields can be parameterized: their range of influence can be limited and any continuous function can express the change in magnitude over distance (linear, exponential, etc.). Furthermore, a two-dimensional field can usually be extended into a three-dimensional field, and so behaviors developed for 2D will work for 3D.

Building a reactive system with potential fields is not without disadvantages. The most commonly cited problem with potential fields is that multiple fields can sum to a vector with 0 magnitude; this is called the local minima problem. Return to Fig. 4.19, the box canyon. If the robot was being attracted to a point behind the box canyon, the attractive vector would cancel the repulsive vector and the robot would remain stationary because all forces would cancel out. The box canyon problem is an example of reaching a local minima. In practice, there are many elegant solutions to this problem. One of the earliest was to always have a motor schema producing vectors with a small magnitude from random noise.¹² The noise in the motor schema would serve to bump the robot off the local minima.

NAVIGATION

Another solution is that of *navigation templates* (NaTs), as implemented by Marc Slack for JPL. The motivation is that the local minima problem most often arises because of interactions between the avoid behavior's repulsive field and other behaviors, such as move-to-goal's attractive field. The minima problem would go away if the avoid potential field was somehow smarter. In NaTs, the avoid behavior receives as input the vector summed from the other behaviors. This vector represents the direction the robot would go if there were no obstacles nearby. For the purposes of this book, this will be referred to as the strategic vector the robot wants to go. If the robot has a strategic vector, that vector gives a clue as to whether an obstacle should be passed on the right or the left. For example, if the robot is crossing a bridge (see Fig. 4.27), it will want to pass to the left of obstacles on its right in order to stay in the middle. Note that the strategic vector defines what is left and what is right.

TEMPLATES

NaTs implement this simple heuristic in the potential field for

4.4 Potential Fields Methodologies
RUNAWAY, promoting it to a true AVOID. The repulsion field is now
supplemented with a tangential orbit field. The direction of the orbit
(clockwise or counter-

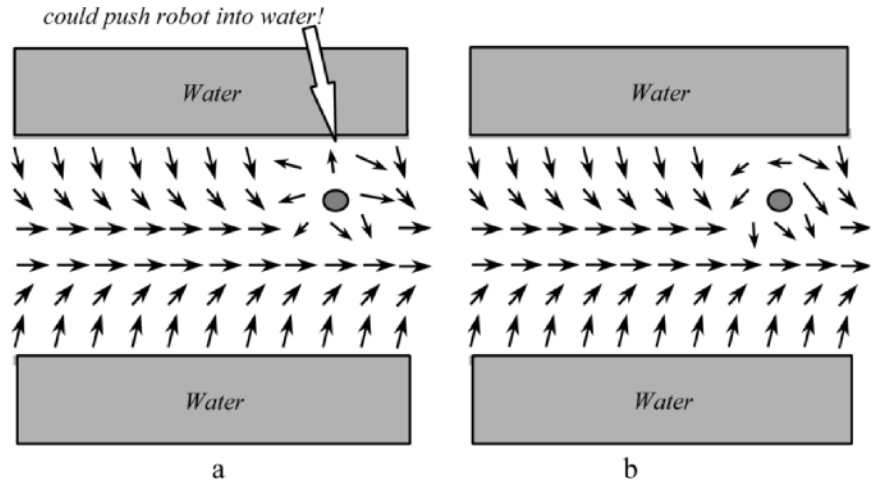


Figure 4.27 Problem with potential fields: a.) an example, and b.) the use of NaTs to eliminate the problem.

clockwise) is determined by whether the robot is to the right or left of the strategic vector. The output of the avoid behavior can be called a tactical vector, because it carries out the strategic goal of the robot in the face of immediate challenges, just as a military general might tell a captain to capture a fort but not specify each footstep.

A more recent solution to the local minima problem has been to express the fields as harmonic functions.⁴⁰ Potential fields implemented as harmonic functions are guaranteed not to have a local minima of 0. The disadvantage of this technique is that it is computationally expensive, and has to be implemented on a VLSI chip in order to run in real-time for large areas.

To summarize the major points about potential fields architectures:

Behaviors are defined as consisting of one or more of both motor and perceptual schemas and/or behaviors. The motor schema(s) for a behavior must be a potential field.

All behaviors operate concurrently and the output vectors are summed. Behaviors are treated equally and are not layered, although as will be seen in Ch. 5, there may be abstract behaviors which

haviors. The coordinated control program is not specified; the designer can use logic, finite state machines, whatever is deemed appropriate. Sequencing is usually controlled by perceived cues or affordances in the environment, which are releasers.

Although all behaviors are treated equally, behaviors may make varying contributions to the overall action of the robot. A behavior can change the gains on another behavior, thereby reducing or increasing the magnitude of its output. This means that behaviors can inhibit or excite other behaviors, although this is rarely used in practice.

Perception is usually handled by direct perception or affordances.

Perception can be shared by multiple behaviors. *A priori* knowledge can be supplied to the perceptual schemas, to emulate a specialized sensor being more receptive to events such as hall boundary spacing.

4.5 Evaluation of Reactive Architectures

As seen by the follow-corridor example, the two styles of architectures are very similar in philosophy and the types of results that they can achieve. Essentially, they are equivalent.

In terms of support for modularity, both decompose the actions and perceptions needed to perform a task into behaviors, although there is some disagreement over the level of abstraction of a behavior. Subsumption seems to favor a composition suited for a hardware implementation, while potential fields methods have nice properties for a software-oriented system.

The niche targetability is also high for both, assuming that the task can be performed by reflexive behaviors. Indeed, the use of direct perception emphasizes that reactive robots are truly constructed to fill a niche.

The issue of whether these architectures show an ease of portability to other domains is more open. Reactive systems are limited to applications which can be accomplished with reflexive behaviors. They cannot be transferred to domains where the robot needs to do planning, reasoning about resource allocation, etc. (this led to the Hybrid Paradigm to be described in Ch. 7). In practice, very few of the subsumption levels can be ported to new applications of navigating in an environment without some changes. The different applications create layers which need to

subsume the lower layers differently. The potential fields methodology performs a bit better in that the

designer can create a library of behaviors and schemas to choose from, with no implicit reliance on a lower layer.

Neither architecture presents systems which could be called genuinely robust. The layering of subsumption imparts some graceful degradation if an upper level is destroyed, but it has no mechanisms to notice that a degradation has occurred. The finite state mechanisms in the docking behavior show some resilience, but again, only for situations which can be anticipated and incorporated into the state diagram. As with animals, a reactive robot will also do something consistent with its perception of the world, but not always the right thing.

4.6 Summary

Under the Reactive Paradigm, systems are composed of behaviors, which tightly couple sensing and acting. The organization of the Reactive Paradigm is SENSE-ACT or S-A, with no PLAN component. Sensing in the Reactive Paradigm is local to each behavior, or behavior-specific. Each behavior has direct access to one or more sensors independently of the other behaviors. A behavior may create and use its own internal world representation, but there is no global world model as with the Hierarchical Paradigm. As a result, reactive systems are the fastest executing robotic systems possible.

There are four major characteristics of robots constructed under the Reactive Paradigm. Behaviors serve as the basic building blocks for robot actions, even though different designers may have different definitions of what a behavior entails. As a consequence of using behaviors, the overall behavior of the robot is emergent. Only local, behavior-specific sensing is permitted. The use of explicit representations in perceptual processing, even locally, is avoided in most reactive systems. Explicit representations of the world are often referred to as maintaining the state of the world internally, or internal state. Instead, reactive behaviors rely on the world to maintain state (as exemplified by the gripper controlling whether the robot was looking for soda cans or for the recycling bin). Animal models are often cited as a basis for a behavior or the architecture. Behaviors and groups of behaviors which were inspired by or simulate animal behavior are often considered desirable and more interesting than hacks. Finally, reactive systems exhibit good software engineering principles due to the “programming by behavior” ap-

proach. Reactive systems are inherently modular from a software design perspective. Behaviors can be tested independently, since the overall behavior-

ior is emergent. More complex behaviors may be constructed from primitive behaviors, or from mixing and matching perceptual and motor components. This supports good software engineering practices, especially low coupling and high cohesion.

The subsumption architecture is a popular reactive system. Behaviors are purely reflexive and may not use memory. Behaviors are arranged in layers of competence, where the lower levels encapsulate more general abilities. The coordination of layers is done by higher layers, which have more specific goal-directed behaviors, subsuming lower layers. Behaviors within a layer are coordinated by finite state automata, and can be readily implemented in hardware.

Potential fields methodologies are another popular reactive system. Behaviors in potential field systems must be implemented as potential fields. All active behaviors contribute a vector; the vectors are summed to produce a resultant direction and magnitude for travel. Pfields provide a continuous representation, which is easier to visualize than rule encoding, and are continuous. The fields can be readily implemented in software, and parameterized for flexibility and reuse. The vector summation effect formalizes how to combine behaviors, eliminating issues in how to design behaviors for subsumption. The fields are often extensible to three dimensions, adding to the re-usability and portability. In the example in this chapter, behaviors using potential fields were able to encapsulate several layers in subsumption into a set of concurrent peer behaviors with no layers. Ch. 5 will give examples of how to sequence, or assemble, behaviors into more abstract behaviors.

Despite the differences, subsumption and potential fields appear to be largely equivalent in practice. Both provide support for modularity and niche targetability. The ease of portability to other domains is relative to the complexity of the changes in the task and environment. Neither style of architecture explicitly addresses robustness, although in theory, if only a higher layer of a subsumption system failed, the lower layers should ensure robot survivability.

5 *Designing a Reactive*

Chapter objectives:

Use schema theory to design and program behaviors using object-oriented programming principles.

Design a complete behavioral system, including coordinating and controlling multiple concurrent behaviors.

For a given behavioral system, draw a *behavioral table* specifying the releasers, perceptual schemas, and motor schemas for each behavior.

Describe the two methods for assembling and coordinating primitive behaviors within an *abstract behavior*: *finite state automata* and *scripts*. Be able to represent a sequence of behaviors with a *state diagram* or with a pseudo-code *script*.

5.1 Overview

By this point, the sheer simplicity and elegance of reactive behaviors tend to incite people to start designing and implementing their own robots. Kits such as Lego Mindstorms and the Rug Warrior permit the rapid coupling of sensors and actuators, enabling users to build reactive behaviors. However, the new issues are how to program more intelligence into the software and how to exploit better sensors than come with kits. Unfortunately, good intentions in robot programming are often frustrated by two deficits. First, designing behaviors tends to be an art, not a science. Novice roboticists often are uncertain as to how to even start the design process, much less how to know when they've got a reasonable system. The second deficit is more subtle. Once the designer has a few well-designed and tested behaviors, how

EMERGENT BEHAVIOR

are they integrated into a system? The Reactive movement and early work described in Ch. 4 was characterized by robots running a very small set of behaviors which were combined internally to produce an overall *emergent behavior*. The key components of a reactive architecture were shown to be the behaviors plus the mechanism for merging the output of the concurrent behaviors.

SERIES OF BEHAVIORS

However, many applications are best thought of as a *series of behaviors*, each operating in a recognizable sequence. One of the first popular applications that many roboticists looked at was picking up and disposing of an empty soft drink can. This involved search for a can, move toward the can when it is found, pick up the can, search for the recycle bin, move toward the recycle bin, drop the can.^{39;129;66} It's counterintuitive to think of these behaviors as being concurrent or merged. (There is certainly the possibility of concurrency, for example avoiding obstacles while moving to the soda can or recycle bin.) Therefore, new techniques had to be introduced for controlling sequences of behaviors. Most of these techniques are conceptually the equivalent of constructing a macro behavior, where the schema structure is used recursively to simplify programming the control program.

This chapter attempts to aid the novice designer in constructing a reactive robot system by addressing each of these two deficits. First, an object-oriented programming approach to designing behaviors is introduced. This approach is based on schema theory, which was introduced in Ch. 3. The "art" of design is presented in a flow chart following the waterfall method of software engineering, along with a case study of a winning entry in the 1994 International Association for Unmanned Systems' Unmanned Ground Robotics Competition. The case study emphasizes the importance of establishing the ecological niche of a reactive robot. Second, two techniques for managing sequences of behaviors are introduced: *finite-state automata* and *scripts*. As could be expected from the material presented in Ch. 3, both of these techniques will look very similar to the Innate Releasing Mechanisms from Tinbergen and Lorenz. Finally, the chapter shows how these techniques were applied to entries in the "Pick Up the Trash" events of the 1994 AAAI and 1995 IJCAI Mobile Robot Competitions. The focus of these examples is how the logic for coordinating a sequence of behaviors is developed, represented, and implemented. The use of schemas to "factor out" and program a small set of generic behaviors rather than designing a large set of specialized behaviors is emphasized throughout

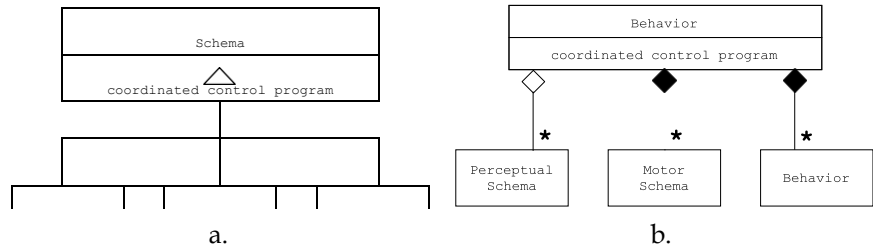


Figure 5.1 Classes: a.) schema and b.) behavior.

5.2 Behaviors as Objects in OOP

Although Object-Oriented Programming (OOP) had not become popular during the time that the Reactive Paradigm was developed, it is useful to cast behaviors in OOP terms. Schema theory is well suited for transferring theoretical concepts to OOP. Furthermore, schema theory will be used as a bridge between concepts in biological intelligence and robotics, enabling a practical implementation of reactivity exploiting innate releasing mechanisms and affordances.

Recall from software engineering that an object consists of data and methods, also called attributes and operations. As noted in Ch. 3, schemas contain specific knowledge and local data structures and other schemas. Fig. 5.1 shows how a schema might be defined. Following Arbib,⁶ a schema as a programming object will be a class. The class will have an optional method called a *coordinated control program*. The coordinated control program is a function that coordinates any methods or schemas in the derived class.

Three classes are derived from the Schema Class: Behavior, Motor Schema, and Perceptual Schema. Behaviors are composed of at least one Perceptual Schema and one Motor Schema; these schemas act as the methods for the Behavior class. A Perceptual Schema has at least one method; that method

takes sensor input and transforms it into a data structure called a *percept*. A Motor Schema has at least one method which transforms the percept into a vector or other form of representing an action. Since schemas are independent, the Behavior object acts as a place holder or local storage area for the percept. The Perceptual Schema is linked to the sensor(s), while the Motor Schema is linked to the robot's

COORDINATED

CONTROL PROGRAM

PERCEPT

actuators. The sensors and actuators can be represented by their own software classes if needed; this is useful when working with software drivers for the hardware.

Using the Unified Modeling Language representation,⁵⁵ the Schema and Behavior classes look like Fig. 5.1. The OOP organization allows a behavior to be composed of multiple perceptual schema and motor schema and even behaviors. Another way of stating this is that the definition of a behavior is recursive. Why is it useful to have multiple perceptual schema and motor schema? In some cases, it might be helpful to have two perceptual schema, one for say daylight conditions using a TV camera and one for nighttime using infra-red. Sec. 5.2.2 provides a more detailed example of why multiple schemas in a behavior can be helpful.

PRIMITIVE BEHAVIOR

Recall that a *primitive behavior* is composed of only one perceptual schema and one motor schema; there is no need to have any coordinated control program. Primitive behaviors can be thought of being *monolithic*, where they do only one (“mono”) thing. Because they are usually a simple mapping from stimulus to response, they are often programmed as a single method, not composed from multiple methods or objects. The concept of Perceptual and Motor Schema is there, but hidden for the sake of implementation.

ABSTRACT BEHAVIORS

Behaviors which are assembled from other behaviors or have multiple perceptual schema and motor schema will be referred to as *abstract behaviors*, because they are farther removed from the sensors and actuators than a primitive behavior. The use of the term “abstract behavior” should not be confused with an abstract class in OOP.

5.2.1 Example: A primitive move-to-goal behavior

This example shows how a primitive behavior can be designed using OOP principles. In 1994, the annual AAAI Mobile Robot Competitions had a “Pick Up the Trash” event, which was repeated in 1995 at the joint AAAI-IJCAI Mobile Robot Competition.^{129;66} The basic idea was for a robot to be placed in an empty arena about the size of an office. The arena would have Coca-Cola cans and white Styrofoam cups placed at random locations. In two of the four corners, there would be a blue recycling bin; in the other two, a different colored trash bin. The robot who picked up the most trash and placed them in the correct bin in the allotted time was the winner. In most years, the strategy was to find and recycle the Coca-Cola cans first, because it was easier for the robot’s vision processing algorithms to perceive red and blue.

One of the most basic behaviors needed for picking up a red soda can

5.2 Behaviors as Objects in OOP 161
and moving to a blue bin is `move_to_goal`. When the robot sees a red
can, it must move to it. When it has a can, it must find and then move to
a blue bin.

It is better software engineering to write a general move to goal behavior, where only what is the goal—a red region or a blue region—varies. The goal for the current instance can be passed in at instantiation through the object constructor.

Writing a single generic behavior for `move_to_goal(color)` is more desirable than writing a `move_to_red` and a `move_to_blue` behaviors. From a software engineering perspective, writing two behaviors which do the same thing is an opportunity to introduce a programming bug in one of them and not notice because they are supposed to be the same. Generic behaviors also share the same philosophy as factoring in mathematics. Consider simplifying the equation $45x^2 + 90x + 45$. The first step is to factor out any common term to simplify the equation. In this case, 45 can be factored and the equation rewritten as $45(x + 1)^2$. The color of the goal, red or blue, was like the common coefficient of 45; it is important, but tends to hide that the key to the solution was the move-to-goal part, or x .

Modular, generic code can be handled nicely by schemas as shown in Fig. 5.2. The behavior `move_to_goal` would consist of a perceptual schema, which will be called `extract-goal`, and a motor schema, which uses an attractive field called `pfields.attraction`. `extract-goal` uses the affordance of color to extract where the goal is in the image, and then computes the angle to the center of the colored region and the size of the region. This information forms the percept of the goal; the affordance of the Coke can is the color, while the information extracted from the perception is the angle and size. The attraction motor schema takes that percept and is responsible for using it to turn the robot to center on the region and move forward. It can do this easily by using an attractive field, where the larger the region, the stronger the attraction and the faster the robot moves.

The `move_to_goal` behavior can be implemented as a primitive behavior, where `goal_color` is a numerical means of representing different colors such as red and blue:

`move to goal(goal_color):`

Object	Behavioral Analog	Identifier
Data	percept	goal_angle goal_strength
Methods	perceptual_schema motor_schema	extract_goal(goal_color) pfields.attraction(goal_angle, goal_strength)

The above table implies some very important points about

AFFORDANCE

The behavior is the “glue” between the perceptual and motor schemas. The schemas don’t communicate in the sense that both are independent entities; the perceptual schema doesn’t know that the motor schema exists. Instead, *the behavior puts the percept created by the perceptual schema in a local place where the motor schema can get it.*

Behaviors can (and should) use libraries of schemas. The `pfields` suffix on the `pfields.attraction()` meant that attraction was a method within another object identified as `pfields`. The five primitive potential fields could be encapsulated into one class called `PFIELDS`, which any motor schema could use. `PFIELDS` would serve as a library. Once the potential fields in `PFIELDS` were written and debugged, the designer doesn’t ever have to code them again.

Behaviors can be reused if written properly. In this example, the move to goal behavior was written to accept a structure (or object) defining a color and then moving to a region of that color. This means the behavior can be used with both red Coke cans and blue trash cans.

5.2.2 Example: An abstract follow-corridor behavior

The move to goal example used a single motor schema with a single perceptual schema. This example shows how a potential fields methodology can be implemented using schemas. In the corridor following example in Ch. 4, the `follow_corridor` potential field consisted of two primitive fields: two instances of perpendicular to the walls and one uniform parallel to the walls. The `follow_corridor` field could be implemented in schemas in at least two different ways as shown in Fig. 5.2. In one way, each of the primitive fields would be a separate motor schema. The follow corridor motor schema would consist of the three primitives and the coordinated control program. The coordinated control program would be the function that knows that one field is perpendicular from the left wall going towards the center of the corridor, which way is forward, etc. They were summed together by the coordinated control program in the behavioral schema to produce a single output vector. The perceptual schema for the follow corridor would examine the sonar polar plot and extract the relative location of the corridor walls. The perceptual schema would return the distance to the left wall and the right wall.

Another way to have achieved the same overall behavior is to have

fol- low_wall composed of two instances of a follow wall behavior:
follow_-

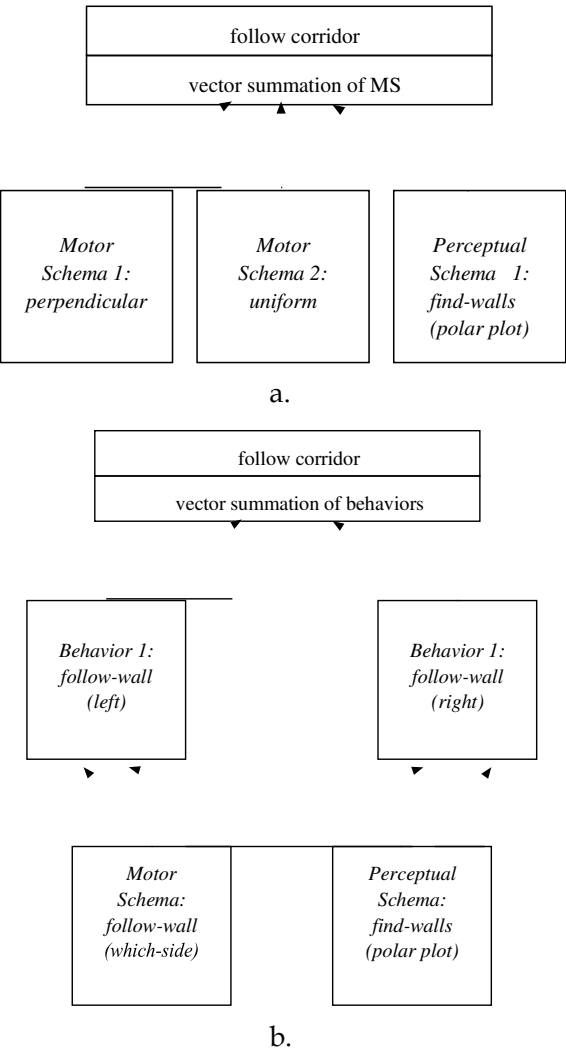


Figure 5.2 Class diagrams for two different implementations of `follow_corridor`: a.) use of primitive fields, and b.) reuse of fields grouped into a `follow_wall` behavior.

`wall (left)` and `follow_wall(right)`. Each instance of `follow_wall` would receive the sonar polar plot and extract the relevant wall. The associated class diagram is shown on the right in Fig. 5.2.

In both implementations, the motor schema schemas ran continuously and the vectors were summed internally in order to produce a single output vector. Since there were multiple motor schemas, the coordinated control program for follow-corridor is not null as it was for move-to-goal. The vector summation and the concurrency form the conceptual coordinated control program in this case.

5.2.3 Where do releasers go in OOP?

The previous examples showed how behaviors can be implemented using OOP constructs, such as classes. Another important part of a behavior is how

TWO PURPOSES OF

it is activated. As was discussed in Ch. 3, perception serves two purposes:

PERCEPTION *to release a behavior and to guide it.* Perceptual schemas are clearly used for guiding the behavior, either moving toward a distinctively colored goal or following a wall. But what object or construct contains the releaser and how is it “attached” to the behavior?

The answer to the first part of the question is that *the releaser is itself a perceptual schema*. It can execute independently of whatever else is happening with the robot; it is a perceptual schema not bound to a motor schema. For example, the robot is looking for red Coke cans with the `extract_color` perceptual schema. One way to implement this is when the schema sees red, it can signal the main program that there is red. The main program can determine that that is the releaser for the move to goal behavior has been satisfied, and instantiate `move_to_goal` with `goal=red`. `move_to_goal` can instantiate a new instance of `extract_color` or the main program can pass a pointer to the currently active `extract_color`. Regardless, `move_to_goal` has to instantiate `pfield.attraction`, since the attraction motor schema wouldn’t be running. In this approach, the main program is responsible for calling the right objects at the right time; the releaser is attached to the behavior by the designer with little formal mechanisms to make sure it is correct. This is awkward to program.

Another, more common programming approach is to have the releaser be part of the behavior: the single perceptual schema does double duty.

This programming style requires a coordinated control program. The behavior is always active, but if the releaser isn't satisfied, the coordinated control program short-circuits processing. The behavior returns an identity function,

in the case of potential fields, a vector of (0.0,0.0), which is the same as if the behavior wasn't active at all. This style of programming can tie up some resources, but is generally a simple, effective way to program. Fig. 5.2 shows the two approaches.

Either way, once the robot saw red, the observable aspect of move to goal (e.g., moving directly toward the goal) would commence. The extract goal schema would update the percept data (relative angle of the goal and size of red region) every time it was called. This percept would then be available to the motor schema, which would in turn produce a vector.

As will be covered in Sec. 5.5, the releaser must be designed to support the correct sequence. Depending where the robot was in the sequence of activities, the robot uses move to goal to move to a red Coke can or a blue recycling bin. Otherwise, the robot could pursue a red Coke can and a blue recycling bin simultaneously. There is nothing in the OOP design to prevent that from happening—in fact, OOP makes it easy. In this situation, there would be two move to goal objects, one instantiated with goal of “red” and the other with goal of “blue.” Notice that the move to goal behavior can use any perceptual schema that can produce a goal angle and goal strength. If the robot needed to move to a bright light (phototropism), only the perceptual schema would need to be changed. This is an example of software reusability.

5.3 Steps in Designing a Reactive Behavioral System

Fig. 5.3 shows the steps in designing a reactive behavioral system, which are taken from *Behavior-Based Robotics*¹⁰ and a case study by Murphy.⁹⁸ This section will first give a broad discussion of the design process, then work through each step using the winning approach taken in the 1994 Unmanned Ground Vehicle Competition.

The methodology in Fig. 5.3 assumes that a designer is given a task for the robot to do, and a robot platform (or some constraints, if only budgetary). The goal is to design a robot as a situated agent.

Therefore, the first three steps serve to remind the designer to specify the *ecological niche* of the robot. The fourth step begins the iterative process of identifying and refining the set of behaviors for the task. It asks the question: *what does the robot do?* Defining the ecological niche defines constraints and opportunities but doesn't necessarily introduce

major insights into the *situatedness* of the robot: *how it acts and reacts to the range of variability in its ecological niche*. This step is where a novice begins to recognize that designing behaviors is an art. Sometimes,

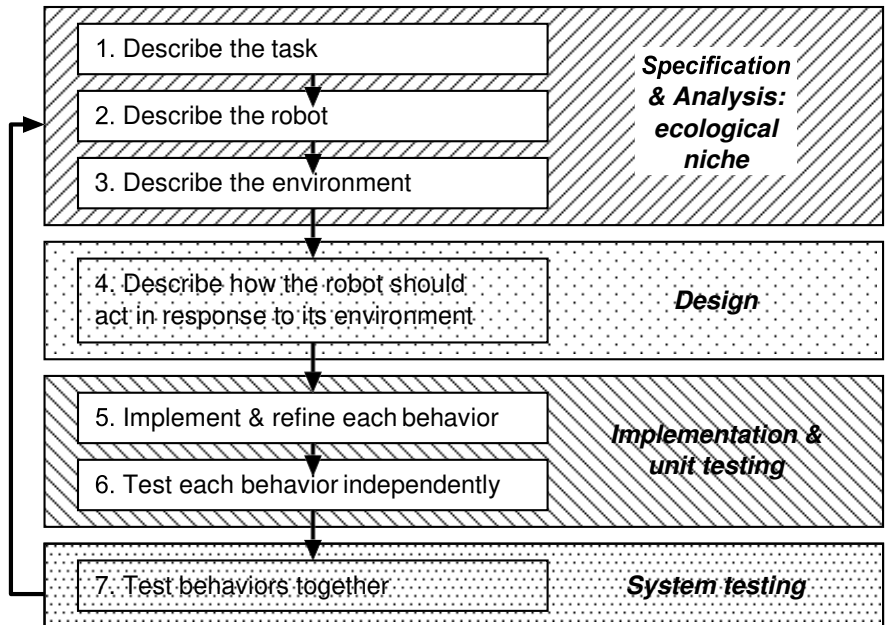


Figure 5.3 Steps in designing a reactive behavioral system, following basic software engineering phases.

a behavioral decomposition appears obvious to a roboticist after thinking about the ecological niche. For example, in the 1994 and 1995 Pick Up the Trash events, most of the teams used a partitioning along the lines of: random search until see red, move to red, pick up can, random search until see blue, move to blue, drop can.

Roboticists often attempt to find an analogy to a task accomplished by an animal or a human, then study the ethological or cognitive literature for more information on how the animal accomplishes that class of tasks. This, of course, sidesteps the question of how the roboticist knew what class of animal tasks the robot task is similar to, as well as implies a very linear thinking process by roboticists. In practice, roboticists who use biological and cognitive insights tend to read and try to stay current with the ethological literature so that they can notice a connection later on.

Steps 5-7 are less abstract. Once the candidate set of behaviors has been proposed, the designer works on designing each individual behavior, spec-

ifying its motor and perceptual schemas. This is where the designer has to write the algorithm for finding red blobs in a camera image for the random search until find red and move to red behaviors. The designer usually programs each schema independently, then integrates them into a behavior and tests the behavior thoroughly in isolation before integrating all behaviors. This style of testing is consistent with good software engineering principles, and emphasizes the practical advantages of the Reactive Paradigm.

The list of steps in implementing a reactive system can be misleading. Despite the feedback arrows, the overall process in Fig. 5.3 appears to be linear. In practice, it is iterative. For example, a supposed affordance may be impossible to detect reliably with the robot's sensors, or an affordance which was missed in the first analysis of the ecological niche suddenly surfaces. The single source of iteration may be testing all the behaviors together in the "real world." Software that worked perfectly in simulation often fails in the real world.

5.4 Case Study: Unmanned Ground Robotics Competition

This case study is based on the approach taken by the Colorado School of Mines team to the 1994 Unmanned Ground Robotics Competition.⁹⁸ The objective of the competition was to have a small unmanned vehicle (no larger than a golf cart) autonomously navigate around an outdoor course of white lines painted on grass. The CSM entry won first place and a \$5,000 prize. Each design step is first presented in boldface and discussed. What was actually done by the CSM team follows in italics. This case study illustrates the effective use of only a few behaviors, incrementally developed, and the use of affordances combined with an understanding of the ecological niche. It also highlights how even a simple design may take many iterations to be workable.

Step 1: Describe the task. The purpose of this step is to specify what the robot has to do to be successful.

The task was for the robot vehicle to follow a path with hair pin turns, stationary obstacles in the path, and a sand pit. The robot which went the furthest without going completely out of bounds was the winner, unless two or more robots went the same distance or completed the course, then the winner was whoever went the fastest. The maximum speed was 5 mph. If the robot went partially out of bounds (one wheel or a portion of a tread remained inside), a distance penalty was subtracted. If the robot

hit an obstacle enough to move it, another distance penalty was levied. Therefore,

the competition favored an entry which could complete the course without accruing any penalties over a faster entry which might drift over a boundary line or bump an obstacle. Entrants were given three runs on one day and two days to prepare and test on a track near the course; the times of the heats were determined by lottery.

Step 2: Describe the robot. The purpose of this step is to determine the basic physical abilities of the robot and any limitations. In theory, it might be expected that the designer would have control over the robot itself, what it could do, what sensors it carries, etc. In practice, most roboticists work with either a commercially available research platform which may have limitations on what hardware and sensors can be added, or with relatively inexpensive kit type of platform where weight and power restrictions may impact what it can reasonably do. Therefore, the designer is usually handed some fixed constraints on the robot platform which will impact the design.

In this case, the competition stated that the robot vehicle had to have a footprint of at least 3ft by 3.5ft but no bigger than a golf cart. Furthermore, the robot had to carry its own power supply and do all computing on-board (no radio communication with an off-board processor was permitted), plus carry a 20 pound payload.

The CSM team was donated the materials for a robot platform by Omnitech Robotics, Inc. Fig. 5.4 shows Omnibot. The vehicle base was a Power Wheels battery powered children's jeep purchased from a toy store. The base met the minimum footprint exactly. It used Ackerman (car-like) steering, with a drive motor powering the wheels in the rear and a steering motor in the front. The vehicle had a 22 turning angle. The on-board computing was handled by a 33MHz 486 PC using Omnitech CANAMP motor controllers. The sensor suite consisted of three devices: shaft encoders on the drive and steer motors for dead reckoning, a video camcorder mounted on a mast near the center of the vehicle and a panning sonar mounted below the grille on the front. The output from the video camcorder was digitized by a black and white framegrabber. The sonar was a Polaroid lab grade ultrasonic transducer. The panning device could sweep 180°. All coding was done in C++.

Due to the motors and gearing, Omnibot could only go 1.5 mph. This limitation meant that it could only win if it went farther with less penalty points than any other entry. It also meant that the steering had to have at least a 150ms update rate or the robot could veer out of bounds without ever perceiving it was going off course. The black and white framegrabber eliminated the use of color. Worse yet, the update rate of the framegrabber was almost 150ms; any vision processing algorithm would have to be very fast or else the robot would be moving faster than it could react. The reflections from uneven grass reduced the standard range of the sonar from 25.5 ft to about 10 ft.



Figure 5.4 Omnibot, a mobile robot built from a Power Wheels battery-powered toy jeep by students and Omnitech Robotics, Inc.

Step 3: Describe the Environment. This step is critical for two reasons. First, it is a key factor in determining the situatedness of the robot. Second, it identifies perceptual opportunities for the behaviors, both in how a perceptual event will instantiate a new behavior, and in how the perceptual schema for a behavior will function. Recall from Chapter 4 that the Reactive Paradigm favors direct perception or affordance-based perception because it has a rapid execution time and involves no reasoning or memory.

The course was laid out on a grassy field with gentle slopes. The course consisted of a 10 foot wide lane marked in US Department of Transportation white paint, roughly in the shape of a kidney (see Fig. 5.5). The exact length of the course and layout of obstacles of the course were not known until the day of the competition, and teams were not permitted to measure the course or run trials on it. Obstacles were all stationary and consisted of bales of hay wrapped in either white or red plastic. The bales were approximately 2 ft by 4 ft and never extended more than 3 feet into the lane. The sonar was able to reliably detect the plastic covered bales at most angles of approach at 8 feet away. The vehicles were scheduled to run between 9am and 5pm on May 22, regardless of weather or cloud cover. In addition to the visual challenges of changing lighting due to clouds, the bales introduced shadows on the white lines between 9–11am and 3–5pm. The sand pit was only 4 feet long and placed on a straight segment of the course.

The analysis of the environment offered a simplification of the task. The placing of the obstacles left a 4 ft wide open area. Since Omnibot was only 3 ft wide, the course could be treated as having no obstacles if the robot could stay in the center of the lane

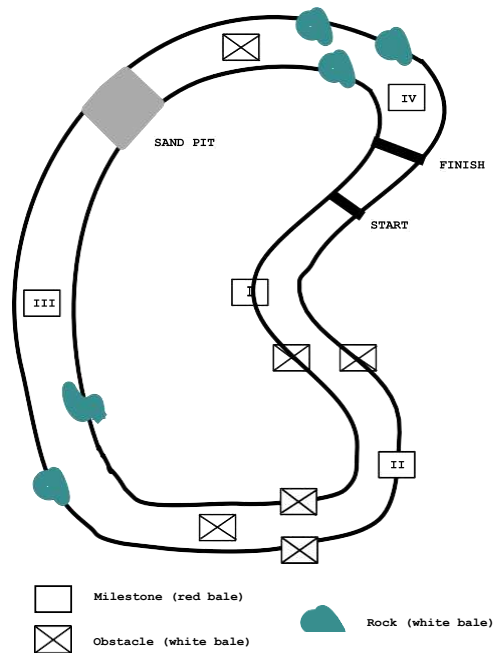


Figure 5.5 The course for the 1994 Ground Robotics Competition.

with a 0.5 ft tolerance. This eliminated the need for an avoid obstacle behavior.

The analysis of the environment also identified an affordance for controlling the robot. The only object of interest to the robot was the white line, which should have a high contrast to the green (dark gray) grass. But the exact lighting value of the white line changed with the weather. However, if the camera was pointed directly at one line, instead of trying to see both lines, the majority of the brightest points in the image would belong to the line (this is a reduction in the signal to noise ratio because more of the image has the line in it). Some of the bright points would be due to reflections, but these were assumed to be randomly distributed. Therefore, if the robot tried to keep the centroid of the white points in the center of the image, it would stay in the center of the lane.

Step 4: Describe how the robot should act in response to its environment. The purpose of this step is to identify the set of one or more candidate primitive behaviors; these candidates will be refined or eliminated later. As the designer describes how the robot should act, behaviors usually become apparent. It should be emphasized that the point of this step is to concen-

trate on what the robot should do, not how it will do it, although often the designer sees both the what and the how at the same time.

In the case of the CSM entry, only one behavior was initially proposed: follow-line. The perceptual schema would use the white line to compute the difference between where the centroid of the white line was versus where it should be, while the motor schema would convert that to a command to the steer motor.

BEHAVIOR TABLE

In terms of expressing the behaviors for a task, it is often advantageous to construct a *behavior table* as one way of at least getting all the behaviors on a single sheet of paper. The releaser for each behavior is helpful for confirming that the behaviors will operate correctly without conflict (remember, accidentally programming the robotic equivalent of male sticklebacks from Ch. 3 is undesirable). It is often useful for the designer to classify the motor schema and the percept. For example, consider what happens if an implementation has a purely reflexive move-to-goal motor schema and an avoid-obstacle behavior. What happens if the avoid-obstacle behavior causes the robot to lose perception of the goal? Oops, the perceptual schema returns no goal and the move-to-goal behavior is terminated! Probably what the designer assumed was that the behavior would be a fixed-action pattern and thereby the robot would persist in moving toward the last known location of the goal.

Behavior
Table

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	follow-line()	stay-on-path(c_x)	c_x	compute-centroid(image,white)

As seen from the behavior table above, the CSM team initially proposed only one behavior, follow-line. The follow-line behavior consisted of a motor schema, stay-on-path(centroid), which was reflexive (stimulus-response) and taxis (it oriented the robot relative to the stimulus). The perceptual schema, compute-centroid(image,white), extracted an affordance of the centroid of white from the image as being the line. Only the x component, or horizontal location, of the centroid was used, c_x.

Step 5: Refine each behavior. By this point, the designer has an overall idea of the organization of the reactive system and what the activities are. This step concentrates on the design of each individual behavior. As the designer constructs the underlying algorithms for the motor and perceptual schemas, it is important to be sure to consider both the normal range of environmental conditions the robot is expected to operate in (e.g., the steady-state case) and when the behavior will fail.

The follow-line behavior was based on the analysis that the only white things in the environment were lines and plastic covered bales of hay. While this was a good assumption, it led to a humorous event during the second heat of the competition.

5.4 Case Study: Unmanned Ground Robotics Competition 171

As the robot was following the white line down the course, one of the judges

stepped into view of the camera. Unfortunately, the judge was wearing white shoes, and Omnibot turned in a direction roughly in-between the shoes and the line. The CSM team captain, Floyd Henning, realized what was happening and shouted at the judge to move. Too late, the robot's front wheels had already crossed over the line; its camera was now facing outside the line and there was no chance of recovering. Suddenly, right before the leftmost rear wheel was about to leave the boundary, Omnibot straightened up and began going parallel to the line! The path turned to the right, Omnibot crossed back into the path and re-acquired the line. She eventually went out of bounds on a hair pin further down the course. The crowd went wild, while the CSM team exchanged confused looks.

What happened to make Omnibot drive back in bounds? The perceptual schema was using the 20% brightest pixels in the image for computing the centroid. When it wandered onto the grass, Omnibot went straight because the reflection on the grass was largely random and the positions cancelled out, leaving the centroid always in the center of the image. The groundskeepers had cut the grass only in the areas where the path was to be. Next to the path was a parallel swatch of uncut grass loaded with dandelion seed puffs. The row of white puffs acted just as a white line, and once in viewing range Omnibot obligingly corrected her course to be parallel to them. It was sheer luck that the path curved so that when the dandelions ran out, Omnibot continued straight and intersected with the path. While Omnibot wasn't programmed to react to shoes and dandelions, it did react correctly considering its ecological niche.

Step 6: Test each behavior independently. As in any software engineering project, modules or behaviors are tested individually. Ideally, testing occurs in simulation prior to testing on the robot in its environment. Many commercially available robots such as Khepera and Nomads come with impressive simulators. However, it is important to remember that simulators often only model the mechanics of the robot, not the perceptual abilities. This is useful for confirming that the motor schema code is correct, but often the only way to verify the perceptual schema is to try it in the real world.

Step 7: Test with other behaviors. The final step of designing and implementing a reactive system is to perform integration testing, where the behaviors are combined. This also includes testing the behaviors in the actual environment.

Although the follow-line behavior worked well when tested with white lines, it did not perform well when tested with white lines and obstacles. The obstacles, shiny plastic-wrapped bales of hay sitting near the line, were often brighter than the grass. Therefore the perceptual schema for follow-line included pixels belonging to the bale in computing the centroid. Invariably the robot became fixated on the bale,

and followed its perimeter rather than the line. The bales were referred to as “visual distractions.”

Fortunately, the bales were relatively small. If the robot could “close its eyes” for about two seconds and just drive in a straight line, it would stay mostly on course. This was called the move-ahead behavior. It used the direction of the robot (steering angle, *dir*) to essentially produce a uniform field. The issue became how to know when to ignore the vision input and deploy move-ahead.

The approach to the issue of when to ignore vision was to use the sonar as a releaser for move-ahead. The sonar was pointed at the line and whenever it returned a range reading, move-ahead took over for two seconds. Due to the difficulties in working with DOS, the CSM entry had to use a fixed schedule for all processes. It was easier and more reliable if every process ran every update cycle, even if the results were discarded. As a result the sonar releaser for move-ahead essentially inhibited follow-line, while the lack of a sonar releaser inhibited move-ahead. Both behaviors ran all the time, but only one had any influence on what the robot did. Fig. 5.6 shows this inhibition, while the new behavioral table is shown below.

New Behavior Table

Releaser	Inhibited by	Behavior	Motor Schema	Percept	Perceptual Schema
always on	near=read_sonar()	follow_line()	stay-on-path(c_x)	c_x	compute_centroid(image,white)
always on	far=read_sonar()	move_ahead(dir)	uniform(dir)	dir	dead_reckon(shaft-encoders)

The final version worked well enough for the CSM team to take first place. It went all the way around the track until about 10 yards from the finish line. The judges had placed a shallow sand pit to test the traction. The sand pit was of some concern since sand is a light color, and might be interpreted as part of the line. Since the sand was at ground level, the range reading could not be used as an inhibitor. In the end, the team decided that since the sand pit was only half the length of a bale, it wouldn't have enough effect on the robot to be worth changing the delicate schedule of existing processes.

The team was correct that the sand pit was too small to be a significant visual distraction. However, they forgot about the issue of traction. In order to get more traction, the team slipped real tires over the slick plastic wheels, but forgot to attach them. Once in the sand, the robot spun its wheels inside the tires. After the time limit was up, the team was permitted to nudge the robot along (done with a frustrated kick by the lead programmer) to see if it would have completed the entire course. Indeed it did. No other team made it as far as the sand pit.

It is clear that a reactive system was sufficient for this application. The use of primitive reactive behaviors was extremely computationally inexpensive,

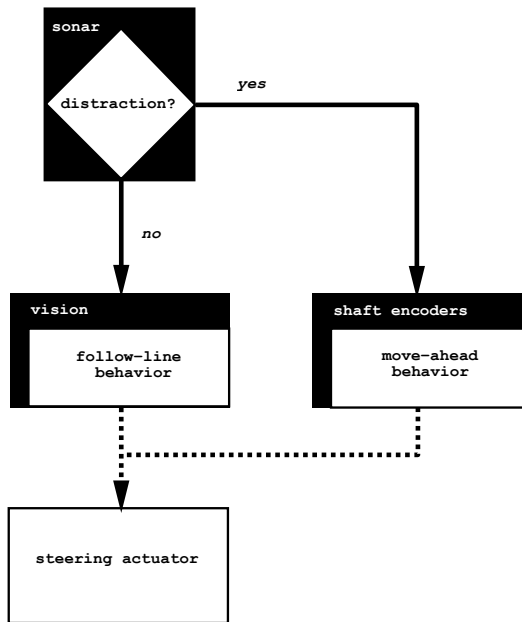


Figure 5.6 The behavioral layout of the CSM entry in the 1994 Unmanned Ground Vehicle Competition.

allowing the robot to update the actuators almost at the update rate of the vision framegrabber.

There are several important lessons that can be extracted from this case study:

The CSM team evolved a robot which fit its ecological niche. However, it was a very narrow niche. The behaviors would not work for a similar domain, such as following sidewalks, or even a path of white lines with an intersection. Indeed, the robot reacted to unanticipated objects in the environment such as the judge's white shoes. The robot behaved "correctly" to features of the open world, but not in a desirable fashion.

This example is a case where the releaser or inhibitor stimulus for a behavior was not the same perception as the percept needed to accomplish the task. The sonar was used to inhibit the behaviors. Follow-line used vision, while move-ahead integrated shaft encoder data to continue to move in the last good direction.

This example also illustrates the tendency among purely reactive motor schema to assign one sensor per behavior.

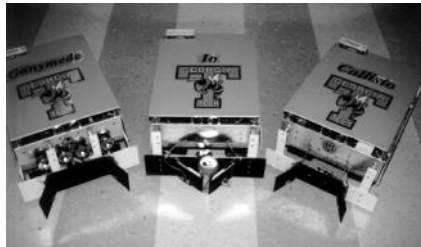
5.5 Assemblages of Behaviors

The UGV competition case study illustrated the basic principles of the design of reactive behaviors. In that case, there were a trivial number of behaviors. What happens when there are several behaviors, some of which must run concurrently and some that run in a sequence? Clearly there are releasers somewhere in the system which determine the sequence. The issue is how to formally represent the releasers and their interactions into some sort of sequencing logic. If a set of behaviors form a prototypical pattern of action, they can be considered a “meta” or “macro” behavior, where a behavior is assembled from several other primitive behaviors into an *abstract behavior*. This raises the issue of how to encapsulate the set of behaviors and their sequencing logic into a separate module.

The latter issue of encapsulation is straightforward. The same OOP schema structure used to collect a perceptual schema and a motor schema into a behavior can be used to collect behaviors into an abstract behavior, as shown by a behavior being composed of behaviors in Fig. 5.1. The *coordinated control program* member of the abstract behavior expresses the releasers for the component behaviors.

SKILLS This leaves the issue of how to formally represent the releasers in a way that both the robot can execute and the human designer can visualize and troubleshoot. There are three common ways of representing how a sequence of behaviors should unfold: *finite state automata*, *scripts* and *skills*. Finite state automata and scripts are logically equivalent, but result in slightly different ways about thinking about the implementation. Skills collect behavior-like primitives called Reaction-Action Packages (RAPs) into a “sketchy plan” which can be filled in as the robot executes. FSA-type of behavioral coordination and control were used successfully by the winning Georgia Tech team¹⁹ in the 1994 AAAI Pick Up the Trash event, and the winning LOLA team in the 1995 IJCAI competition for the Pick Up the Trash event. Scripts were used by the Colorado School of Mines team in the 1995 competition; that entry performed behaviorally as well as the winning teams but did not place due to a penalty for not having a manipulator. Those three teams used at most eight behaviors,

even though LOLA had a more sophisticated gripper than the Georgia Tech team. In contrast, *CHIP* the second place team in the IJCAI



a.



b.

Figure 5.7 Two award-winning Pick Up the Trash competitors: a.) Georgia Tech’s robots with the trash gripper (photograph courtesy of Tucker Balch and AAIL), and b.) University of North Carolina’s LOLA (photograph courtesy of Rich LeGrand and AAIL).

competition required 34 skills and 80 RAPs to do the same task, in part because of the complexity of the arm.^{53;54} Since in general skills lead to a more complex implementation than FSA and scripts, they will not be covered here. The most important point to remember in assembling behaviors is to try to make the world trigger, or *release*, the next step in the sequence, rather than rely on an internal model of what the robot has done recently.

5.5.1 Finite state automata

FINITE STATE

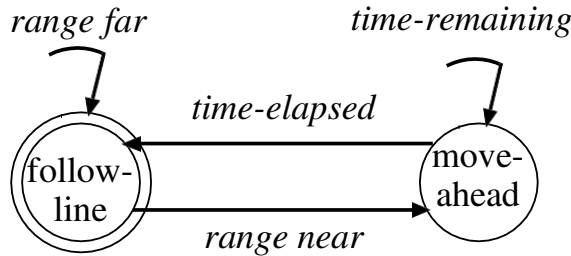
Finite state automata (FSA) are a set of popular mechanisms for specifying

AUTOMATA

STATE DIAGRAM

what a program should be doing at a given time or circumstance. The FSA can be written as a table or drawn as a *state diagram*, giving the designer a

visual
representation.
Most designers do
both. There are
many variants of



a.

$M : K = \text{follow-line, move-ahead}, = \text{range near, range far},$
 $s = \text{follow-line}, F = \text{follow-line, move-ahead}$

q		$(q;)$
follow-line	range near	move-ahead
follow-line	range far	follow-line
move-ahead	time remaining	move-ahead
move-ahead	time elapsed	follow-line

b.

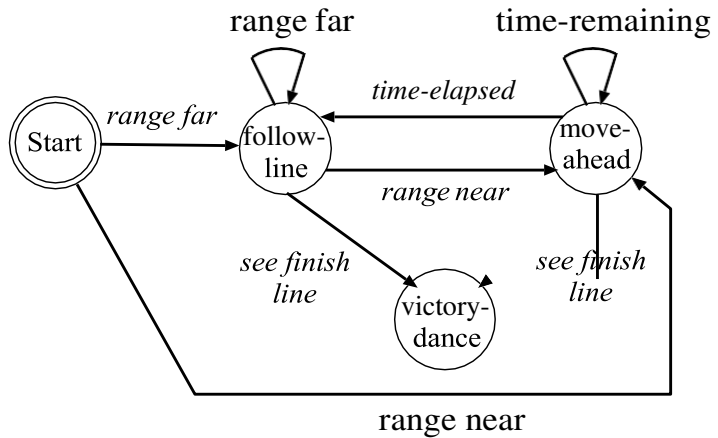
Figure 5.8 A FSA representation of the coordination and control of behaviors in the UGV competition: a.) a diagram and b.) the table.

FSA, but each works about the same way. This section follows the notation used in algorithm development.⁸⁶

STATES To begin with, the designer has to be able to specify a finite number of discrete *states* that the robot should be in. The set of states is represented by K , and each state, $q \in K$ is a listing of the behaviors that should be active at the same time. In the case of the UGR competition, there were only two states: following the line and moving forward. States are represented in a table under the heading q , and by circles in a graph. (See Fig. 5.8.) By convention, there is always a *Start state*, and the robot would always start there. The Start state is often written as s or q_0 and drawn with a double circle. In the case of the UGR entry, the following-line state was the start state since the robot always starts with the follow-line behavior active and not suppressed.

START STATE

The next part of the FSA is the inputs (also called the alphabet). Inputs are the behavioral releasers, and appear under the column heading $.$ Unlike the IRM diagrams, the FSM table considers what happens to each



a.

$M : K = \{ \text{follow-line, move-ahead} \}, = \{ \text{range near, range far,} \\ s = \text{follow-line, } F = \{ \text{follow-line, move-ahead} \}$

q		$(q;)$
follow-line	range near	move-ahead
follow-line	range far	follow-line
move-ahead	time remaining	move-ahead
move-ahead	time elapsed	follow-line

b.

Figure 5.9 An alternative FSA representation of the coordination and control of behaviors in the UGV competition: a.) a diagram and b.) the table.

all possible inputs. As shown in Fig. 5.8, there are only two releasers for the UGR example, so the table doesn't have many rows.

TRANSITION FUNCTION

The third part of the FSM is the *transition function*, called δ , which specifies what state the robot is in after it encounters an input stimulus, s . The set of stimulus or affordances that can be recognized by the robot is represented by S (a capital letter). These stimuli are represented by arrows. Each arrow represents the *releaser* for a behavior. The new behavior triggered by the releaser depends on the state the robot is in. This is the same as with Innate Releasing Mechanisms, where the animal literally ignores releasers that aren't relevant to its current state.

Recall also in the serial program implementations of IRMs in Ch. 3 that the agent “noticed” releasers every second. At one iteration through the loop, it might be hungry and “enter” the state of feeding. In the next iteration, it might still be hungry and re-enter the state of feeding. This can be represented by having arrows starting at the feeding state and pointing back to the feeding state.

For the example in Fig. 5.8, the robot starts in the state of following a line. This means that the robot is not prepared to handle a visual distraction (range near) until it has started following a line. If it does, the program may fail because the FSA clearly shows it won’t respond to the distraction for at least one update cycle. By that time, the robot may be heading in the wrong direction. Starting in the following-line state fine for the UGR competition, where it was known in advance that there were no bales of hay near the starting line. A more general case is shown in Fig. 5.9, where the robot can start either on a clear path or in the presence of a bale. The FSA doesn’t make it clear that if the robot starts by a bale, it better be pointed straight down the path!

FINAL STATE

The fourth piece of information that a designer needs to know is when the robot has completed the task. Each state that the robot can reach that terminates the task is a member of the set of *final state*, F . In the UGR example, the robot is never done and there is no final state—the robot runs until it is turned off manually or runs out of power. Thus, both states are final states. If the robot could recognize the finish line, then it could have a finish state. The finish state could just be stopped or it could be another behavior, such as a victory wag of the camera. Notice that this adds more rows to the FSA table, since there must be one row for each unique state.

In many regards, the FSA table is an extension of the behavioral table. The resulting table is known as a finite state machine and abbreviated M for machine. The notation:

$$M = fK; ; ; s; F g$$

is used as reminder that in order to use a FSA, the designer must know all the q states (K), the inputs the transitions between the states, the special starting state q_0 , and the terminating state(s) (F). There must be one arrow in the state diagram for every row in the table. The table below summarizes the relationship of FSA to behaviors:

FSA	Behavioral analog
K	all the behaviors for a task
	the releasers for each behavior in K
	the function that computes the new state
s	the behavior the robot starts in when turned on
F	the behavior(s) the robot must reach to terminate

In more complex domains, robots need to avoid obstacles (especially people). Avoidance should always be active, so it is often implicit in the FSA. move-to-goal often is shorthand for move-to-goal *and* avoid. This implicit grouping of “interesting task-related behaviors” and “those other behaviors which protect the robot” will be revisited in Ch. 7 as strategic and tactical behaviors.

Another important point about using FSA is that they describe the over- all behavior of a system, but the implementation may vary. Fig. 5.8 is an accurate description of the state changes in the UGV entry. The control for the behaviors could have been implemented exactly as indicated by the FSA: if following-line is active and range returns range near, then move- ahead. However, due to timing considerations, the entry was programmed differently, though with the same result. The following examples will show how the FSA concept can be implemented with subsumption and schema- theoretic systems.

5.5.2 A Pick Up the Trash FSA

As another example of how to construct and apply an FSA, consider the Pick Up the Trash task. Assume that the robot is a small vehicle, such as the ones used by Georgia Tech shown in Fig. 5.7 or the Pioneer shown in Fig. 5.10, with a camera, and a bumper switch to tell when the robot has collided with something. In either case, the robot has a simple gripper. Assume that the robot can tell if the gripper is empty or full. One way to do this is to put an IR sensor across the jaw of the gripper. When the IR returns a short range, the gripper is full and it can immediately close, with a grasping reflex. One problem with grippers is that they are not as good as a human hand. As such, there is always the possibility that the can will slip or fall out of the gripper. Therefore the robot should respond appropriately when it is carrying a can or trash and loses it.

The Pick Up the Trash environment is visually simple, and there are obvious affordances. Coca-cola cans are the only red objects in the

and trash cans are the only blue objects. Therefore visually extracting red and blue blobs should be sufficient. All objects are on the floor, so the robot only has to worry about where the objects are in the x axis. A basic scenario is for the robot to start wandering around the arena looking for red blobs. It should head straight for the center of the largest red blob until it scoops the can in the forklift. Then it should try three times to grab the can, and if successful it should begin to wander around looking for a blue blob. There should only be one blue blob in the image at a time because the two trash cans are placed in diagonal corners of the arena. Once it sees a blue blob, the robot should move straight to the center of the blob until the blob gets a certain size in the image (looming). The robot should stop, let go of the can, turn around to a random direction and resume the cycle. The robot should avoid obstacles, so moving to a red or blue blob should be a fixed pattern action, rather than have the robot immediately forget where it was heading.

The behavior table is:

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	avoid()	pfields.nat(goal_dir)	bumper_on	read_bumper()
EMPTY=gripper_status()	wander()	pfields.random()	time-remaining	countdown()
EMPTY=gripper_status() AND SEE_RED=extract_color(red)	move-to-goal(red)	pfields.attraction(c_x)	c_x	extract-color(red, c_x)
FULL=gripper_status()	grab-trash()	close_gripper()	status	gripper_status()
FULL=gripper_status() AND NO_BLUE=extract_color(blue)	wander()	pfields.random()	time_remaining	countdown()
FULL=gripper_status() AND SEE_BLUE=extract_color(blue)	move-to-goal(blue)	pfields.attraction(c_x)	c_x	extract-color(blue)
FULL=gripper_status() AND AT_BLUE=looming(blue, size=N)	drop_trash()	open_gripper() turn_new_dir(curr_dir)	curr_dir	read_encoders()

The function calls in the table only show the relevant arguments for brevity. The avoid behavior is interesting. The robot backs up either to the right or left (using a NaT) when it bumps something. It may bump an arena wall at several locations, but eventually a new wander direction will be set. If the robot bumps a can (as opposed to captures it in its gripper), backing up gives the robot a second chance. This table shows that the design relies on the gripper to maintain where the robot is in the nominal sequence. An empty gripper means the robot should be in the collecting the trash phase, either looking for a can or moving toward one.

A full gripper means the robot is in the deposit phase. The looming releaser extracts the size of the blue region in pixels and compares it to the size N . If the region is greater than or equal

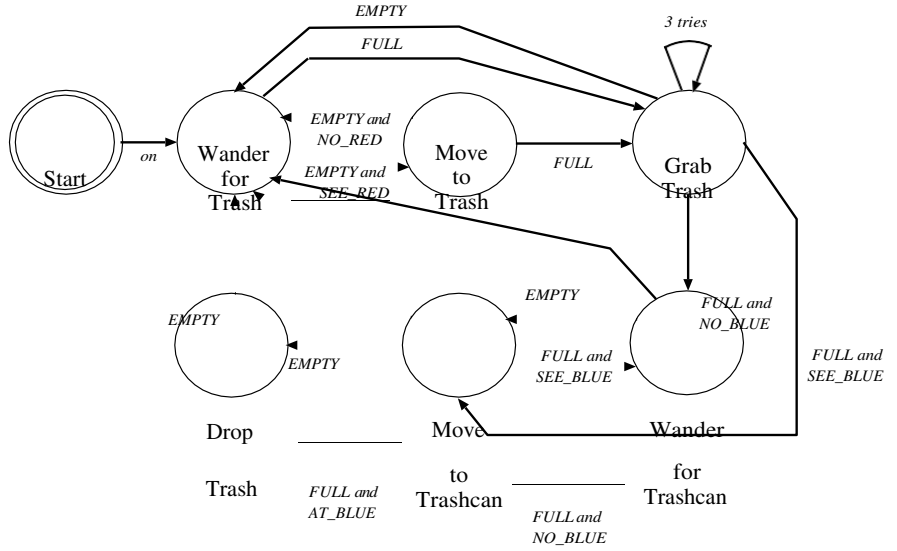


Figure 5.10 A Pioneer P2-AT with a forklift arm suitable for picking up soda cans. (Photograph courtesy of ActivMedia, Incorporated.)

to N, then the robot is “close enough” to the trash can and the robot can drop the can.

There are two problems with the behavior table. The first is that it doesn’t show the sequence, or flow of control, clearly. The second is how did the designer come up with those behaviors? This is where a FSA is particularly helpful. It allows the designer to tinker with the sequence and represent the behavioral design graphically.

Fig. 5.11 shows a FSA that is equivalent to the behavior table. The FSA may be clearer because it expresses the sequence. It does so at the cost of not showing precisely how the sequence would be implemented and encouraging the designer to create internal states. A programmer might implement two wander behaviors, one which is instantiated by different releasers and terminates under different conditions, and two move-to-goal behaviors. Many designers draw and interpret FSA as carrying forward previous releasers. For example, the correct transition from Grab Trash to Wander For Trash can is FULL and NO_BLUE, but a designer may be tempted to label the arrow as only NO_BLUE, since to get that state, the gripper had to be FULL. This is a very dangerous mistake because it assumes that the implementation will be keeping up with what internal state the robot is in (by setting a vari-



a.

$K = \text{fwander for trash, move to trash, grab trash, wander for trash can,}$
 $\text{move to trash can, drop trash } g, = \text{fon, EMPTY, FULL, SEE\_RED,}$
 $\text{NO\_BLUE, SEE\_BLUE, AT\_BLUE } g, s = \text{Start, } F = K$

q		$(q;)$
start	on	wander for trash
wander for trash	EMPTY, SEE_RED	move to trash
wander for trash	FULL	grab trash
move to trash	FULL	grab trash
move to trash	EMPTY, NO_RED	wander for trash
grab trash	FULL, NO_BLUE	wander for trash can
grab trash	FULL, SEE_BLUE	move to trash can
grab trash	EMPTY	wander for trash
wander for trash can	EMPTY	wander for trash
wander for trash can	FULL, SEE_BLUE	move to trash can
move to trash can	EMPTY	wander for trash
move to trash can	FULL, AT_BLUE	drop trash
drop trash	EMPTY	wander for trash

b.

180

5 Designing a Reactive Implementation

Figure 5.11 A FSA for picking up and recycling Coke cans: a.) state diagram, and
b.) table showing state transitions.

able), instead of letting directly perceivable attributes of the world inform the robot as to what state it is in. Internal state is incompatible with reactivity.

The FSA also hid the role of the avoid behavior. The avoid behavior was always running, while the other behaviors were asynchronously being in-stantiated and de-instantiated. It is difficult to show behavioral concurrency with an FSA. Other techniques, most especially Petri Nets, are used in soft-ware engineering but have not been commonly used in robotics. The avoid behavior was not a problem in this case. It was always running and the out-put of the avoid potential field vector could be summed with the output of whatever other behavior was active.

5.5.3 Implementation examples

In a schema-theoretic implementation, the FSA logic would exist in one of two places. If the robot's sole task was to recycle soda cans, the controlling logic could be placed in the main program. If the robot had many tasks that it could do, the ability to recycle trash would be an *abstract behavior*, called by the main program whenever the robot needed to recycle trash. In that case, the FSA logic would be placed in the coordinated control program slot of the behavior schema.

Although the current discussion is on where the FSA goes, it might be use-ful to spend some time on the overall implementation. While the wander-to- goal and move-to-goal behaviors can be easily implemented with a potential fields methodology, drop-trash cannot. Drop-trash really isn't a navigation behavior. It fits the overall profile of a behavioral schema: it has an obvious motor schema (open the gripper, turn the wheels), a perceptual schema (read gripper encoders and wheel encoders), a coordinated control program (open THEN turn), and a releaser (at trash can). While schema-theoretic implemen-tations use potential field methodologies and vector summation for effector control, not every behavior will generate a vector based on a potential field. One advantage of FSA is that they are abstract, and can be implemented in a number of ways. The behavior table illustrated one way the FSA could be implemented with a schema-theoretic system. Fig. 5.12 shows one way it could be implemented in subsumption. This example shows the power of inhibition and suppression which is not well represented by FSA state diagrams.

In keeping with the idea of modularity and incrementally adding behaviors, the system starts with an explicit avoid behavior running on top of Level 0 (not shown). At the next level the robot wanders until it sees red. Then

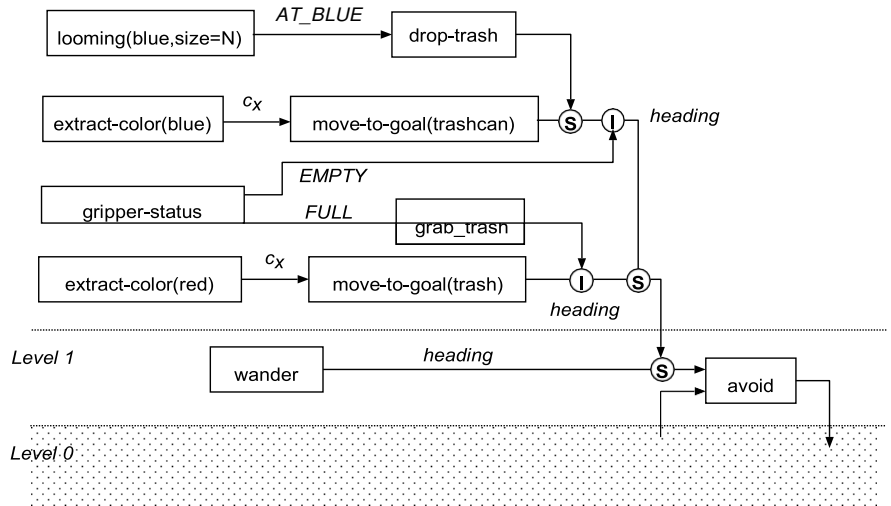


Figure 5.12 One possible Pick Up the Trash abstract behavior using subsumption.

move-to-trash suppresses wandering and replaces the heading with the direction to the trash. The move-to-trash behavior continues until the can is in the gripper. Once the gripper is full, the gripper closes and grabs the trash. Then the move-to-trash behavior is inhibited from executing. This prevents move-to-trash from sending any directions out, so the wandering direction is now again the output.

The next level of competency is to deposit trash in the trash can.

When it sees the blue trash can, move-to-trash can begins to suppress wander and direct the robot to the trash can. If the gripper is empty, the output for move-to-trash can is inhibited. The robot is simultaneously

looking for red and blue blobs, but as long as the gripper is empty, it only responds to red blobs. Dropping trash also is constantly executing. If the robot happens to wander next to a blue trash can, it will signal to drop trash and turn around. The new heading suppresses any heading direction from move-to-trashcan. But the robot will not open its gripper and turn around if the gripper is empty, because empty inhibits that whole line. The subsumption example produces

a much less complex system than a direct FSA mapping.

5.5.4 Abstract behaviors

Finite state automata are a useful tool for expressing the coordinated control program of an abstract behavior. They fall short as a programming tool for abstract behaviors in a number of ways. First, in many cases, the assemblage of behaviors represents a prototypical sequence of events that should be

TEMPLATE slightly adapted to different situations, in essence, a *template* or *abstract behav-*

ABSTRACT BEHAVIOR *ior*. In the Pick Up the Trash event, recycling Coke cans was only part of the task; the robots were also supposed to find white Styrofoam cups and deposit them in yellow trash cans. The behaviors represented by the FSA could be collected into an abstract behavior: `pick-up-the-trash(trash-color, trash can-color, size-trash can)`.

Second, templates need to handle different initialization conditions. Initialization wasn't a big problem for the Pick Up the Trash task, but it could be one for other applications. For example, the emergent behavior of the robot described in the Unmanned Ground Vehicle competition in Sec. 5.4 could be described as an abstract "follow-path" behavior. Recall that the robot's program assumed that it started facing the line. A more general purpose, reusable follow-path behavior would need to handle a broader range of starting conditions, such as starting facing a bale or not perfectly lined up

IMPRINTING with the line. Another common initialization behavior is *imprinting*, where the robot is presented with an object and then records the perceived color (or other attribute) of the object for use with the nominal behavior. In the Pick Up the Trash competitions, several teams literally showed the robot the Coke can and let it determine the best values of "red" for the current lighting conditions. Likewise, some abstract behaviors would need special termination behavior. In the case of the UGR competition, the termination behaviors was NULL, but it could have been the victory dance.

EXCEPTIONS Third, some times robots fail in their task; these events are often called *exceptions*. An exception might be when the robot does not pick up a soda can in over 10 minutes. Another behavior can be substituted (do a raster scan rather than a random wander) or alternative set of parameters (use different values for red).

5.5.5 Scripts

5.5 Assemblages of Behaviors
Abstract behaviors often use *scripts*,^{49;50;87;100} or a related construct called *skills*,^{53;54} to create generic templates of assemblages of behaviors. Scripts provide a different way of generating the logic for an assemblage of behav-

SCRIPTS
SKILLS

Script	Collection of Behaviors	Example
Goal	Task	pick up and throw away a Coca-Cola can
Places	Applicability	an empty arena
Actors	Behaviors	WANDER_FOR_GOAL, MOVE_TO_GOAL, GRAB_TRASH, DROP_TRASH
Props, Cues	Percepts	red, blue
Causal Chain	Sequence of Behaviors	WANDER_FOR_GOAL(TRASH), MOVE_TO_GOAL(TRASH), GRAB_TRASH,WANDER_FOR_GOAL(TRASH CAN), MOVE_TO_GOAL(TRASH CAN), DROP_TRASH
Subscripts	Exception Handling	if have trash and drop, try GRAB_TRASH three times

Figure 5.13 Comparison of script structures to behaviors.

iors. They encourage the designer to think of the robot and the task literally in terms of a screenplay. Scripts were originally used in natural language processing (NLP) to help the audience (a computer) understand actors (people talking to the computer or writing summaries of what they did).¹²³ In the case of robots, scripts can be used more literally, where the actors are robots reading the script. The script has more room for improvisation though, if the robot encounters an unexpected condition (an exception), the robot begins following a *sub-script*.

SUB-SCRIPT

CAUSAL CHAIN

Fig. 5.13 shows how elements of an actor's script compares to a robot script. The main sequence of events is called a *causal chain*. The causal chain is critical, because it embodies the coordination control program logic just as a FSA does. It can be implemented in the same way. In NLP, scripts allow the computer to keep up with a conversation that may be abbreviated. For example, consider a computer trying to read and translate a book where the main character has stopped in a restaurant. Good writers often eliminate all the details of an event to concentrate on the ones that matter. This missing, but implied, information is easy to extract. Suppose the book started with "John ordered lobster." This is a clue that serves as an index into the current or relevant event of the script (eating at a restaurant), skipping over past events (John arrived at the restaurant, John got a menu, etc.). They also focus the system's attention on the next likely event (look for a phrase that indicates John has placed an order), so the computer can instantiate the function which looks for this event. If the next sentence is "Armand brought out the lobster and refilled the white wine," the computer can infer that Armand is the waiter and that John had previously ordered and received white wine,

without having
been explicitly told.

INDEXING

FOCUS-OF-ATTENTION

In programming robots, people often like to abbreviate the routine portions of control and concentrate on representing and debugging the important sequence of events. Finite state automata force the designer to consider and enumerate every possible transition, while scripts simplify the specification. The concepts of *indexing* and *focus-of-attention* are extremely valuable

for coordinating behaviors in robots in an efficient and intuitive manner. Effective implementations require asynchronous processing, so the implementation is beyond the scope of this book. For example, suppose a Pick Up the Trash robot boots up. The first action on the causal chain is to look for the Coke cans. The designer though realizes that this behavior could generate a random direction and move the robot, missing a can right in front of it. Therefore, the designer wants the code to permit the robot to skip searching the arena if it immediately sees a Coke can, and begin to pick up the can without even calling the wander-for-goal(red) behavior. The designer also knows that the next releaser after grab-trash exits to look for is blue, because the cue for moving to the trash can and dropping off trash is blue.

The resulting script for an abstract behavior to accomplish a task is usually the same as the programming logic derived from an FSA. In the case of Pick Up the Trash, the script might look like:

```
for each update...
\\ look for props and cues first: cans, trash cans, gripper
rStatus=extract_color(red, rcx, rSize); \\ ignore rSize
if (rStatus==TRUE)
    SEE_RED=TRUE;
else
    SEE_RED=FALSE;
bStatus=extract_color(blue, bcx, bSize);
if (bStatus==TRUE){
    SEE_BLUE=TRUE; NO_BLUE=FALSE;
} else {
    SEE_BLUE=FALSE; NO_BLUE=TRUE;
}
AT_BLUE=looming(size, bSize);
gStatus=gripper_status();
if (gStatus==TRUE) {
    FULL=TRUE; EMPTY=FALSE;
} else {
    FULL=FALSE; EMPTY=TRUE;
```



```

\\index into the correct step in the causal chain
if (EMPTY){
    if (SEE_RED){
        move_to_goal(red);
    } else
        wander();
} else{
    grab_trash();
    if (NO_BLUE)
        wander();
    else if (AT_BLUE)
        drop_trash();
    else if (SEE_BLUE)
        move_to_goal(blue);
}

```

5.6 Summary

As defined in Ch. 4, a reactive implementation consists of one or more behaviors, and a mechanism for combining the output of concurrent behaviors. While an architectural style (subsumption, potential fields) may specify the structure of the implementation, the designer must invest significant effort into developing individual behaviors and into assembling them into a sequence or an abstract behavior.

Schema theory is highly compatible with Object Oriented Programming. A behavior is derived from the schema class; it is a schema that uses at least one perceptual and motor schema. If a behavior is composed of multiple schema, it must have a coordinated control program to coordinate them. Finite State Automata offer a formal representation of the coordination logic needed to control a sequence of behaviors. Scripts are an equivalent mechanism with a more natural story-like flow of control.

The steps in designing robot intelligence under the Reactive Paradigm are:

1. Describe the task,
2. Describe the robot,
3. Describe the environment,

4. Describe how the robot should act in response to its environment,	193
3.5 Assemblies of Behaviors	

5. Refine each behavior,
6. Test each behavior independently,
7. Test with other behaviors,

and repeat the process as needed. A *behavior table* serves as a means of representing the component schemas and functions of the behavioral system. For each behavior, it shows the releasers, the motor schemas, the percept, and the perceptual schemas.

These steps emphasize the need to fully specify the ecological niche of the robot in order to design useful behaviors. Since the idea of behaviors in the Reactive Paradigm is derived from biology, it is not strange that the idea of a robot being evolved to fit its environment should also be part and parcel of the design process. Regardless of the implementation of the coordination program, the control should rely on the world to inform the robot as to what to do next, rather than rely on the program to remember and maintain state.

6 *Common Sensing Techniques for Reactive Robots*

Chapter Objectives:

Describe the difference between *active* and *passive* sensors, and give examples of each.

Define each of the following terms in one or two sentences: *proprioception*, *exteroception*, *exproprioception*, *proximity sensor*, *logical sensor*, *false positive*, *false negative*, *hue*, *saturation*, *image*, *pixel*, *image function*, *computer vision*.

List the metrics for rating individual sensors and a sensor suite, and apply these metrics to a particular application.

Describe the problems of *specular reflection*, *cross-talk*, and *foreshortening* with an ultrasonic transducer, and if given a 2D line drawing of surfaces, illustrate where each of these problems would likely occur.

Describe the types of *behavioral sensor fusion* and be able to apply to a real-world problem.

Write perceptual schemas from any logically equivalent range sensor to produce a polar plot percept for obstacle avoidance behavior.

If given a small interleaved RGB image and a range of color values for a region, be able to 1) threshold the image on color and 2) construct a color histogram.

Write computer vision code to enable a robot to imprint on and track a color.

6.1 Overview

To review Ch. 3, perception in a reactive robot system has two roles: *to release a behavior*, and *to support or guide the actions of the behavior*. All sensing is behavior-specific, where behaviors may tap into the same sensors, but use the data independently of each other. Also, the connotation of reactive robots is that behaviors are most often stimulus-response, relying on direct perception rather than require memory. Recognition is not compatible with reactivity. In order for a reactive robot to be successful in theory, it must have reliable perception since perception and action are tightly coupled. For it to be successful in practice, a robot has to have perceptual hardware and software which updates quickly. This chapter covers the most common sensing techniques for reactive robots. The sensors and sensing techniques described in this chapter also apply to robot architectures in the Hybrid paradigm, since Hybrid architectures use reactive behaviors.

SENSOR Regardless of sensor hardware or application, sensing and sensors can be thought of interacting with the world and robots as shown in Fig. 6.1. The *sensor* is a device that measures some attribute of the world. The term *trans-*

TRANSDUCER *ducer* is often used interchangeably with *sensor*. A transducer is the mechanism, or element, of the sensor that transforms the energy associated with what is being measured into another form of energy.² A sensor receives energy and transmits a signal to a display or a computer. Sensors use transducers to change the input signal (sound, light, pressure, temperature, etc.) into an analog or digital form capable of being used by a robot. In a reactive robot, the sensor observation is intercepted by a perceptual schema which extracts the relevant percept of the environment for the behavior. This percept is then used by the motor schema, which leads to an action.

PASSIVE SENSOR A sensor is often classified as being either *passive sensor* or *active sensor*.

ACTIVE SENSOR Passive sensors rely on the environment to provide the medium for observation, e.g., a camera requires a certain amount of ambient light to produce a usable picture. Active sensors put out energy in the environment to either change the energy or enhance it. A sonar sends out sound, receives the echo, and measures the time of flight. An X-ray machine emits X-rays and measures the amount blocked by various types of tissue. Although a camera is a passive device, a camera with a flash is active. The term

the same as *active sensing*. Active sensing connotes the system for using an effector to dynamically position a sensor for a “better look.” A camera with a flash is an active sensor; a camera on a pan/tilt head with algorithms to direct the camera to turn to get a better view is using active sensing.

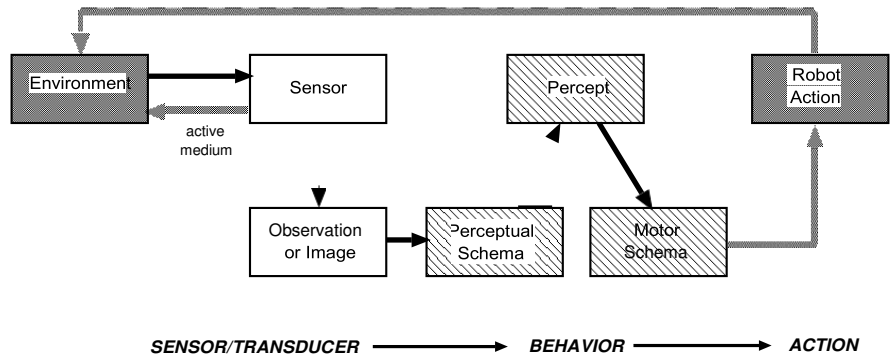


Figure 6.1 A model of sensing.

MODALITY

Different sensors measure different forms of energy. This in turns leads to different types of processing. Sensors which measure the same form of en- ergy and process it in similar ways form a sensor *modality*. A sensor modality refers to what the raw input to the sensor is: sound, pressure, temperature, light, and so on. In some regards, modalities are similar to the five senses in humans. A modality can be further subdivided, for instance, vision can be decomposed into visible light, infrared light, X-rays, and other modalities.

6.1.1 Logical sensors

LOGICAL SENSORS

A powerful abstraction of sensors is *logical sensors*, first introduced by Hen- derson and Shilcrat.⁶⁵ A logical sensor is a unit of sensing or module that supplies a particular percept. It consists of the signal processing from the physical sensor and the software processing needed to extract the percept; it is the functional building block for perception. A logical sensor can be easily implemented as a perceptual schema.

An overlooked aspect of a logical sensor is that it contains all available alternative methods of obtaining that percept. For example, a percept com- monly used for obstacle avoidance is a polar plot of range data. The logical sensor for the percept might be named `range_360` and return a data struc- ture or object specifying the polar plot. The logical sensor would go further and list all the possible ways the robot had for constructing a polar plot of that form. The robot might be able to use sonar, a laser, stereo vision, or tex- ture. Each of those modules would be

LOGICAL EQUIVALENCE

logically equivalent;
that is, they would
return the same
percept data
structure so they
can be used
interchangeably.

However, they wouldn't necessarily be equivalent in performance or update rate. As will be seen in this chapter, the sonar is liable to produce a noisy percept in a second or two, while stereo vision may take minutes. Even different stereo vision algorithms may produce different results on the same data stream. Therefore, the logical sensor contains a selector function which specifies the conditions under which each alternative is useful and should be selected.

Notice that a logical sensor can be implemented as a perceptual schema, where the methods are the alternative means of generating the percept and the coordinated control strategy contains the knowledge as to when a particular method is appropriate. Also note that each individual method can be implemented as a perceptual schema, leading to the recursive, building-block effect.

In reactive systems, the term logical sensor has degenerated somewhat from its original usage and is essentially equivalent to a perceptual schema. "Logical sensor" is often used to connote information hiding, where the particular sensor and processing algorithm is hidden in the "package." This is useful because a robot might use the same physical sensor in two different ways. An avoid behavior might use a polar plot of sonar range data, while a panic-stop behavior might use the minimum of all the incoming sonar data. Since the perceptual schema use the raw sonar data differently, it is as if they were different sensors.

6.2 Behavioral Sensor Fusion

SENSOR FUSION

Sensor fusion is a broad term used for any process that combines information from multiple sensors into a single percept. The motivation for sensor fusion stems from three basic combinations of sensors: *redundant* (or competing), *complementary*, and *coordinated*. Although many researchers treat sensor fu-

REDUNDANT

COMPLEMENTARY

COORDINATED

sion as a means of constructing a global world model in a hierarchical or deliberative system, sensor fusion can be incorporated into behaviors through *sensor fission*, *action-oriented sensor fusion*, and *sensor fashion*.

In some cases multiple sensors are used when a particular sensor is too imprecise or noisy to give reliable data. Adding a second sensor can give another "vote" for the percept. When a sensor leads the robot to believe that

a percept is present, but it is not, the error is called a *false positive*. The robot has made a positive identification of percept, but it was false. Likewise, an error where the robot misses a percept is known as a *false negative*.

Sensors



Figure 6.2 Example of redundant top and bottom sonar rings.

will often produce different false positive and false negative rates. Whether a robot can tolerate a higher false positive or false negative rate depends on the task.

When the sensors are both returning the same percept, the sensors are considered *redundant*. An example of *physical redundancy* is shown in Fig. 6.2, where a Nomad 200 has two sonar rings. The sonar software returns the minimum reading (shortest range) from the two, providing a more reliable reading for low objects which would ordinarily specularly reflect the beam from the upper sonar. Sensors can also be *logically redundant*, where they return identical percepts but use different modalities or processing algorithms. An example is extracting a range image from stereo cameras and from a laser

range finder. Sometimes redundant sensors are called *competing sensors*, because the sensors can be viewed as competing to post the “winning” percept. Complementary sensors provide disjoint types of information about a percept. In behavioral sensor fusion for urban search and rescue, a robot may search for survivors by fusing observations from a thermal sensor for body heat with a camera detecting motion. Both logical sensors return some aspect of a “survivor,” but neither provides a complete view. Coordinated sensors use a sequence of sensors, often for cue-ing or providing focus-of-attention.

REDUNDANT
PHYSICAL
REDUNDANCY

LOGICALLY

REDUNDANT

COMPETING SENSORS

A predator might see motion, causing it to stop and examine the scene more closely for signs of prey.

Most of the work on sensor fusion treats it as if it were a deliberative pro-

SENSOR FISSION

cess: one that requires a global world model. Early work in reactive systems used robots with only a few simple sensors, a sonar or sonar ring for range and a camera for color, texture, or motion affordances. As a result, there was a philosophy of one sensor per behavior. Behaviors could share a sensor stream, but without knowing it. This philosophy led to the approach taken by Brooks that sensor fusion at the behavioral level was a mirage. Instead sensor fusion was really multiple instances of the same behavior with different sensor inputs being combined together. To an outside observer, it would look like some complicated process was being enacted inside the robot, but in fact it would be a simple competition with an emergent behavior. Brooks dubbed this *sensor fission* in part as a take off on the connotations of the word “fusion” in nuclear physics. In nuclear fusion, energy is created by forcing atoms and particles together, while in fission, energy is created by separating atoms and particles. Fig. 6.3a shows a diagram of sensor fission.

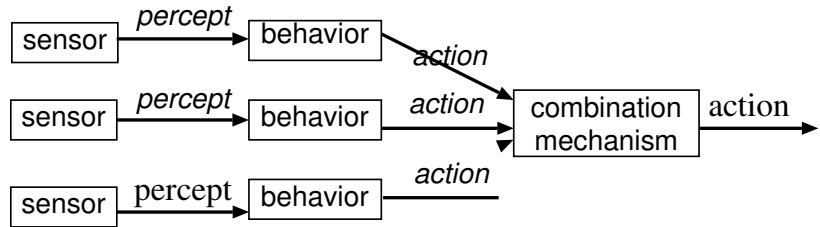
ACTION-ORIENTED

Murphy reported on studies from cognitive psychology and neuropsychology showing that behavioral sensor fusion does occur in animals, and therefore should be part of a robot’s behavioral repertoire.⁹⁹ The gist of the studies was that sensor fusion does occur in behaviors. The sensor pathways throughout the brain remain separate and can be routed to multiple behaviors in the superior colliculus. Only when the sensor signals routed to the portion of the brain associated with a particular behavior arrive at that location does there appear to be any transformation. The transformation appears to be a new common representation. Any or all of these sensor streams can be active and influence the resulting behavior. For example, consider the predation behavior in cats. If a cat hears a noise and sees a movement, it will react more strongly than if it has only a single stimulus. This type of sensor fusion is called *action-oriented sensor fusion* to emphasize that the sensor data

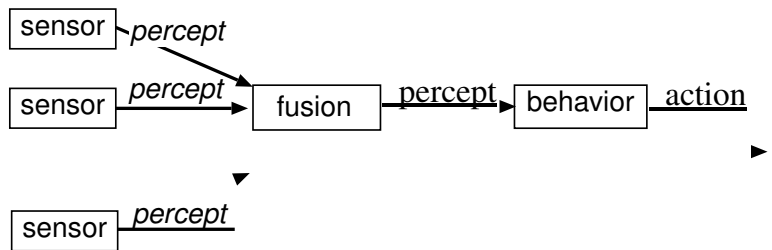
is being transformed into a behavior-specific representation in order to support a particular action, not for constructing a world model. Fig. 6.3b shows a diagram of action-oriented sensor fusion.

Sensor fission and action-oriented sensor fusion cover competing and complementary sensing. Sensor fission is by definition a competitive method, though complementary sensors may be used to support a particular instance of a behavior. Action-oriented sensor fusion is not restricted to either competing or complementary sensors, since the behavior makes a local transformation anyway. This leaves the category

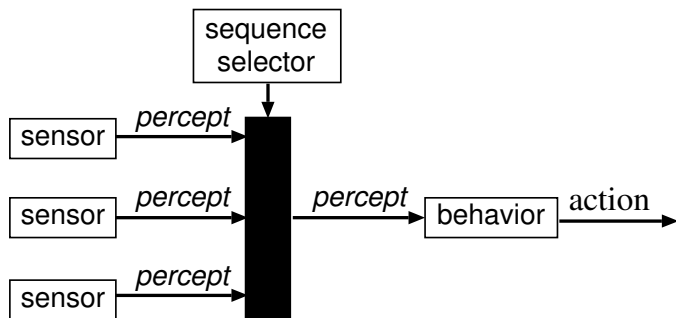
Arkin filled in the apparent gap by calling coordination *sensor fashion*, an alliterative name intended to imply the robot was changing sensors with changing circumstances just as people change styles of clothes with the seasons. A



a.



b.



c.

Figure 6.3 Three types of behavioral sensor fusion: a.) sensor fission, b.) action-oriented sensor fusion, and c.) sensor fashion.

diagram of sensor fusion is shown in Fig. 6.3c.

Murphy incorporated pathways for all three types of behavioral sensor fusion as well as other deliberative forms into the Sensor Fusion Effects (SFX) architecture. SFX will be covered in Ch. 7.

6.3 Designing a Sensor Suite

Historically, reactive robots used either inexpensive infrared (IR) or ultrasonic transducers to detect range. The earliest behaviors focused on basic navigational skills such as obstacle avoidance and wall following. The percept for these behaviors all involve knowing the distance to an occupied area of space. Now with the advent of inexpensive miniature cameras and laser range finders for consumer applications, computer vision is becoming increasingly common. In agricultural and transportation applications of reactive robots, GPS technology has become popular as well. This chapter attempts to cover the basics of these sensing modalities, and how they are used in mobile robots. Because the sensor market is rapidly changing, the chapter will focus on how to design a suite of sensors for use by a robot, rather the device details.

An artificially intelligent robot has to have some sensing in order to be considered a true AI robot. If it cannot observe the world and the effects of its actions, it cannot react. As noted in the chapter on “Action-Oriented Perception” in *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, the design of a set of sensors for a robot begins with an assessment of the type of information that needs to be extracted.

¹⁴ This information can be either from *proprioception* (measurements of movements relative to an internal frame of reference), *exteroception* (measurements of the layout of the environment and objects relative to the robot’s frame of reference) or *exproprioception* (measurement of the position of the robot body or parts relative to the layout of the environment).

The Colorado School of Mines fielded an entry to the 1995 UGV competition entry discussed in Ch. 5. This provides an example of different types of sensing for a path following robot. In 1995, the follow-path behavior was expanded to track both lines of the path using a wide angle lens on the camera. follow-path could be considered exteroceptive because it acquired information on the environment. However, the camera for the robot was mounted on a panning mast,

PROPRIOCEPTION

EXTEROCEPTION

EXPROPRIOCEPTION

which was intended to turn to keep the line in view, no matter what direction the path turned in. Therefore, the

robot had to know where the camera was turned relative to the robot's internal frame of reference in order to correctly transform the location of a white line in image coordinates to a steering direction. This meant the information needed for `follow-path` had both proprioceptive and exteroceptive components, making the perception somewhat exproprioceptive. (If the robot was extracting the pose of its camera from exteroception, it would be clearly exproprioceptive.)

Due to a programming error, the `follow-path` behavior incorrectly assumed that the exteroceptive camera data had been transformed by the proprioceptive shaft encoder data from the panning mast into exproprioceptive data. The robot needed the exproprioception to determine where it should move next: turn to follow the path in camera coordinates, plus the compensation for the current camera pan angle. The programming error resulted in the robot acting as if the camera was aligned with the center of the robot at all times. But the camera might be turned slightly to maintain the view of both lines of the path through the `pan-camera` behavior. The resulting navigational command might be to turn, but too little to make a difference, or even to turn the wrong way. This subtle error surfaced as the robot went around hair pin turns, causing the robot to go consistently out of bounds.

6.3.1 Attributes of a sensor

As can be seen by the above example, robots may have dead reckoning capabilities, but will always have some type of exteroceptive sensor. Otherwise, the robot cannot be considered reactive: there would be no stimulus from the world to generate a reaction. The set of sensors for a particular robot is

SENSOR SUITE

FIELD OF VIEW (FOV)

called a *sensor suite*. Following *Sensors for Mobile Robots*,⁵² in order to construct a sensor suite, the following attributes should be considered for each sensor:

1. Field of view and range. Every exteroceptive sensor has a region of space that it is intended to cover. The width of that region are specified by the sensor's *field of view*, often abbreviated as FOV. The field of view is usually expressed in degrees; the number of degrees covered vertically may be different from the number of degrees covered horizontally. Field of view is frequently used in photography, where different lenses capture different size and shape areas. A wide angle lens will often cover up to 70°, while a "regular" lens may only have a field of

HORIZONTAL FOV

coordinates, where one angle is the *horizontal FOV* and the other is the *vertical*

VERTICAL FOV
RANGE

FOV. The other aspect is the *range*, or how far the sensor can make reliable measurements. In spherical coordinates, this would be the values of r that defined the depth of the operating range.

Field of view and range are obviously critical in matching a sensor to an application. If the robot needs to be able to detect an obstacle when it's 8 feet away in order to safely avoid it, then a sensor with a range of 5 feet will not be acceptable.

RESOLUTION

2 Accuracy, repeatability, and resolution. Accuracy refers to how correct the reading from the sensor is. But if a reading for the same conditions is accurate only 20% of the time, then the sensor has little repeatability. If the sensor is consistently inaccurate in the same way (always 2 or 3 cm low), then the software can apply a bias (add 2 centimeters) to compensate. If the inaccuracy is random, then it will be difficult to model and the applications where such a sensor can be used will be limited. If the reading is measured in increments of 1 meter, that reading has less *resolution* than a sensor reading which is measured in increments of 1 cm.

3 Responsiveness in the target domain. Most sensors have particular environments in which they function poorly. Another way of viewing this is that the environment must allow the signal of interest to be extracted from noise and interference (e.g., have a favorable signal-to-noise ratio). As will be seen below, sonar is often unusable for navigating in an office foyer with large amounts of glass because the glass reflects the sound energy in ways almost impossible to predict. It is important to have characterized the ecological niche of the robot in terms of what will provide, absorb, or deflect energy.

4 Power consumption. Power consumption is always a concern for robots. Since most robots operate off of batteries, the less power they consume, the longer they run. For example, the battery life on a Nomad 200, which carries five batteries, was improved from four hours to six by shutting off all sensors. Power is so restricted on most mobile robots that many robot manufacturers will swap microprocessor chips just to reduce the power drain (which was part of the motivation for the Transmeta Crusoe chip). Sensors which require a large amount of power are less desirable than those which do not. In general, passive sensors have less power demands than active sensors because they are not emitting energy into the environment.

HOTEL LOAD

The amount of power on a mobile robot required to support a sensor pack- age (and any other electronics such as a microprocessor and communications links) is sometimes called the *hotel load*. The sensor suite

is the “guest” of
the

LOCOMOTION LOAD

platform. The power needed to move the robot is called the *locomotion load*. Unfortunately, many robot manufacturers focus on only the locomotion load, balancing power needs with the desire to reduce the overall weight and size. This leads to a very small hotel load, and often prevents many sensors from being added to platform.

5. Hardware reliability. Sensors often have physical limitations on how well they work. For example, Polaroid sonars will produce incorrect range reading when the voltage drops below 12V. Other sensors have temperature and moisture constraints which must be considered.

6. Size. The size and weight of a sensor does affect the overall design. A microrover on the order of a shoebox will not have the power to transport a large camera or camcorder, but it may be able to use a miniature “Quick-Cam” type of camera.

The above list concentrated on considerations for the physical aspects of the sensor. However, the sensors only provide observations; without the software perceptual schemas, the behaviors cannot use the sensors. Therefore, the software that will process the information from a sensor must be considered as part of the sensor selection process.

7. Computational complexity. Computational complexity is the estimate of how many operations an algorithm or program performs. It is often written as a function o , called the “order,” where $O(x)$ means the number of operations is proportional to x . x is often a function itself. Lower orders are better. An algorithm that executes with $O(n)$ equally consuming operations is faster than one with $O(n^2)$ operations. (If you doubt this, see if you can find a positive, whole number value of n such that $n > n^2$.) Computational complexity has become less critical for larger robots, with the rapid advances in processors and miniaturization of components. However, it remains a serious problem for smaller vehicles.

8. Interpretation reliability. The designer should consider how reliable the sensor will be for the ecological conditions and for interpretation. The robot will often have no way of determining when a sensor is providing incorrect information. As a result the robot may “hallucinate” (think it is seeing things that are not there) and do the wrong thing. Many sensors produce output which are hard for human to interpret without years of training; medical X-rays are one example, and synthetic aperture radar (SAR) which produces polar plots is another. If a sensor algorithm was not working properly in these modalities, the designer might not be skilled enough to notice it. Therefore, the algorithms themselves must be reliable.

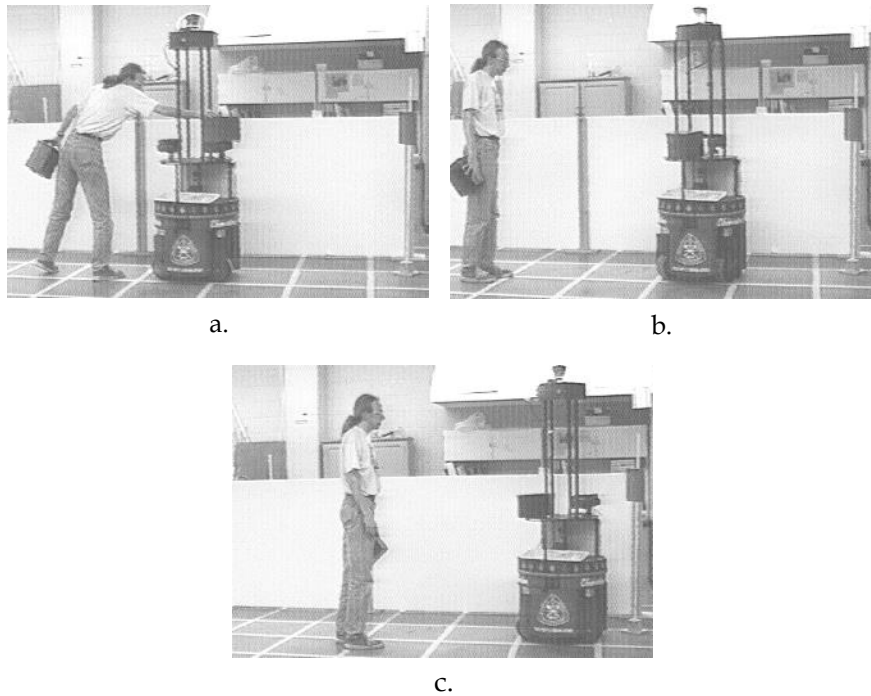


Figure 6.4 Sequence showing a Denning mobile robot with redundant cameras, responding to sensing failures introduced by Dave Hershberger.

6.3.2 Attributes of a sensor suite

Everett⁵² recommends that the following attributes should be considered for the *entire sensing suite*:

1. **Simplicity.** As with any design, simple sensors which have straightforward hardware and operating principles are more desirable than sensors which are complex and require constant maintenance.
2. **Modularity.** In order to reconfigure a sensor suite (add new sensors, remove ones that aren't appropriate for the task), the designer must be able to remove one sensor and/or its perceptual schema without impacting any other sensing capability.
3. **Redundancy.** In most military and industrial applications, it is imperative that the robot function correctly, since it would put people at risk to try

to retrieve the robot and fix it. Sensors are especially important because a faulty sensor can cause the robot to “hallucinate.” As a result, sensor suites may offer some sort of redundancy.

PHYSICAL There are two types of redundancy. *Physical redundancy* means that there

REDUNDANCY are several instances of physically identical sensors on the robot. Fig. 6.4 shows a robot with redundant cameras. In this case, the cameras are mounted 180° apart, and when one sensor fails, the robot has to “drive backwards” in order to accomplish its task.

LOGICAL *Logical redundancy* means that another sensor using a different sensing mo-

REDUNDANCY dality can produce the same percept or releaser. For example, the Mars Sojourner mobile robot had a laser striping system for determining the range to obstacles, which emitted a line of light. If the surface was flat, a camera would see a flat line, whereas if the surface was rough the line would appear jagged. The robot also carried a second camera which could do obstacle detection via stereo triangulation. The laser striping sensor and the stereo cameras are logically redundant. They are not physically redundant, but they produce the same overall information: the location of obstacles relative to the robot. However, logically redundant sensors are not necessarily equivalent in processing speed or accuracy and resolution. A stereo range sensor and algorithm computes the range much slower than a laser striping system.

Physical redundancy introduces new issues which are the area of active research investigation. Possibly the most intriguing is how a robot can determine that a sensor (or algorithm) has failed and needs to be swapped out.

FAULT TOLERANCE Surviving a failure is referred to as *fault tolerance*. Robots can be programmed in most cases to tolerate faults as long as they can identify when they occur.

6.4 Proprioceptive Sensors

Proprioception is dead reckoning, where the robot measures a signal originating within itself. In biology, this is usually some measure of how far an arm or leg has been extended or retracted. In robotics, actuators are generally motors. Many motors come with a *shaft encoder* which

SHAFT ENCODER

measures the number of turns the motor has made. If the gearing and wheel size is known, then the number of turns of the motor can be used to compute the number of turns of the robot's wheels, and that number can be used to estimate how far the robot has traveled.

Proprioception is often only an estimate. This is due to the impact of the environment on the actual movement of the robot. Arkin in his PhD thesis⁸ showed that the same wheeled robot, HARV, traveled different distances for the same encoder count on a sidewalk, grass, and wet grass. The texture of the different surfaces caused the wheels of HARV mobile to slip to varying degrees. A robot on a tiled floor might slip twice as much as on dry grass.

6.4.1 Inertial navigation system (INS)

INERTIAL NAVIGATION

Aircraft, submarines, and missiles use *inertial navigation systems (INS)*. INS

SYSTEMS (INS)

measure movements electronically through miniature accelerometers. As long as the movements are smooth, with no sudden jarring, and the samples are taken frequently, an INS can provide accurate dead reckoning to 0.1 percent of the distance traveled.⁵² However, this technology is unsuitable for mobile robots for several reasons. The cost of an INS is prohibitive; units run from \$50,000 to \$200,000 USD. The cost is due in part to having to stabilize the accelerometers with gyroscopes, as well as the nature of precision electronics. Mobile robots often violate the constraint that motion must be smooth. A hard bump or sudden turn can exceed the accelerometers' measurement range, introducing errors. INS systems are typically big; smaller devices have less accuracy. Sojourner, the Mars rover, carried an INS system. In one trek, it would have stopped 30 cm from a rock it was to sample if it had just used proprioception. Instead, by using exteroception, it got within 4 cm.

6.4.2 GPS

PRECISION

GPS, or Global Positioning System, is becoming more common on robots, especially those used to automate farm equipment (an effort called *precision*

AGRICULTURE

agriculture). GPS systems work by receiving signals from satellites orbiting the Earth. The receiver triangulates itself relative to four GPS satellites, computing its position in terms of latitude, longitude, altitude, and change in time. GPS isn't a proprioceptive sensor per se since the robot must receive signals from the satellites, external to the robot. However, they are not exteroceptive sensors either, since the robot isn't computing its position relative to its environment. Since they tend to be used in place

6.4.1 *Proprietary Sensors* of dead reckoning on outdoor robots, only GPS will be covered here. 210

Currently the only sets of GPS satellites that a receiver can triangulate it- self against are the Navstar “constellation” maintained by the United States Air Force Space Command or the Russian counterpart, GLONASS, main-

tained by the Russian Federation Ministry of Defense. There are two types of channels on Navstar, one public, called the Standard Positioning System, and an encrypted signal, the Precise Positioning System. Until early in the year 2000, the U.S. military actually introduced an error in the satellite message as to where the satellite actually is, which could result in triangulation errors of up to 100 meters. The error was called *selective availability*, because it

SELECTIVE

made accurate positioning available only to those users selected by the U.S. military. This was intended to prevent a hostile country from putting a GPS receiver on a guided missile and precisely targeting where the President is giving a talk. Selective availability was turned off in part because of the rise of civilian uses of GPS, and because it led to interoperability with groups working with the U.S. military who were using commercial, not military, GPS.

AVAILABILITY

Many inexpensive hand-held receivers sold to hunters and hikers attempt to improve on localization by averaging or filtering the readings. This can reduce the error down to 10-15 meters. Surveyors and GPS specialty companies such as Trimble and Rockwell have found a way to subtract the error in the public channel and get performance near the Y-code's rumored accuracy

DIFFERENTIAL GPS

of centimeters. The method is called *differential GPS (DGPS)*, where two GPS

(DGPS) receivers are used. One remains stationary, while the other is put on the robot. If the two receivers are observing the same satellites, then any sudden change in position on the stationary "base" receiver is due to the induced error and can be subtracted from the readings at the robot GPS. The ultimate fix to the induced error will probably come in a few years due to the commercial sector. A consortium of private companies is planning to launch a new constellation of GPS satellites, Teledesic, which will emit accurate data at all times in a format that can be decoded only by chips licensed by the consortium. Teledesic is scheduled to go on-line in 2004.

GPS and DGPS are not complete solutions to the dead reckoning problem in mobile robots for at least two reasons. First, GPS does not work indoors in most buildings, especially offices or factories with large amounts of steel-reinforced concrete. As with cellular phones, these structures interrupt the reception of radio signals. Likewise, GPS may not work outdoors in major

URBAN CANYONS

cities where skyscrapers act as *urban canyons* and interfere with reception.

Second, commercial DGPS systems cost on the order of \$30,000 USD, which is prohibitively high. Several web sites now offer free “do-it-yourself” DGPS code to create a DGPS from two inexpensive receivers.

6.5 Proximity Sensors

Proximity sensors measure the relative distance (range) between the sensor and objects in the environment. Since the sensor is mounted on the robot, it is a straightforward computation to translate a range relative to the sensor to a range relative to the robot at large. Most proximity sensors are active. Sonar, also called ultrasonics, is the most popular proximity sensor, with infrared, bump, and feeler sensors not far behind.

6.5.1 Sonar or ultrasonics

Sonar refers to any system for using sound to measure range. Sonars for different applications operate at different frequencies; for example, a sonar for underwater vehicles would use a frequency appropriate for traveling through water, while a ground vehicle would use a frequency more suited for air. Ground vehicles commonly use sonars with an ultrasonic frequency, just at the edge of human hearing. As a result the terms “sonar” and “ultrasonics” are used interchangeably when discussing extracting range from acoustic energy.

Ultrasonics is possibly the most common sensor on commercial robots operating indoors and on research robots. They are active sensors which emit a

TIME OF FLIGHT

sound and measure the time it takes for the sound to bounce back. The *time of flight* (time from emission to bounce back) along with the speed of sound in that environment (remember, even air changes density with altitude) is sufficient to compute the range of the object.

Ultrasonics is common for several reasons. Its evolution paralleled the rise of the Reactive Paradigm. In the mid-1980's, Hans Moravec did impressive robot navigation with a ring of sonars. The ring configuration gave a 360 coverage as a polar plot. This ring was developed by one of the first mobile robot manufacturers, Denning Robotics, and since then sonar rings are often referred to as “Denning rings,” regardless of manufacturer. Besides providing direct range measurements, the transducers were cheap, fast, and had terrific coverage. In the early 1980's, the Polaroid Land Corporation had developed small, inexpensive sonars for use as camera range finders. A bigger version, the Polaroid Lab Grade ultrasonic transducer, costs on the order of \$30 USD and can measure ranges from 1 to 25 feet with inch resolution over a field of view of 30°. Furthermore, the measurement time was on

the ²¹⁰order of
seconds versus
hours for
computer vision.
Ultrasonics
became the
sensor of choice
for behavior-
based robots.

6 Common Sensing Techniques for Reactive Robots



Figure 6.5 Polaroid ultrasonic transducer. The membrane is the disk.

ECHO

A robotic sonar transducer is shown in Fig. 6.5. The transducer is about the size and thickness of a dollar coin, and consists of a thin metallic membrane. A very strong electrical pulse generates a waveform, causing the membrane on the transducer to produce a sound. The sound is audible as a faint click- ing noise, like a crab opening and closing its pinchers. Meanwhile a timer is set, and the membrane becomes stationary. The reflected sound, or *echo*, vibrates the membrane which is amplified and then thresholded on return signal strength; if too little sound was received, then the sensor assumes the sound is noise and so ignores it. If the signal is strong enough to be valid, the timer is tripped, yielding the time of flight.

SIDE LOBES

The key to how useful the data is requires understanding how the sound wave is generated by the transducer. In reality, the sound beam produces multiple secondary sound waves which interact over different regions of space around the transducer before dissipating. Secondary sound waves are called *side lobes*. Most robot systems assume that only sound from the main, or centermost, lobe is responsible for a range measurement. The width of the main lobe is often modeled as being 30 wide at about 5 meters away. However, in practice, reactive robots need

to respond to
obstacles in the

6 Common Sensing Techniques for Reactive Robots

to 3 meter range. As a result many algorithms only treat the lobe as being between 8 and 15 wide depending on how reliable the range readings are in a particular environment. Ch. 11 will go over this in more detail.

The strength of the main lobe in the environment determines the maximum range that the sonar can extract reliability. In ideal indoor venues, a sonar might return ranges of up to 25 feet, while in the outdoors, the same sonar might go no further than 8 feet with any repeatability. So while the upper limit of the range reading depends on the sensor and the environment, the lower limit does not. Ultrasonic transducers have a “dead time” immediately following emission while the membrane vibration decays. The decay time translates into an inability to sense objects within 11 inches because measurements made during this period are unreliable because the membrane may not have stopped ringing.

Regardless of the maximum allowed range return (i.e., does the program ignore any reading over 3 meters?) and the width of the lobe, most computer programs divide the area covered by a sonar into the three regions shown in Fig. 6.6. Region I is the region associated with the range reading. It is an arc, because the object that returned the sound could be anywhere in the beam. The arc has a width, because there are some resolution and measurement errors; the width of Region I is the tolerance. Region II is the area that is empty. If that area was not empty, the range reading would have been shorter. Region III is the area that is theoretically covered by the sonar beam, but is unknown whether it is occupied or empty because it is in the shadow of whatever was in Region I. Region IV is outside of the beam and not of interest.

SPECULAR REFLECTION

Although they are inexpensive, fast, and have a large operating range, ultrasonic sensors have many shortcomings and limitations which a designer should be aware of. Ultrasonic sensors rely on reflection, and so are susceptible to *specular reflection*. Specular reflection is when the wave form hits a surface at an acute angle and the wave bounces away from the transducer. Ideally all objects would have a flat surface perpendicular to the transducer, but of course, this rarely happens. To make matters worse, the reflected signal may bounce off of a second object, and so on, until by coincidence return some energy back to the transducer. In that case, the time of flight will not correspond to the true relative range.

Even with severely acute angles, the surface is usually rough enough to send some amount of sound energy back. An exception to this is glass, which is very common in hospitals and offices where mail robots operate, but induces serious specular reflection. Fortunately this energy is often

sufficiently

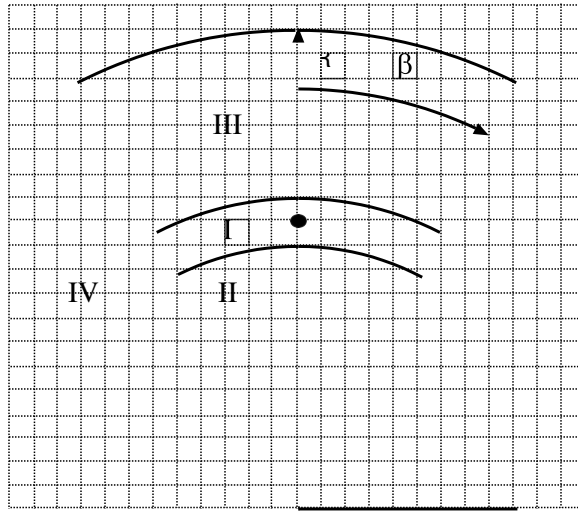


Figure 6.6 The regions of space observed by an ultrasonic sensor.

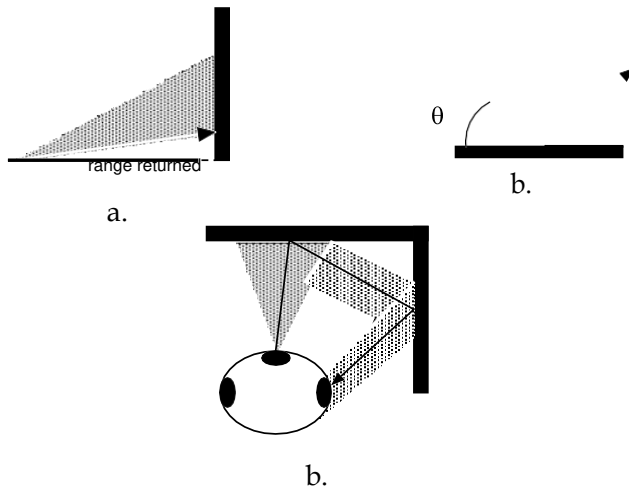


Figure 6.7 Three problems with sonar range readings: a.) foreshortening, b.) specular reflection, and c.) cross-talk.

FORESHORTENING

strong to pass the thresholding in the transducer circuit. However, a new problem, *foreshortening*, may occur. Recall that a sonar has a 30° field of view. This means that sound is being broadcast in a 30° wide cone. If the surface is not perpendicular to the transducer, one side of the cone will reach the object first and return a range first. Most software assumes the reading is along the axis of the sound wave. If it uses the reading (which is really the reading for 15°), the robot will respond to erroneous data. There is no solution to this problem.

SPECULAR REFLECTION

Specular reflection is not only by itself a significant source of erroneous readings; it can introduce a new type of error in rings of sonars. Consider a ring of multiple sonars. Suppose the sonars fire (emit a sound) at about the same time. Even though they are each covering a different region around the robot, some specularly reflected sound from a sonar might wind up getting received by a completely different sonar. The receiving sonar is unable to tell the difference between sound generated by itself or by its peers. This source of wrong reading is called *cross-talk*, because the sound waves are getting crossed. Most robot systems stagger the firing of the sonars in a fixed pattern of four sonars, one from each quadrant of the ring) at a time. This helps some with cross-talk, but is not a complete or reliable solution. If the sonar sound frequency and firing rate can be changed (which is generally not the case), then sophisticated aliasing techniques can be applied. These techniques are outside the scope of this book.

CROSS-TALK

One researcher, Monet Soldo, told a story of when she developed a reactive mobile robot for IBM's T.J. Watson Laboratories during the late 1980's. The robot used sonar as its primary sensors, and she had written behaviors to guide the robot through doors, rooms, and hall successfully at reasonable speeds. The day came for the big demonstration, which was to be held not in the hallways of the laboratory but in the front reception area. The robot navigated successfully out of the lab, down the halls, and then went berserk when it got to the atrium. She rebooted, and tried again, but with the same result. After days of trying to debug the code, she realized it wasn't a code problem, it was an environment problem: most of the atrium reception area consisted of glass partitions. The specular reflection and cross-talk caused the robot to hallucinate, although in different ways each time.

I had a similar problem when my robot started navigating in an office environment. In that environment, the robot was expected to navigate among office cubes, or work areas delimited by partitions. The partitions

were covered with cloth to dampen the sound from the workers.

Unfortunately, the cloth also absorbed the sound from the sonars! These stories emphasize the

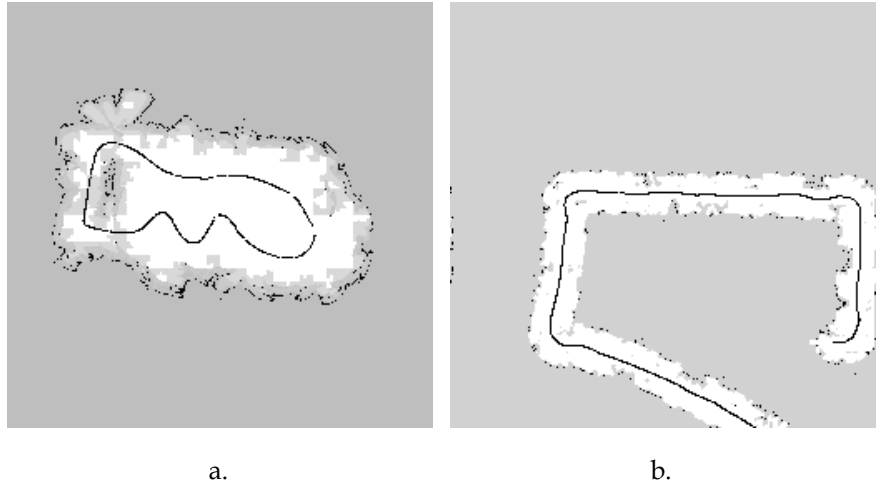


Figure 6.8 Maps produced by a mobile robot using sonars in: a.) a lab and b.) a hallway. (The black line is the path of the robot.)

need to consider the operating environment for the sensor and its impact on the signal.

The impact of the problems of specular reflection and cross talk become easier to see with plots of sonar returns overlaid with the perimeter of the area they were taken in; see Fig. 6.8. Some walls are invisible, others are too close. Clearly sonar readings have to be taken with a grain of salt.

The 30 cone also creates resolution problems. While sonars often have excellent resolution in depth, they can only achieve that at large distances if the object is big enough to send back a significant portion of the sound wave. The further away from the robot, the larger the object has to be. Most desk chairs and table tops present almost no surface area to the sensor and so the robot will often not perceive their presence and run right into them.

In practice, another problem leads to spurious sonar readings: power. The generation of a sound wave requires a significant pulse of energy. If the robot is operating at low power levels, the correct waveform will not

be generated and the return signal worthless. This problem is often difficult to debug

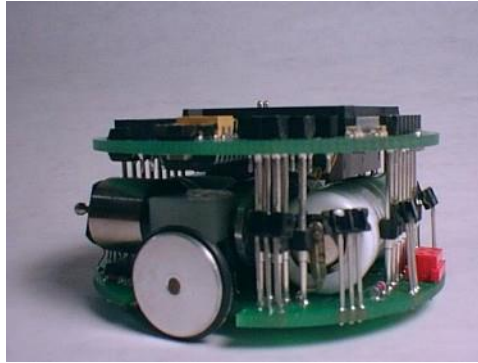


Figure 6.9 The ring of IR sensors on a Khepera robot. Each black square mounted on three posts is an emitter and receiver.

by looking at the sonar returns, which often suggest specular reflection or crosstalk. One team in a robot competition attached a voltmeter next to the emergency stop button on their robot to ensure that this problem would not go undetected.

One method of eliminating spurious readings, regardless of cause, is to take the average of three readings (current plus the last two) from each sensor. This method is fairly common on purely reactive robots, but is ad hoc. As will be seen in later chapters, other approaches treat the reading as being uncertain and apply formal evidential reasoning techniques to refine the reading. These uncertainty techniques are employed by architectures operating under the Hybrid Paradigm, and will be covered in Ch. 11.

6.5.2 Infrared (IR)

Infrared sensors are another type of active proximity sensor. They emit near-infrared energy and measure whether any significant amount of the IR light is returned. If so, there is an obstacle present, giving a binary signal. IR sensors have a range of inches to several feet, depending on what frequency of light is used and the sensitivity of the receiver. The simplest IR proximity sensors can be constructed from LEDs, which emit light into the environment and have a range of 3-5 inches. Fig. 6.9 shows the IR emitters and receivers placed side by side in a single rectangular

package on a Khepera robot. These often fail in practice because the light emitted is often “washed out” by bright

ambient lighting or is absorbed by dark materials (i.e., the environment has too much noise).

In more sophisticated IR sensors, different IR bands can be selected or modulated to change the signal-to-noise ratio. This typically ensures that an object in range doesn't absorb the light, causing the sensor to miss the presence of the object. Nomad robots have an IR sensor option.

6.5.3 Bump and feeler sensors

TACTILE

Another popular class of robotic sensing is *tactile*, or touch, done with bump and feeler sensors. Feelers or whiskers can be constructed from sturdy wires. Bump sensors are usually a protruding ring around the robot consisting of two layers. Contact with an object causes the two layers to touch, creating an electrical signal. In theory, the sensitivity of a bump sensor can be adjusted for different contact pressures; some robots may want a "light" touch to create a signal rather than a "heavier" touch. In practice, bump sensors are annoyingly difficult to adjust. In the "Hors d'Oeuvres, Anyone?" event of the 1997 AAAI Mobile Robot Competition, humans were served finger food by robot "waiters." The robot waiters were various research robots with serving trays attached. Humans were supposed to communicate to the Colorado School of Mines' Nomad 200 robot waiter (seen in Fig. 6.10) that they were done eating by kicking the bump sensor. The sensitivity of the bump sensor was so low that it often required many kicks, producing a very comical scene with Bruce Lee overtones.

Placement of bump sensors is a very important issue. The bump sensors on a Nomad 200 base clearly protect the robot only from low obstacles not perceivable by sonar. The Denning mobile robot platforms built in the 1980's used a bump sensor that looked much like a thick piece of gray tape. Denning mobile robots look like a fat version of the Nomad 200's, and the bump sensor is wrapped around the cowling of the robot at the waist level. Unfortunately, in certain turning configurations, the wheels extend beyond the cowling. In those situations, the bump sensor is totally useless in preventing the expensive synchro-drive mechanism from being damaged in a collision.

Feeler sensors are whiskers or antennae, only not as sensitive as those on animals. Contact of a whisker and an object will trip a binary switch, whereas there is reason to believe that an insect or animal can extract

much more information. Bump and feeler sensors are actually tactile sensors since they require the robot to be touching something in order to generate a reading.



a.



b.

Figure 6.10 CRISbot, the Colorado School of Mines' entry in the 1997 AAAI Mobile Robot Competition Hors d'Oeuvres, Anyone? event. a.) Interacting with the audience. b.) Audience members "communicated" by kicking the two protruding bumpers near the bottom of the robot. (Photographs courtesy of AAAI.)

However, they are often interchangeable with IR sensors because IR sensors often operate over the short range (inches) with less reliability.

6.6 Computer Vision

COMPUTER VISION
IMAGE

PIXELS

Computer vision refers to processing data from any modality which uses the electromagnetic spectrum which produces an image. An *image* is essentially a way of representing data in a picture-like format where there is a direct physical correspondence to the scene being imaged. Unlike sonar, which returns a single range reading which could correspond to an object anywhere within a 30 cone, an image implies multiple readings placed in a two-dimensional array or grid. Every element in the array maps onto a small region of space. The elements in

image arrays are called *pixels*, a contraction

IMAGE FUNCTION

of the words “picture element.” The modality of the device determines what the image measures. If a visible light camera is used, then the value stored at each pixel is the value of the light (e.g., color). If a thermal camera is used, then the value is the heat at that region. The function that converts a signal into a pixel value is called an *image function*.

Computer vision includes cameras, which produce images over the same electromagnetic spectrum that humans see, to more exotic technologies: thermal sensors, X-rays, laser range finders, and synthetic aperture radar. Simple forms of computer vision are becoming more popular due to the drop in prices and miniaturization of cameras and because reactive robots need to exploit affordances such as color or texture.

As noted in the Introduction, computer vision is a separate field of study from robotics, and has produced many useful algorithms for filtering out noise, compensating for illumination problems, enhancing images, finding lines, matching lines to models, extracting shapes and building 3D representations. Reactive robots tend not to use those algorithms. Most of the algorithms, especially those that remove noise, require many computations on each pixel in the image; until recently, the algorithms were too computationally expensive to run in real-time. Also, there was a resistance to algorithms which required any type of memory or modeling. Therefore a robot designed to follow paths which used vision to extract the path boundary lines in the current image based on knowledge of the width, then predicted where the path boundary lines should be in the next image would be on the borderline of reactivity.

6.6.1 CCD cameras

Computer vision on reactive robots is most often from a video camera, which uses CCD (charged couple device) technology to detect visible light. A video camera, such as a camcorder, is arranged so that light falls on an array of closely spaced metal-oxide semiconductor (MOS) capacitors. Interestingly, the MOS capacitors are rectangular, not square, so there is some distortion in creating the image. The capacitors form a shift register, and output is either a line at a time (“line transfer”) or the whole array at one time (“frame transfer”).

The output of most consumer video signals is analog, and must be digitized for use by a computer. Consumer digital cameras post an analog signal, but the update rate is too slow at this time for real-time reactive

robot control. The A/D conversion process takes longer than the CCD array can sense

FRAMEGRABBER

light, so the camera device can either have many frame buffers, which create a pipeline of images (but is expensive), or have a low frame rate.

A *framegrabber* is a card which fits inside a computer, accepts analog camera signals and outputs the digitized results. The card has a software driver which allows the robot software to communicate with the board. Framegrabbers can produce a grayscale or a color digital image. In the early part of the 1990's, color-capable framegrabbers were prohibitively expensive, costing around \$3,000 USD. Now color framegrabbers can be purchased from \$300 to \$500 USD, and TV tuners which can capture a single frame are available for \$50 USD.

6.6.2 Grayscale and color representation

The framegrabber usually expresses the grayscale value of a pixel as an 8 bit number (1 byte of computer memory). This leads to 256 discrete values of gray, with 0 representing black and 255 representing white. (Remember, 256 values means 0. 255.)

Color is represented differently. First, there are many different methods of expressing color. Home PC printers use a subtractive method, where cyan plus yellow make green. Most commercial devices in the U.S. use a NTSC (television) standard. Color is expressed as the sum of three measurements:

RGB

COLOR PLANES

red, green, and blue. This is simply abbreviated as *RGB*.

RGB is usually represented as three *color planes*, or axes of a 3D cube as shown in Fig. 6.11. The cubic represents all possible colors. A specific color is represented by a tuple of three values to be summed: (R, G, B). Black is (0,0,0) or 0+0+0, or no measurements on any of the three color planes. White is (255, 255, 255). The pure colors of red, green, and blue are represented by (255,0,0), (0,255,0), and (0,0,255) respectively. This is the same as in color graphics.

Notice that the cube dimensions in the figure are 256 by 256 by 256, where 256 is the range of integers that can be expressed with 8 bits. Since there are three color dimensions, a manufacturer may refer to this as 24-bit color (3×8), to distinguish their framegrabber from ones which map color onto a linear grayscale. The 8-bit color model is what is used to colorize old black and white movies. There are only 256 values of color, which is quite limited, and the gray values are often ambiguous. The pixel values of a person's red lips might be 185, while their dark blue

dress²²⁰ is also 185.
A person may
have to indicate
which regions in
each frame of the
film where
185=red and
185=dark blue. 8-
bit color is not
often used for
robots, unless the
robot will

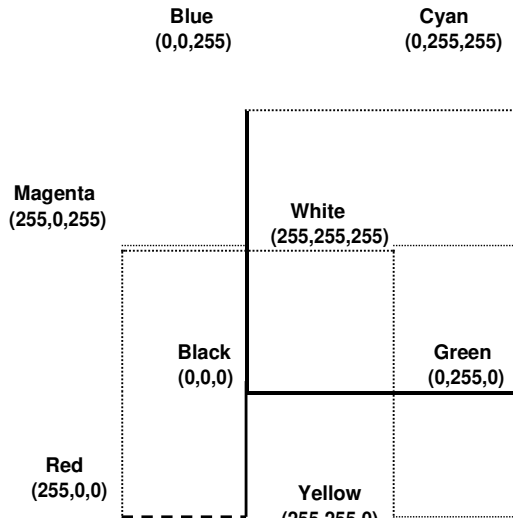


Figure 6.11 RGB color cube.

be operating in an environment where the only visible colors will not have an ambiguity.

24-bit color is usually sufficient for robotics. For other applications of computer vision such as medical imaging, weather forecasting, or military reconnaissance, an 8 bit resolution is often insufficient. Those applications may use 10 bits for each color plane. Since 10 bits do not fall on a byte boundary, it can be awkward to program algorithms for representing and manipulating this kind of image. Companies such as Pixar make special computers for these applications.

In programming terms, 24-bit color image is often declared in computer programs in two ways:

1. *Interleaved.* Interleaved means the colors are stored together, `RGB RGB RGB . . .`, and it is the more common representation. The order is almost always red, green, then blue, although there may be framegrabbers which do not follow that convention. Below is a code fragment where the color is displayed for a pixel at location `row`, `col`.

```
#define RED 0
#define GREEN 1
#define BLUE 2
```

```

int          image[ROW][COLUMN][COLOR_PLANE];
...
red = image[row][col][RED];
green = image[row][col][GREEN];
blue = image[row][col][BLUE];
display_color(red, green, blue);

```

- 2 *Separate*. Some framegrabbers store an image as three separate two-dimensional arrays, as shown below. Some framegrabbers have functions which return each color plane separately or interleaved. The equivalent code fragment is:

```

int          image_red[ROW][COLUMN];
int          image_green[ROW][COLUMN];
int          image_blue[ROW][COLUMN];
...
red = image_red[row][col];
green = image_green[row][col];
blue = image_blue[row][col];
display_color(red, green, blue);

```

The RGB representation has disadvantages for robotics. Color in RGB is a function of the wavelength of the light source, how the surface of the object modifies it (surface reflectance), and the sensitivity of the sensor. The first problem is that color is not absolute. RGB is based on the sensitivity of the three color sensing elements to reflected light. An object may appear to be at different values at different distances due to the intensity of the reflected light. Fig. 6.12 shows the same program and parameters to segment an orange landmark that is serving as a “flag” for tracking a small robot just outside of the image. The RGB segmentation in Fig. 6.12a is more correct than in Fig. 6.12b. The only difference is that the flagged robot has moved, thereby changing the incidence of the light. This degradation in segmentation quality is called *visual erosion*, because the object appears to erode with changes in lighting. Moreover, CCD devices are notoriously insensitive to red. This means that one of the three color planes is not as helpful in distinguishing colors.

Clearly a device which was sensitive to the absolute wavelength of the reflected light (the hue) would be more advantageous than having to work

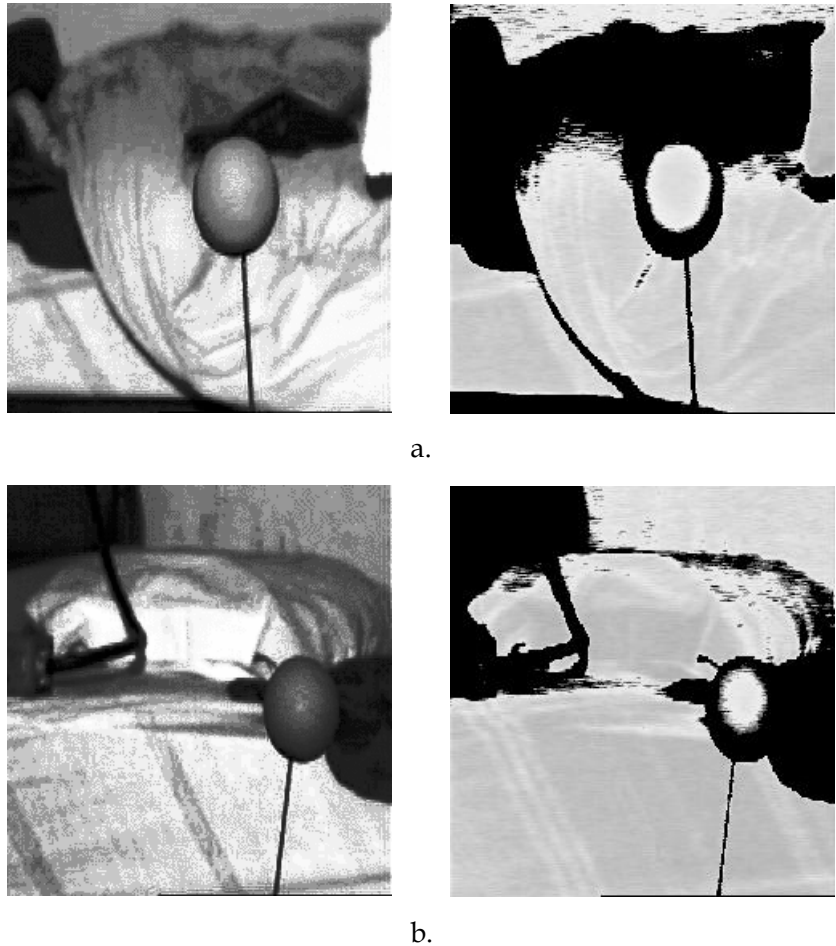


Figure 6.12 Images showing visual erosion of an orange landmark sticking up from a small robot (not visible): a.) Original image and RGB segmentation and b.) original image and degradation in RGB segmentation as robot moves farther from camera.

HSI	around the limitations of RGB. Such a device would work on the <i>HSI</i>
HUE	(hue, saturation, intensity) representation of color. The <i>hue</i> is the
SATURATION	dominant wave- length and does not change with the robot's relative
VALUE	position or the object's shape. <i>Saturation</i> is the lack of whiteness in the
INTENSITY	color; red is saturated, pink is less saturated. The <i>value</i> or <i>intensity</i>
	measure is the quantity of light received
	by the sensor. So HSV is a very different color scheme than RGB.

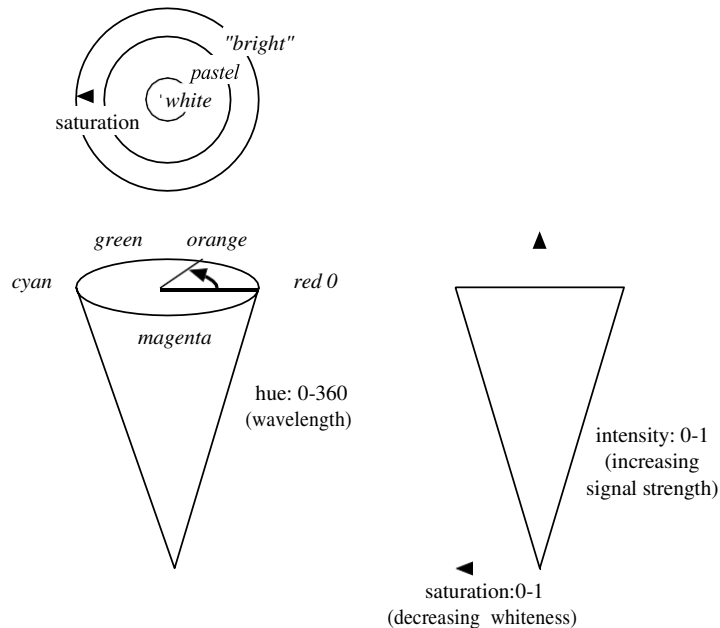


Figure 6.13 HSV space representation.

HSV is a three-dimensional space in that it has three variables, but it is definitely not a cube representation, more of a cone as seen in Fig. 6.13. The hue, or color, is measured in degrees from 0 to 360. Saturation and intensity are real numbers between 0 and 1. These are generally scaled to 8-bit numbers. Accordingly, red is both 0 and 255, orange is 17, green is at 85, blue is 170, with magenta at 200.

HSV space is challenging for roboticists for many reasons. First, it requires special cameras and framegrabbers to directly measure color in HSV space. This equipment is prohibitively expensive. Second, there is a software conversion from the RGB space measured by consumer electronics, but it is computationally expensive and has singularities (values of RGB where the conversion fails). These singularities occur at places where the three colors for a pixel are the same; the flatness of the red color plane in CCD cameras increases the likelihood that a singularity will occur.

An alternative color space that is currently being explored for robotics is the Spherical Coordinate Transform (SCT).¹⁴⁰ That color space was

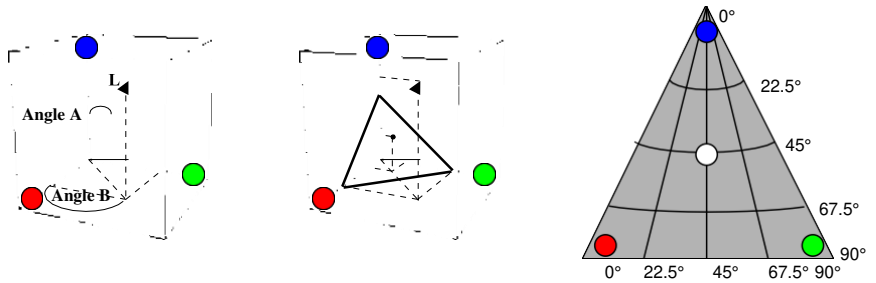


Figure 6.14 Spherical coordinate transform.

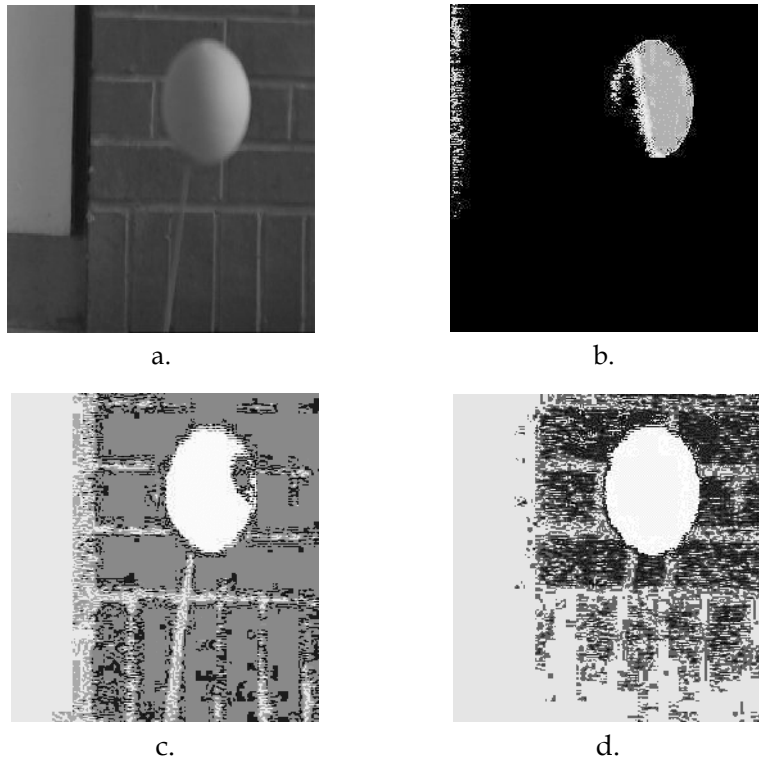


Figure 6.15 A comparison of color spaces: a.) a landmark in the original image, b.) RGB segmentation, c.) HSI segmentation, and d.) SCT segmentation.

to transform RGB data to a color space that more closely duplicates the response of the human eye. It is used in biomedical imaging, but has not been widely considered for robotics. The shape of the color space is triangular, as shown in Fig. 6.14. Initial results indicate it is much more insensitive to changes in lighting.⁷⁴ Fig. 6.15 shows an image and the results of segmenting a color in RGB, HSI, and SCT spaces.

6.6.3 Region segmentation

The most ubiquitous use of computer vision in reactive robotics is to identify a region in the image with a particular color, a process called *region segmentation*.

Region segmentation and color affordances are a staple perceptual algorithm for successful entries to many different international robot competitions, including the AAAI Mobile Robot Competition, RoboCup, and MIROSOT. There are many color region segmentation algorithms available and Newton Labs sells a *Cognachrome* board dedicated to the rapid extraction of colored regions. The basic concept is to identify all the pixels in an image which are part of the region and then navigate to the region's center (centroid). The first step is to threshold all pixels which share the same color (thresholding), then group those together and throw out any pixels which don't seem to be in the same area as the majority of the pixels (region growing).

Ch. 5 described a robot which used red to signify a red Coca-Cola can for recycling. Ideally, the robot during the searching for the can behavior would

BINARY IMAGE
THRESHOLDING

see the world as a *binary image* (having only 2 values) consisting of red, not-red. This partitioning of the world can be achieved by *thresholding* the image and creating a binary image. A C/C++ code example is shown below:

```
for (i = 0; i < numberRows; i++)
  for (j = 0; j < numberColumns; j++) {
    if ((ImageIn[i][j][RED] == redValue)
        && (ImageIn[i][j][GREEN] == greenValue)
        && (ImageIn[i][j][BLUE] == blueValue)) {
      ImageOut[i][j] = 255;
    }
    else {
      ImageOut[i][j] = 0;
    }
  }
```


Note that the output of a thresholded color image is a two-dimensional array, since there is no need to have more than one value attached at each pixel. Also, in theory a binary image would permit only values of 0 and 1. However, on many compilers there is no particular benefit to doing a bit level representation, and it can complicate code reuse. Also, most display software is used to displaying at least 256 values. The difference between 1 and 0 is not detectable by the human eye. Therefore, it is more common to replace the 1 with a 255 and use a full byte per pixel.

Thresholding works better in theory than in practice due to the lack of color constancy. The shape of an object will mean that even though a human sees the object as a solid color, the computer will see it as a set of similar colors. The common solution is to specify a range of high and low values on each color plane. The C/C++ code would now become:

```
for (i = 0; i < numberRows; i++)
    for (j = 0; j < numberColumns; j++) {
        if (((ImageIn[i][j][RED] >= redValueLow)
            && (ImageIn[i][j][RED] <= redValueHigh))
            && ((ImageIn[i][j][GREEN] >= greenValueLow)
                && (ImageIn[i][j][GREEN] <= greenValueHigh))
            && ((ImageIn[i][j][BLUE] >= blueValueLow)
                && (ImageIn[i][j][BLUE] <= blueValueHigh))) {
            ImageOut[i][j] = 255;
        }
        else {
            ImageOut[i][j] = 0;
        }
    }
}
```

The change in viewpoints and lighting means that the range which defines the object from the robot's current position is likely to change. The color range for the object has to be made even wider to include the set of color values the object will take. If the object color is unique for that environment, this increase in the color range is acceptable. Otherwise, if there are objects which have a color close enough to the object of interest, those objects may be mistaken for the target. In some circles, the object of interest is called the *foreground*, while everything else in the image is called the *background*. Thresholding an image requires a significant contrast between the background and foreground to be successful.

FOREGROUND

BACKGROUND

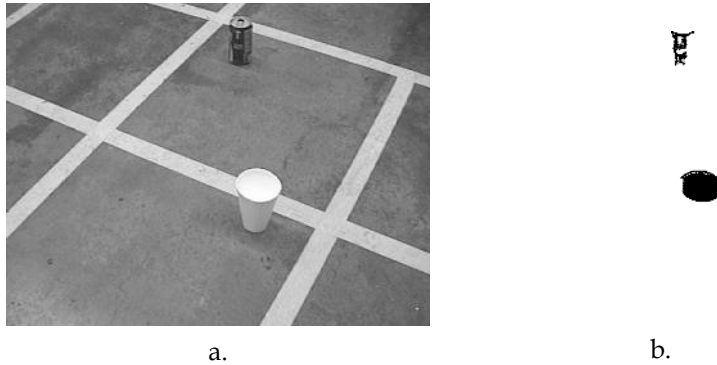


Figure 6.16 Segmentation of a red Coca Cola can: a.) original image and b.) resulting red regions. Note that some non-object pixels showed a reddish color.

Fig. 6.16 is the output of a threshold on color alone. If a robot was to move to the “red” in the image, how would it know which pixel to follow? The perceptual schema could be instantiated for each red pixel; this is simple, but it would waste a lot of execution cycles. The perceptual schema could take the weighted centroid of all the red pixels. In this case, it would be somewhat about where most people would say the center of the can was. Or, the perceptual schema could attempt to find the largest region where red pixels were adjacent to each other, then take the centroid of that region. (The region is often referred to as a “blob,” and the extraction process is known as blob analysis.)

Color regions can also be helpful in cluttered environments. Fig. 6.17 shows a Denning mobile robot simulating a search of a collapsed building. The international orange vest of the workman provides an important cue. The robot can signal a teleoperator when it sees bright colors.

6.6.4 Color histogramming

Thresholding works well for objects which consist of one color or one dominant color. A different technique popularized by Mike Swain, called *color*

HISTOGRAMMING *histogramming*, can be used to identify a region with several colors.¹³⁷ Essentially, color histogramming is a way to match the proportion of colors in

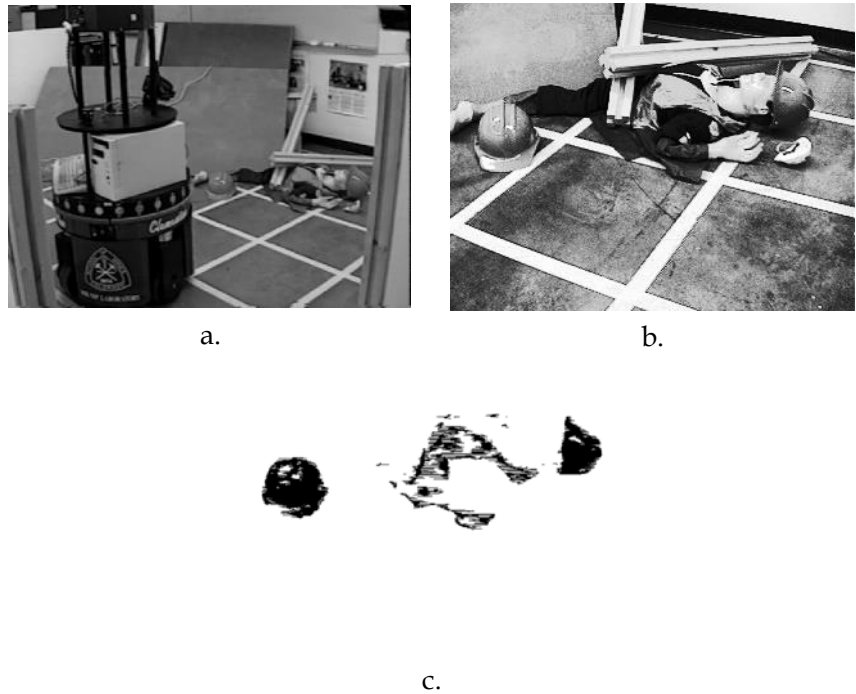
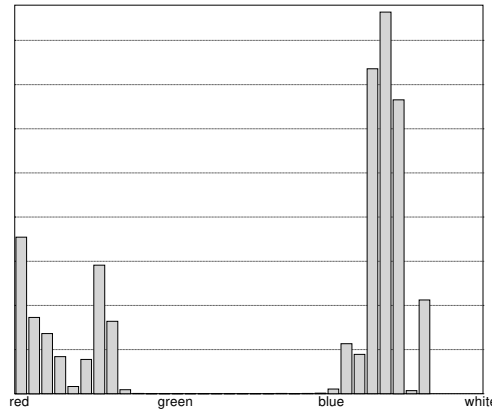


Figure 6.17 An urban search and rescue scene. a.) a Denning mobile robot searching, b.) image from the camera, and c.) segmentation for orange.

A color histogram is a type of histogram. A histogram is a bar chart of data. The user specifies the range of values for each bar, called buckets. The size of the bar is the number of data points whose value falls into the range for that bucket. For example, a histogram for a grayscale image might have 8 buckets (0-31, 32-63, 64-95, 96-127, 128-159, 160-191, 192-223, 224-251) and

each bucket contains the number of pixels in the image that fell into that range. Constructing a color histogram is straightforward for a region in hue space, as shown in Fig. 6.18.

A color histogram in RGB or any other distributed color space is a bit harder to visualize. The grayscale and hue image histograms had only one axis for buckets, because these images have only one plane that matters. But a color image has 3 planes in the RGB coordinate system. As a result, it has buckets for each color plane or axis. Assuming that each



a.



b.

Figure 6.18 a.) A histogram for b.) the image of the children's toy, Barney.

into 8 buckets, the first bucket would be the number of pixels which fell into the range of (R, G, B) of (0-31, 0-31, 0-31).

The real advantage of a color histogram for reactive robots is that color histograms can be subtracted from each other to determine if the current image (or some portion), I , matches a previously constructed histogram, E . The histograms are subtracted bucket by bucket (j buckets total), and the difference indicates the number of pixels that didn't match. The number of mismatched pixels divided by the number of pixels in the image gives a percentage match. This is called the histogram intersection:

$$(6.1) \quad \text{intersection} = \frac{\sum_{j=1}^n \min(I_j, E_j)}{\sum_{j=1}^n E_j}$$

For example, a robot can "wake up" and imprint the object in front of it by constructing the color histogram. Then a perceptual schema for a releaser or behavior can compute the color histogram intersection of the current image with the imprint. The robot can use the color histogram to determine if a particular object is of interest or not.

Because the color histogram of a current image can be matched with another image, the technique appears to be model-based, or recognition. But reactive systems do not permit recognition types of perception. Is this a contradiction? No; a color histogram is an example of a local, behavior-specific representation which can be directly extracted from the environment. For example, a robot could be shown a Barney doll with its distinct purple color with green belly as the percept for the goal for a move-to-goal behavior. However, the robot will follow a purple triangle with a green region, because the ratio of colors is the same. There is no memory and no inference, just a more complex stimulus.

Note that the intersection can be considered to be a measure of the strength of the stimulus, which is helpful in reactive robotics. In one set of experiments, a robot was presented a poster of Sylvester and Tweety. It learned the histogram, then after learning the object (e.g., fixating on it), it would begin to move towards it, playing a game of tag as a person moved the poster around. The robot used a simple attractive potential fields-based move-to-goal behavior, where the perceptual schema provided the location of the poster and the percent intersection. The motor schema used the location to compute the direction to the poster, but the intersection influenced the magnitude of the output vector. If the person moved the poster into a dark area or turned it at an angle, the intersection would be low and the robot would move slower. If the match was strong, the robot would speed up. Overall, it produced a very dog-like behavior where the robot appeared to play tag quickly (and happily) until the human made it too difficult. Then if the human moved the poster back to a more favorable position, the robot would resume playing with no hard feelings.

6.7 Range from Vision

An important topic in computer vision is how to extract range information. Humans extract depth from vision. In most cases, though not always, depth perception is due to having two eyes and being able to triangulate as shown in Fig. 6.20, also known as *stereopsis*. Other times, perception of depth is related to *optic flow* and simple cues such as shadows, texture, and expected size of objects. This section covers three types of vision sensors which are commonly used to create an image

STEREOPSIS

OPTIC FLOW

230
representing a
depth map: stereo
camera pairs, light
stripers, and laser
ranger finders.

6 Common Sensing Techniques for Reactive Robots



Figure 6.19 A Denning mobile robot using a color histogram to play tag with a poster of Sylvester and Tweety.

6.7.1 Stereo camera pairs

Using two cameras to extract range data is often referred to as *range from stereo*, *stereo disparity*, *binocular vision*, or just plain “stereo.” One way to extract depth is to try to superimpose a camera over each eye as in Fig. 6.20a. Each camera finds the same point in each image, turns itself to center that point in the image, then measures the relative angle. The cameras are known as the *stereo pair*.

STEREO PAIR

This method has two challenges. The first is that it is hard to design and build a mechanism which can precisely move to verge on the points. (It is even harder to design and build an inexpensive vergence mechanism.) The second challenge is even more fundamental: *how does the robot know that it is looking at the same point in both images?* This problem is referred to as the *correspondence* problem, since the task is to find a point in one image that corresponds to a point in the other image. A common approach is to identify “interesting” or potentially uniquely valued pixels in the image, such as very bright or dark spots or edges. The algorithm that selects interesting pixels is called an *interest operator*. Since even minute changes in lighting make a difference in the image, there is no guarantee that the two images, even though acquired at precisely the same time from two cameras, will “see” the same values for corresponding pixels. Therefore, interest operator algorithms usually

CORRESPONDENCE

INTEREST OPERATOR

return a list of interesting pixels, not just one, and a matching algorithm tries to find the best correspondence between all of them. After the interest

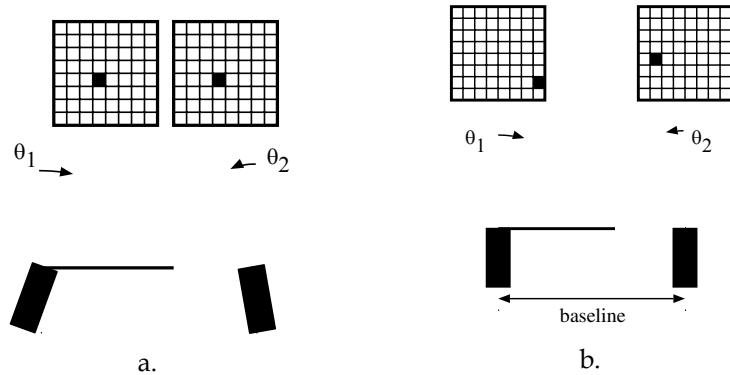


Figure 6.20 Ways of extracting depth from a pair of cameras: a.) vergence of the cameras to determine the depth of a point, and b.) a set of rectified stereo images.

points are established, the rest of the pixels have to be labeled with a depth relative to those points.

RECTIFIED IMAGES

DISPARITY

EPIPOLAR LINES

EPIPOLAR PLANE

Fortunately, it is not necessary to have a mechanical vergence system. Instead, cameras can be mounted in place with the optic axes parallel to each other and perpendicular to the mount, producing *rectified images*.⁴¹ This type of traditional stereo “head” is shown in Fig. 6.21. The space between the axes of two cameras is called the *baseline*. The distance in the location of the point of interest between the images is called the *disparity*; the distance of the point from the cameras is inversely proportional to disparity.⁶⁹ Fig. 6.20b shows the geometry behind a stereo pair.

While rectified images eliminate the need for expensive mechanical vergence systems, they don’t solve the correspondence problem. If the cameras are aligned precisely, then one row in the left image will correspond to a row in the right image. These rows are said to be *epipolar lines* or projections of a *epipolar plane*. Whenever the robot finds an interesting point in one image, it only has to consider the pixels along the epipolar line in the other image. This is a tremendous computational

savings. However,
it only works if the



Figure 6.21 A stereo camera pair mounted on a pan/tilt head.

CAMERA CALIBRATION

cameras are perfectly matched optically and remain in alignment. In practice, robots move, bump, and suffer alignment drifts, plus the cameras may have some flaws in their optics. The alignment can be periodically compensated for in software through a *camera calibration* process, where the robot is presented with a standard and then creates a calibration look up table or function. Fig. 6.22 shows the CMU Uranus robot calibrating its camera system. As a result, many researchers are turning to units which package a stereo pair in one fixed case, where the alignment cannot be altered. Fig. 6.25 shows the results using a stereo range system using three cameras in a fixed configuration.

The first robot to use stereo vision successfully was Hans Moravec's Stanford Cart shown in Fig. 6.23a Moravec worked on the Cart while at graduate school at Stanford between 1973 and 1980. Fig. 6.23b shows the Marsokhod rover developed in the late 1990's which used a stereo pair for real-time navigation. Longer baselines tend to be more accurate because a slight measurement error has a smaller impact, but smaller baselines have a smaller "footprint," in effect, take up less room. The same point in both images still has to be identified.

RANGE IMAGE

Fig. 6.24 shows the simplified flow of operations in extracting range from a pair of images. The process begins with two images, the left-right pair, and results in a third image called the *range image* or the *depth map*. The left-right

pair can be grayscale or color, but the depth map is a grayscale map, where intensity is proportional to the distance the pixel is away from the

6.7 Range from Vision cameras. Fig. 6.25 shows two stereo images and the resulting depth map.²³⁷

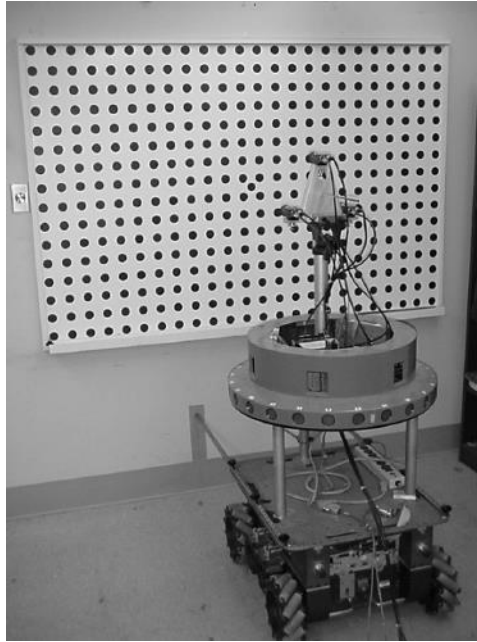
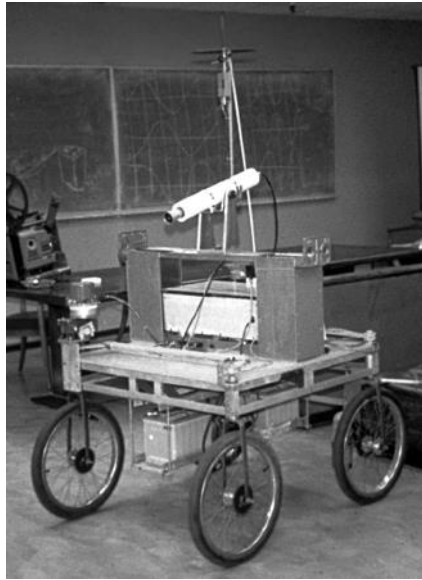


Figure 6.22 Uranus undergoing calibration. (Photograph courtesy of Hans Moravec.)

The major drawback with extracting range information from vision is that the algorithms tend to be computationally complex. Stereo matching algorithms are typically on the order of $O(n^2m^2)$. This means that to process an image of size 640 by 480 takes on the order of $9 \cdot 10^{10}$ instructions, while to simply segment a color is on the order of $O(nm)$ or $3 \cdot 10^5$ instructions. Even with advances in microprocessors, a stereo range map can take minutes to compute.

6.7.2 Light stripers

Light striping, light stripers or structured light detectors work by projecting a colored line (or stripe), grid, or pattern of dots on the environment. Then a regular vision camera observes how the pattern is distorted in the image. For example, in Fig. 6.26a, the striper projects four lines. The lines should occur at specific, evenly spaced rows in the camera image if the surface is flat. If the surface is not flat, as shown in Fig. 6.26b, the lines will have breaks or



a.



b.

Figure 6.23 Robots and stereo: a.) The Stanford Cart developed in the late 1970's. (Photograph courtesy of Hans Moravec.) b.) The Marsokhod rover developed in the late 1990's jointly by scientists from McDonnell Douglas, Russia, and NASA Ames Research Center. (Image courtesy of the National Aeronautics and Space Administration.)

discontinuities. A vision algorithm can quickly scan each of the designated rows to see if the projected line is continuous or not. The location of the breaks in the line give information about the size of the obstacle. The vision algorithm can also look for where the dislocated line segments appears, since the distance in image coordinates is proportional to the depth of the object. The relative placement of the lines indicates whether the object is above the ground plane (an obstacle) or below (a hole or *negative obstacle*). The more lines or finer-grained grid, the more depth information.

Light stripers are less expensive for many reasons. First, since they are producing a line or pattern to be measured, expensive time-of-flight detectors are unnecessary. The detection is done by the camera. Second, producing a thick line that can be detected by a camera does not require a laser. Instead it can be done with structured light, an optical method which allows "regular" light to mimic desirable properties of laser light.

NEGATIVE OBSTACLE

Finally, light
stripers pro-

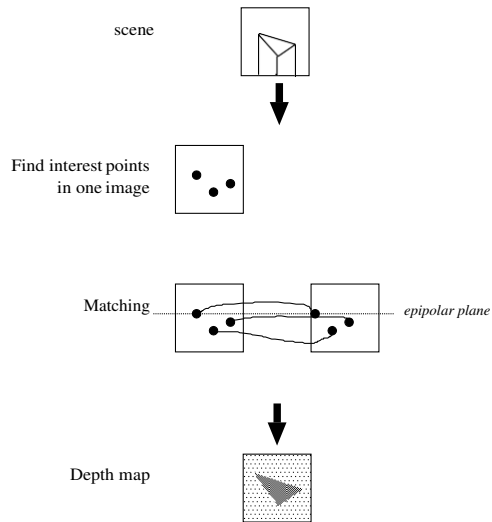


Figure 6.24 A simplified flow of operations in extracting range from a stereo pair.

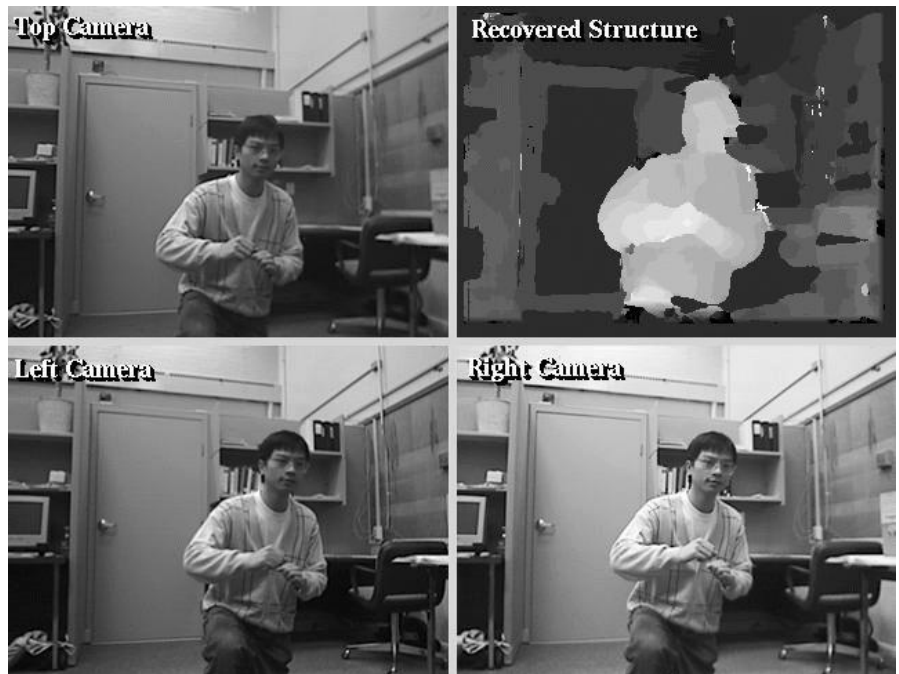


Figure 6.25 A set of stereo images from a Triclops stereo camera and resulting depth map. (Images courtesy of Chandra Kambhamettu.)

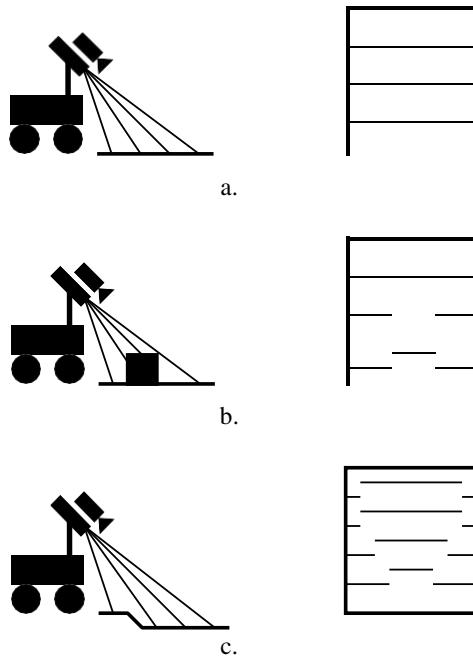


Figure 6.26 Situations and resulting images of a.) flat surface, b.) an obstacle, and c.) a negative obstacle.

duce a fairly coarse pattern; they don't project a line or grid onto every pixel in the image. This means the device is overall less demanding, and therefore less expensive to build.

Light stripers were popular in the late 1980's and early 1990's, and laboratories such as the University of Pennsylvania's GRASP Lab under the direction of Ruzena Bajcsy produced excellent results in extracting 3D information. However, these efforts focused on using the depth map to recognize an object under laboratory conditions. The results did not transfer particularly well to mobile robots. Reactive robots are not concerned with recognition, so many of the algorithms were not particularly useful or provided a quick, reflexive response. Also, in the open world, objects were often the same color as the projected light or near enough to it to confuse the vision system. The amount and direction of lighting could also confuse the striper, with brightly lit rooms making it difficult to see the bright laser or structured light.

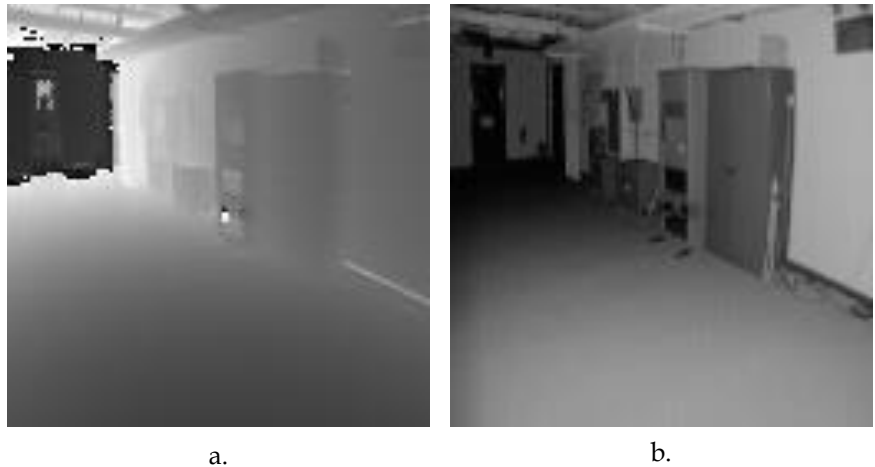


Figure 6.27 Lidar images from an Odetics LADAR camera: a.) range and b.) intensity. (Images courtesy of University of South Florida and Oak Ridge National Laboratory.)

The Sojourner Mars rover used a light striping system for obstacle avoidance. The system projected five lines ahead of the vehicle.¹³⁴ This worked well because Mars has little color or light diversity. Interestingly enough, the light striper used one member of a stereo pair, but unlike the Marsokhod, Sojourner did not use stereo for navigation. Instead, the robot would periodically take a stereo pair for reconstruction and map-making on Earth.

6.7.3 Laser ranging

Ultrasonics introduces acoustic energy into the environment and measures the time of flight of the signal to return. The same principle can be used with lasers; a laser beam can be emitted and then the reflectance measured. Unlike a sonar beam which has a very wide field of view (almost 30°), a laser produces an almost infinitesimal field of view. If the laser is directed in a scan, just like a raster scan on a CRT, the device can cover a reasonable area and produce an image, where the image function produces depth values. Devices which use lasers to produce a depth map or image are often called *laser radar*, *ladar*,

LADAR
LIDAR

6.7 *Range from Vision*
or *Haas*. They can generate up to one million range pixels per second, ⁵²⁴ with a range of 30 meters and an accuracy of a few millimeters. The mechanical



Figure 6.28 Sick laser, covering a 180 area.

scanning component makes lidars very expensive, on the order of \$30,000 to \$100,000 USD. A less expensive solution for navigation is to create a planar laser range finder.

A lidar produces two images: intensity and range. Fig. 6.27 shows the images produced by a Odetics laser range (LADAR) camera. The intensity map is essentially a black and white photograph and measures the intensity of the light reflected or absorbed by objects in the scene. This corresponds to how humans perceive the scene. The image function for the range image is depth from the camera. Pixels that are black, or have a value of 0, are closer than white pixels. A flat floor usually appears as a radiating set of semi-circles going from near to far; trigonometry is then used to compute that the circles represent a flat surface. This process is called *range segmentation* and can be quite difficult.

Lidars have some problems in practice. For example, Fig. 6.27 shows an area on the range image that is pure black or very near. But as can be seen from the intensity image the area is actually far away. Likewise, the black moulding between the wall and floor appear to be very far away on the range image. The errors were due to out of range conditions, absorption of the light (not enough light returned), or to the optical equivalent of specular reflection (light hitting corners gets reflected away from the receiver).

A planar laser range finder, such as the Sick shown in Fig. 6.28, provides

a 240° narrow
horizontal range
map. The map is
essentially a high
resolution polar

6 Common Sensing Techniques for Reactive Robots

plot. Robots such as Nomads and Pioneers originally came with Sick lasers mounted in parallel to the floor. This was useful for obstacle avoidance (as long as the obstacle was tall enough to break the laser plane), but not particularly helpful for extracting 3D information. Also, as with sonars, robots ran the risk of being decapitated by obstacles such as tables which did not appear in the field of view of the range sensor but could hit a sensor pod or antenna. To combat this problem, researchers have recently begun mounting planar laser range finders at a slight angle upward. As the robot moves forward, it gets a different view of upcoming obstacles. In some cases, researchers have mounted two laser rangefinders, one tilted slightly up and the other slightly down, to provide coverage of overhanging obstacles and negative obstacles.

6.7.4 Texture

The variety of sensors and algorithms available to roboticists can actually distract a designer from the task of designing an elegant sensor suite. In most cases, reactive robots use range for navigation; robots need a sensor to keep it from hitting things. Ian Horswill designed the software and camera system of Polly, shown in Fig. 6.29, specifically to explore vision and the relationship to the environment using subsumption.⁷⁰ Horswill's approach

LIGHTWEIGHT VISION

is called *lightweight vision*, to distinguish its ecological flavor from traditional model-based methods.

Polly served as an autonomous tour-guide at the MIT AI Laboratory and Brown University during the early 1990's. At that time vision processing was slow and expensive, which was totally at odds with the high update rates needed for navigation by a reactive mobile robot. The percept for the obstacle avoidance behavior was based on a clever affordance: texture. The halls of the AI Lab were covered throughout with the same carpet. The "color" of the carpet in the image tended to change due to lighting, but the overall texture or "grain" did not. In this case, texture was measured as edges per unit area, as seen with the fine positioning discussed in Ch. 3.

RADIAL DEPTH MAP

The robot divided the field of view into angles or sectors, creating a *radial depth map*, or the equivalent of a polar plot. Every sector with the texture of the carpet was marked empty. If a person was standing on the carpet, that patch would have a different texture and the robot would

mark²⁴⁰ the area as occupied.

Although this methodology had some problems—for example, strong shadows on the floor created “occupied” areas—it was fast and elegant.

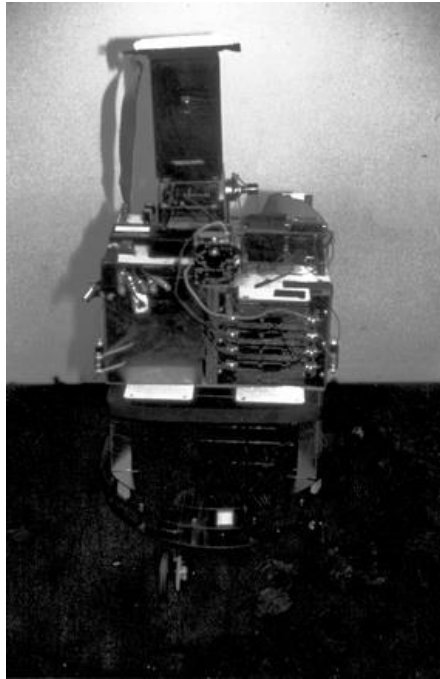


Figure 6.29 Polly, a visually guided reactive robot using a black and white camera. (Photograph courtesy of Ian Horswill and AAIL.)

6.8 Case Study: Hors d'Oeuvres, Anyone?

The University of South Florida's (USF) entry in the 1999 AAIL Mobile Robot Competition Hors d'Oeuvres, Anyone? event provides a case study of selecting sensors, constructing reactive behaviors, and using behavioral sensor fusion. The entry used two cooperative robots. The goal was to push the envelope in robotic sensing by using six different sensing modalities with 40 different physical devices on one robot and four modalities with 23 devices on the other. Although the robots executed under a hybrid deliberative/reactive style of architecture (covered later in Ch. 7), the basic design process followed the steps in Ch. 5, and produced a largely behavioral system. The entry won a Technical Achievement award of Best Conceptual and Artistic Design after a series of hardware and software timing failures prevented it from working at the competition.



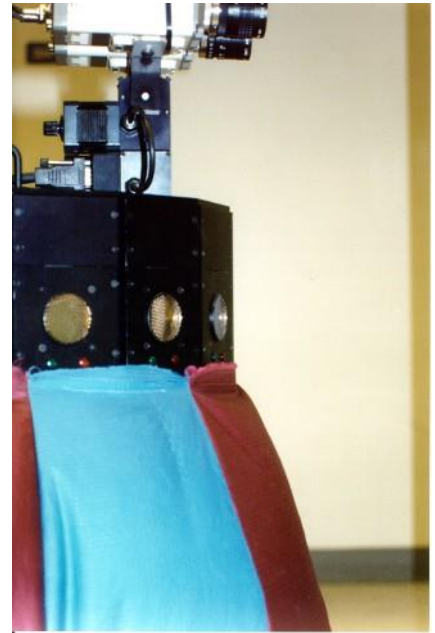
a.



b.



c.



d.

Figure 6.30 USF robots in the “Hors d’Oeuvres, Anyone?” event. a.) Family portrait, where Borg Shark is on the left, Puffer Fish on the right, with b.) the thermal sensor located as the Borg Shark’s “third eye,” c.) the SICK laser located behind the Borg Shark’s teeth (head piece is removed for better view), and d.) a profile of Puffer Fish’s skirt showing spatial relationship to sonar.

Step 1: Describe the task. The “Hors d’Oeuvres, Anyone?” event required fully autonomous robots to circulate in the reception area at the AAAI conference with a tray of finger food, find and approach people, interact with them, and refill the serving tray. Each robot was scored on covering the area, noticing when the tray needed refilling, interacting with people naturally, having a distinct personality, and recognizing VIPs. The USF entry used two robots, shown in Fig. 6.30, costumed by the USF Art Department in order to attract attention. The Borg Shark was the server robot, and navigated through audience following a pre-planned route. It would stop and serve at regular intervals or whenever a treat was removed from the tray. It used a DEC-talk synthesizer to broadcast audio files inviting audience members to remove a treat from its mouth, but it had no way of hearing and understanding natural language human commands. In order to interact more naturally with people, the Borg Shark attempted to maintain eye contact with people. If it saw a person, it estimated the location in image coordinates of where a VIP’s colored badge might be, given the location of the face.

When the Borg Shark was almost out of food, she would call over radio ethernet her assistant robot, Puffer Fish. Puffer Fish would be stationary in sleep mode, inhaling and exhaling through her inflatable skirt and turning her cameras as if avoiding people crowding her. When Puffer Fish awoke, she would head with a full tray of food (placed on her stand by a human) to the coordinates given to her by Borg Shark. She would also look for Borg Shark’s distinctive blue costume, using both dead reckoning and visual search to move to goal. Once within 2 meters of Borg Shark, Puffer Fish would stop. A human would physically swap trays, then kick the bumpers to signal that the transfer was over. Borg Shark would resume its serving cycle, while Puffer Fish would return to its home refill station.

Both robots were expected to avoid all obstacles: tables, chairs, people. Since there was a tendency for people to surround the robots, preventing coverage of the area or refilling, the robots had different responses. Borg Shark, who was programmed to be smarmy, would announce that it was coming through and begin moving. Puffer Fish, with a grumpy, sullen personality, would vocally complain, then loudly inflate her skirt and make a rapid jerk forward, usually causing spectators to back up and give her room.

Step 2: Describe the robots. The robots used for the entry were Nomad 200 bases, each with a unique sensor suite.

The original sensors on Borg Shark included a pair of color cameras mounted on a pan-tilt head, redundant sonar rings, and a SICK planar laser:

Type	Modality	Devices
exteroceptive	vision	2 cameras
exteroceptive	laser	1 planar ranger
exteroceptive	sonar	15 ultrasonics (upper) 15 ultrasonics (lower)
exteroceptive	tactile	1 bumper switch
proprioceptive	motor encoders	3 drive, steer, turret control 2 pan, tilt control

The sensors on Puffer Fish were:

Type	Modality	Devices
exteroceptive	vision	2 cameras
exteroceptive	sonar	15 ultrasonics
exteroceptive	tactile	1 bumper switch
proprioceptive	motor encoders	3 drive, steer, turret control 2 pan, tilt control

Step 3: Describe the environment. The environment was a convention center arena, with variable lighting and many people causally dressed. The major perceptual challenge was to maintain eye contact with people and to determine if a person is a VIP. The VIPs' badges were marked with distinctively colored ribbons, so if the robot was sure it was looking in the right area, the color would afford a VIP. However, the colors of the ribbon were fairly common. If the robot scanned at about chest height for ribbons, it would likely find a shirt of that color in the crowd who would be wearing bright tourist wear. This would cause it to misidentify a VIP, losing points. Another approach was to use sonars to find a person: read a range, point the camera in that direction and tilt to the angle where the eyes would be for a person of average height. This focus-of-attention mechanism could be used to look in the likely location of a badge. However, it was known from experience that people rarely stood along the acoustic axis of a sonar. If a single person was picked up by two sonars, the robot would look for 10 seconds to the left of the person and then 10 seconds to the right. If multiple people were present, the robot would seem even more dysfunctional, and the focus-of-attention mechanism would not work correctly.

A better solution would be to detect a person using vision. Notice that detection is not the same thing as recognition. Detection means that the robot is able to identify a face, which is reactive. Recognition would label the face and be able to recognize it at a later time, which is a deliberative function, not reactive. Is there a simple visual affordance of a face? Actually, to a vision system human skin is remarkably similar in color, regardless of ethnicity. Once the robot had found a colored region about the size and shape of a head, it could then more reliably find the VIP badges.

The other opportunity for an affordance was Puffer Fish's navigation to Borg Shark. Although Puffer Fish would receive Borg Shark's coordinates, it was unlikely that Puffer Fish could reliably navigate to Borg Shark using only dead reckoning. The coordinates were likely to be incorrect from Borg Shark's own drift over time. Then Puffer Fish would accumulate dead reckoning error, more so if it had to stop and start and avoid people. Therefore, it was decided that Puffer Fish should look for Borg Shark. Borg Shark's head piece was deliberately made large and a distinctive blue color to afford visibility over the crowd and reduce the likelihood of fixating on someone's shirt.

Step 4-7: Design, test, and refine behaviors. The choice of sensors for other behaviors, such as treat removal, was influenced by the physical location of the sensors. For example, the SICK laser for Borg Shark came mounted on the research platform as shown in Fig. 6.30b. The research platform, nominally the top of the robot, was at hand height, making it a logical place to attach a tray for holding food. It was obvious that the laser could be used to monitor the food tray area. Other teams tried various approaches such as having a colored tray and counting the amount of that color visible (more color means fewer treats on the tray, covering the color). Another approach was to build a scale and monitor the change in weight.

An interesting aspect of the robots that impacted the sensor suite indirectly were the costumes. As part of giving the robots personality, each robot had a costume. The Puffer Fish had an inflatable skirt that puffed out when the robot was crowded or annoyed. The team had to empirically test and modify the skirt to make sure it would not interfere with the sonar readings. Fig. 6.30c shows the profile of the skirt.

As seen in the behavior table below, the only behavior using any form of sensor fusion was `move-to-goal` in Puffer Fish, which had two

6.8 Case Study: *Hors d'Oeuvres, Anyone?*
competing instances of the goal making it sensor fission.

248

The initial behaviors for Borg Shark are given in the behavior table below:

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	avoid()	vfh()	most-open-direction	polar-plot(sonar)
FOOD-REMOVED= treat-removal(laser)	track-face	center-face(face-centroid)	face-centroid	find-face(vision)
		track-face() check-VIP()	ribbon-color	look-for-ribbon(VIP-color)
SERVING-TIME-OUT, TRAY-FULL=bumper()	move-to-goal	pfields.attraction(waypoint)	waypoint	list of waypoints
FOOD-DEPLETED= treat-removal(laser)	track-face	center-face(face-centroid)	face-centroid	find-face(vision)

The initial behavior table for Puffer Fish was:

Releaser	Behavior	Motor Schema	Percept	Perceptual Schema
always on	avoid()	vfh()	most-open-direction	polar-plot(sonar)
AT-HOME= dead-reckoning(encoders)	sleep()	turn-camera-head() cycle-skirt()	obstacle	polar-plot(sonar)
AWAKE=radio-signal()	move-to-goal()	pfields.attraction(location)	relative-location	read-encoders()
AWAKE=radio-signal()	move-to-goal()	pfields.attraction(shark)	shark	find-shark-blue(camera)
TRAY-FULL=bumper()	move-to-goal()	pfields.attraction(home)	relative-location	read-encoders()

The vfh behavior is an obstacle avoidance behavior using polar plots derived from models described in Ch. 11. As the team tested the behaviors individually, the find-face and treat-removal behaviors proved to be unreliable. While color was a reasonable affordance for a face, the algorithm often returned false negatives, missing faces unless in bright light. Meanwhile the laser appeared to occasionally get a reflection from the teeth, also generating false positives, and more than 75% of the time it would miss a person's hand if the motion was quick.

The rates were:

logical sensor	False Positives	False Negatives
Face-Find	1.7%	27.5%
Food-Count	6.7%	76.7%

The solution to the find-face performance was to exploit another affordance of a human, one used by mosquitoes: heat. The problem was partial segmentation; candidate regions were getting rejected on being too small. Heat would make a good the decision criteria. If a candidate region was co-located with a hot region, then it was declared a face. Fortunately, the team was able to transfer an E^2T digital thermometer

used on another robot to Borg Shark. The thermal sensor shown in Fig. 6.30 was intended for deter-

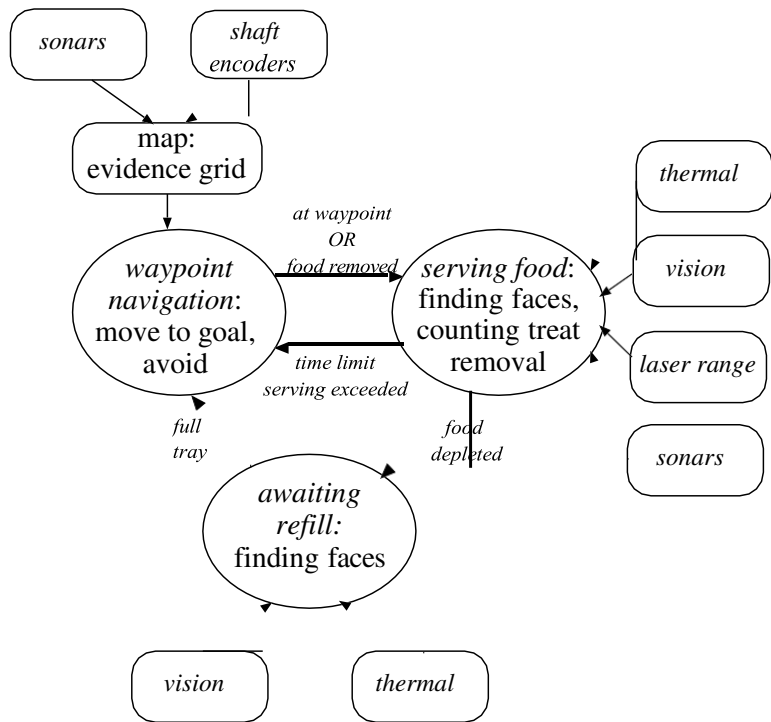


Figure 6.31 State diagrams for the Borg Shark, annotated to show sensors being used.

mining the temperature of a person on contact, but it was able to detect an increase in temperature above the ambient from a person up to two meters away.

Why wasn't the vision system simply replaced with the thermal sensor? This gets back to the attributes of sensors and how they fit with the environment. The thermal sensor has a 2 field of view, making it too slow to scan. Instead, the vision system covered a much wider field of view and could generate a small list of candidate regions. Then as the camera turned to center on the largest region, the thermal probe could decide whether this was really a person or not.

The problem with the food-count was greatly reduced by a simple AND function with the sonars. The system counted a treat as being removed only if there was a close range reading in front of the tray at the same time.

The false reading rate dropped considerably as seen below:

Logical sensor	W/O Fusion		Fusion	
	FP	FN	FP	FN
Face-Find	1.7%	27.5%	2.5%	0%
Food-Count	6.7%	76.7%	6.7%	1.7%

At this point it is helpful to step back and examine the sensing for the Hors d'Oeuvres, Anyone? entry in terms of the attributes listed in Sec. 6.3. Recall that the attributes for evaluating the suitability of an individual sensor were *field of view*, *range*, *accuracy*, *repeatability*, *resolution*, *responsiveness in target domain*, *power consumption*, *reliability*, and *size*. The field of view and range of the sensors was an issue, as seen by the differences in vision and thermal sensors for the face-finding behavior. The camera had a much better field of view than the thermal sensor, so it was used to focus the attention of the heat sensor. Repeatability was clearly a problem for laser with its high false positive/false negative rate. The sonars could not be used for estimating the location of a face because the resolution was too coarse. Each of the sensors had reasonable responsiveness from a hardware perspective, though the algorithms may not have been able to take advantage of them. Power consumption was not an issue because all sensors were on all the time due to the way the robots were built. Reliability and size of the hardware were not serious considerations since the hardware was already on the robots.

The algorithmic influences on the sensor design were *computational complexity* and *reliability*. Both were definitely a factor in the design of the perceptual schemas for the reactive behaviors. The robots had the hardware to support stereo range (two cameras with dedicated framegrabbers). This could have been used to find faces, but given the high computational complexity, even a Pentium class processor could not process the algorithm in real-time. Reliability was also an issue. The vision face finding algorithm was very unreliable, not because of the camera but because the algorithm was not well-suited for the environment and picked out extraneous blobs.

Finally, the sensing suite overall can be rated in terms of *simplicity*, *modularity*, and *redundancy*. The sensor suite for both Nomad robots can be considered simple and modular in that it consisted of several separate sensors, mostly commercially available, able to operate

independently of each other. The sensor suite did exhibit a high degree of physical redundancy: one robot had dual sonar rings, and the sonars, laser, and camera pair could have

been used for range, ignoring the placement of the shark teeth and the computational complexity. There was also a large amount of logical redundancy, which was exploited through the use of behavioral sensor fusion.

6.9 Summary

The success of a reactive system depends on the suitability of the robot's sensor suite. It is often more useful to think of sensing in terms of perceptual schemas or logical sensors needed to accomplish a task, rather than focus on the characteristics of a particular transducer or modality. Reactive perception may appear limited since it uses only behavior-specific representations and does not involve recognition. However, it supports a diversity of forms of perception, including behavioral sensor fusion. Advances in electronics have led to a plethora of range sensors and algorithms. Many of these are logically equivalent and can be used interchangeably; for example, with obstacle avoidance behaviors.

The design of a perceptual schema or a sensor suite requires a careful analysis. Each individual sensor should fit the task, power, and processing constraints. Likewise, the entire sensor suite should provide complete coverage of all perceptual processing required by the robot.

Almost all mobile robots have some form of proprioception, most likely shaft or wheel encoders used to estimate the distance traveled based on the number of times the motor has turned. Outdoor robots may carry GPS, and this trend is expected to increase as the cost of receivers goes down and inexpensive DGPS systems emerge.

Reactive navigation requires exteroception, whereby the robot observes the environment. Proprioception can guide a robot on a path, but exteroception can prevent it from hitting an unmodeled obstacle or falling off a cliff. The most common exteroceptive sensor on reactive robots is an ultrasonic transducer or sonar. An ultrasonic transducer is an active sensor which returns a single range reading based on the time-of-flight of an acoustic wave. Some of the difficulties associated with ultrasonics include erroneous readings due to specular reflection, crosstalk, and foreshortening. Other popular proximity sensors are IR and laser rangefinders.

Due to the low price and availability of consumer electronics, computer vision is becoming more common in robotic systems.

Computer vision processing operates on images, regardless of the modality which generated it. Color coordinate systems tend to divide an image into 3 planes. The two

most common color coordinate systems are RGB and HSV. HSV treats color in absolute terms, but RGB is favored by equipment manufacturers. A color space used in biomedical imaging, SCT, appears to be less sensitive to lighting conditions than RGB and RGB-derived HSV. Many reactive robots exploit color as an affordance. This can be done by thresholding an image and identifying regions of the appropriate color. A color affordance method which works well for objects with multiple colors is color histogramming. Stereo range finding is an important class of algorithms for navigation, though the computational complexity has prevented it being ported to many mobile robot applications. Laser range finders, particularly the inexpensive planar rangefinders, have grown in popularity over the past few years.

Despite the diversity of sensors and affordances inherent in the environment, reactive robotics is remarkable for its lack of sophistication in sensing. This may stem from the split between computer vision and robotics in the formative years of the field. Many roboticists still assume algorithms developed by computer vision specialists are too computationally expensive to work on commercially available on-board processors. This is no longer true, in part because of the increased computational power of general purpose chips. Readers are encouraged to explore the large body of literature on computer vision and free tools on the web.

7 *The Hybrid Deliberative/Reactive Paradigm*

Chapter objectives:

Be able to describe the Hybrid Deliberative/Reactive paradigm in terms of i) sensing, acting, and planning and ii) sensing organization.

Name and evaluate one representative Hybrid architecture in terms of: support for modularity, niche targetability, ease of portability to other domains, robustness.

Given a list of responsibilities, be able to say whether it belongs in the deliberative layer or in the reactive layer.

List the five basic components of a Hybrid architecture: sequencer agent, resource manager, cartographer, mission planner, performance monitoring and problem solving agent.

Be able to describe the difference between managerial, state hierarchy, and model-oriented styles of Hybrid architectures.

Be able to describe the use of state to define behaviors and deliberative responsibilities in state hierarchy styles of Hybrid architectures.

7.1 Overview

By the end of the 1980's, the trend in artificially intelligent robots was to design and program using the Reactive Paradigm. The Reactive Paradigm allowed robots to operate in real-time using inexpensive, commercially available processors (e.g., HC6811) with no memory. But the cost of reactivity, of course, was a system that eliminated planning or any functions which involved remembering or reasoning about the global state of the robot relative

7 *Multi-agents*

Chapter Objectives:

Define the types of *control regimes*, *cooperation strategies*, and *goals* in multi-agents.

Given a description of an intended task, a collection of robots, and the permitted interactions between robots, design a multi-agent system

250

6 Common Sensing Techniques for Reactive Robots

and describe the system in terms of heterogeneity, control, cooperation, and goals.

Compute the *social entropy* of a team.

Be able to program a set of homogeneous reactive robots to accomplish a foraging task.

Describe use of social rules and internal motivation for emergent societal behavior.

8.1 Overview

SOCIETIES

MULTI-AGENTS

SWARM ROBOTS

This chapter explores artificial intelligence methods for coordinating and controlling collections of mobile robots working on completing a task. Collections of two or more mobile robots working together are often referred to as teams or *societies* of multiple mobile robots, or more concisely *multi-agents*.

Multi-agent teams are desirable for many reasons. In the case of planetary explorers or removing land mines, more robots should be able to cover more area. Like ants and other insects, many cheap robots working together could replace a single expensive robot, making multi-agents more cost effective.

Indeed, the term *swarm robots* is becoming popular to refer to large numbers

of robots working on a single task. Another motivation of multiple robots is redundancy: if one robot fails or is destroyed, the other robots can continue and complete the job, though perhaps not as quickly or as efficiently. Rodney Brooks at MIT first proposed to NASA that teams of hundreds of inexpensive ant-like reactive robots be sent to Mars in a technical report entitled “Fast, Cheap and Out of Control”³⁰ in part because having many robots meant that several robots could be destroyed in transit or during landing without a real impact on the overall mission.

Multi-agent teams are becoming quite popular in robot competitions, especially two international robot soccer competitions: RoboCup and MIROSOT. In these competitions, teams of real or simulated robots play soccer against other teams. The soccer task explicitly requires multiple robots that must cooperate with each other, yet react as individuals.

Readers with a strong background in artificial intelligence may notice similarities between teams of mobile robots and teams of software agents (“we-bots” which search the web and “knowbots” which do data mining). Those similarities are not accidental; software and physical agents fall into a research area in Artificial Intelligence often referred to as *Distributed Artificial Intelligence (DAI)*. Most of the issues in organizing teams of robots apply to software agents as well. Arkin,¹⁰ Bond and Gasser,¹¹⁹ Brooks,²⁶ and Oliveira et al.¹¹³ all cite the problems with teams of multiple agents, condensed here as:

DISTRIBUTED
ARTIFICIAL
INTELLIGENCE (DAI)

Designing teams is hard. How does a designer recognize the characteristics of a problem that make it suitable for multi-agents? How does the designer (or the agents themselves) divide up the task? Are there any tools to predict and verify the social behavior?

INTERFERENCE

There is a “too many cooks spoil the broth” effect. Having more robots working on a task or in a team increases the possibility that individual robots with unintentionally *interfere* with each other, lowering the overall productivity.

It is hard for a team to recognize when it, or members, are unproductive. One solution to the “too many cooks spoil the broth” problem is to try engineering the team so that interference cannot happen. But this may not be possible for every type of team or the vagaries of the open world may undermine that engineering. To defend itself, the team should be capable of monitoring itself to make sure it is productive. This in turn returns to the issue of communication.

It is not clear when communication is needed between agents, and what to say. Many animals operate in flocks, maintaining formation without explicit communication (e.g., songs in birds, signals like a deer raising its tail to display white, speaking). Formation control is often done simply by perceiving the proximity to or actions of other agents; for example, schooling fish try to remain equally close to fish on either side. But robots and modern telecommunications technology make it possible for all agents in a team to literally know whatever is in the mind of the other robots, though at a computational and hardware cost. How can this unparalleled ability be exploited? What happens if the telecommunications link goes bad? Cell phones aren't 100% reliable, even though there is tremendous consumer pressure on cell phones, so it is safe to assume that robot communications will be less reliable. Is there a language for multi-agents that can abstract the important information and minimize explicit communication?

The "right" level of individuality and autonomy is usually not obvious in a problem domain. Agents with a high degree of individual autonomy may create more interference with the group goals, even to the point of seeming "autistic."¹¹³ But agents with more autonomy may be better able to deal with the open world.

The first question in the above list essentially asks *what are the architectures for multi-agents?* The answer to that question at this time is unclear. Individual members of multi-agent teams are usually programmed with behaviors, following either the Reactive (Ch. 4) or Hybrid Deliberative/Reactive (Ch. 7) paradigms. Recall that under the Reactive Paradigm, the multiple behaviors acting concurrently in a robot led to an *emergent behavior*. For example, a robot might respond to a set of obstacles in a way not explicitly programmed in. Likewise in multi-agents, the concurrent but independent actions of each robot leads to an *emergent social behavior*. The group behavior can be different

EMERGENT SOCIAL

BEHAVIOR from the individual behavior, emulating "group dynamics" or possibly "mob psychology." As will be seen in this chapter, fairly complex team actions such as flocking or forming a line to go through a door emerge naturally from reactive robots with little or no communication between each other. But as with emergent behavior in individual robots, emergent social behavior is often hard to predict. Complete architectures for designing teams of robots are still under development; Lynne Parker's ALLIANCE architecture¹¹⁴ is possibly the most comprehensive system

to date. The whole field of multi-agents is so new that there is no consensus on what are the important dimensions,

or characteristics, in describing a team. For the purposes of this chapter, *heterogeneity*, *control*, *cooperation*, and *goals* will be used as the dimensions.¹¹⁷

8.2 Heterogeneity

HETEROGENEITY *Heterogeneity* refers to the degree of similarity between individual robots that are within a collection. Collections of robots are characterized as being either heterogeneous or homogeneous. *Heterogeneous teams* have at least two

HOMOGENEOUS TEAMS *homogeneous teams* members with different hardware or software capabilities, while in *homogeneous teams* the members are all identical. To make matter more confusing, members can be homogeneous for one portion of a task by running identical behaviors, then become heterogeneous if the team members change the behavioral mix or tasks.

8.2.1 Homogeneous teams and swarms

Most multi-agent teams are homogeneous swarms. Each robot is identical, which simplifies both the manufacturing cost and the programming. The biological model for these teams are often ants or other insects which have large numbers of identical members. As such, swarms favor a purely reactive approach, where each robot operates under the Reactive Paradigm. Insect swarms have been modeled and mimicked since the 1980's. The proceedings of the annual conference on the Simulation of Adaptive Behavior (also called "From Animals to Animats") is an excellent starting point.

An example of a successful team of homogeneous robots is Ganymede, Io, and Callisto fielded by Georgia Tech. These three robots won first place in the "Pick Up the Trash" event of the 1994 AAAI Mobile Robot Competition,¹²⁹ also discussed in Ch. 5. Recall that the objective of that event was to pick up the most trash (coca-cola cans) and deposit it in a refuse area. The majority of the entries used a single agent, concentrating on model-based vision for recognizing trash, cans, and bins and on complex grippers.

The three identical robots entered by Georgia Tech were simple, both physically and computationally, and are described in detail in a 1995 *AI Magazine* article.¹⁹ The robots are shown in Fig. 8.1, and were constructed from an Intel 386 PC motherboard mounted on a radio-controlled toy tracked vehicle. The robots had a miniature wide-angle video camera and

8.2 Heterogeneity framegrabber. The flapper-style grippers had an IR to indicate when something was in the gripper. The robots also had a bump sensor in front for collisions. The robots were painted fluorescent green.

Each robot was programmed with a sequence of simple reactive behaviors (renamed here for clarity), following the reactive layer of Arkin's AuRA architecture described in Ch. 4:

wander-for-goal This behavior was instantiated for two goals: trash and trashcan. The motor schema was a random potential field, the perceptual schema was color blob detection, where trash="red" and trashcan="blue."

move-to-goal This behavior also had two different goals: trash and trashcan. The motor schema was an attractive potential field, and the perceptual schema for the trash and trashcan were the same as in the wander-for- goal.

avoid-obstacle This behavior used the bump switch as the perceptual schema, and a repulsive field as the motor schema.

avoid-other-robots The three robots did not communicate with each other, instead using only the repulsive field created by avoid-other-robots to reduce interference. The motor schema was a repulsive potential field (linear dropoff), while the perceptual schema detected "green."

grab-trash The robot would move toward the trash until the perceptual schema reported that the IR beam on the gripper was broken; the motor schema would close the gripper and back up the robot.

drop-trash When the robot reached the trashcan with trash in its gripper, the motor schema would open the gripper and back up the robot, and turn 90 degrees.

8.2.2 Heterogeneous teams

A new trend in multi-agents is *heterogeneous teams*. A common heterogeneous team arrangement is to have one team member with more expensive computer processing. That robot serves as the team leader and can direct the other, less intelligent robots, or it can be used for special situations. The danger is that the specialist robot will fail or be destroyed, preventing the team mission from being accomplished.

One interesting combination of vehicle types is autonomous air and ground vehicles. Researchers at the University of Southern California

under the direction of George Bekey have been working on the coordination of teams of ground robots searching an area based on feedback from an autonomous

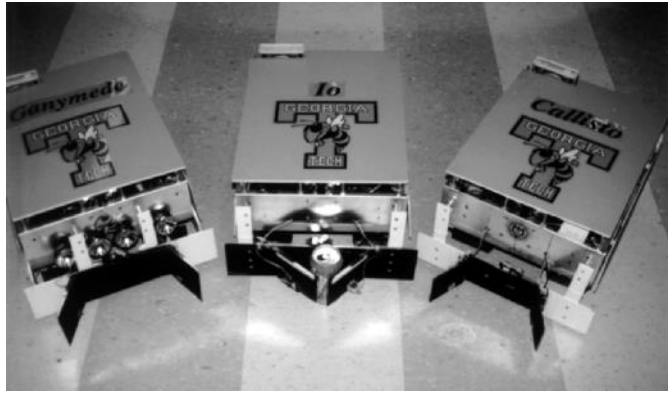


Figure 8.1 Georgia Tech's winning robot team for the 1994 AAI Mobile Robot Competition, Pick Up the Trash event. (Photograph courtesy of Tucker Balch and AAI.)

miniature helicopter. This combination permits the team to send a human observer a comprehensive view of a particular site, such as a hostage situation.

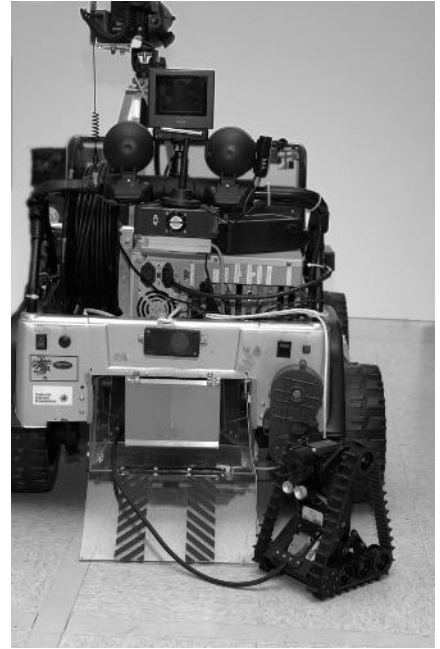
A special case of a cooperative, heterogeneous team of robots has been dubbed *marsupial robots*. The motivation for marsupial robots stemmed from concerns about deploying micro-rovers for applications such as UrbanSearch and Rescue. Micro-rovers often have limited battery power, which they can't afford to spend just traveling to a site. Likewise, micro-rovers may not be able carry much on-board processing power and need to have another, more computationally powerful workstation do proxy (remote) processing. A marsupial team consists of a large robot which carries one or more smaller robots to the task site, much like a kangaroo mother carries a joey in her pouch. Like a joey, the daughter robot is better protected in the pouch and can conserve energy or be recharged during transport. The mother can protect a delicate mechanism or sensor from collisions while it navigates through an irregular void. The mother can also carry a payload of batteries to recharge (feed) the daughter. It can serve as a proxy workstation, moving to maintain communications. The mother is likely to be a larger robot, while the daughter might be a micro-rover with sensors very close to the ground. The mother will have a better viewpoint and sensors, so in some circumstances it can communicate advice to the smaller daughter to help it cope with a

8.2 *Heterogeneity*
“mouse’s eye” view of the world. A teleoperator can also control

301



a.



b.

Figure 8.2 Two views of a marsupial robot team at University of South Florida. a.) Silver Bullet is the “mother” connected by an umbilical tether to a tracked chemical inspection robot Bujold, the “daughter.” b.) Bujold exits from the rear of the jeep. (Photographs by Tom Wagner.)

the daughter more easily in some situations by looking through the mother’s camera.

At this time, there appear to be only two physically realized implementations of autonomous marsupials: the University of South Florida teams, one of which is shown in Fig. 8.2, and the robots at the US Department of Energy’s Idaho National Energy and Engineering Laboratory (INEEL).³ The USF team is the only one where a mother robot carries a micro-rover inside the structure to protect it. The Mars Pathfinder mission is similar to a marsupial robot in that a micro-rover was transported to a mission site and the transport vehicle served as a support mechanism. However, our definition of marsupial assumes the mother is a fully mobile agent and can recover and retask the micro-rover.

8.2.3 Social entropy

SOCIAL ENTROPY

The above examples show how different heterogeneous teams can be. One rough measure of the degree of heterogeneity is the *social entropy* metric created by Tucker Balch.¹⁶ (Entropy is a measure of disorder in a system, especially in the sense of the Third Law of Thermodynamics. It was also adapted by Shannon for use in information theory to quantify the amount or quality of information in a system.) The point of social entropy is to assign a numerical value for rating diversity (or disorder) in a team. The number should be 0 if all team members are the same (homogeneous). The number should have the maximum value if all the team members are different. The number of team members which are different should make the overall number higher.

To compute social entropy, consider a marsupial team R with a mother robot and three identical (hardware and software) micro-rovers. The formula for the social entropy, $Het(R)$, is:

$$(8.1) \quad Het(R) = \sum_{i=1}^c p_i \log_2(p_i)$$

CASTES

There are two types of robots in the team, called *castes* or c : the mother and the daughters. Therefore $c = 2$. The term p_i is the decimal percent of robots belonging to caste c_i . If $i = 1$ for the mother, and $i = 2$ for the daughters:

$$(8.2) \quad \begin{aligned} p_1 &= \frac{1}{4} = 0.25 \\ p_2 &= \frac{3}{4} = 0.75 \end{aligned}$$

Substituting into Eqn. 8.1 (and remembering that $n = \frac{\log_{10} n}{\log_{10} 2}$), the social entropy is:

$$(8.3) \quad Het(R) = \sum_{i=1}^c p_i \log_2(p_i)$$

$$\begin{aligned}
 & \quad i=1 \quad 8.2 \text{ Heterogeneity} \\
 & = (0.25 \log_2 \\
 & \quad 0.25 + 0.75 \\
 & \quad \log_2 0.75) \\
 & = ((0.50) + (\\
 & \quad 0.31)) \\
 & = 0.81
 \end{aligned}$$

Now consider a case where the daughters are not identical. Suppose that one of the three micro-rovers has a different sensor suite and behaviors from the other two. In that case $c = 3$, where $p_1 = \frac{1}{3}$, $p_2 = \frac{2}{3}$, and $p_3 = \frac{1}{3}$. Substituting into Eqn. 8.1 yields 1.5. Since $1.5 > 0.81$, the marsupial team with the different daughter is more diverse than the marsupial team with all identical daughters.

8.3 Control

CENTRALIZED Control of multi-agents can fall in a spectrum bounded by *centralized control*

and *distributed control* regimes. In centralized control, the robots communicate with a central computer. The central computer distributes assignments, goals, etc., to the remote robots. The robots are essentially semi-autonomous, with the centralized computer playing the role of a teleoperator in a teleoperated system. In distributed control, each robot makes its own decisions and acts independently. Of course, there is a range of regimes between fully centralized and fully distributed; the robots can interact with a central controller to receive new goals, then operate for the duration of the mission in a distributed manner.

Examples of full and partial centralized control can be found by comparing the RoboCup and MIROSOT robot soccer competitions. In those soccer competition events, teams of robots are controlled remotely by a central computer. In the small sized league of RoboCup and MIROSOT, teams of three, very small self-contained robots (7.5cm x 7.5cm x 7.5cm) play on a 130cm x 90cm arena with an orange golf ball serving as the miniature soccer ball. Each robot had a unique pattern of bright colors to make it visible from the overhead cameras, and the overhead camera is connected to a central processor. The robots communicate with the central processor over a radio link. In MIROSOT, the central processor commands each robot by supplying the direction to move. In RoboCup, the central processor can give either explicit directions or just locations of other robots and the ball, letting the robot's on-board behaviors generate the (one hopes) correct response. Fig. 8.3 shows a view of the small-sized league from the 1998 RoboCup World Cup.

MIROSOT robots are more drone-like than their RoboCup

8.2 Heterogeneity
counterparts, since they are not required to carry any on-board sensing. They represent the extreme of centralized control, where everything must go through a single computer, much like the battle-droids in the *Star Wars* movie, *The Phantom Menace*. RoboCup robots are required to have some type of on-board

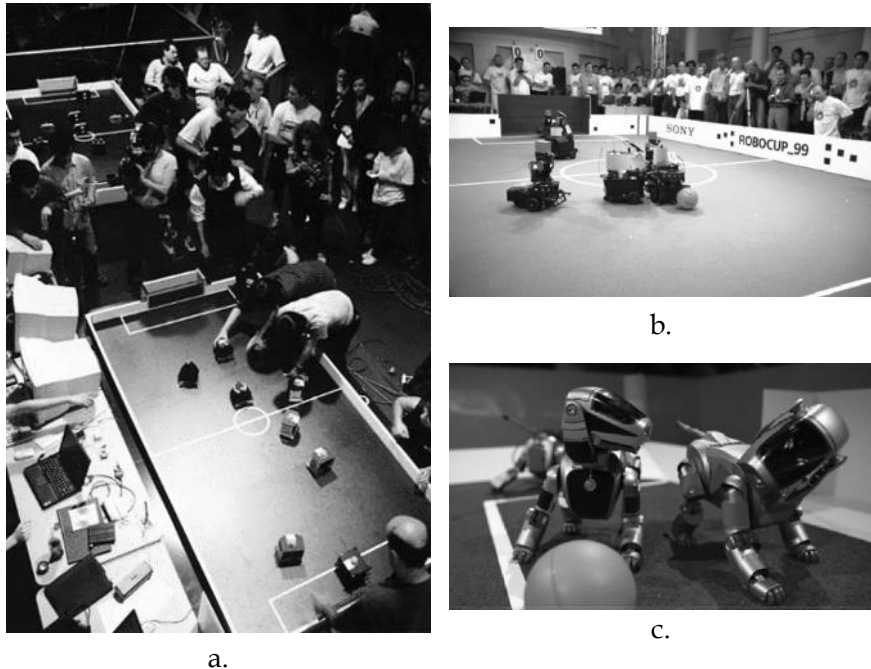


Figure 8.3 RoboCup soccer competition: a.) Overhead view of the RoboCup playing field and teams for the small-sized league (overhead camera is not shown), b.) Mid-sized league, and c.) Legged league using Sony Aibo robots. (Photographs c The RoboCup Federation 1999. All rights reserved.)

sensing for reflexive obstacle avoidance. In the RoboCup case, the robots must have a set of basic tactical behaviors (see Ch. 7), but may either receive strategic commands from the central computer or have on-board strategic behaviors. This type of control is conceptually equivalent to the Hybrid Reactive-Deliberative Paradigm, where the reactive layer physically resides on the robot and the deliberative layer resides on the central workstation.

Distributed control is more natural for soccer playing than centralized control, because each player reacts independently to the situation. An example of distributed control in robot soccer playing is the mid-sized league in RoboCup. Notice that in robot soccer the robots are inherently heterogeneous. Although they may be physically the same, each robot is programmed with a different role, most especially Goalie, Striker, and

Likewise, there are significant advantages to planning and learning strategies, two deliberative functions. Manuela Veloso and Peter Stone have used RoboCup as a test domain for research in deliberation.

8.4 Cooperation

COOPERATION	<i>Cooperation</i> refers to how the robots interact with each other in pursuing a goal. Robots can show <i>active cooperation</i> by acknowledging one another and working together. Note that this does not necessarily mean the robots communicate with each other. For example, in robot soccer, one robot can pass the ball to another robot as part of an offensive play. The cooperation does not require communication—if a robot has the ball, can't see goal and can see team mate, then it passes to team mate, but this does require being aware of the teammates.
ACTIVE COOPERATION	
NON-ACTIVE	More often robots are programmed to exhibit <i>non-active cooperation</i> , where-
COOPERATION	by they individually pursue a goal without acknowledging other robots but cooperation emerges. The choice of cooperation schemes is often influenced by the sensory capabilities of the robots. Active cooperation requires that robot be able to distinguish its peer robots from other aspects of the environment. In the case of the Georgia Tech entry, each robot was covered in fluorescent green poster paper easily segmented as a color region. If the robots had not been green, they would have been treated as obstacles to be avoided. Non-active cooperation has attracted much interest in the robotics community because it requires very little sensing or behaviors.
PHYSICAL	It is easy to think of cooperation in terms of robots working together on a task. Another aspect of cooperation is <i>physical cooperation</i> , where the robots
COOPERATION	physically aid each other or interact in similar ways. Marsupial robots are certainly a type of physical cooperation, especially during deployment and
RECONFIGURABLE	docking. An even more exciting type of cooperation occurs between <i>reconfig-</i>
ROBOTS	<i>urable robots</i> . One of the first such systems was proposed by Toshio Fukuda, called CEBOT for "cellular robot system." ³¹ These are small identical

8.2 Heterogeneity
robots that hook up to form a useful robot. Another aspect³⁰⁵ of reconfigurable robots

COOPERATIVE is *cooperative mobility*, where one robot might come over and help another

MOBILITY robot in trouble. Shigeo Hirose simulated robots which could link up with each other to gain more stability or traction in rough terrain.⁶⁷

8.5 Goals

The final dimension for characterizing a collection of multi-agents is how the robot works on a goal. If all the robots in the collection work on attaining the same explicit goal, then they are said to share a single goal, versus having individual goals.

An example of robots working a single goal is the winning team for the Office Navigation event in the 1996 AAAI Mobile Robot Competition.⁷⁸ The office navigation event had a robot that was supposed to search a series of rooms, find an empty conference room, and then go to a list of rooms where people were and tell them that a meeting was going to begin in the empty conference room. The event was originally conceptualized as a single agent task, but the SRI entry under the direction of Kurt Konolige consisted of three robots.⁶² Each of the three robots ran the Saphira architecture (see Ch. 7) and were coordinated by a central workstation. While the robots were responsible for autonomous navigation, their goals were set by the central strategy agent. Even though they were navigating through different parts of the office maze, the robots were working on a single goal and the software agents on the central workstation were explicitly coordinating the actions of the robots. The robots were able to find an empty room and inform the attendees in 4 minutes and 30 seconds. The next best time was close to 10 minutes.

An example of purely reactive robots working on individual goals is a problem originally posed by Ron Arkin:⁹ a group of robotic space “ants” foraging for mineral rich asteroids or Near Earth Objects (NEOs). If each robot in the group forages for its own asteroid, then they have individual goals. (Notice that a behavior that permits them to notice other robots and be repulsed will help disperse the robots.) If the robots are programmed so that they will all go to one specific asteroid, then they share a common goal. Emergent cooperation is not the same thing as having a single goal. For example, suppose the robotic space ants are programmed to go to the nearest *non-moving* asteroid and bring it back to base. Each robot might have a set of behaviors: find-stationary-asteroid, move-to-asteroid, push-asteroid-to-home, and avoid-robots. The find-stationary-asteroid could be done with a random potential field (in 3 dimensions, of course). An attractive “asteroid-tropic” potential field could be used for the move-to-asteroid behavior. Likewise an attractive field could be used for the push-asteroid-to-home behavior, where the robot tries to stay behind the

asteroid as it moves to home rather than avoid the asteroid. Avoid-robot could be done with a repulsive field. These behaviors give the robots individual goals, since there is

no awareness of the goals of the other team members.

Now consider what happens when a robot ant encounters an asteroid it can't move. The robot stays there pushing. Eventually another robot will come along because the asteroid is not moving. As it is attracted to the "dark side" of the asteroid, it will come into range of the first robot. What happens? The avoid-robot behavior should be instantiated, causing the first robot to move over a bit. The second robot will also feel a repulsive force and slow down. As the first robot moves out of the way, the angle of repulsion changes, forcing the second robot to move sideways as well, as it continues to move to the asteroid. Together, the interaction between the two robots should cause them to naturally balance themselves behind the asteroid and push together. The point is that the robots were not explicitly directed to all work on the same NEO; they were each directed to find their own NEO, but circumstances led them to the same one.

8.6 Emergent Social Behavior

The examples of heterogeneity, cooperation, control, and goals give some hint of how an overall social behavior emerges from the actions of autonomous robots. The robot teams often are the result of extensive design efforts, where the teams aren't too large to interfere with each other, and are optimally sized for the particular task, etc. Many researchers are exploring the issues of what happens when the designer doesn't have a choice about the size of the robot population. How do social behaviors emerge in those cases? And how can social rules or conventions be established to make the team self-regulating and productive? This section summarizes two approaches: creating social rules for the robots to follow, and allowing internal motivation to cause the robots to adapt their behavior to problems.

8.6.1 Societal rules

Maja Mataric has focused her research on how group dynamics might emerge in herds of multiple agents operating under fully distributed control. She explored the impact of density and the impact of societal rules on overall team performance.⁹⁰ Each IS Robotics R2 robot was programmed with behaviors using the Subsumption architecture. She set up a scenario where up to 20 identical robots (now known as "The Nerd

Herd") were given the same location as a goal. The goal, however, was on the other side of a partition with a narrow door, permitting only one robot to pass through the partition at a



Figure 8.4 The Nerd Herd. (Photograph courtesy of USC Interaction Laboratory.)

time. The robots were placed randomly on the same side of the partition and started moving at the same time.

IGNORANT

In the first set of demonstrations, the robots functioned with *ignorant coex-*

COEXISTENCE *istence*. The robots coexisted in a team, but did not have any knowledge of each other. A robot treated another robot as an obstacle. Each robot had the equivalent of a move-to-goal and an avoid-obstacle behavior. Since robots were treated as obstacles, once the robots gathered at the opening, they spent most of their time avoiding each other. The team as a whole made slow progress through the door to the goal location. Worse yet, the larger the number of robots fielded, the larger the traffic jam, and the longer to get all the team members through.

INFORMED

In the second demonstration, *informed coexistence*, the robots were allowed

COEXISTENCE to recognize each other and given a simple social rule governing inter-robot interactions. In addition to move-to-goal and avoid-obstacle, a third behavior was created for avoiding robots. If a robot detected another robot, it would stop and wait for time p . If the blocking robot was still in the way after p , the robot would turn left and then resume moving to the goal. The result of the new behavior was to reduce the traffic jams, and the group got through the door in about the same time as a single agent going back and forth through the opening 20 times.



Figure 8.5 Robots cooperatively tracking an object under the ALLIANCE architecture. (Photograph courtesy of Oak Ridge National Laboratories.)

The social behavior for avoiding robots was replaced with another heuristic: the robots were repulsed from other robots, but as it moves away, it tries to move in the same direction as a majority of other robots. (Each robot broadcast its heading over a radio transmitter to compensate for the inability to recognize each other by vision or sonar, so that isn't considered communication.) As a result, the robots exhibited a flocking behavior and went through the door in single file! The need to go in the same direction created a tendency to form a line, while repulsion caused the robots to essentially create spaces for robots to merge into line. Together the two effects created a strong need to go through the door single file, even though there was no such explicit direction. Not only were traffic jams reduced, but the overall task was accomplished faster.

8.6.2 Motivation

In Mataric's work, the robots reduced interference through simple social rules with no communication, but the members of the team could not actively help out failed colleagues or change tasks dynamically. Lynne Parker has attempted to address the larger issues of robustness and fault tolerance with the ALLIANCE architecture,¹¹⁴ an outgrowth of the Subsumption architecture. The central idea is that members in the team

“hear” the progress of others in teams, as well as their own progress. If they get frustrated with their own progress, they should stop what they’re doing and move on to something else. Likewise, if a robot is free and another robot has been unable to accomplish a task, it should try to complete the unfinished task. This is particularly useful for tasks where there is a logical sequence of behaviors, where all of a particular task (like dusting) needs to be done for an area before the robots begin working on another task (e.g., sweeping). These changes in behaviors are regulated by a simple mechanism: *motivation*. The motivation of a robot to do a task is regulated by two internal motivations, *robot impatience* and *robot acquiescence*. The more frustrated a robot with gets with another robot’s performance on t_i , the higher the impatience associated with that task t_i . Likewise, the more frustrated a robot gets with its own performance for a task, the higher the acquiescence. If the frustration threshold is exceeded, then the robot either takes over the unfinished task or abandons its current task and changes behavior.

Fig. 8.6 shows the time trace for an example of motivation for two space ants foraging for asteroids. (This example isn’t really a sequential series of tasks in the manner used by ALLIANCE, but this conveys the elegance of motivation.) In this case, the reactive space ants have to either broadcast what they’re doing or be able to perceive the other’s progress. This makes it a bit different than the “no communication” approach. At time 0, both robots start by looking for asteroids. (We assume there is no frustration for the find task.) Both see asteroid A1, but Robot 1 is the first there. Robot 1 has now taken responsibility for Task 1 (T1), pushing A1 to home. Even though A1 is still stationary at time 3, Robot 2 does not join in as it would in the no-communication method. Instead, it begins to accrue impatience about T1. Once Robot 1 begins to push A1, it starts accruing frustration in the form of acquiescence. As with the no-communication example, a single robot cannot push the asteroid.

While Robot 1 is trying to push asteroid A1, Robot 2 sees and moves to asteroid A2. All the while its impatience over T1 is growing. At time 7, Robot 2 is trying unsuccessfully to push asteroid A2 (task T2) and its acquiescence counter is increasing. Also at time 7, Robot 2’s patience with Robot 1 and task T1 has been exceeded. It pushes T1 onto its stack of things to do when it completes its current task. Meanwhile, at time 9, Robot 1 gives up on T1. Although it is frustrated with Robot 2, it assumes that T2 is still under control and so begins to forage again. Finally, at time 10, the frustration over T2 reaches the limit and Robot 1 is free to help

time	Robot 1	Robot 2
0	find-stationary-asteroid	find-stationary-asteroid
1	sees A1	sees A1
2	move-to-asteroid(A1)	move-to-asteroid(A1)
3	arrives at A1	resumes find-stationary-asteroid
4	push-asteroid-to-home(A1) T1-acquiescence++	find-stationary-asteroid T1-impatience++
5	push-asteroid-to-home(A1) T1-acquiescence++	sees A2 T1-impatience++
6	push-asteroid-to-home(A1) T1-acquiescence++	move-to-asteroid(A2) T1-impatience++
7	push-asteroid-to-home(A1) T1-acquiescence++ T2-impatience++	push-asteroid-to-home(A2) T1-impatience>limit put T1 on stack T2-acquiescence++
8	push-asteroid-to-home(A1) T1-acquiescence++ T2-impatience++	push-asteroid-to-home(A2) A1-impatience++ T2-acquiescence++
9	T1-acquiescence>limit gives up on T1 find-stationary-asteroid T2-impatience++	push-asteroid-to-home(A2) T2-acquiescence++
10	T2-impatience>limit now attempts T2 move-to-asteroid(A2)	T2-acquiescence++
11	push-asteroid-to-home(A2) T2-acquiescence = 0	push-asteroid-to-home(A2) T2-acquiescence = 0
12	arrives at HOME	arrives at HOME

Figure 8.6 Example of how the internal motivation in ALLIANCE might be extended to work with two space ants.

8.7 Summary

In summary, many tasks favor the use of many cheap robots rather than a single expensive one. These collections of multiple robots are often referred to as multi-agents. Individual robots in a multi-agent team are generally programmed with behaviors, most often as purely reactive systems, but occasionally with a hybrid architecture. As with an overall behavior emerging

on a single reactive agent, societies of reactive agents often exhibit an emergent societal behavior.

Multi-agent societies can be characterized according to where they fall on at least four dimensions (since multi-agent theory is relatively new, even for robotics, new dimensions may surface over time).

Heterogeneity refers to whether the member robots are identical in software and hardware. Cooperation may be either active or non-active, while control may fall in the spectrum from fully distributed to fully centralized. A robot society may have a single, explicitly shared goal or each robot may have its own goal. When communication between agents is appropriate is a pervasive, open question. From a practical side, the emphasis in multi-agents has been on how favorable group dynamics emerge from teams of homogeneous, purely reactive robots while operating under distributed control. Problems in emergent societal behaviors such as interference and the need to adapt to the open world can often be addressed by specifying social rules and internal motivation. However, more interest is emerging in robots that have either heterogeneous software or hardware capabilities, such as marsupial and reconfigurable robots. The diversity of a heterogeneous team can be captured somewhat by the social entropy metric.

DELIBERATIVE

to its environment. This meant that a robot could not plan optimal trajectories (path planning), make maps, monitor its own performance, or even select the best behaviors to use to accomplish a task (general planning). Notice that not all of these functions involve planning per se; map making involves handling uncertainty, while performance monitoring (and the implied objective of what to do about degraded performance) involves problem solving and learning. In order to differentiate these more cognitively oriented functions from path planning, the term *deliberative* was coined.

The Reactive Paradigm also suffered somewhat because most people found that designing behaviors so that the desired overall behavior would emerge was an art, not a science. Techniques for sequencing or assembling behaviors to produce a system capable of achieving a series of sub-goals also relied heavily on the designer. Couldn't the robot be made to be smart enough to select the necessary behaviors for a particular task and generate how they should be sequenced over time?

Therefore, the new challenge for AI robotics at the beginning of the 1990's was how to put the planning, and deliberation, back into robots, but without disrupting the success of the reactive behavioral control. The consensus was that behavioral control was the "correct" way to do low level control, because of its pragmatic success, and its elegance as a computational theory for both biological and machine intelligence. As early as 1988, Ron Arkin was publishing work on how to add more cognitive functions to a behavioral system in the form of the *Autonomous Robot Architecture (AuRA)*. Many roboticists looked at adding layers of higher, more cognitive functions to their behavioral systems, emulating the evolution of intelligence. This chapter will cover five examples of architectures which illustrate this bottom-up, layering approach: *AuRA*, *Sensor Fusion Effects (SFX)*, *3T*, *Saphira*, and *TCA*. Other robot systems which do not strongly adhere to an architectural style, such as *Rhino* and *Minerva*, will be discussed in later chapters.

REACTIVE PLANNING

During the 1990's, members of the general AI community had become exposed to the principles of reactive robots. The concept of considering an intelligent system, or agent, as being situated in its environment, combined with the existence proof that detailed, Shakey-like world representations are not always necessary, led to a new style of planning. This change in planning was called *reactive planning*. Many researchers who had worked in traditional AI became involved in robotics. One type of reactive planner for robots, Jim Firby's *reactive-action packages (RAPs)*,

⁵³ was²⁵⁸ integrated
as a layer within
the 3T
architecture.²¹
Architectures
stemming from
the planning
community roots
showed their
traditional AI
roots. They use a
more top-down,
hierar-

chical flavor with global world models, especially Saphira⁷⁷ and TCA.¹³¹

Regardless of the bottom-up or top-down inspiration for including non-behavioral intelligence, architectures which use reactive behaviors, but also incorporate planning, are now referred to as being part of the Hybrid Deliberative/Reactive Paradigm. At first, Hybrids were viewed as an artifact of research, without any real merit for robotic implementations. Some researchers went so far as to recommend that if a robot was being designed to operate in an unstructured environment, the designer should use the Reactive Paradigm. If the task was to be performed in a knowledge-rich environment, easy to model, then the Hierarchical Paradigm was preferable, because the software could be engineered specifically for the mission. Hybrids were believed to be the worst of both worlds, saddling the fast execution times of reactivity with the difficulties in developing hierarchical models.

The current thinking in the robotics community is that Hybrids are the best general architectural solution for several reasons. First, the use of asynchronous processing techniques (multi-tasking, threads, etc.) allow deliberative functions to execute independently of reactive behaviors. A planner can be slowly computing the next goal for a robot to navigate to, while the robot is reactively navigating toward its current goal with fast update rates. Second, good software modularity allows subsystems or objects in Hybrid architectures to be mixed and matched for specific applications. Applications which favor purely reactive behaviors can implement just the subset of the architecture for behaviors, while more cognitively challenging domains can use the entire architecture.

7.2 Attributes of the Hybrid Paradigm

The organization of a Hybrid Deliberative/Reactive system can be described as: PLAN, then SENSE-ACT. It is shown in Fig. 7.1. The PLAN box includes all deliberation and global world modeling, not just task or path planning. The robot would first plan how to accomplish a mission (using a global world model) or a task, then instantiate or turn on a set of behaviors (SENSE-ACT) to execute the plan (or a portion of the plan). The behaviors would execute until the plan was completed, then the planner would generate a new set of behaviors, and so on.

The idea of PLAN, then SENSE-ACT evolved from two assumptions of the Hybrid Paradigm. First, planning covers a long time horizon and

re- quires global knowledge, so it should be decoupled from real-time execution

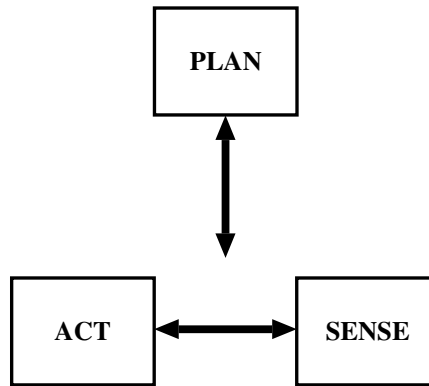


Figure 7.1 P,SA Organization of Hybrid Deliberative/Reactive Paradigm in terms of Primitives.

DELIBERATION AND

SYMBOLS a just on the software engineering principle of coherence (dissimilar functions should be placed in different objects). It is good for setting objectives and selecting methods, but not for making finely grained decisions. Another way of stating this is that *deliberation works with symbols*, e.g., the goal is to pick up a “Coca-Cola can,” while reaction works with sensors and actuators, e.g., the percept is a “red blob” which exerts an attractive field. Second, planning and global modeling algorithms are computationally expensive, so they should be decoupled from real-time execution just from a standpoint of practicality because they would slow down the reaction rate.

The organization of sensing in the Hybrid architecture is more complex. Sensing is truly hybrid, as seen in Fig. 7.2. In the behaviors, sensing remains as it was for the Reactive Paradigm: local and behavior specific. But planning and deliberation requires global world models. Therefore, planning functions have access to a global world model. The model is constructed by processes independent of the behavior-specific sensing. However, both the perceptual schemas for the behaviors and the model making processes can share the same sensors. Furthermore, the model making processes can share the percepts created by the perceptual schemas for behaviors (eavesdrop) or it can have sensors which are dedicated to providing observations which are useful for world modeling but aren’t used for any active behaviors.

REACTOR
DELIBERATOR

The organization of the SENSE, PLAN, ACT primitives in the Hybrid Paradigm is conceptually divided into a reactive (or *reactor*) portion and a deliberation (or *deliberator*) portion. Although many architectures will have discrete

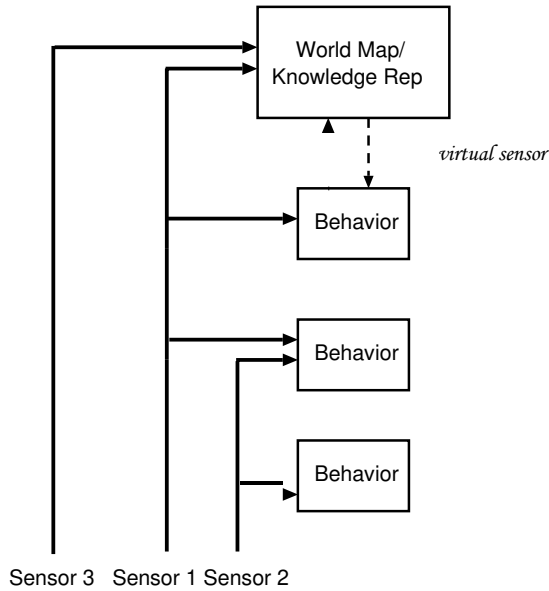


Figure 7.2 Sensing Organization in Hybrid Paradigm, showing that the global world model can have its own sensors, can “eavesdrop” on sensors or percepts used/created by behaviors, and can act as a virtual sensor for use by a behavior.

layers of functionalities within the reactor and deliberator, each architecture in the Hybrid Paradigm has obvious partitions between reactive and deliberative functions.

7.2.1 Characteristics and connotation of reactive behaviors in hybrids

The Hybrid Paradigm is an extension of the Reactive Paradigm, and from the above description, it would appear that the behavioral component is untouched. That is not entirely true. Behaviors in the Hybrid Paradigm have a slightly different connotation than in the Reactive Paradigm. In the Reactive Paradigm, “behavior” connotes purely reflexive behaviors. In the Hybrid Paradigm, the term “behavior” is usually more consistent with the ethological use and includes reflexive, innate, and learned behaviors. This can be

260
havior to avoid
confusion with
purely reflexive
behaviors. Also,
Hybrid
implementations
tend to use
assemblages of
behaviors
sequenced over
time,

7 *The Hybrid Deliberative/Reactive Paradigm*

rather than primitive behaviors. Because the Hybrid implementations are interested in more complex emergent behaviors, there is more diversity in methods for combining the output from concurrent behaviors.

7.2.2 Connotations of “global”

The term “global” is used almost synonymously with “deliberative” and “local” with “reactive.” This can lead to significant confusion, because “global” isn’t always truly global in Hybrids.

The deliberative portion of a Hybrid architecture contains modules and functions for things which are not easy to represent in reactive behaviors. Some of these functions clearly require a global world model; path planning and map making are probably the best examples. But other activities

BEHAVIORAL	require global knowledge of a different sort. <i>Behavioral management</i> (plan-
MANAGEMENT	ning which behaviors to use) requires knowing something about the current mission and the current (and projected) state of the environment. This is global knowledge in that it requires the module to know something outside of itself, as compared to a reactive behavior which can function without any knowledge of whether there are other behaviors actively executing. Like-
PERFORMANCE	wise, <i>performance monitoring</i> to see if the robot is actually making progress
MONITORING	to its goal, and problem solving is a global activity. Consider writing a program to diagnose whether the robot’s not moving forward is a problem with the terrain (it’s stuck in the mud) or a sensor (the shaft encoders don’t report wheel turns correctly). In order to perform the diagnostics, the program has to know what the behaviors were trying to accomplish, if there are any other sensors or knowledge sources to corroborate any hypotheses, etc. Therefore, a deliberative function may not need a global world model, but may need to know about the internal workings of the robot on a global scale, if only to know what other modules or deliberative capabilities the program should interact with.

7.3 Architectural Aspects

The differences between various Hybrid Deliberative/Reactive architectures fall into three areas based on how they answer the following questions: *How*

does the architecture distinguish between reaction and deliberation? How does it organize responsibilities in the deliberative portion? How does the overall behavior emerge? The difference between reaction and deliberation is a critical issue in building a successful, reusable object-oriented implementation. This deter-

mines what functionality goes in what modules, what modules have access to global knowledge (which leads to specifying public and friend classes in C++), and what that global knowledge (shared data structures) should be. Likewise, it is important to subdivide the deliberative portion into modules or objects. A good decomposition will ensure portability and reusability. While Hybrid architectures are most noteworthy for how they incorporate deliberation into mobile robotics, they also introduce some changes in the way reaction is organized. Many researchers found the two primary means of combining reactive behaviors—subsumption and potential field summation—to be limited. Since then at least three other mechanisms have been introduced: *voting* (in the DAMN architecture),¹²¹ *fuzzy logic* (Saphira),⁷⁷ and *filtering* (SFX).¹⁰⁷

The number of Hybrid architectures is rapidly increasing. This section attempts to introduce some conceptual organization on Hybrids in two ways. First, it offers a set of common components—essentially, things to look for in a Hybrid architecture. Second, it divides Hybrid into three broad categories: *managerial*, *state hierarchies*, and *model-oriented*.

7.3.1 Common components of hybrid architectures

While Hybrid architectures vary significantly in how they implement deliberative functionality, what they implement is fairly similar. Generally a Hybrid architecture has the following modules or objects:

SEQUENCER

A *Sequencer* agent which generates the set of behaviors to use in order to accomplish a subtask, and determines any sequences and activation conditions. The sequence is usually represented as a dependency network or a finite state machine, but the sequencer should either generate this structure or be able to dynamically adapt it. Recall from Ch. 5 that assemblages of reactive behavior are manually constructed.

RESOURCE MANAGER

A *Resource manager* which allocates resources to behaviors, including selecting from libraries of schemas. For example, a robot may have stereo vision, sonars, and IR sensors, all of which are capable of range detection. The behavioral manager would ascertain whether the IR sensors can detect at a sufficient range, the stereo vision can update fast enough to match the robot's desired velocity, and the sonars have enough power to produce reliable readings. In reactive architectures, the resources for a behavior were often hard-coded or hardwired,

260
despite the
ability of hu-

7 *The Hybrid Deliberative/Reactive Paradigm*

mans to use alternative sensors and effectors (e.g., opening a door with the other hand when the holding something in the preferred hand).

CARTOGRAPHER

A *Cartographer* which is responsible for creating, storing, and maintaining map or spatial information, plus methods for accessing the data. The cartographer often contains a global world model and knowledge representation, even if it is not a map.

MISSION PLANNER

A *Mission Planner* which interacts with the human, operationalizes the commands into robot terms, and constructs a mission plan. For example, the ideal robot assistant might be given a command: "Lassie, go get the sheriff." The mission planner would interpret that command to mean to first physically search for a person, identify him as a sheriff because of the uniform, attract his attention, and then lead him back to the current location. The mission planner might have access to information that sheriffs are most likely to be in their office, donut shops, or the last place they were seen. The plan might first be to navigate to the sheriff's office.

PERFORMANCE MONITORING AND PROBLEM SOLVING

A *Performance Monitoring and Problem Solving* agent which allows the robot to notice if it is making progress or not. Notice that this requires the robot to exhibit some type of self-awareness.

7.3.2 Styles of hybrid architectures

MANAGERIAL STYLES

Architectural styles can be loosely divided into three categories. *Managerial styles* focus on subdividing the deliberative portion into layers based on the scope of control, or managerial responsibility, of each deliberative function. A Mission Planning module would be able to direct other, subordinate deliberative modules such as navigation, because Mission Planning (where to go) is more abstract than Path Planning (how to get there). *State hierarchies* use the knowledge of the robot's state to distinguish between reactive and deliberative activities. Reactive behaviors are viewed as having no state, no self-awareness, and function only in the Present. Deliberative functions can be divided into those that require knowledge about the robot's Past state (where it is in a sequence of commands) and about the Future (mission and path planning). *Model-oriented styles* are more nebulous. They are character-

STATE HIERARCHIES

MODEL-ORIENTED

STYLES 174 *Managerial Architectures* 265
ized by behaviors that have access to portions of a world model, often to the
point that they appear to have returned to the Hierarchical monolithic
global world model.

7.4 Managerial Architectures

Managerial styles of Hybrid architectures are recognizable by their decomposition of responsibilities similar to business management. At the top are agents which do high level planning, then pass off the plan to subordinates, who refine the plan and gather resources, and pass those down to the lowest level workers, the reactive behaviors. Higher level agents can see the results of their subordinate lower level agents (essentially eavesdrop on them), and can give them directions. As with subsumption, a layer can only modify the layer below it. In Managerial styles, each layer attempts to carry out its directive, identify problems and correct them locally. Only when an agent cannot solve its own problem does it ask for help from a superior agent; the agent is said to *fail upwards* in this case.

FAIL UPWARDS

7.4.1 Autonomous Robot Architecture (AuRA)

AUTONOMOUS ROBOT
ARCHITECTURE
(AuRA)

Autonomous Robot Architecture (AuRA) is the oldest of the Hybrids. It was actually designed and implemented by Arkin at the same time Brooks was beginning to publish his work with subsumption. AuRA is based on schema theory, and consists of five subsystems, equivalent to object-oriented classes. Two of the subsystems comprise the deliberative portion: the Planner, and the Cartographer. The Planner is responsible for mission and task planning. It is subdivided into three components, equivalent to the Nested Hierarchical Controller⁹³ discussed in Ch. 2. The Cartographer encapsulates all the map making and reading functions needed for navigation. The Cartographer can also be given an *a priori* map. For example, a human operator might load in a map file for the Cartographer to use. The three components of the Planner would interact with the Cartographer through methods to obtain a path to follow, broken down into subsegments.

The Planner subsystem is divided into the Mission Planner, Navigator, and Pilot. The Mission Planner serves as the interface with the human, and the current implementation of AuRA has one of the most extensive and user friendly robot interfaces available. The Navigator works with the Cartographer to compute a path for the robot and break it into subtasks (go over the mountain to the water tower, follow the road along the ridge to the camp). The Pilot takes the first subtask (go over the

mountain to the water tower) and gets relevant information (terrain, foliage types, what the water tower looks like) to generate behaviors. The Pilot portion of the Planning subsystem interacts with the Motor Schema Manager in the Motor subsystem, giv-

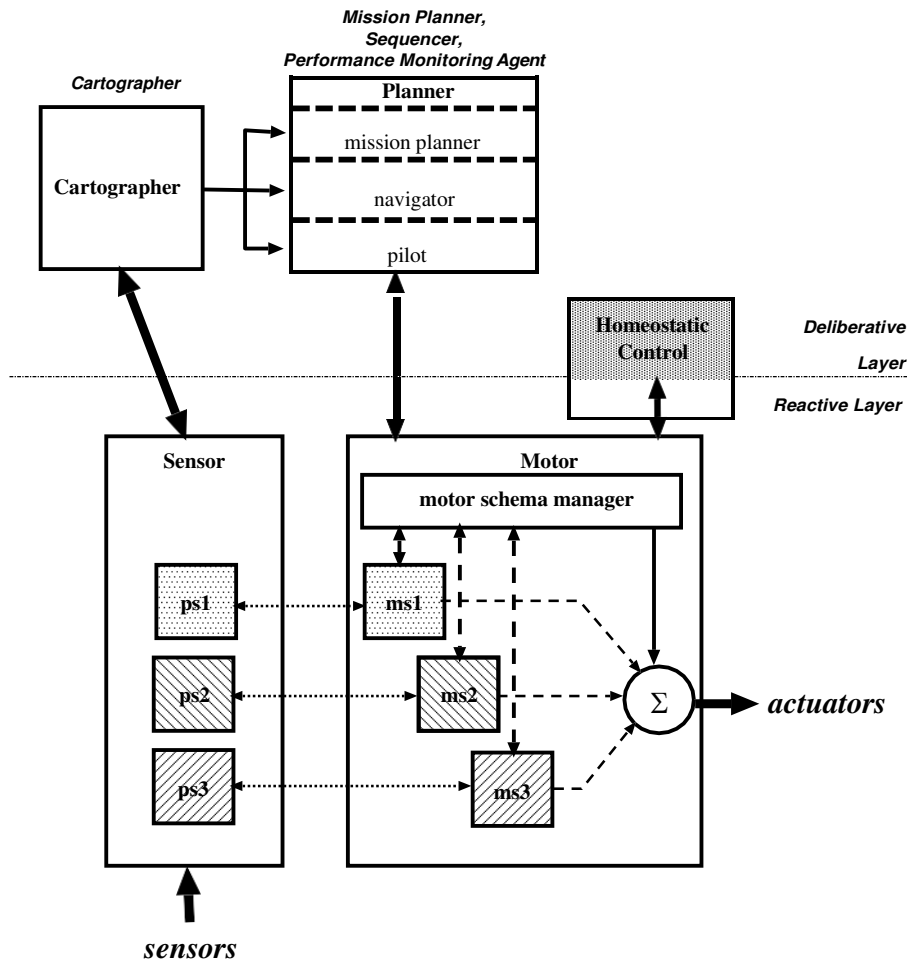


Figure 7.3 Layout of AuRA, showing the five subsystems.

ing it the list of behaviors it needs to accomplish the current subsegment. The Motor Schema Manager composes each behavior by examining the libraries of perceptual schemas in the Sensing subsystem and of motor schemas in the Motor subsystem. The motor schemas represent actions with potential fields, and the overall behavior emerges due to vector summation.

The Sensor and Motor subsystems make up the reactive portion of the architecture. These classes contain libraries of perceptual and motor

which form the behavior schema. The schemas themselves can consist of assemblages of primitive schemas, coordinated by finite state machines. Schemas can share information, if necessary, through links established by the Motor Schema Manager. Behaviors are not restricted to being purely reflexive; behavior specific knowledge, representations, and memory is permitted within the schemas. The motor schemas, however, are restricted to potential fields.

The fifth subsystem, Homeostatic Control, falls into a gray area between deliberation and reaction. The purpose of Homeostatic control is to modify the relationship between behaviors by changing the gains as a function of the “health” of the robot or other constraints. As an example, consider a planetary rover operating on a rocky planet. The robot is tasked to physically remove rock samples from various locations around the planet and deliver them to a return vehicle. The return vehicle has a fixed launch date; it will blast off, returning to Earth on a set day and time no matter what. Now, the rover may be provided with default gains on its behaviors which produce a conservative behavior. It may stay two meters away from each obstacle, giving itself a wide margin of error. At the beginning of the mission, such a conservative overall behavior appears reasonable. Now consider what happens towards the time when the return vehicle is set to launch. If the robot is near the return vehicle, it should be willing to shave corners and reduce the margin by which it avoids obstacles in order to ensure delivery. The robot should be willing to perform the equivalent of sacrificing its own existence for the sake of the mission.

The issue becomes how to do homeostatic. Many aspects of AuRA are motivated by biology, and homeostatic control is no exception. Rather than put a module in the deliberative portion to explicitly reason about how to change the overall behavior of the robot, biology suggests that animals subconsciously modify their behaviors all the time in response to internal needs. For example, an animal who needs food becomes increasingly focused on finding food. Human behavior changes in response to insulin. Fortunately, changing the emergent behavior is straightforward in a potential fields representation of behaviors, since the output vector produced by each behavior can be scaled by a gain term. Returning to the case of the planetary rover rushing to make the last delivery, the gain on the move-to-goal behavior attracting the rover to the return vehicle should start going up, while the gain on the avoid-obstacle behavior should start going down as a function of time to launch.

The table below summarizes AuRA in terms of the common components and style of emergent behavior:

AuRA Summary	
Sequencer Agent	Navigator, Pilot
Resource Manager	Motor Schema Manager
Cartographer	Cartographer
Mission Planner	Mission Planner
Performance Monitoring Agent	Pilot, Navigator, Mission Planner
Emergent behavior	Vector summation, spreading activation of behaviors, homeostatic control

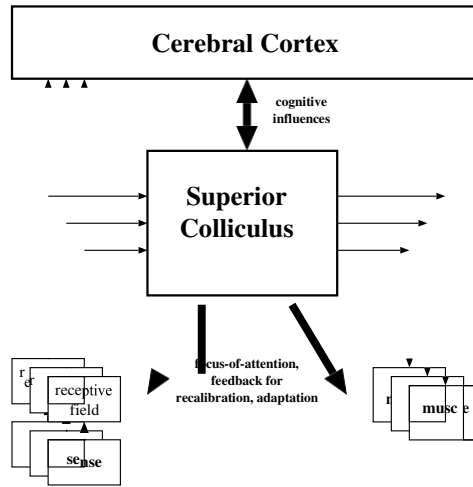
7.4.2 Sensor Fusion Effects (SFX)

Another managerial style of architecture is the Sensor Fusion Effects (SFX) architecture, which started out as an extension to AuRA by Murphy to incorporate pathways for sensing. The extension was to add modules to specify how sensing, including sensor fusion and handling sensor failures, are handled. Over time, SFX has reorganized both the reactive and deliberative components of AuRA although the two architectures remain philosophically identical. SFX is an example of how robustness can be built into an architecture. SFX has been used on eight robots for indoor office navigation, outdoor road following, and urban search and rescue.

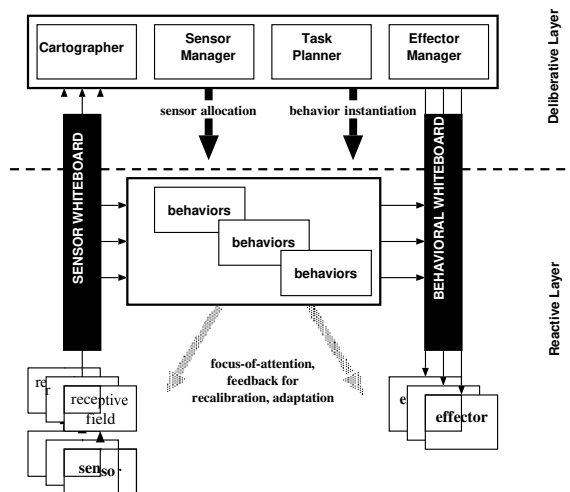
Fig. 7.4 shows a neurophysiological model of sensing based on studies with sensing in cats. The model suggests that sensory processing is initially local to each sensor, and may have its own sensor-dependent receptive field. This is consistent with reactive robot behaviors, and at least with the motivation for sensor fission. Sensor processing then appears to branch, with duplicates going to the *superior colliculus* (a mid-brain structure responsible for motor behaviors) and the other to the *cerebral cortex* (responsible for more cognitive functions). The branching allows the same sensor stream to be used simultaneously in multiple ways. In SFX, the equivalent superior colliculus functions are implemented in the reactive layer, and cortical activities in the deliberative layer. Branching of perception is done through the use of whiteboards, common in numerous AI systems as global cognitive data structures.

The deliberative component is divided into modules or object classes,

each of which is actually a software agent, a self-contained software program which specializes in some area of competence and can interact with other



cognitive model



SFX implementation

Figure 7.4 The cognitive model used for the implementation of the Sensor Fusion Effects Architecture.

agents. The dominant agent is called the Mission Planner agent. This agent serves to interact with the human and specify the mission constraints to the other agents in the deliberative layer. The agents in the deliberative layer attempt to find (and maintain) a set of behaviors which can accomplish the mission while meeting the constraints. The software agents in each deliberative layer are peers; just as with behaviors, they operate independently of each other. But the nature of deliberation suggests that they have to negotiate with other peers to find a satisfactory set of behaviors to accomplish the current task. One way to think of this partitioning is that the Mission Planner acts as a president or CEO in a large company, giving directions, while the behaviors are the workers. The agents in the lower deliberative layer are middle-management, planning how to organize the workers' assignments, monitoring productivity, and adapting assignments if necessary.

Within the deliberative layer, the Task Manager, Sensing Manager, and Effector Manager serve as the resource managers. The resource manager functions are divided across managers because the types of knowledge and algorithms are different for managing sensors and actions. The managers use AI planning, scheduling, and problem solving techniques to determine the best allocation of effector and sensing resources given the set of motor and perceptual schemas for a behavior. They are not allowed to relax any constraints specified by the Mission Planner, so they essentially know what the robot is supposed to do, but only the Mission Planner knows why. The advantage of this middle-management layer is that it simplifies the AI techniques needed for behavioral management.

The Sensing Manager in SFX is particularly noteworthy because of its explicit commitment to performance monitoring and problem solving. It has two software agents for monitoring both the task performance and whether the habitat has changed (if so, a performance failure is likely to occur). If a behavior fails or a perceptual schema detects that sensor values are not consistent or reasonable, the Sensing Manager is alerted. It can then identify alternative perceptual schemas, or even behaviors, to replace the problematic behavior immediately. Imagine a mobile robot in a convoy of robots hauling food to refugees. If the robot had a glitch in a sensor, it shouldn't suddenly just stop and think about the problem. Instead, it should immediately switch to a back-up plan or even begin to smoothly slow down while it identifies a back-up plan. Otherwise, the whole convoy would stop, there might be wrecks, etc. Then working in a background mode, the Sensing Manager can attempt to diagnose the

cause of the problem and correct it. In one demonstration, a robot using SFX resorted to shaking its camera to shake off a T-

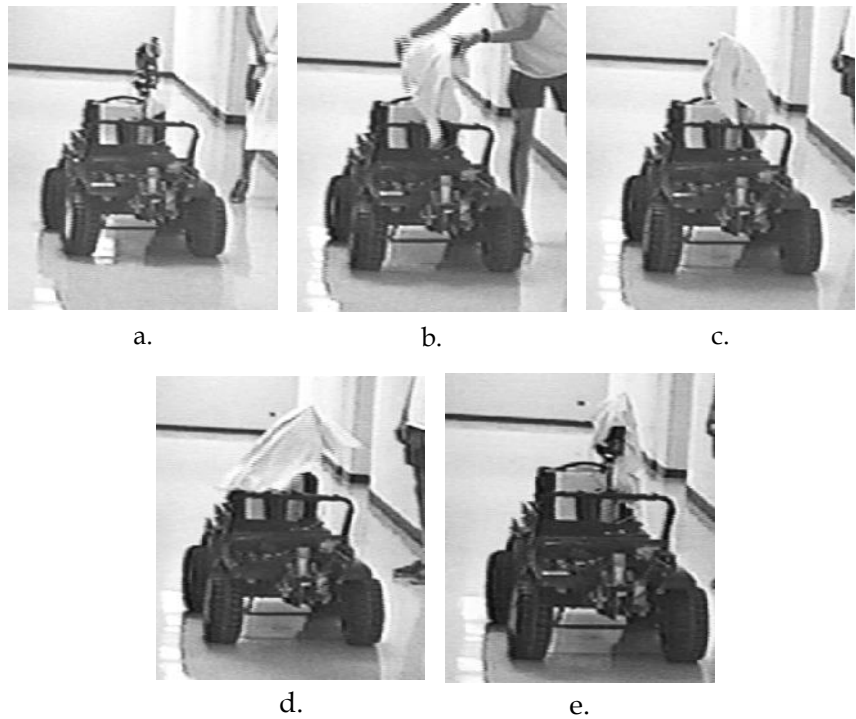


Figure 7.5 A robot using SFX to figure out that something is wrong with the camera. It cannot identify the problem, so it attempts to correct the problem by shaking.

Shirt covering the lens, as shown in Fig. 7.5.

The lower deliberative layer also contains a Cartographer agent and performance monitoring agents. The Cartographer is responsible for map making and path planning, while the performance agents attempt to observe the progress of the robot toward its goals and notice when the robot is unsuccessful, using work by Hughes.⁷¹ An example is when a robot gets stuck in the mud on a dirt road. The follow-road behavior is executing correctly; it is seeing the road and giving actuator commands correctly. The shaft encoders show that the wheels are turning. So some other process with more self-awareness than a reactive behavior is needed to notice that the robot isn't moving. In this case, if the robot is moving, the vision should change over time, not remain the same.

STRATEGIC The reactive component is also divided into two layers. In this case, the layers reflect *strategic* versus *tactical behaviors*, defined below. As noted ear-

TACTICAL BEHAVIORS tier, just about every hybrid architecture has its own method of combining behaviors in the reactive portion. AuRA uses potential field combination, whereas SFX uses a filtering method. The SFX philosophy is similar to subsumption: there are some fundamental behaviors that should override other behaviors. The difference is that in SFX, it is the “lowest” tactical behaviors which do the subsuming, not the “higher” strategic ones.

The idea of a tactical behavior is best seen by example. Recall the case of obstacle avoidance with potential fields. The use of a repulsive field was simple, but could lead to a global minima, where the repulsion could cancel out any other motive fields such as move-to-goal. The NaTs¹³² solution was to use the vector created by that other field as an input to the avoid-obstacle field. That vector would lead to a tangential field in addition to the repulsive field, resulting in an avoid-obstacle field which repulses the robot towards the direction it was heading in to begin with. In this case, the move-to-goal field was offering a strategic command; like a general, it was telling the troops to move ahead. But the avoid-obstacle field was like an infantry man; it was trying to go in the direction given by the general but not hit anything. The avoid-obstacle behavior filters the strategic direction (given by move-to-goal) with the immediate tactical situation (the presence of an obstacle).

Another example of a tactical behavior in SFX is speed-control. Speed-control in AuRA and many architectures is a by-product of the mechanism used for combining behaviors. The emergent speed in AuRA is the magnitude of the vector summed from all the active behaviors. In SFX, speed-control is considered a separate behavior. The safe velocity of a robot depends on many influences. If the robot cannot turn in place (in effect, turns like a car), it will need to be operating at a slow speed to make the turn without overshooting. Likewise, it may need to go slower as it goes up or down hills. These influences are derived from sensors, and the action is a template (the robot always slows down on hills), so speed control is a legitimate behavior. But the other behaviors should have some influence on the speed as well. So these other, strategic behaviors contribute a strategic speed to the speed-control behavior. If the strategic speed is less than the safe speed computed from tactical influences, then the output speed is the strategic speed. But if the tactical

7.4 Managerial Architectures 273
safe speed is lower, the output speed to the actuator is the tactical speed.²⁷³
Tactical behaviors serve as filters on strategic commands to

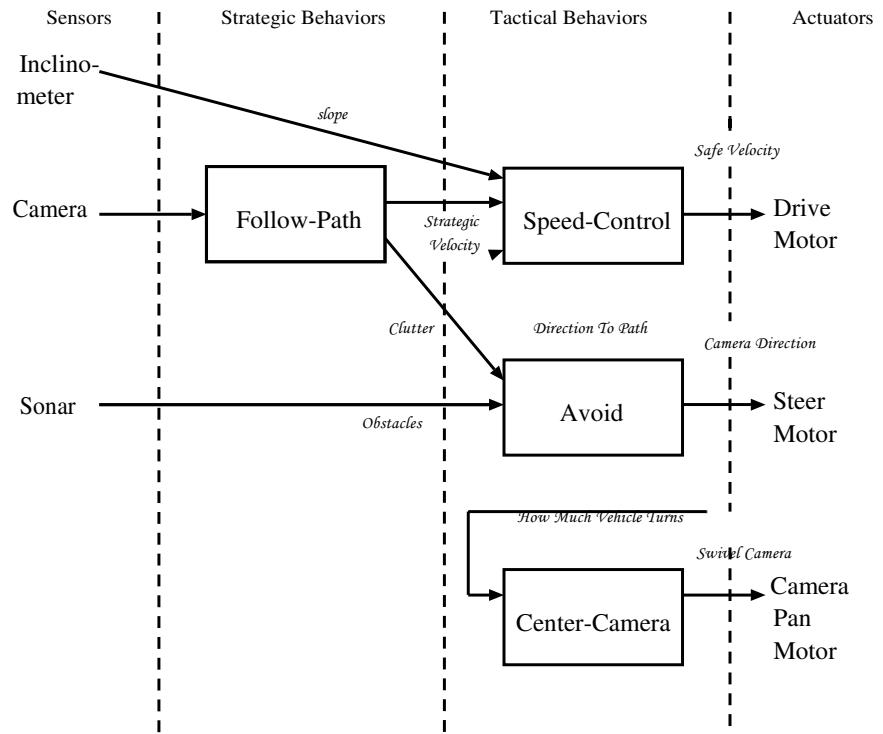


Figure 7.6 Strategic and tactical behaviors for following an outdoor path, used in the 1995 UGV Unmanned Ground Robot competition.

ensure that the robot acts in a safe manner in as close accordance with the strategic intent as possible. The interaction of strategic and tactical behaviors is still considered emergent behavior.

One outcome of the strategic-tactical partitioning was the discovery that every task to date could be done with one strategic behavior and several tactical behaviors. This means that the need to combine behaviors does not occur often, and so the combination mechanism is not particularly important. However, it should be emphasized that the strategic behaviors were often assemblages in the form of scripts. There were many strategic behaviors, but they were explicitly coordinated and controlled according to behavior-specific knowledge.

The table below summarizes SFX in terms of the common components and style of emergent behavior:

SFX Summary	
Sequencer Agent	Task Manager
Resource Manager	Sensing and Task Managers
Cartographer	Cartographer
Mission Planner	Mission Planner
Performance Monitoring Agent	Performance Monitor, Habitat Monitor
Emergent behavior	Strategic behaviors grouped into abstract behaviors or scripts, then filtered by tactical behaviors

7.5 State-Hierarchy Architectures

State-hierarchy styles of architectures organize activities by the scope of time knowledge. Since time is generally thought of as being **Present**, **Past**, and **Future**, state-hierarchy styles of architectures usually have 3 layers. Unfortunately, many other Hybrid architectures have 3 layers but with the layers meaning something totally different. Therefore, this book refers to them as state-hierarchies to emphasize the layering based on the state of knowledge. As with managerial styles, the organization is broken down into layers. Within each layer are peers of software agents or functions that accomplish the goals of that layer. As with managerial styles, a higher layer has access to the output of lower layers and can operate on the next lower layer.

7.5.1 3-Tiered (3T)

The 3T, or 3-Tiered, mobile robot architecture is the best example of a state-hierarchy system and is predominately used at NASA. The roots of 3T stem from continuing work at NASA, where aspects of Slack's NaT system,¹³² Gat's subsumption-style ATLANTIS architecture,⁵⁷ and Firby's RAPs system⁵³ were merged at JPL under the initial direction of David Miller, and refined by Pete Bonasso and Dave Kortenkamp at NASA's Johnson Space Center. As the name suggests, 3T divides itself into 3 layers, one clearly reactive, one clearly deliberative, and one which serves as the interface between the two. Fig. 7.7 show the layers actually running on three different computers at NASA Johnson Space Center controlling the Extra-Vehicular Activity Helper-Retriever (EVAHR) robot simulator. 3T has been used primarily for planetary

7.4 Managerial Architectures
rovers, underwater vehicles, and robot assistants for astronauts.

273



Figure 7.7 Each of the 3T layers running on a separate computer, controlling the EVAHR robot. (Photograph courtesy of the National Aeronautics and Space Administration, Johnson Space Center.)

The top layer of 3T is Planner. It fulfills the duties of the mission planner and cartographer by setting goals and strategic plans. These goals are passed to the middle layer, called the Sequencer. The Sequencer uses a reactive planning technique called RAPs to select a set of primitive behaviors from a library and develops a task network specifying the sequence of execution for the behaviors for the particular subgoal. The Sequencer is responsible for the sequencer and performance monitoring functions of a generic Hybrid architecture. The Sequencer layer instantiates a set of behaviors (called *skills*) to carry out the plan. These behaviors form the bottom layer, called the Controller or Skill Manager. In order to avoid confusion with the connotations of purely reflexive behaviors left over from the Reactive Paradigm, 3T does not call its behaviors “behaviors.” Its behaviors have the same broader scope as AuRA and SFX, permitting sensor fusion and assemblages of primitive behaviors. The preferred term is “skill” to distinguish its behaviors from the connotations of behaviors popularized by the subsumption architecture. A skill is often an assemblage of primitive skills; indeed, one of the interesting aspects of 3T is its foundation as a tool for learning

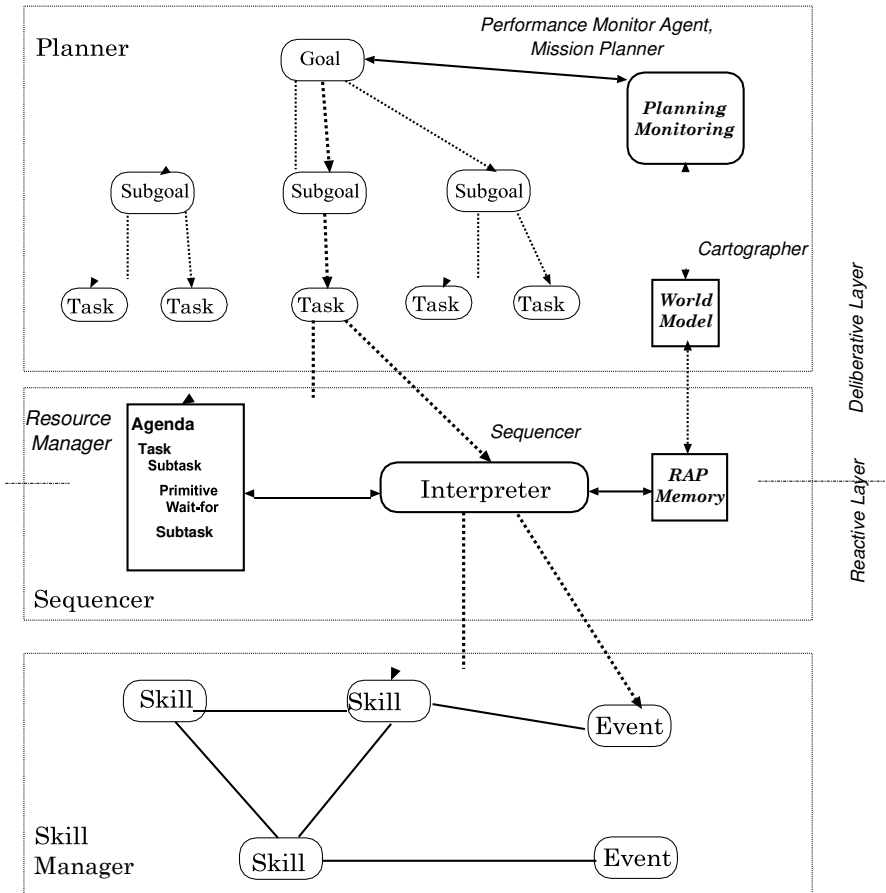


Figure 7.8 3T architecture.

EVENTS A powerful attribute of the lower level is that the skills have associated *events*, which serve as checkpoints to verify explicitly that an action has had the correct effect. In some regards, events are equivalent to the innate releasing mechanisms; both let the world be its own best representation.

The three layers represent true deliberation, reactive planning, and reactive control. They also represent a philosophy organized by scope of state, rather than scope of responsibility. The Skill Manager layer is composed of skills that operate only in the Present (although with some allowances to permit the persistence of some behaviors when the

stimulus
temporarily

disappears). The components in the Sequencer layer operate on state information reflecting memories about the **Past**, as well as the **Present**. Therefore, sequences of behaviors can be managed by remembering what the robot has already done and whether that was successful or not. This adds a great deal of robustness and supports performance monitoring. The Planner layer works with state information predicting the **Future**. It can also use information from the **Past** (what the robot has done or tried) and **Present** (what the robot is doing right now). In order to plan a mission, the planner needs to project what the environment will be and other factors.

UPDATE RATE.

In practice, 3T does not strictly organize its functions into layers by state (**Past**, **Present**, **Future**); instead it often uses *update rate*. Algorithms that update slowly are put in the Planner, while fast algorithms go in the Skill Manager layer. This appears to be a situation where the pragmatic considerations of computation influenced the design rules; in the early 1990's behaviors were very fast, reactive planning (especially RAPs and Universal Plans) were fast, and mission planning was very slow. However, many sensor algorithms involving computer vision were also slow, so they were placed in the Planner despite their low-level sensing function.

The table below summarizes 3T in terms of the common components and style of emergent behavior:

3T	
Sequencer Agent	Sequencer
Resource Manager	Sequencer (Agenda)
Cartographer	Planner
Mission Planner	Planner
Performance Monitoring Agent	Planner
Emergent behavior	Behaviors grouped into skills, skills grouped into task networks

7.6 Model-Oriented Architectures

Both the managerial and state-hierarchy styles of architectures evolved directly from the Reactive Paradigm. The designers sought to add more cognitive functionality to the Reactive Paradigm. As such, managerial and state-hierarchy styles are more bottom-up in flavor, emphasizing

7.6 *Model-Oriented Architectures* 279
behaviors or skills as the basic building blocks. However, a new influx of researchers from the traditional AI community, particularly Stanford and SRI, has en-

tered robotics and made a distinct contribution. Their architectures have a more top-down, symbolic flavor than managerial and state-hierarchies. One hallmark of these architectures is that they concentrate symbolic manipulation around a global world model. However, unlike most other Hybrid architectures, which create a global world model in parallel with behavior-specific sensing, this global world model also serves to supply perception to the behaviors (or behavior equivalents). In this case, the global world model serves as a virtual sensor.

The use of a single global world model for sensing appears to be a throw-back to the Hierarchical Paradigm, and conceptually it is. However, there are four practical differences. First, the monolithic global world model is often less ambitious in scope and more cleverly organized than earlier systems. The world model is often only interested in labeling regions of the sensed world with symbols such as: hallway, door, my office, etc. Second, perceptual processing is often done with distributed processing, so that slow perceptual routines run asynchronously of faster routines, and the behaviors have access to the latest information. In effect, the “eavesdropping” on perception for behaviors is an equivalent form of distributed processing. Third, sensor errors and uncertainty can be filtered using sensor fusion over time. This can dramatically improve the performance of the robot. Fourth, increases in processor speeds and optimizing compilers have mitigated the processing bottleneck.

Two of the best known model-oriented architectures are the Saphira architecture developed by Kurt Konolige with numerous others at SRI, and the Task Control Architecture (TCA) by Reid Simmons which has been extended to do multi-task planning with the Prodigy system. The Saphira architecture comes with the ActivMedia Pioneer robots.

7.6.1 Saphira

SAPHIRA

COORDINATION,
COHERENCE,
COMMUNICATION

The *Saphira* architecture, shown in Fig. 7.9, has been used at SRI on a variety of robots, including Shakey’s direct descendents: Flakey and Erratic. The motivation for the architecture stems from the basic tenet that there are three keys to a mobile robot operating successfully in the open world: *coordination*, *coherence*, and *communication*.⁷⁷ A robot must coordinate its actuators and sensors (as has been seen through in the Reactive Paradigm), but it must also coordinate its goals over a period of time (which is not addressed by the Reactive Paradigm). Whereas the

motivation for
coordination is
compatible with
reactivity,
coherence is an
explicit break
from the Reactive
Paradigm.

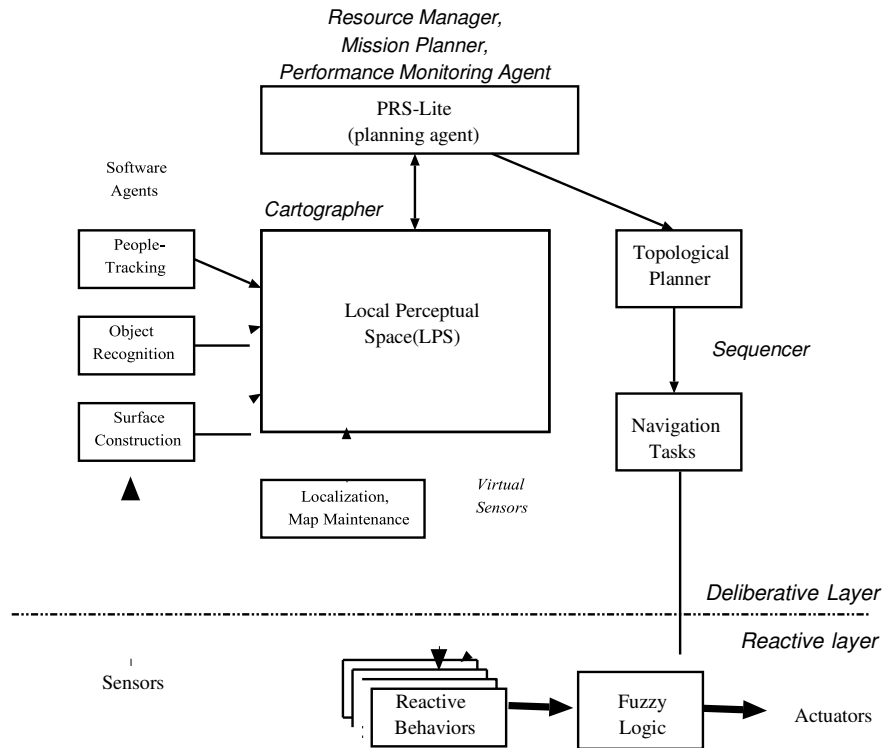


Figure 7.9 Simplified view of Saphira architecture.

Coherence is essentially the ability of the robot to maintain global world models, which Konolige and Myers argue is essential for good behavioral performance and interacting with humans.⁷⁷ Finally, communication is important because robots have to interact with humans and, as will be seen in Ch. 8, other robots. This introduces the problem of having common frames of reference (anchors) for communicating.

The bulk of the architecture is concerned with planning, and uses a type of reactive planner called *PRS-lite* for Procedural Reasoning System-lite.⁷⁷ PRS-lite is capable of taking natural language voice commands from humans (“deliver this to John”) and then operationalizing that into navigation tasks and perceptual recognition routines. Both planning and execution relies on the Local Perceptual Space, the central world model. Much processing is

devoted to maintaining an accurate model of the world based on the robot’s sensors and assigning symbolic labels to regions. Saphira also divides the deliberation activities among software agents. This provides a high degree of flexibility. Since software agents are independent, they don’t even have to run on-board the robot they’re controlling. In the 1996 AAAI Mobile Robot Competition, ⁷⁸ robots running Saphira actually had some of their planning reside on a local workstation, transmitted through a radio link. ⁶² This will be covered in more detail in Ch. 8.

VIRTUAL SENSOR

The reactive component of Saphira consists of behaviors. The behaviors extract *virtual sensor* inputs from the central world model, the Local Perceptual Space. The behavioral output is fuzzy rules, which are fused using fuzzy logic into a velocity and steer command. Fuzzy logic turns out to be a very natural way of fusing competing demands, and is less ad hoc than Boolean logic rules (e.g., “if x and y but not z, then turn left”). The behaviors are managed by the plan execution of the planned navigation tasks. The fuzzy logic mechanism for combining behaviors produces essentially the same results as a potential field methodology, as described by Konolige and Myers. ⁷⁷ The Local Perceptual Space can improve the quality of the robot’s overall behavior because it can smooth out sensor errors. Although this central processing introduces a computational penalty, the increases in processor power and clock speeds have made the computational costs acceptable.

The table below summarizes Saphira in terms of the common components and style of emergent behavior:

Saphira	
Sequencer Agent	Topological planner, Navigation Tasks
Resource Manager	PRS-lite
Cartographer	LPS
Mission Planner	PRS-lite
Performance Monitoring Agent	PRS-lite
Emergent behavior	Behaviors fused with fuzzy logic

7.6.2 Task Control Architecture (TCA)

TASK CONTROL

Reid Simmon’s *Task Control Architecture (TCA)* has been used extensively by

ARCHITECTURE (TCA)

robots designed for NASA, including Ambler, Dante (Fig. 7.11), and service robots. It is also the intelligence inside Xavier (shown in Fig. 7.11, the

280 Carnegie Mellon University robot which is accessible over the web).⁷ *The Hybrid Deliberative/Reactive Paradigm*.
Xavier has has traveled autonomously over 210 kilometers in the
hallways at CMU

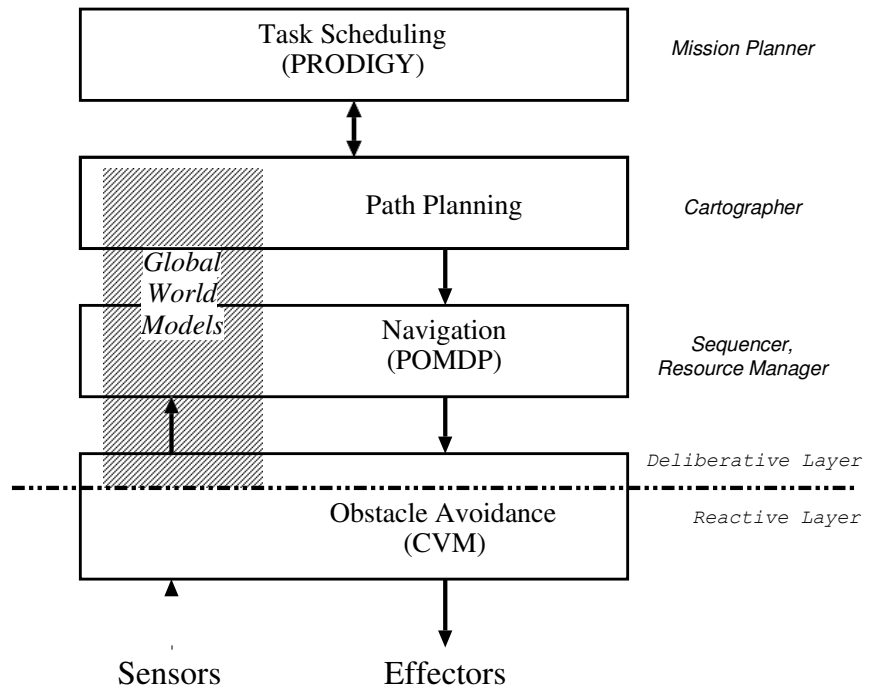


Figure 7.10 Layout of the Task Control Architecture.

in response to 30,000 requests by web users to navigate to a particular site.¹³⁰ TCA is more difficult to evaluate as a Hybrid architecture, in part because it has more of an operating system flavor than a general purpose architecture. Also, there are no behaviors per se. However, many of the low level tasks resemble behaviors, and TCA shares the hybrid philosophy of layering intelligence, and having lower modules fail upwards.

The basic layout of TCA for indoor navigation is presented in a somewhat different form than usual¹³¹ in Fig. 7.10 to provide consistency with the terminology of this chapter. TCA uses dedicated sensing structures such as evidence grids (see Ch. 11) which can be thought of as a distributed global world model. Sensor information percolates up through the global models. The basic task flow is determined by the Task Scheduling Layer, which uses the Prodigy planner. (A layer is more along the lines of a software agent or a subsystem in other architectures.) This layer interacts with the user and determines the goals and order of

execution. For example, if the robot is

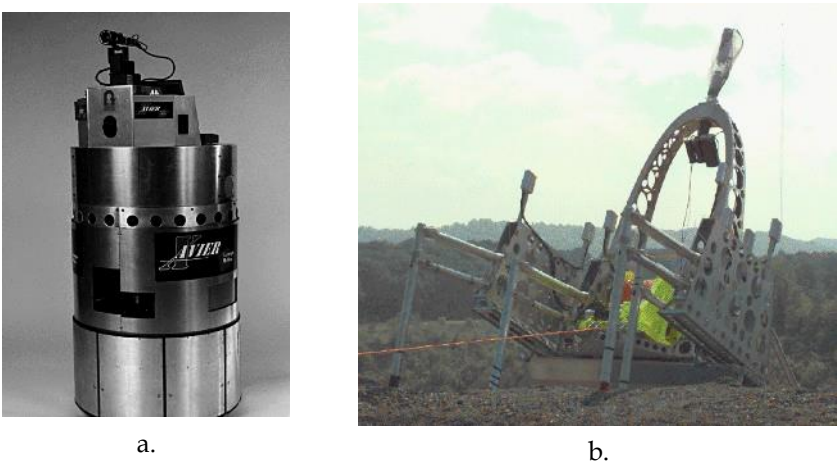


Figure 7.11 Two robots using TCA: a.) Xavier, a RWI robot at Carnegie Mellon University (photograph courtesy of Reid Simmons) and b.) Dante (photograph courtesy of NASA Ames Research Center).

CURVATURE-VELOCITY

METHOD

given several jobs to drop, Prodigy can prioritize and optimize the schedule. Once the current task has been established, the Path Planning layer is engaged. Navigation is handled by a Partially Observable Markov Decision Process (POMDP, pronounced “pom D P”) module which determines what the robot should be looking for, where it is, and where it has been. As with the relationship between strategic and tactical behaviors in SFX, the Obstacle Avoidance Layer takes the desired heading and adapts it to the obstacles extracted from the evidence grid virtual sensor. The layer uses a *curvature-velocity method* (CVM) to factor in not only obstacles but how to respond with a smooth trajectory for the robot’s current velocity.

The table below summarizes TCA in terms of the common components and style of emergent behavior:

TCA	
Sequencer Agent	Navigation Layer
Resource Manager	Navigation Layer
Cartographer	Path-Planning Layer
Mission Planner	Task Scheduling Layer
Performance Monitoring Agent	Navigation, Path-Planning, Task-Scheduling
Emergent behavior	Filtering

7.7 Other Robots in the Hybrid Paradigm

One implicit criterion for evaluating the utility of a Paradigm and derivative architectures is its popularity. The Hybrid Paradigm describes many of the robots being developed in laboratories today. It is clearly popular. A large number of these robots fall into the Hybrid category by virtue of being “neither fish nor fowl,” in this case being neither purely Reactive nor Hierarchical. Almost every robot has a set of functions that are equivalent to behaviors, though several have followed the lead of 3T in calling those functions by other names (e.g., “skills”) to reduce the connotations of purely reflexive behaviors. Even architectures that started out as Hierarchical, namely Albus’ Real-time Control Architecture¹ and Simmons’ Task Control Architecture,¹³¹ recast many of their functions into behavioral components.

One of the major influences on the mobile robot community has been the DARPA UGV Demo II and Demo III projects, which forwarded the state of the art in outdoor ground vehicle control and navigation. The tasks were well-suited for a hybrid approach: the idea was to give a HMMWV military jeep a map, a set of directions, and send it off. It would autonomously plan a route, drive along roads, then go off roads as needed. In a military reconnaissance mode, the HMMWV was expected to autonomously stay behind hills, avoid trees and rocks, and covertly position itself so that Reconnaissance, Surveillance, and Target Acquisition (RSTA, pronounced “rist-tah”) sensors could peek over and find enemy locations. Demo II even had multiple HMMWVs traveling autonomously in formation.

The European Community ESPRIT agency has also significantly influenced mobile robotics, sponsoring research in automating highway vehicles via initiatives in the 1990’s. The United States has shown some interest through several iterations of “intelligent vehicle highway systems” programs. The most notable of these automated vehicles were constructed by Dickmanns and Graefe in Germany⁴⁵ and the CMU Navlab led by Dean Pomerleau.¹¹⁶

A third influence on mobile robots has been NASA’s push for autonomous planetary rovers, which culminated in the Mars Pathfinder mission with the Sojourner rover. Mapping a planet’s surface is intrinsically a deliberative function, as is planning paths for optimal exploration. Unfortunately, from the Viking lander missions, Mars was known to be very rocky. The approaches quickly fell into camps. One, led by MIT,

favored deploying many small, legged robots, such as Genghis, that would climb over small rocks and go around others. The “low-to-the-ground” viewpoint, which limits what a robot—and therefore, the scientists on Earth—can sense, would be over-

come by the large number of robots. The other approach, characterized by Carnegie Mellon University's Ambler robot, proposed a single large robot with a higher viewpoint and stability. Ambler was built by "Red" Whitaker at the CMU Field Robotics Institute to be able to maintain a sensor platform at a level height by stepping over the majority of rocks, but at a tremendous penalty in size, weight, and power. In the end, planetary rover researchers have gravitated towards wheeled vehicles with some type of articulation to maintain stability, such as seen with Sandia National Laboratories' Rattler. An extension of the Ambler design philosophy was manifested in the Dante robots. These were built to rappel down steep canyons and volcanoes on Mars (and Earth). Dante was able to lower itself successfully most of the way into a volcano in Antarctica, but could not climb back out. It was dropped while being lifted out by a helicopter, twisting its frame.

7.8 Evaluation of Hybrid Architectures

In some regards, it is difficult to evaluate Hybrid architectures individually. Each architecture is still evolving and the deliberative component is being expanded practically daily. Returning to the four criteria set forth in the overview in Part I, it is interesting to evaluate the architectures as a whole.

In terms of support for modularity, each architecture is highly modular. Most are divided into layers, which are then subdivided into modules. As the software agent programming style for AI gains in popularity, probably more architectures will implement deliberative modules as independent specialists. AuRA and SFX clearly exhibit an organization which lends itself to object-oriented programming. The use of specialists also enhances the ease of portability.

Hybrids tend to have a high degree of niche targetability. The addition of the deliberative component allows Hybrids to be used for applications not appropriate for purely Reactive systems. However, the partitioning between reaction and deliberation allows the reactive portion to be used alone for purely reactive applications.

Another attractive aspect of Hybrid architectures is that they often explicitly attempt to ensure robustness. In the SFX and 3T architecture, modules within the various deliberative components attempt to monitor the performance of the reactive behaviors and either replace or adapt the configuration as needed.

An evaluation of Hybrid architectures would not be complete without asking the question, “so what’s the difference between what Shakey could do (if it had had infinite computation power) and what a Hybrid can do?” Speed of execution aside, it is interesting to ponder whether the only fundamental difference is how Hierarchical and Hybrids achieve the same ends. Hybrids certainly explicitly reflect more of the principles of software engineering (modularity, coherence, design for reuse, etc.). The two paradigms also certainly reflect different attitudes toward world modeling. In the Hybrid Paradigm, global models are used only for symbolic functions. The frame problem either doesn’t exist or is minor because 1) execution is reactive and therefore well-suited for unstructured environments, and 2) software agents can use agent-specific abstractions to exploit the structure of an environment in order to fulfill their particular role in deliberation. Global models are generally closed world, but the world is “closed” at the deliberative level. The robot can think in terms of a closed world, while it acts in an open world. Another major philosophical difference is the role of planning and execution. Under STRIPS, Shakey planned every move, down to the lowest level of granularity, and had problems confirming that an action had been accomplished. It should be noted that modern planners often produce only partial plans, then execute that part of the plan, note the results and plan the next step. It is not clear whether a SENSE, PLAN, ACT paradigm with a more suitable planner would produce a more intuitively appealing robot architecture. The final distinction between the Hierarchical and Hybrid Paradigms may be the influence of biology. The Hybrid Paradigm has its roots in ethology, and provides a framework for exploring cognitive science. The Hierarchical Paradigm is less clearly cognitively plausible, although both share the same cognitive motivation.

7.9 Interleaving Deliberation and Reactive Control

The primary contribution of a Hybrid architecture is to provide a template for merging deliberation and reaction. However, interleaving the two functions is often task dependent. In navigation, there is the issue of interleaving path planning and reactive execution of the path. Another issue is monitoring the progress of behaviors and terminating behaviors correctly.

One of the obvious task domains for a mobile robot is navigation, simply getting around in the world. Many robot architectures have well-developed modules for handling navigational tasks. In the early 1990's, navigation

served as an excellent example of why deliberation and reaction were largely separate, and reinforced a “plan once, execute lots” mentality. However, advances in technology and new domains are causing researchers to rethink the strict separation.

By the early 1990’s, many algorithms existed which if given a two or even three dimensional map could compute an optimal path for a robot to take through the world. These path planning algorithms exhibited several drawbacks. One was that they were all computationally expensive, taking between 5 and 15 minutes to execute. This prevented the robot from continuously generating the path. But as was seen with Shakey, if the robot tried to execute a pre-computed path, it would be vulnerable to unexpected changes in the world. That map would represent the best, currently available knowledge about the world: where obstacles were, unpassable terrain, etc. But a map is at best a representation of a closed world; it can’t show what has changed since it was built, unless a robot goes there and senses it. So the robot can generate an optimal path, but it may discover an unmodeled obstacle blocking that path. If the robot was just using the precomputed path to navigate, it would have to stop, update the map with the newly discovered obstacle, then replan the optimal path, resulting in slow progress.

If the robot used reactive behaviors to navigate, there was no guarantee that it would make it to the goal any faster. Even if it headed off in the right direction, it might spend time getting into and out of box canyons. By letting the robot solely react to the world, it might make poor choices.

WAYPOINT

The common solution was to mix path planning and reaction by having a planning algorithm in the cartographer generate a complete optimal route for the robot to take, but then decompose that route into segments, usually straight lines. The end of each path segment is called a *waypoint*, because it is a point along the way to the final goal. Each waypoint can be a goal to reach, which can be accomplished by behaviors (move-to-goal(), avoid-obstacles()). When the robot reaches the first goal, the sequencer agent can give the behavioral manager the coordinates or landmarks for the next goal, and so on. Notice this allows computationally expensive path planning to occur only once, and then the behaviors take care of actually navigating.

As will be seen in Part II, this strategy has its own set of drawbacks. But for the most part it is a well-accepted means of partitioning deliberation and reaction in navigation. However, with the advent of the

Intel Pentium and Sparc microprocessors, the computational demands of path planning algorithms have been mitigated.

Algorithms which ran on the order of once every 10 minutes can now execute once every second. As a result, the mis-

matches in planning and reaction times are no longer a compelling reason to enforce a strict separation of deliberation and reaction. However, the software engineering reason for the partition remains: things which operate on symbols and global information should be in the deliberator; things which operate directly on sensors and actuators should be in the reactor.

The above example of Navigation illustrates a top-down interleaving of deliberation and reaction. The deliberative layer(s) decompose the mission into finer and finer steps until it arrives at a set of behaviors capable of accomplishing the first subgoal. Another example of interleaving deliberation and reaction is the role of the deliberative layer(s) in supplying the reactor with expectations and virtual sensors. And it is important to note that deliberation can be triggered bottom-up as well.

It is not always the case that the deliberator generates the set of behaviors, turns them on, and lets them execute until they either complete the subtask or fail. In the case of sensing, it might be desirable for the deliberator to make some of the global models being constructed available to the behaviors. For example, consider a robot vehicle following a road. Now suppose that it is looking for the large tree, since it is supposed to turn right at the intersection after the tree. In order to maintain the correct sequence of behaviors, the deliberator has a global world model where trees are noted. The reactive behaviors of following a road and avoid an obstacle do not need the explicit representation of a tree. But what if the tree casts a shadow which might confuse the follow-path behavior and cause it to give the robot incorrect steering commands? In this case, it would be useful if the information about the presence of a tree could be absorbed by the behavior. One way to do this is to permit methods on the world model to act as virtual sensors or perceptual schema. Then the follow-road behavior could use the virtual sensor, which tells the behavior when the road boundaries extracted by the vision sensor are probably distorted, to ignore affected regions in the image that would normally be observed by the vision sensor. This is an example of how the deliberative layer can aid with *selective attention*, or filtering of perception for a context.

But the reactor may wish to trigger deliberation under many circumstances. Most of these cases involve failure or exception handling. Suppose that a sensor breaks and the perceptual schema cannot produce a percept. The schema cannot diagnose itself. Therefore, its failure would

cause a deliberative function, such as the Sensing Manager in SFX, to be triggered. If the Sensing Manager cannot find a replacement perceptual schema or equivalent behavior, then it can no longer meet the constraints imposed by the Mission

COGNIZANT FAILURE

Planner. In the SFX system, the Sensing Manager would trigger the Mission Planner. The Mission Planner presumably has the intelligence to relax constraints on the mission, or in practice to alert the human supervisor. This is an example of “failing upwards,” where a lower level module fails and a higher level, or smarter, module has to take over.

Failing upwards is also known as a *cognizant failure*, meaning that the failure takes some amount of awareness of the context to resolve, not that the cognition itself has failed. Erann Gat motivated the need for cognizant failures through the “Wesson Oil” problem.⁵⁷ In old Wesson Oil TV advertisements, a mother would be cooking chicken in oil when she would have to stop to rush her son to the emergency room for treatment of a broken arm. Ideally the mother would not only turn off the stove but remove the chicken from the oil. In the commercial, the mother neglects to remove the chicken. Since she is using Wesson Oil, the chicken isn’t ruined when they return. The point of the Wesson Oil problem for robotics is what happens when a set of reactive behaviors are turned off in the middle of a sequence? In many cases, it may not be just a simple act of de-instantiating the behaviors and activating new ones. There may be behaviors in the sequence that have to execute or disaster will occur (turn off the stove) as well behaviors which must run or lead to merely undesirable side effects (chicken sits in oil if it is not removed). All of this means that the sequencer must know why a failure or need to change behaviors has occurred and what the intent of the current sequence is. Deliberation is definitely not trivial!

7.10 Summary

The Hybrid Deliberative-Reactive Paradigm can be thought of as PLAN, then SENSE-ACT (P,S-A). The SENSE-ACT portion is always done with reactive behaviors, while PLAN includes a broader range of intelligent activities than just path or mission planning. Planning can be interleaved with execution; then for the most part, the robot plans a set of behaviors which will execute for a long time. Sensing is also hybrid; the reactive portion uses local, behavior-specific representations while the deliberative portion uses global world models.

Architectures in the Hybrid paradigm usually encapsulate functionality into modules. The basic modules are the *mission planner*, *sequencer agent*, *behavioral manager*, *cartographer*, and *performance monitor*. Hybrid architectures

⁷⁹*Interleaving Deliberation and Reactive Control* always have portions which are recognizably deliberative and reactive. The²⁹²

rule of thumb is that functions which operate on symbolic information go in the deliberative layer, or deliberator, while functions which transform sensor data into actuator commands go in the reactive layer, or reactor. The reactive component is organized by behaviors, but the definition of behavior is wider than the purely reflexive behaviors in the Reactive Paradigm. Behaviors may be called skills or by other names to eliminate confusion with reflexive behaviors. Hybrid architectures also exhibit a wide variety of mechanisms for combining behaviors, including subsumption, potential field combination, filtering, voting, and fuzzy logic. The reactive portion tends to make more use of assemblages of behaviors than in the Reactive Paradigm.

The deliberative component is often subdivided into layers. The layering may reflect scope in managerial responsibilities or in time horizon. Layers often consist of modules implemented as software agents. These agents may have agent-specific world models or share a single global world model. The world model can also serve as a virtual sensor for use by behaviors.

Managerial styles of architectures favor a bottom-up organization, concentrating on layering planning on top of behaviors and behavior management. Deliberation is based on whether symbols are being manipulated. The overall behavior emerges as a function of the behavioral layer. State-hierarchies divide deliberation and reaction by the state, or scope of knowledge, available to the modules or agents operating at that layer. The three states are Past, Present, Future. Responsibilities are organized into planning and sequencing components. The overall emergent behavior is more due to the sequencing of behaviors, or skills, rather than concurrency. Model-based styles favor a top-down organization, focusing on the creation and maintenance of a global world model. The world model then serves as a virtual sensor to the behaviors and as a data structure for planning. Often the software agents within the deliberative layer in all three architectural styles exhibit self monitoring and problem solving abilities, and fail upwards if the mission cannot be accomplished. As such, they distribute planning and problem-solving.

8

Topological Path Planning

Chapter Objectives:

Define the difference between a *natural* and *artificial landmark* and give one example of each.

Given a description of an indoor office environment and a set of behaviors, build a relational graph representation labeling the *distinctive places* and *local control strategies* using gateways.

Describe in one or two sentences: *gateway*, *image signature*, *visual homing*, *viewframe*, *orientation region*.

Given a figure showing landmarks, create a topological map showing landmarks, landmark pair boundaries, and orientation regions.

9.1 Overview

TOPOLOGICAL *Topological, route, or qualitative navigation* is often viewed as being more sim-

NAVIGATION ple and natural for a behavior-based robot. Certainly people frequently give other people routes as directives; therefore it seems natural to expect a robot to be able to parse commands such as “go down the hall, turn to the left at the dead end, and enter the second room on the right.” Even without a map of where everything is, there is enough information for navigation as long as the robot knows what a “hall,” “dead-end,” and “room” is.

Route representations fall into one of two approaches:

1. **Relational.** Relational techniques are the most popular, and can be thought of as giving the robot an abbreviated, “connect the dots” graph-style of spatial memory.

- 2 **Associative.** Relational techniques tend to focus on the graph-like representation of spatial memory. Associative techniques focus on coupling sensing with localization in a manner which parallels the tight coupling of sensing to acting found in reflexive behaviors.

Because relational techniques use an explicit representation, they can support path planning. Associative techniques are better for retracing known paths.

9.2 Landmarks and Gateways

LANDMARK

Topological navigation depends on the presence of landmarks. A *landmark* is one or more perceptually distinctive features of interest on an object or locale of interest. Note that a landmark is not necessarily a single, self-contained object like “red door.” A landmark can be a grouping of objects; for example, “McDonald’s” means a tall sign, bright building of a certain shape, and a parking lot with a lot of activity. Another outdoor landmark might be a “stand of aspen trees.”

Landmarks are used in most aspects of navigation. If a robot finds a landmark in the world and that landmark appears on a map, then the robot is localized with respect to the map. If the robot plans a path consisting of segments, landmarks are needed so the robot can tell when it has completed a segment and another should begin. If a robot finds new landmarks, they can be added to its spatial memory, creating or extending a map.

GATEWAYS

Dave Kortenkamp popularized a particularly interesting special case of landmarks: *gateways*.⁷⁹ A gateway is an opportunity for a robot to change its overall direction of navigation. For example, an intersection of two hallways is a gateway; the robot can choose to go straight or turn left or right. Because gateways are navigational opportunities, recognizing gateways is critical for localization, path planning, and map making.

ARTIFICIAL LANDMARKS

NATURAL LANDMARKS

Landmarks may be either *artificial* or *natural*. The terms “artificial” and “natural” should not be confused with “man-made” and “organic.” An artificial landmark is a set of features added to an existing object or locale in order to either support recognition of the landmark or some other perceptual activity. An interstate highway exit sign is an example of an artificial landmark. It is put there with the purpose of being easy to see (retro-reflective) and the white-on-green font is sized for optimal visibility (perceptual activity is reading the sign). A natural landmark

is a configuration of exisiting features selected for recognition which were not expressly designed for the

CRITERIA FOR

LANDMARKS

perceptual activity. If someone gives directions to their house, “take the sec- ond right after the McDonald’s,” the McDonald’s is being used as an orienta- tion cue for navigation to their house. Clearly, the McDonald’s was not built with the purpose of being a navigational cue to a private home. The fact that it was used for another purpose means it is a natural landmark.

Regardless of whether a landmark is artificial or natural, it must satisfy three criteria:

1. *Be readily recognizable.* If the robot can’t find the landmark, it isn’t useful.
2. *Support the task dependent activity.* If the task dependent activity is simply an orientation cue (“take the second right after the McDonald’s”), then being recognized is enough. Suppose a landmark is intended to provide position estimation to guide docking the space shuttle to a space station. In that case, the landmark should make it easy to extract the relative distance to contact.
3. *Be perceivable from many different viewpoints.* If the landmark is widely visible, the robot may never find it.

Fig. 9.1 shows artificial landmarks constructed for use in the 1992 AAAI Mobile Robot Competition.⁴² Each robot was supposed to follow a route be- tween any sequence of waypoints in an arena. The teams were allowed to mark the waypoints with artificial landmarks. Each waypoint is readily rec- ognizable by the checkerboard pattern. The task dependent activity of going to the correct waypoint is facilitated by the cylindrical barcodes which are unique for each station. Notice that the use of a cylinder guaranteed that landmarks would be perceivable from any viewpoint in the arena.

A good landmark has many other desirable characteristics. It should be passive in order to be available despite a power failure. It should be perceiv- able over the entire range where the robot might need to see it. It should have distinctive features, and, if possible, unique features. Distinctive features are those which are locally unique; they appear only as part of the landmark from every viewpoint of the robot in that region of the world (e.g., there is only one McDonald’s on Busch Boulevard). If the feature occurs only once in the entire region of operations (e.g., there is only one McDonald’s in Tampa), then the feature would be considered globally unique. In addition to being perceivable for recognition

7.9 Interleaving Deliberation and Reactive Control. If the robot needs³²⁹ to position itself to within 0.5 meters of the landmark, then the landmark must be designed to achieve that accuracy.



Figure 9.1 Artificial landmarks used in the 1992 AAAI Mobile Robot Competition. (Photograph courtesy of AAAI.)

9.3 Relational Methods

Relational methods represent the world as a graph or network of nodes and edges. Nodes represent gateways, landmarks, or goals. Edges represent a navigable path between two nodes, in effect that two nodes have a spatial relationship. Additional information may be attached to edges, such as direction (N,S,E,W), approximate distance, terrain type, or the behaviors needed to navigate that path. Paths can be computed between two points using standard graph algorithms, such as Dijkstra's single source shortest path algorithm. (See any algorithm textbook for details.)

One of the earliest investigations of relational graphs for navigation was by Smith and Cheeseman.¹³³ They represented the world as a relational graph, where the edges represented the direction and distance between

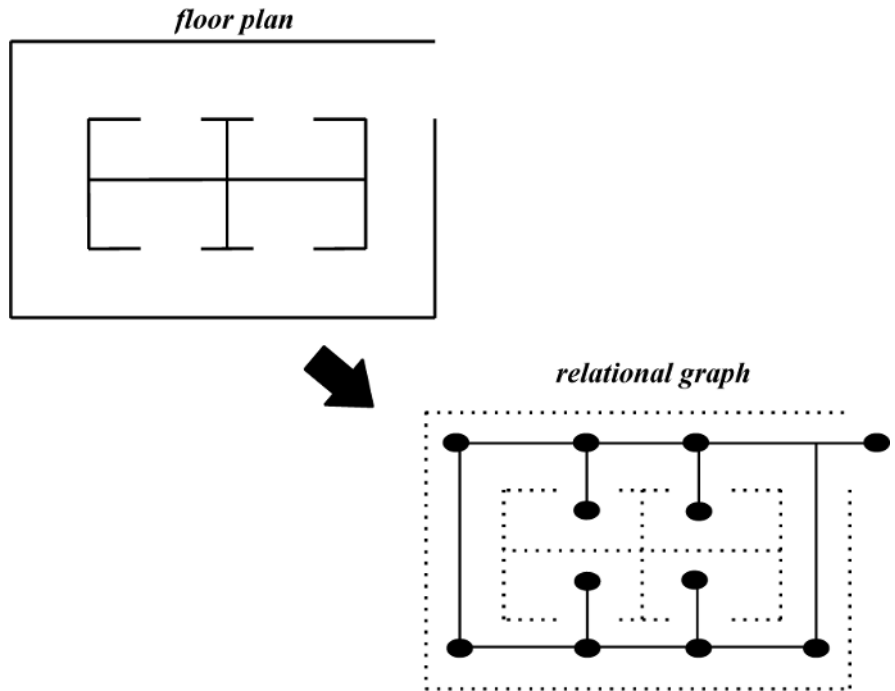


Figure 9.2 Representation of a floor plan as a relational graph.

simulated what would happen if a robot used dead-reckoning to navigate. As would be expected from the section on proprioception in Ch. 6, they found that the error would continually increase and soon the robot would be unable to reach any of the nodes.

9.3.1 Distinctive places

DISTINCTIVE PLACE

Kuipers and Byun tied relational graphs to sensing in their seminal work with distinctive places.⁸¹ A *distinctive place* is a landmark that the robot could detect from a nearby region called a neighborhood. Their work was motivated by research in cognitive science indicating that spatial representation in the animal kingdom forms a multi-level hierarchy. (More recent studies suggest this hierarchy isn't as clearly partitioned as previously thought.) The lowest level, or most primitive way of representing space, was by identifying landmarks (doors, hallways) and the procedural knowledge to travel

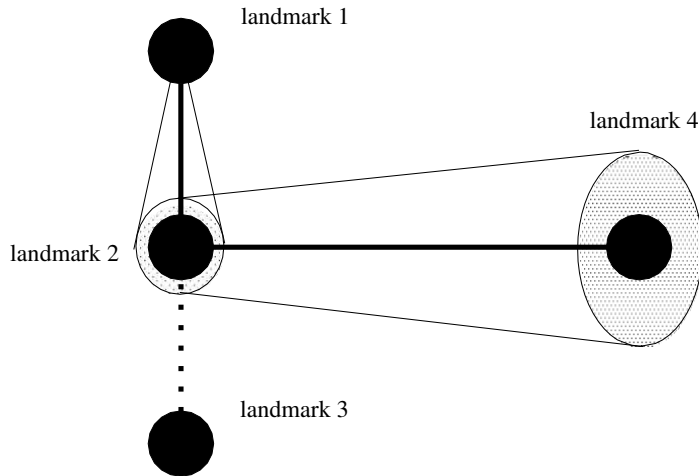


Figure 9.3 Propagation of error in a relational graph.

between them (follow-hall, move-thru-door). The next layer up was topological. It represented the landmarks and procedural knowledge in a relational graph, which supported planning and reasoning. The uppermost level was metric, where the agent had learned the distances and orientation between the landmarks and could place them in a fixed coordinate system. Higher layers represented increasing intelligence.

Kuipers and Byun's representation is of particular interest. Each node represents a distinctive place. Once in the neighborhood, the robot can position itself in a known spot relative to the landmark using sensor readings. One example of a distinctive place was a corner. (Kuipers and Byun worked in simulation; this did not turn out to be realistic with sonars.) The idea was that the robot could repeatably move around in the neighborhood the corner until, for example, 1 meter from each wall. Then the robot would be localized on the map.

LOCAL CONTROL

STRATEGY

HILL-CLIMBING

ALGORITHM

An arc or edge in the relational graph was called a *local control strategy*, or *lcs*. The local control strategy is the procedure for getting from the current node to the next node. When the robot senses the landmark it is filling

in values for a set of features. The robot uses a *hill-climbing algorithm*. The

hill-climbing algorithm directs the robot around in the neighborhood until

a measurement function (e.g., how far away the walls are) indicates when the robot is at a position where the feature values are maximized (e.g., both

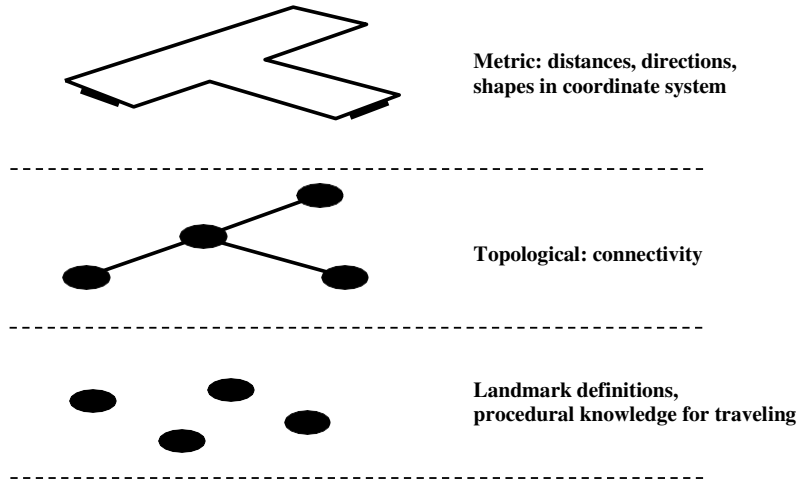


Figure 9.4 Multi-level spatial hierarchy, after Byun and Kuipers. ⁸¹

are 1 meter away). The point where the feature values are maximized is the distinctive place. The hill-climbing algorithm is a very simple approach. The idea is that for most hills, if you always choose the next step which is the highest (you can't look ahead), then you will get to the top quickly. Hence, the robot always moves in the direction which appears to cause the most positive change in the measurement function.

Although developed independently of each other, reactive behaviors map nicely onto distinctive places and local control strategies, as shown in Fig. 9.6. Consider a robot navigating down a wide hall to a dead-end. The local control strategy is a behavior, such as `follow-hall`, which operates in conjunction with a releaser, `look-for-corner`. The releaser is an exteroceptive cue. When it is triggered, the robot is in the neighborhood of the distinctive place, and the released behavior, `hillclimb-to-corner(distance=1)`, directs the robot to 1 meter from each wall.

9.3.2 Advantages and disadvantages

The distinctive place concept eliminates any navigational errors at each node. The robot may drift off-course because the hall is wide, but as soon as it reaches the neighborhood, it self-corrects and localizes itself. Kuipers and Byun were able to show in simulation how a robot with

dead reckoning er-

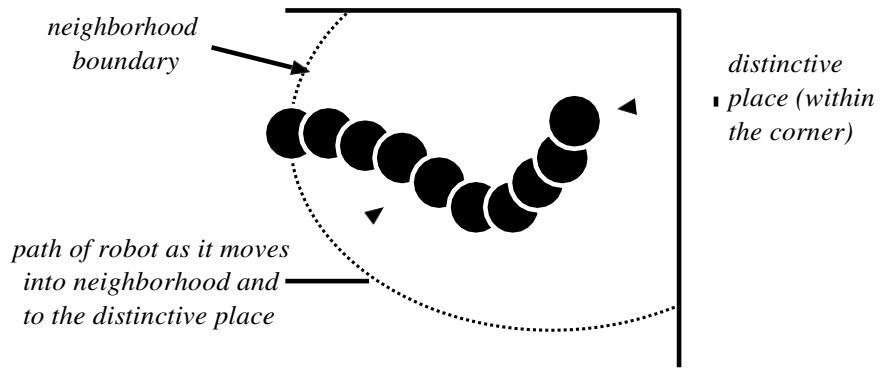


Figure 9.5 A robot reacting to a corner and moving to a distinctive place within the neighborhood.

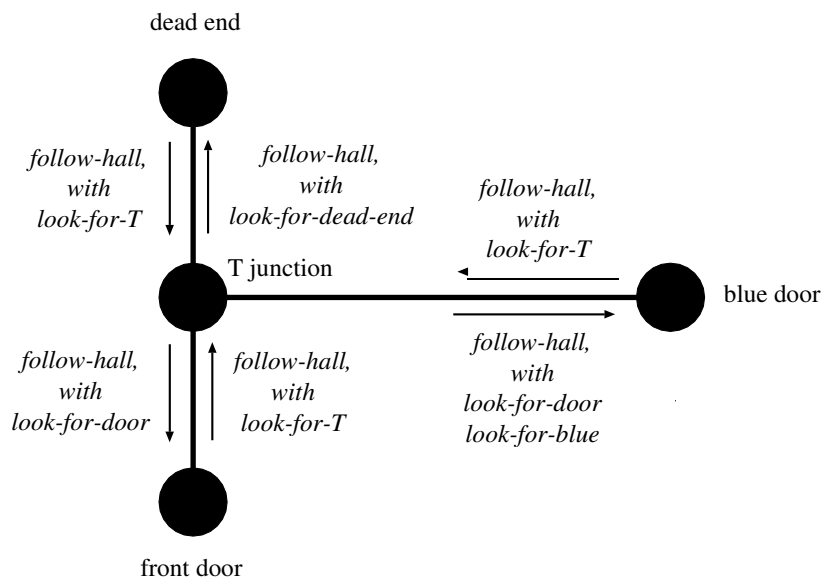


Figure 9.6 Behaviors serving as local control strategies, and releasers as means of signalling the entrance to a neighborhood.

rors could use multiple trips between nodes to build up a reasonable metric map, since most of the errors would average out. Another attractive aspect of the distinctive place approach is that it supports discovery of new landmarks as the robot explores an unknown environment. As long as the robot found something distinctive that it could reliably localize itself to, it could be put on a topological map. Then as it repeatedly moved to it, the robot could construct a metric map.

Returning to the discussion of landmarks, it should be noticed that a landmark must be unique to a node pair. There can't be any corners in the real world that are not on the graph between the nodes or else the robot will localize itself incorrectly.

The distinctive place approach as originally formulated encountered some problems when behavior-based roboticists began to apply it to real robots. One of the most challenging problems was perception. Good distinctive places are hard to come by; configurations that seemed useful to humans, like corners, proved difficult to reliably sense and localize against. Features that were easy to sense often were too numerous in the world, and so were not locally unique. Another challenge was learning the local control strategy. As the robot explored an unknown environment, it was easy to imagine that it could find distinctive places. But how did it learn the appropriate local control strategy? In an indoor environment, the robot might resort to always using wall following, even though other behaviors would be better suited. How would it ever try something different? Another open issue is the problem of indistinguishable locations. The issue of indistinguishable locations has also been tackled to some degree by work with probabilistic methods, which will be covered in Ch. 11.

9.4 Associative Methods

ASSOCIATIVE METHODS

PERCEPTUAL DISTINGUISHABILITY

PERCEPTUAL STABILITY

Associative methods for topological navigation essentially create a behavior which converts sensor observations into the direction to go to reach a particular landmark. The underlying assumption is that a location or landmark of interest for navigation usually has two attributes:

1. *perceptual stability*: that views of the location that are close together should look similar
2. *perceptual distinguishability*: that views far away should look different.

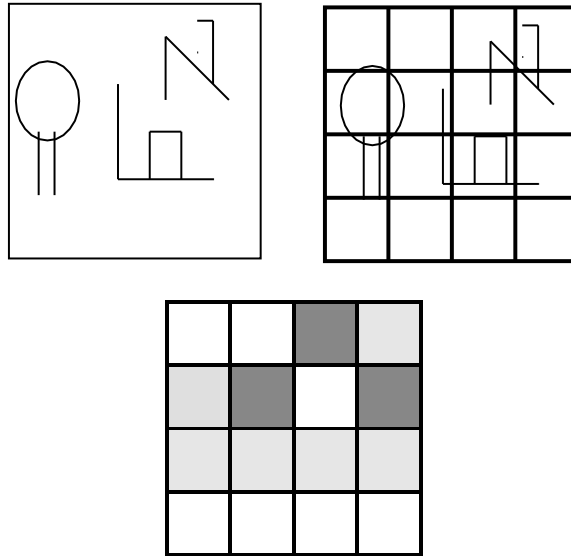


Figure 9.7 Derivation of an image signature. Clockwise from the upper left: a raw image, the image partitioned into 16 sections, and the image with the sections values shown as crosshatching.

These principles are implicit in the idea of a neighborhood around a distinctive place. If the robot is in the neighborhood, the views of the landmark look similar. The main difference is that associative methods use very coarse computer vision.

9.4.1 Visual homing

Work done by Nelson,¹¹¹ and later by Engelson,⁵⁰ relied on an image signature. An image signature is created by partitioning an image into sections. Fig. 9.7 shows an image partitioned into 16 sections. Next, each section is examined and a measurement is made of some attribute in the section. Some possible measurements are the edge density (the number of pixels with edges divided by the number of pixels in the section), dominant edge orientation (the angle the majority of the edges are pointing), average intensity, and so on. The image signature reduces the image to n measurements. One way to think of this is that it changes the resolution of the image into 16 pixels.

Notice that the image signature in the figure forms a pattern. If the robot

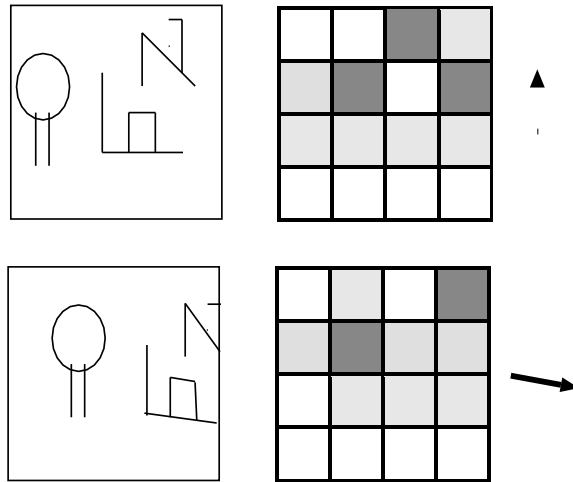


Figure 9.8 Image measurements and the appropriate response from two different views.

is in the neighborhood of the location, then the image measurements should be approximately the same pattern as the image signature, only the pattern may be offset due to the not-quite-at-the-right-place viewpoint.

If the robot can identify the image signature, or portion of it, in the current image, it will then know whether to turn left or right to localize itself relative to the location. The use of image signatures to direct a robot to a specific location is called visual homing. The inspiration for visual homing came from Nelson's research into how bees navigate. It is easy to speculate that baby bees are learning the image signatures of their hive as they execute the zoom in/out behavior described in Ch. 3. In that case, the compound eyes serve as de facto partitions of what to humans would be a single image.

9.4.2 QualNav

Levitt and Lawton took the ideas of neighborhoods and visual homing to an extreme for outdoor navigation over large distances as part of the Defense

VEHICLE (ALV) ^{9.4 Associative Methods} project in the late 1980's. ⁸⁵ At that time, topological navigation using relational graphs appeared promising for indoor environments, but seemed to resist application to outdoor terrains. Part of the challenge was the notion of

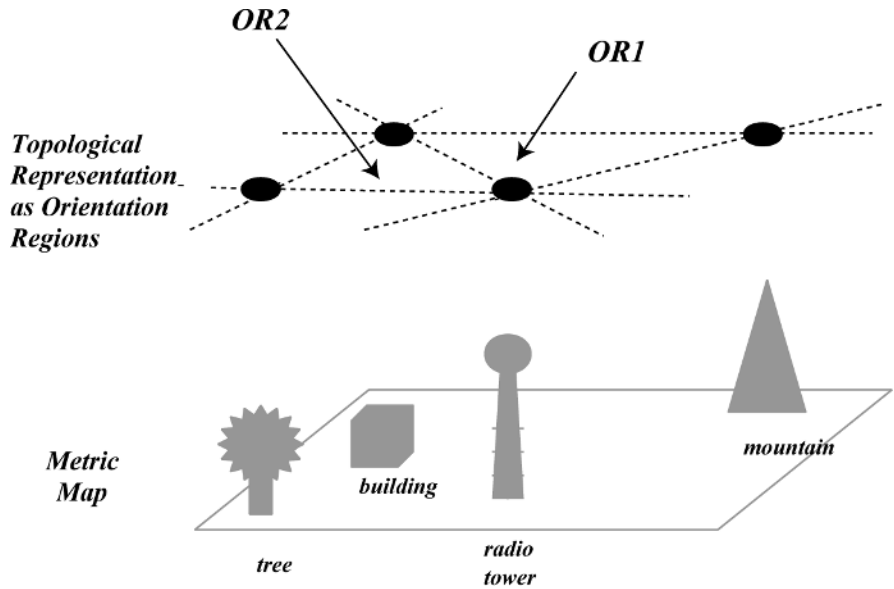


Figure 9.9 Division of an outdoor area into orientation regions, after Levitt and Lawton.⁸⁵

localization. With that challenge went the associated question of what could the robot use as a landmark. The land vehicle might be expected to travel for 50 miles deliberately avoiding obvious landmarks such as houses, road intersections, and radio towers. The idea was to have a robot vehicle capable of stealthily scouting an area and seeing if enemy forces were nearby. The robot vehicle has to provide some indication of where it found the enemy forces as well as return to base.

Inside of trying to localize the vehicle to a particular landmark, the idea behind *QualNav*⁸⁵ (an acronym for “qualitative navigation”) was to localize the vehicle to a particular *orientation region*, or a patch of the world. The orientation region is defined by *landmark pair boundaries*. A landmark pair

boundary is an imaginary line drawn between two landmarks. As seen in the figure below, drawing lines between all the candidate landmark pairs will partition the world into orientation regions, although the orientation regions may be very oddly shaped.

An orientation region is conceptually similar to a neighborhood. Within an orientation region, all the landmarks appear in the same

QualNav
ORIENTATION REGION
LANDMARK PAIR

BOUNDARIES

the vehicle is in OR1, the landmarks are found in the same order if the robot always turns in the same direction (either always clockwise or always counter-clockwise): building-mountain-tower. If the vehicle is in OR2, the landmarks will be tree-building-intersection.

An interesting attribute of orientation regions and landmark pair boundaries is that the vehicle can directly perceive when it has entered a new orientation region. For example, as the robot moves from OR1 to OR2, the building-tower landmark pair boundary is in front of it, then is on both sides of the vehicle, and is finally behind it. The transition is a perceptual event, and does not require knowledge about the distance of the landmarks, just that the relationship of the landmarks to the robot has just changed.

The use of orientation regions allows the robot to create an outdoor topological map as it explores the world, or to localize itself (coarsely) to a metric map. The robot does not have to be able to estimate the range to any of the landmarks.

How the robot navigates inside an orientation region has the flavor of visual homing. The robot may need to retrace its path as precisely as possible through an orientation region. (A military robot might need to return without being seen.) Without knowing the range to the landmarks bounding the orientation region, the robot is helpless. But if it has to remember the angles to each landmark every n minutes, it can move to follow the angles. A set of angles remembered at a point along the path is called a viewframe.

One amusing aspect of the viewframe approach is that it assumed the robot had cameras literally in the back of its head. Unfortunately the ALV vehicle did not; all the sensors faced forward. Daryl Lawton tells the story of trying to convince the union driver of the vehicle to stop every 10 feet and do a 360 turn so he could record the viewframes. After much pleading, the driver finally honored the request, though it wasn't clear if he ever understood why the crazy scientist wanted a high-tech truck capable of autonomous navigation to go in circles every 10 feet!

Associative methods are interesting because of the tight coupling of sensing to homing. The image signature and viewframe concepts do not require the robot to recognize explicitly what a landmark is, only that it is perceptually stable and distinguishable for the region of interest. Unfortunately associative methods require massive storage and are brittle in the presence of a dynamic world where landmarks may be suddenly occluded or change. Of course, this is more problematic for

indoor environments than outdoor ones.



Figure 9.10 A student testing an unidentified entry in the competition arena at the 1994 AAAI Mobile Robot Competition “Office Delivery” event. (Photograph courtesy of AAAI.)

9.5 Case Study of Topological Navigation with a Hybrid Architecture

This section presents a case study of topological navigation using the SFX architecture in the 1994 AAAI Mobile Robot Competition by a team of undergraduates from the Colorado School of Mines. The 1994 competition had an office navigation event.¹²⁹ Each robot was placed in a random room, then had to navigate out of the room and to another room within 15 minutes. Entrants were given a topological map, but weren’t allowed to measure the layout of the rooms and halls. This case study reviews how the topological map was entered, the activities of the Cartographer, how scripts (discussed in Ch. 5) were used to simplify behavioral management, and the lessons learned.

9.5.1 Path planning

The topological map was entered as an ASCII file in Backus-Naur Form. A sample map is shown in Fig. 9.11. The input map consists of three node types: room (R), hall (H), and foyer (F). The world is assumed to be orthogonal. Since edges between nodes can be in only four directions, it is convenient to refer to them as north (N), south (S), east (E), and west (W), where N is set arbitrarily in the map. The robot is given its starting node but as an extra challenge, the robot is not given the direction it is initially facing relative to the map. The topological map is structurally correct, but does not necessarily represent if a corridor or door is blocked. Such blockages may occur or be moved at any time. An additional assumption is that the outside of each door is marked with a landmark such as a room number or room name.

The Cartographer in SFX is responsible for constructing the route. It takes as input a gateway type of topological map and a start node and goal node, and produces a list of nodes representing the best path between the start and goal. The cartographer operates in two steps: preprocessing of the map to support path planning, and path planning. The preprocessing step begins by building a database of the nodes in the input map, reclassifying the corridor nodes which represent a hall to door connection as Hd.

Once the start and goal nodes are known, the Cartographer eliminates extraneous gateways. A Hd node may be connected to a room which is not visitable, that is, it is neither the goal room, the start room, or a room with more than one door. If that occurs, then both the R and Hd node entries are eliminated from the database. A sample input graphical topological representation for a metric map is shown in Fig. 9.11. If R3 was selected as the start node and R7 the goal, the refined graphical representation is as shown in Fig. 9.11. Note that Hd3-R1 and Hd4-R6 were eliminated because they were not associated with the start or goal rooms and could not be used as a shortcut since they had only one entrance. Gateway nodes such as H10 remain in the path because they may be useful if a blocked path occurs. For example, if the robot is going down the hallway from H11 to H8 and the path is blocked, the robot will need to return to a known location in order to reorient itself and replan. However, if the H10 is eliminated, then the robot must return to the very beginning of the corridor, because it does not know where it is with respect to H10. To solve this dilemma of traveling the same part of the

corridor possibly three times, the cartographer maintains nodes which represent possible alternative strategies. Different types of gates, tasks, and robot capabilities will lead to different preprocessing strategies.

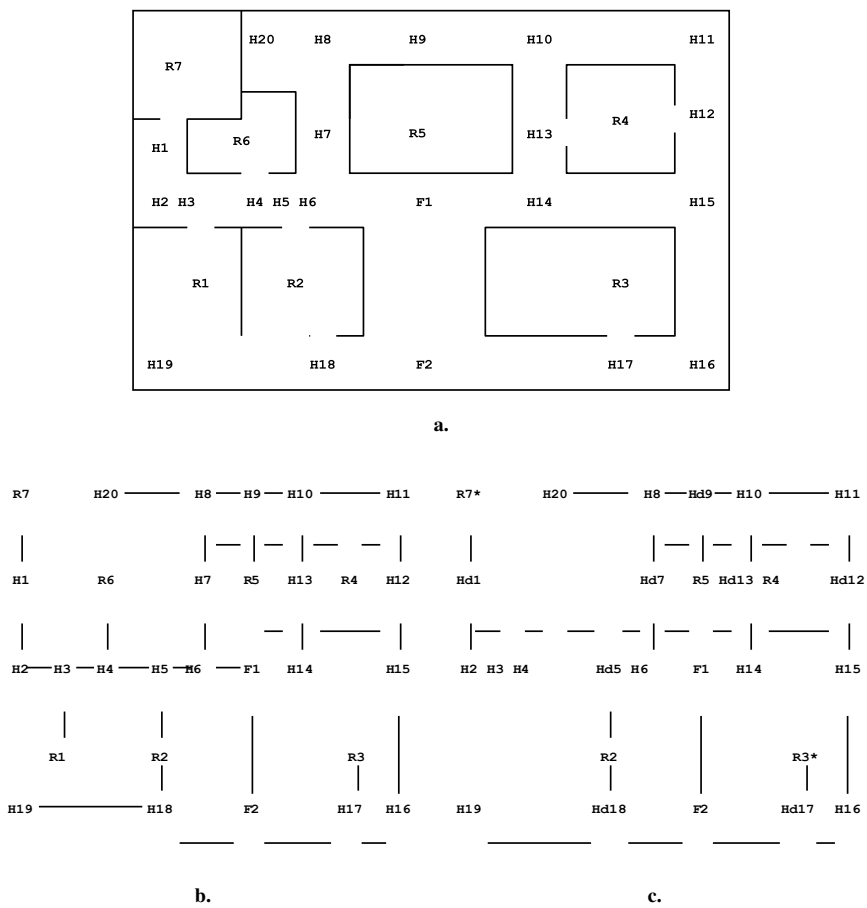


Figure 9.11 a.) Metric map of an office layout, b.) graphical topological representation, and c.) refined graph for traveling from R3 to R7.

An optimal path is computed using Dijkstra’s single source shortest path algorithm. The algorithm generates the shortest path by considering the costs between each pair of connected nodes; the costs are expressed as the length or weight of an edge in the graph. Since the topological representation is not metric, connections between the nodes reflect preferences in constructing routes. These preferences are expressed as edge weights. In this implementation, navigating through foyers was considerably more computationally expensive and unreliable; therefore the edge weighting for any path subsegment originating from a foyer

was set at 3. The edge weight for going from a room to another room⁹ *Topological Path Planning* was set at 2 to discourage the robot from finding solutions which were “impolite” (e.g., cutting through people’s offices). All other connections were set to 1.

TRANSITION TABLE

The Task Manager in SFX uses the path computed by the cartographer to select the appropriate *abstract navigation behavior* (Ch. 5) for traveling between a particular pair of nodes. It maintains a pointer to the current node and the next intended node from the path. The current and next node type determine the appropriate behavior according to the *transition table* shown below:

	To			
From	H	F	R	H
H	Navigate-hall	Navigate-hall	Undefined	Navigate-hall
F	Navigate-hall	Navigate-foyer	Navigate-door	Navigate-hall
R	Undefined	Navigate-door	Navigate-door	Navigate-door
Hd	Navigate-hall	Navigate-hall	Navigate-door	Navigate-hall

The transition table shows that not all combinations of nodes are permitted; by definition, the robot cannot move from a hall node H to a room node R without going through a Hd node. Also, the table is not necessarily symmetrical. In the case of rooms, navigate-door must be employed to either enter or exit, but the case of moving to a foyer will use different strategies depending on whether the robot is traversing a hall or a foyer. The ANB itself uses the information in the database entries for the nodes as parameters for instantiating the script to the current waypoint pair.

One novel aspect of this implementation is how the Task Manager handles a blocked path; it does not reverse the currently instantiated abstract navigation behavior (ANB). If the obstacle avoidance behavior posts to the whiteboard structure that a BLOCKED condition has occurred, the Task Manager terminates the currently active abstract navigation behavior. Because the robot is between nodes, the Task Manager directs the robot to return to the current node. But it triggers a simple move-to-goal behavior, which allows the robot to reorient itself more quickly than reinstantiating the abstract navigation behavior with new parameters.

Once the robot has returned to approximately the last known location, the Task Manager requests a new path from the Cartographer. The Cartographer removes the connection between the current and intended nodes, then recomputes a new path with the current node as the start. Once the new path is completed, the Task Manager resumes control.

Fig. 9.12 demonstrates the robot moving through the map handed out at the actual AAAI Mobile Robot Competition. The start node is R7 and the goal is R2. The cartographer computes the path as R7-H1-H2-H5-

342 R2, eliminating H3 and H4 because they are not relevant for this task. *9 Topological Path Planning*

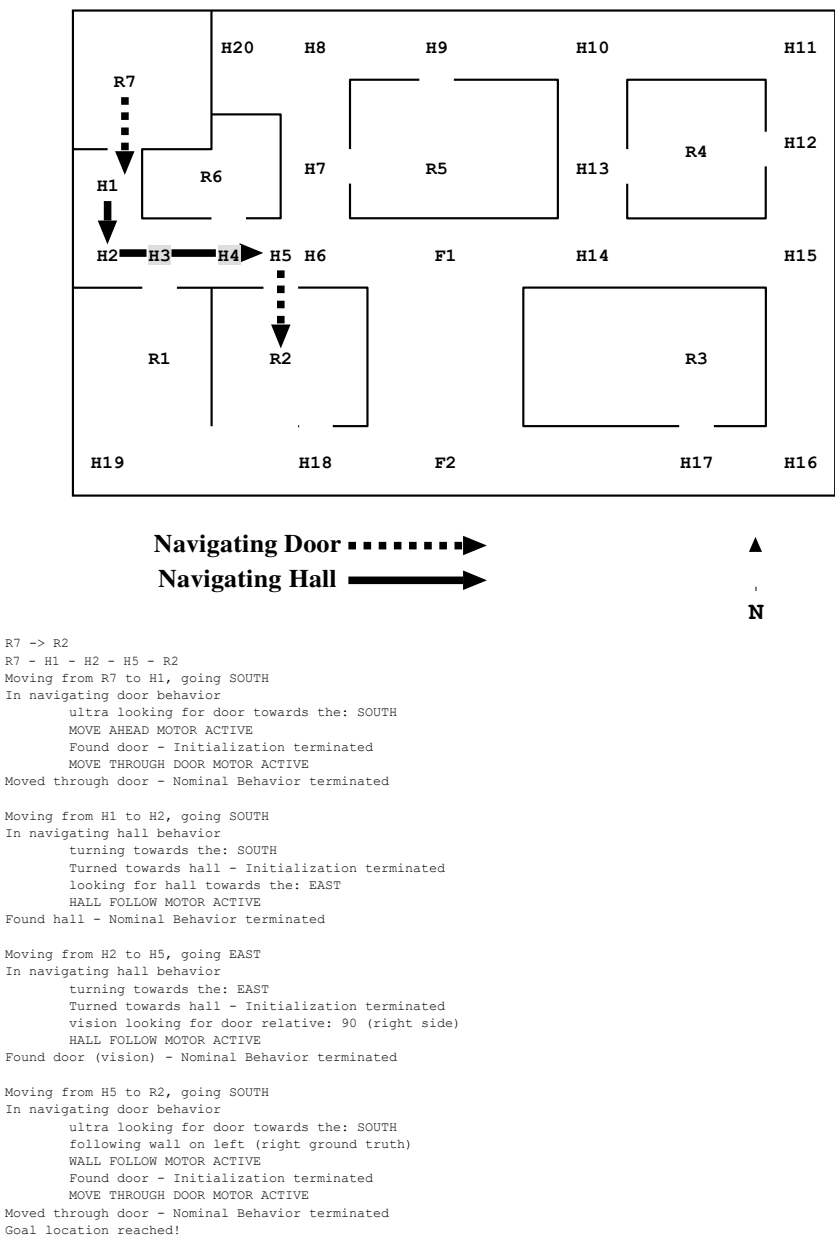


Figure 9.12 Scenario for moving from R7 to R2. Shaded gateways are extraneous and discarded by the planner.

As shown by the output, the Cartographer computes the path shown in the first line. The second line shows that the current navigational task is to move from R7 to H1. The Task Manager selects navigate-door and it begins by looking for the door since the user did not specify where it was. By examining the database, the door is known to be to the south. Therefore, the behavior initiates the move-ahead behavior in the south direction. When the door is found, the initialization phase of the abstract behavior is over and the nominal activity of moving through the door is triggered. Once the robot is through the door, the nominal behavior is terminated, terminating the entire abstract behavior as well.

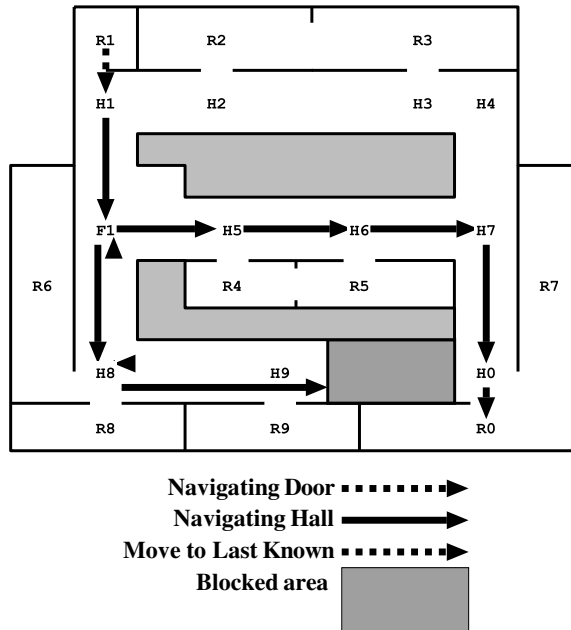
The next navigational task is to move from H1 to H2, again in the direction south. The task manager selects navigate-hall. The task of moving from H2 to H5 is interesting because it shows that the termination condition for hall following is different than for H1 to H2. Since H5 is a gateway to a room of interest, a different perceptual process is used to visually identify the room. Once the door has been identified visually, the robot is considered at H5. The H5-R2 connection instantiates navigate-door. However, in this case, the ultrasonics has not yet identified the door opening and so the initialization phase consists of wall following until the door opening is recognized. Then the nominal behavior is activated and the robot moves into the room, successfully completing the task.

The second simulation shown in Fig. 9.13 uses the sample topological map provided prior to the competition. The intended path was R1-H1-F1-H8-H0-R0, but H8-H0 was blocked. As shown in the figure, the robot detects that the path is blocked and uses move to goal to return to H8. The Cartographer updates the topological map and computes a new route and the robot resumes execution, in this case H8-F1-H5-H6-H7-H0-R0. The output of the script below shows the robot confirming the nodes H5 and H6 even though the plan did not have them enter the associated rooms. These nodes were kept in the path because if another blocked hall was encountered, the robot would then be able to return to either R5 or R6 and attempt to use a route through the door between.

9.5.2 Navigation scripts

The path planning and execution components are clearly deliberative.

The Cartographer is maintaining a map in the form of a graph and is monitoring progress. The transition table takes the role of a high level sequencer. Scripts



```

Moving from H8 to H0, going EAST
In navigating hall behavior
  turning towards the: EAST
  Turned towards hall - Initialization terminated
  HALL FOLLOW MOTOR ACTIVE
User stopped or BLOCKED - behavior terminated
In last known location loop
  MOVE TO GOAL MOTOR ACTIVE

Moving from H8 to F1, going NORTH
In navigating hall behavior
  turning towards the: NORTH
  Turned towards hall - Initialization terminated
  HALL FOLLOW MOTOR ACTIVE
  will follow wall on right (left ground truth) to stay in foyer
Found foyer - Nominal Behavior terminated

Moving from F1 to H5, going EAST
In navigating hall behavior
  following wall on left (right ground truth)
  trying to exit foyer
  looking for hall towards the: EAST
  WALL FOLLOW MOTOR ACTIVE
Found hall - Initialization terminated
  HALL FOLLOW MOTOR ACTIVE
Found door (vision) - Nominal Behavior terminated

```

Figure 9.13 Scenario for moving from R1 to R0 with blocked path.

are used to specify and carry out the implied details of the plan in a modular and reusable way.

The implementation consisted of three scripts for navigating doors, halls, and foyers. The pseudocode for `navigate-door` is shown below in C++ style and takes advantage of the indexing properties of the switch statement:

```
switch(door)
case door-not-found:
    //initialization phase
    //follow wall until find door
    if wall is found
        wallfollow to door
    else
        move-ahead to find a wall
case door-found:
    //nominal activity phase
    move-thru-door(door-location)
```

The nominal behavior, `move-thru-door`, is self-terminating, so there is no separate termination condition. The perception of the door is the key determinant in what the robot does.

The `navigate-hall` is used for traveling between corridors, foyers and corridors, and from corridors to corridor/door nodes. It has two different starting conditions. One condition is that the robot is in a foyer and detects a hall. Since the hall may be somewhat ahead and the `hallfollow` behavior assumes the robot is in the hall, the script has the robot employ `wallfollow` to find the hall, then begin following the hall. The task manager uses the directional information stored in the database to determine which wall of the foyer to follow. In the other condition the robot is already in a hall. Since the robot is not guaranteed to be facing the centerline of the hall (ANBs have no knowledge of what was done prior to their instantiation except via global variables), the sub-script turns it to line itself up with the desired hall.

```
switch(hall)
case not-facing-hall:
```

9.5 Case Study of Topological Navigation with a if starting in a FOYER	Hybrid Architecture	347

```

        if hall-not-found
            wallfollow until find the hall
        else
            if not facing hall
                turn to face hall
            else starting in a HALL
                if not facing hall
                    turn to face hall
        case facing-hall:
            //nominal activity phase
            hallfollow until next gateway

```

The `navigate-hall` ANB terminates when the next expected gateway (the next node in the path) is found. There are three behaviors which look for gateways. `hallwatch` looks for the ultrasonic signature of a hall in the expected direction; `foyerwatch` similarly looks for foyers, and the robot uses vision through the `confirm-door` behavior to detect the landmark associated with the room. These behaviors run concurrently with the nominal behaviors.

The `navigate-foyer` ANB is used to move the robot between two foyers. It assumes that two foyers' nodes connected are a convenient representation of a single large foyer with entries from different directions (i.e., multiple gateways into the foyer). The script moves the robot n feet into the first foyer in the direction of the second foyer. This gets the robot away from potentially confusing ultrasonic signatures. Then the task manager determines which side of the foyer to `wallfollow` until the next expected gateway is detected. There is no case statement in the pseudocode since the sequence of activities is fixed.

```

//step 1
move-to-goal(n, dir) in direction of next foyer
//step 2
wallfollow until next gateway is detected

```

9.5.3 Lessons learned

The CSM team did not place at the AAI competition; the robot suffered a series of hardware failures, causing both the power supply and the sonars to

fail. The robot was eventually fixed and the software worked fine. The tests before and after the competition offered several practical lessons.

First, it is critical to build the abstract navigation behaviors out of robust primitives. The biggest problem with the implementation has not been with the scripts per se, but rather with the quality of the primitive behaviors. If the robot cannot reliably detect an open doorway while it is following a wall around a room, it will never exit no matter what the script says. Second, contention for sensing resources is an emerging issue for robotic control schemes. The video camera on the robot did not have a pan mechanism; essentially, Clementine's directional control is the camera effector. In one instantiation of the `navigate-hall` behavior between a `H` node and a `Hd` node, the nominal behavior of `hallfollow` frequently pointed Clementine and her camera away from the anticipated location of the door, interfering with the `confirm-door` perceptual behavior used to terminate `navigate-hall`. Even at slow speeds, Clementine frequently missed the door. One alternative is to let the two active behaviors take turns with control of the robot. This leads to a move, stop, sense scenario which slowed the robot's progress and still allowed the robot to roll past the door.

Note that the representation and path planning algorithm support the addition of metric information. The metric distance between every visited could be stored in the database. The distance could be attached to the reference index in the node class, much the same as the nodes connected to the `N`, `E`, `S`, `W` are stored. One problem with simply attaching the metric distance is deciding what distance value to use. If `wallfollow` takes the robot around the perimeter of a foyer in order to exit, the distance traveled will not reflect the straight line distance between nodes. Nor is the measure bidirectional; going `N-S` through the foyer could be significantly longer than `S-N` if the foyer is asymmetric and the robot follows different walls each time. On the other hand, if the width of a foyer were known, the robot might possibly be able to use dead reckoning to move directly across it to the desired exit. The impact of obstacles lengthening the distance traveled can also be a problem, especially if the obstacles are people moving down a hall. A short hall can take a long time to navigate if it is cluttered during one visit, but a short time during another. Another difficulty with adding metric distances is how to use it effectively. Metric distances may not be known for all pairs of nodes, making it difficult to apply Dijkstra's algorithm. Likewise, distance may be only one factor in selecting a route; our current

9.5 Case Study of Topological Navigation with a Hybrid Architecture 350
implementation prefers going through halls versus taking short cuts between rooms. Utility theory can be used to quantify the impact of these competing concerns.

9.6 Summary

Landmarks simplify the *where am I?* problem by providing orientation cues. Gateways are a special case of landmarks which reflect a potential for the robot to change directions (turn down a different road or hall, enter a room, etc.). There are two categories of qualitative navigation methods: *relational* and *associative*. Relational methods relate *distinctive places* (nodes) to each other by the *local control strategies (lcs)*, or behaviors, needed to travel between them (edges), forming a graph. The robot can use the graph to plan a path using techniques such as the single source shortest path algorithm. It executes the path by employing the behavior associated with that edge it is traversing. When it sees the landmark of the location, it is in the neighborhood, and then can use another behavior, such as hill-climbing, to localize itself relative to the landmark. Associative methods directly couple perception with acting. An image can be compared to a image signature or a viewframe to generate the next movement for the robot to take. Relational methods are commonly used for topological navigation.

9

Metric Path Planning

Chapter objectives:

Define *Cspace*, *path relaxation*, *digitization bias*, *subgoal obsession*, *termination condition*.

Explain the difference between graph and wavefront planners.

Represent an indoor environment with a *generalized Voronoi graph*, a *regular grid*, or a *quadtrees*, and create a graph suitable for path planning.

Apply the A* search algorithm to a graph to find the optimal path between two locations.

Explain the differences between continuous and event-driven replanning.

10.1 Objectives and Overview

QUANTITATIVE	<i>Metric path planning, or quantitative navigation, is the opposite of topologi-</i>
NAVIGATION	<i>cal navigation. As with topological methods, the objective is to determine a path to a specified goal. The major philosophical difference is that metric methods generally favor techniques which produce an optimal, according to some measure of best, while qualitative methods seem content to produce a route with identifiable landmarks or gateways. Another difference is that</i>
WAYPOINTS	<i>metric paths are usually decompose the path into subgoals consisting of way- points. These waypoints are most often a fixed location or coordinate (x,y). These locations may have a mathematical meaning, but as will be seen with meadow maps, they may not have a sensible correspondence to objects or</i>

landmarks in the world. Topological navigation focused on subgoals which are gateways or locations where the robot could change its primary heading. The terms “optimal” and “best” have serious ramifications for robotics. In order to say a path is optimal, there is an implied comparison. As will be seen, some metric methods are able to produce an optimal path because they consider all possible paths between points. This can be computationally expensive. Fortunately, some algorithms (especially one named “A*” for reasons that will be discussed later) are more clever about rejecting non-optimal paths sooner than others.

Surprisingly, an optimal path may not appear optimal to the human eye; for example, a mathematically optimal path of a world divided into tiles or grids may be very jagged and irregular rather than straight. The ability to produce and compare all possible paths also assumes that the planning has access to a pre-existing (or *a priori*) map of the world. Equally as important, it assumes that the map is accurate and up to date. As such, metric methods are compatible with deliberation, while qualitative methods work well with more reactive systems. As a deliberative function, metric methods tend to be plagued by the same sorts of difficulties that were seen in Hierarchical systems: challenges in world representation, handling dynamic changes and surprises, and computation complexity.

COMPONENTS OF
METRIC PATH
PLANNERS

Metric path planners have two components: the *representation* (data structure) and the *algorithm*. Path planners first partition the world into a structure amenable for path planning. They use a variety of techniques to represent the world; no one technique is dominant, although regular grids appear to be popular. The intent of any representation is to represent only the salient features, or the relevant configuration of navigationally relevant objects in the space of interest; hence the term configuration space. Path planning algorithms generally work on almost any configuration space representation, although as with any algorithm, some methods work better on certain data structures. The algorithms fall into two broad categories: those which treat path planning as a graph search problem, and those which treat path planning as a graphics coloring problem. Regardless of what algorithm is used, there is always the issue in a Hybrid architecture of when to use it. This is sometimes called the issue of interleaving reaction and planning.

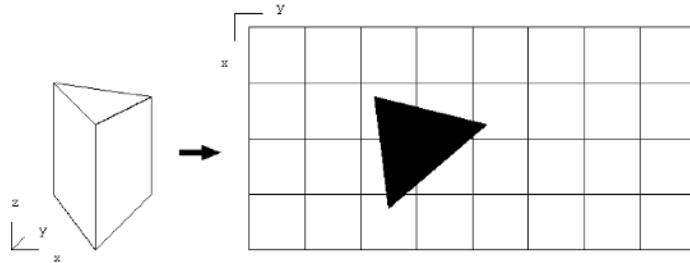


Figure 10.1 Reduction of a 6DOF world space to a 2DOF configuration space.

10.2 Configuration Space

CONFIGURATION SPACE

The physical space robots and obstacles exist in can be thought of as the world space. The *configuration space*, or *Cspace* for short, is a data structure which allows the robot to specify the position (location and orientation) of any objects and the robot.

DEGREES OF FREEDOM

A good Cspace representation reduces the number of dimensions that a planner has to contend with. Consider that it takes six dimensions (also called *degrees of freedom* or DOF) to represent precisely where an object is. A person may specify the location of the object as a $(x; y; z)$ coordinate in some frame of reference. But an object is three-dimensional; it has a front and back, top and bottom. Three more degrees are needed to represent where the front of the chair is facing, whether it is tilted or not, or even upside down. Those are the Euler (pronounced “Oiler”) angles, $\phi; \theta; \psi$, also known as pitch, yaw, and roll.

Six degrees of freedom is more than is needed for a mobile ground robot in most cases for planning a path. The z (height) coordinate can be eliminated if every object the robot sits on the floor. However, the z coordinate will be of interest if the robot is an aerial or underwater vehicle. Likewise, the Euler angles may be unnecessary. Who cares which way the robot is facing if all the robot wants to do is to plan a path around it? But the pitch of a planetary rover or slope of an upcoming hill may be critical to a mission over rocky terrain.

Fig. 10.1 shows a transformation of an object into Cspace. In general, metric path planning algorithms for mobile robots have assumed only two DOF, including for the robot. For path planning purposes, the robot can be modeled as round so that the orientation doesn’t matter. This

implicitly assumes

HOLONOMIC

the robot is *holonomic*, or can turn in place. Many research robots are sufficiently close to holonomic to meet this assumption. However, robots made from radio-controlled cars or golf carts are most certainly not holonomic. Much work is being done in non-holonomic path planning, where the pose of the robot must be considered (i.e., can it actually turn that sharply without colliding?), but there no one algorithm appears to be used in practice. This chapter will restrict itself to holonomic methods.

10.3 Cspace Representations

The number of different Cspace representations is too large to include more than a coarse sampling here. The most representative ones are Voronoi diagrams, regular grids, quadtrees (and their 3D extension, octrees), vertex graphs, and hybrid free space/vertex graphs. The representations offer different ways of partitioning free space. Any open space not occupied by an object (a wall, a chair, an un-navigable hill) is free space, where the robot is free to move without hitting anything modeled. Each partition can be labeled with additional information: “the terrain is rocky,” “this is off-limits from 9am to 5am,” etc.

10.3.1 Meadow maps

MEADOW MAP

Many early path planning algorithms developed for mobile robots assumed that the robot would have a highly accurate map of the world in advance. This map could be digitized in some manner, and then the robot could apply various algorithms to convert the map to a suitable Cspace representation. An example of a Cspace representation that might be used with an *a priori*

map is the *meadow map* or hybrid vertex-graph free-space model.

Meadow maps transform free space into convex polygons. Convex polygons have an important property: if the robot starts on the perimeter and goes in a straight line to any other point on the perimeter, it will not go out of the polygon. The polygon represents a safe region for the robot to traverse. The path planning problem becomes a matter of picking the best series of polygons to transit through. Meadow maps are not that common in robotics, but serve to illustrate the principles of effecting a configuration space and then planning a path over it.

The programming steps are straightforward. First, the planner begins

with a metric layout of the world space. Most planners will increase the size of every object by the size of the robot as shown in Fig. 10.2; this allows the

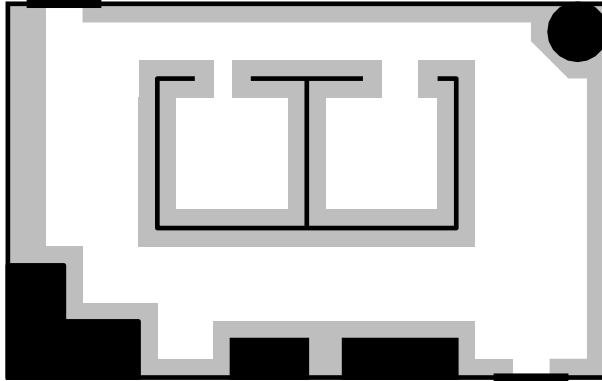


Figure 10.2 A *a priori* map, with object boundaries “grown” to the width of the robot (shown in gray).

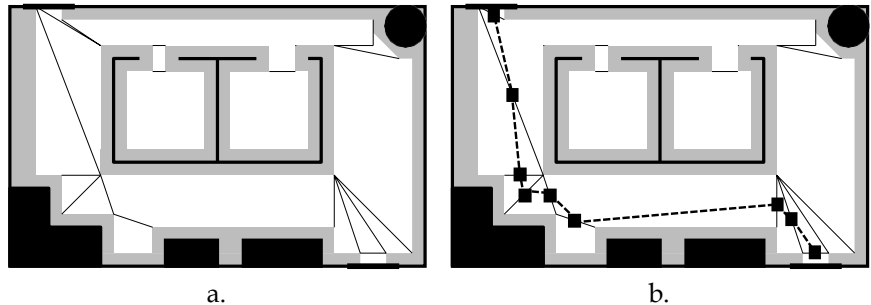


Figure 10.3 Meadow maps: a.) partitioning of free space into convex polygons, and b.) generating a graph using the midpoints.

planner to treat the robot as if it were a point, not a 2D object. Notice how from the very first step, path planners assume holonomic vehicles.

The next step in the algorithm is to construct convex polygons by considering the line segments between pairs of some interesting feature. In the case of indoor maps, these are usually corners, doorways, boundaries of objects, etc. The algorithm can then determine which combination of line segments partitions the free space into convex polygons.

The meadow map is now technically complete, but it is not in a format that supports path planning. Each convex polygon represents a safe passage for the robot. But there is still some work to be done. Some of the

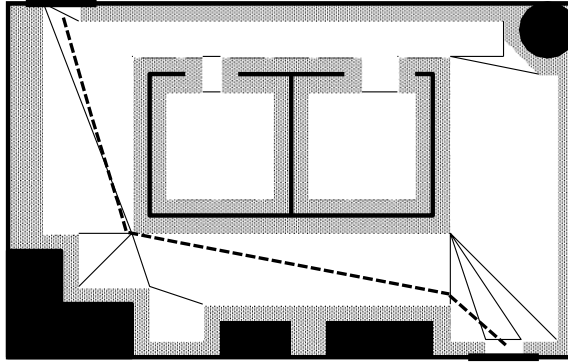


Figure 10.4 String tightening as a relaxation of an initial path.

forming the perimeter aren't connected to another a polygon (i.e., they are part of a wall), so they should be off limits to the planning algorithm. Also, as can be seen by the above figure, some of the line segments are quite long. It would make a difference to the overall path length where the robot cuts across the polygon. It is hard for the planner to discretize a continuous line segment. So the issue becomes how to specify candidate points on the polygon. One solution is to find the middle of each line segment which borders another polygon. Note that each of these midpoints becomes a node, and if edges are drawn between them, an undirected graph emerges. A path planning algorithm would determine the best path to take.

One disadvantage of a meadow map, indeed of any Cspace representation, is evident on inspection of Fig. 10.3: any path which is chosen will be somewhat jagged. Each of the inflection points is essentially a waypoint. One outcome of the partitioning process is that the free space is divided up in a way that makes sense geometrically, but not necessarily for a robot to actually travel. Why go halfway down the hall, then angle off? This may be mathematically optimal on paper, but in practice, it seems downright silly. Chuck Thorpe at CMU devised a solution to paths generated from any kind of discretization of free space.¹³⁸ Imagine that the path is a string. Now imagine pulling on both ends to tighten the string (the technical name for this is *path relaxation*) This would remove most of the kinks from the path without violating the safe zone property of convex polygons.

Meadow maps have three problems which limit their usefulness. One problem is that the technique to generate the polygons is computationally

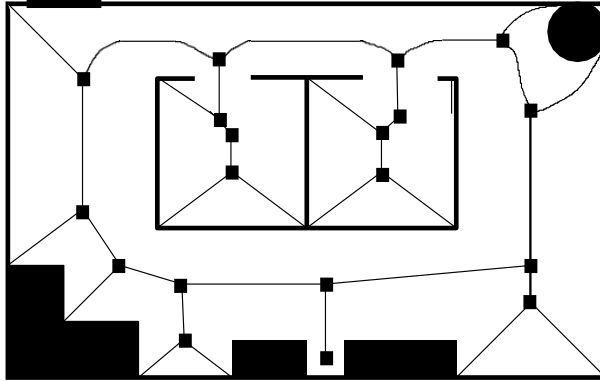


Figure 10.5 Graph from a generalized Voronoi graph (GVG). (Graph courtesy of Howie Choset.)

complex. But more importantly, it uses artifacts of the map to determine the polygon boundaries, rather than things which can be sensed. Unless the robot has accurate localization, how does it know it is halfway down a long, featureless hall and should turn 30°? The third major disadvantage is that it is unclear how to update or repair the diagrams as the robot discovers discrepancies between the *a priori* map and the real world.

10.3.2 Generalized Voronoi graphs

GENERALIZED *Generalized Voronoi Graphs*, or GVGs, are a popular mechanism for represent-

VORONOI GRAPHS ing Cspace and generating a graph. Unlike a meadow map, a GVG can be constructed as the robot enters a new environment, thereby creating a topological map as shown by Howie Choset at CMU.³⁶

VORONOI EDGE The basic idea of a GVG is to generate a line, called a *Voronoi edge*, equidistant from all points. As seen in Fig. 10.5, this line goes down the middle of hallways and openings. The point where many Voronoi edges meet is known as a *Voronoi vertex*. Notice the vertices often have a physical correspondence to configurations that can be sensed in the environment. This makes it much easier for a robot to follow a path generated from a GVG, since there is an implicit local control strategy of staying equidistant from all obstacles.

If the robot follows the Voronoi edge, it will not collide with any

modeled obstacles, because it is staying “in the middle.” This obviates the need to grow the obstacle boundaries. Edges serve as freeways or major thorough-

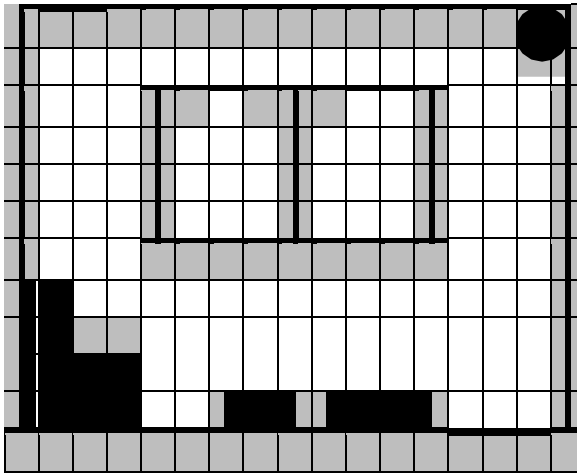


Figure 10.6 Regular grid.

fares. It should also be noted that the curved edges in a GVG do not matter to graph theory or graph algorithms. It is only the length, not the physical reality, of the edges that make any difference.

10.3.3 Regular grids

Another method of partitioning the world space is a regular grid. The regular grid method superimposes a 2D Cartesian grid on the world space, as shown in Fig. 10.6. If there is any object in the area contained by a grid element, that element is marked occupied. Hence, regular grids are often referred to as occupancy grids. Occupancy grids will be detailed in Ch. 11.

Regular grids are straightforward to apply. The center of each element in the grid can become a node, leading to a highly connected graph. Grids

are either considered *4-connected* or *8-connected*, depending on whether they permit an arc to be drawn diagonally between nodes or not.

Unfortunately, regular grids are not without problems. First, they introduce *digitization bias*, which means that if an object falls into even the smallest portion of a grid element, the whole element is marked

4-CONNECTED
NEIGHBORS
8-CONNECTED
NEIGHBORS
DIGITIZATION BIAS

occupied. This leads to wasted space and leads to very jagged objects. To reduce the wasted space, regular grids for an indoor room are often finely grained, on the order of 4- to 6-inches square. This fine granularity means a high storage cost, and a high number of nodes for a path planning algorithm to consider.

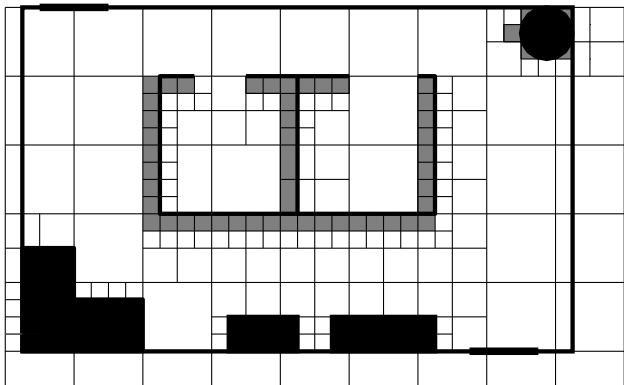


Figure 10.7 Quadtree Cspace representation.

10.3.4 Quadtrees

QUADTREES

A variant on regular grids that attempts to avoid wasted space is *quadtrees*. Quadtrees are a recursive grid. The representation starts out with grid elements representing a large area, perhaps 64-inches square (8 by 8 inches). If an object falls into part of the grid, but not all of it, the Cspace algorithm divides the element into four (hence the name “quad”) smaller grids, each 16-inches square. If the object doesn’t fill a particular sub-element, the algorithm does another recursive division of that element into four more sub-elements, represented a 4-inches square region. A three dimensional quadtree is called an octree.

10.4 Graph Based Planners

INITIAL NODE
GOAL NODE

As seen in the previous section, most Cspace representations can be converted to graphs. This means that the path between the *initial node* and the *goal node* can be computed using graph search algorithms. Graph search algorithms appear in networks and routing problems, so they form a class of algorithms well understood by computer science. However, many of those algorithms require the program to visit each node on the graph to determine the shortest path between the initial and goal nodes. Visiting every node may be computationally tractable for a sparsely connected graph such as derived from a Voronoi diagram, but rapidly becomes computationally expensive for a highly connected graph such as from a regular grid. Therefore, there has

been a great deal of interest in path planners which do a “branch and bound” style of search; that is, ones which prune off paths which aren’t optimal. Of course, the trick is knowing when to prune!

The A* search algorithm is the classic method for computing optimal paths for holonomic robots. It is derived from the A search method. In order to explain how A* works, A search will be first presented with an example, then A*. Both assume a metric map, where the location of each node is known in absolute coordinates, and the graph edges represent whether it is possible to traverse between those nodes.

The A search algorithm produces an optimal path by starting at the initial node and then working through the graph to the goal node. It generates the optimal path incrementally; each update, it considers the nodes that could be added to the path and picks the best one. It picks the “right” node to add to the path every time it expands the path (the “right node” is more formally known as the plausible move). The heart of the method is the formula (or evaluation function) for measuring the plausibility of a node:

$$f(n) = g(n) + h(n)$$

where:

$f(n)$ measures how good the move to node n is

$g(n)$ measures the cost of getting to node n from the initial node. Since A expands from the initial node outward, this is just the distance of the path generated so far plus the distance of the edge to node n

$h(n)$ is the cheapest cost of getting from n to goal

Consider how the formula is used in the example below. Assume that a Cspace representation yielded the graph in Fig. 10.8.

The A search algorithm begins at node A , and creates a decision tree-like structure to determine which are the possible nodes it can consider adding to its path. There are only two nodes to choose from: B and C .

In order to determine which node is the right node to add, the A search algorithm evaluates the plausibility of B and C by looking at the edges. The plausibility of B as the next move is:

$$f(B) = g(B) + h(B) = 1 + 2.24 = 3.24$$

where $g(B)$ is the cost of going from A to B , and $h(B)$ is the cost to get from

B to the goal E .

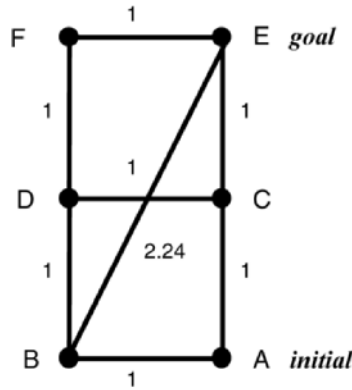


Figure 10.8 Graph for A search algorithm.

The plausibility of C is:

$$f(C) = g(C) + h(C) = 1 + 1 = 2:0$$

where $g(C)$ is the cost of going from A to C , and $h(C)$ is the cost of getting from C to E . Since $f(C) > f(B)$, the path should go from A to C .

But this assumes that $h(n)$ was known at every node. This meant that the algorithm had to recurse in order to find the correct value of $h(n)$. This leads to visiting all the nodes.

A search is guaranteed to produce the optimal path because it compares all possible paths to each other. A* search takes an interesting approach to reducing the amount of paths that have to be generated and compared: it compares the possible paths to the best possible path, even if there isn't a path in the real world that goes that way. The algorithm estimates h rather than checks to see if there is actually a path segment that can get to the goal in that distance. The estimate can then be used to determine which nodes are the most promising, and which paths have no chance of reaching the goal better than other candidates and should be pruned from the search.

Under A* the evaluation function becomes:

$$f(n) = g(n) + h(n)$$

where the means that the functions are estimates of the values that would

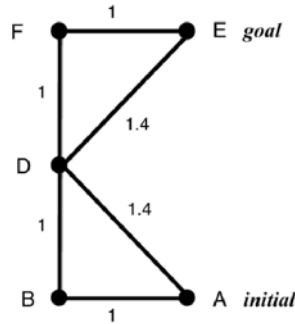


Figure 10.9 An A* example.

have been plugged into the A search evaluation. In path planning, $g(n)$ is the same as $g(n)$: the cost of getting from the initial node to n , which is known through the incremental build of the path. $h(n)$ is the real difference. So what is a way to estimate the cost of going from n to the goal? Furthermore, how can we be sure that the estimate will be accurate enough that we don't end up choosing a path which isn't truly optimal? This can be done by making sure that $h(n)$ will never be smaller than $h(n)$. The restriction that $h(n)$ is called the *admissibility condition*. Since $h(n)$ is an estimate, it is also called a *heuristic function*, since it uses a rule of thumb to decide which is the best node to consider.

ADMISSIBILITY

CONDITION
HEURISTIC FUNCTION

Fortunately, there is a natural heuristic function for estimating the cost from n to the goal: the Euclidean (straight line) distance. Recall that the locations of each node are known independently of the edges. Therefore, it is straightforward to compute the straight line distance between two nodes. The straight line distance is always the shortest path between two points, barring curvature of the earth, etc. Since the real path can never be shorter than that, the admissibility condition is satisfied.

To see how A* uses this to actually eliminate visiting nodes, consider the example in Fig. 10.9. As with A search, the first step in A* is to consider the choices from the initial node.

The choices are B and D , which can be thought of as a search tree (see

Fig. 10.10a) or as a subset of the original graph (see Fig 10.10b).
 Regardless

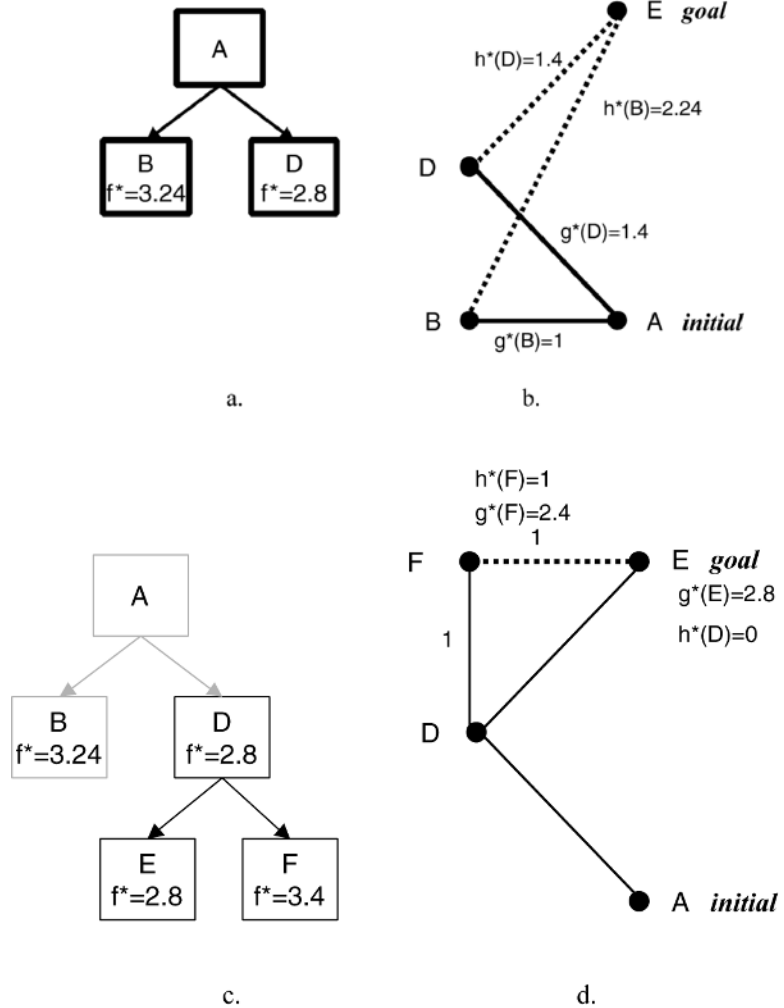


Figure 10.10 a.) What A* “sees” as it considers a path A-?-E, and b.) the original graph. c.) What A* “sees” from considering a path A-D-?-E, and d.) the associated graph.

of how it is visualized, each node is evaluated to determine which is the most plausible move. The above figure shows what the algorithm “sees” at this point in the execution. The choices evaluate to:

$$f(B) = g(B) + h(B) = 1 + 2 \cdot 24 = 3 \cdot 24$$

$$f(D) = g(D) + h(D) = 1 \cdot 4 + 1 \cdot 4 = 2 \cdot 8$$

A path going from $A \rightarrow D \rightarrow E$ has the potential to be shorter than a path going from $A \rightarrow B \rightarrow E$. So, D is the most plausible node. Notice that A^* can't eliminate a path through B because the algorithm can't “see” a path that actually goes from D to E and determine if it is indeed as short as possible.

At step 2, A^* recurses (repeats the evaluation) from D , since D is the most plausible, as shown in Fig. 10.10.

The two options from D are E and F , which are evaluated next:

$$f(E) = g(E) + h(E) = 2 \cdot 8 + 0 = 2 \cdot 8$$

$$f(F) = g(F) + h(F) = 2 \cdot 4 + 1 \cdot 0 = 3 \cdot 4$$

Now the algorithm sees that E is the best choice among the leaves of the search tree, including the branch through B . (If B was the best, then the algorithm would have changed branches.) It is better than F and B . When it goes to expand E , it notices that E is the goal, so the algorithm is complete. The optimal path is $A \rightarrow D \rightarrow E$, and we didn't have to explicitly consider $A \rightarrow B \rightarrow F \rightarrow E$. There are other ways to improve the procedure described so far. $f(F)$ didn't need to be computed if the algorithm looks at its choices and sees that one of them is the goal. Any other choice has to be longer because edges aren't allowed to be negative, so $D \rightarrow F \rightarrow E$ has to be longer than $D \rightarrow E$.

Another important insight is that any path between A and E has to go through D , so the B branch of the search tree could have been pruned. Of course, in the above example the algorithm never had an opportunity to notice this because B was never expanded. It's easy to imagine in a larger graph that there might be a case where after several expansions through D , the leaf at B in the search tree came up the most plausible. Then the algorithm would have expanded A and seen the choices were D . Since D already occurred in another branch, with a

cheaper $g(D)$, the B branch could be safely pruned.

This is particularly useful when A* is applied to a graph created from a regular grid, where the resulting graph is highly connected.

One very attractive feature of the A* path planner is that it can be used with any Cspace representation that can be transformed into a graph. The major impact that Cspace has on the A* planner is how many computations it takes to find the path.

A limitation of A* is that it is very hard to use for path planning where there are factors other than distance to consider in generating the path. For example, the straight line distance may be over rocky terrain or sand that poses a risk to the robot. Likewise, the robot may wish to avoid going over hills in order to conserve energy, but at the same time might wish to go down hills whenever possible for the same reason. In order to factor in the impact of terrain on the path costs, the heuristic function $h(n)$ has to be changed. But recall that the new heuristic function must continue to satisfy the admissibility condition: $h \leq h^*$. If the new h just takes the worst case energy cost or safety cost, it will be admissible, but not particularly useful in pruning paths. Also, gaining energy going downhill is essentially having an edge in the graph with a negative weight, which A* can't handle. (Negative weights pose an interesting problem in that the robot can get into a loop of rolling down a hill repeatedly because it's an energy-efficient solution rather than actually making forward progress! Bellman-Ford types of algorithms deal with this situation.)

10.5 Wavefront Based Planners

Wavefront propagation styles of planners are well suited for grid types of representations. The basic principle is that a wavefront considers the Cspace to be a conductive material with heat radiating out from the initial node to the goal node. Eventually the heat will spread and reach the goal, if there is a way, as shown in the sequence in Fig. 10.11. Another analogy for wavefront planners is region coloring in graphics, where the color spreads out to neighboring pixels. An interesting aspect of wavefront propagation is that the optimal path from all grid elements to the goal can be computed as a side effect. The result is a map which looks like a potential field. One of the many wavefront types of path planners is the Trulla algorithm developed by Ken Hughes.¹⁰⁶ It exploits the similarities with a potential field to let the path itself represent what the

364 robot should do as if the path were a sensor observation. *10 Metric Path Planning*

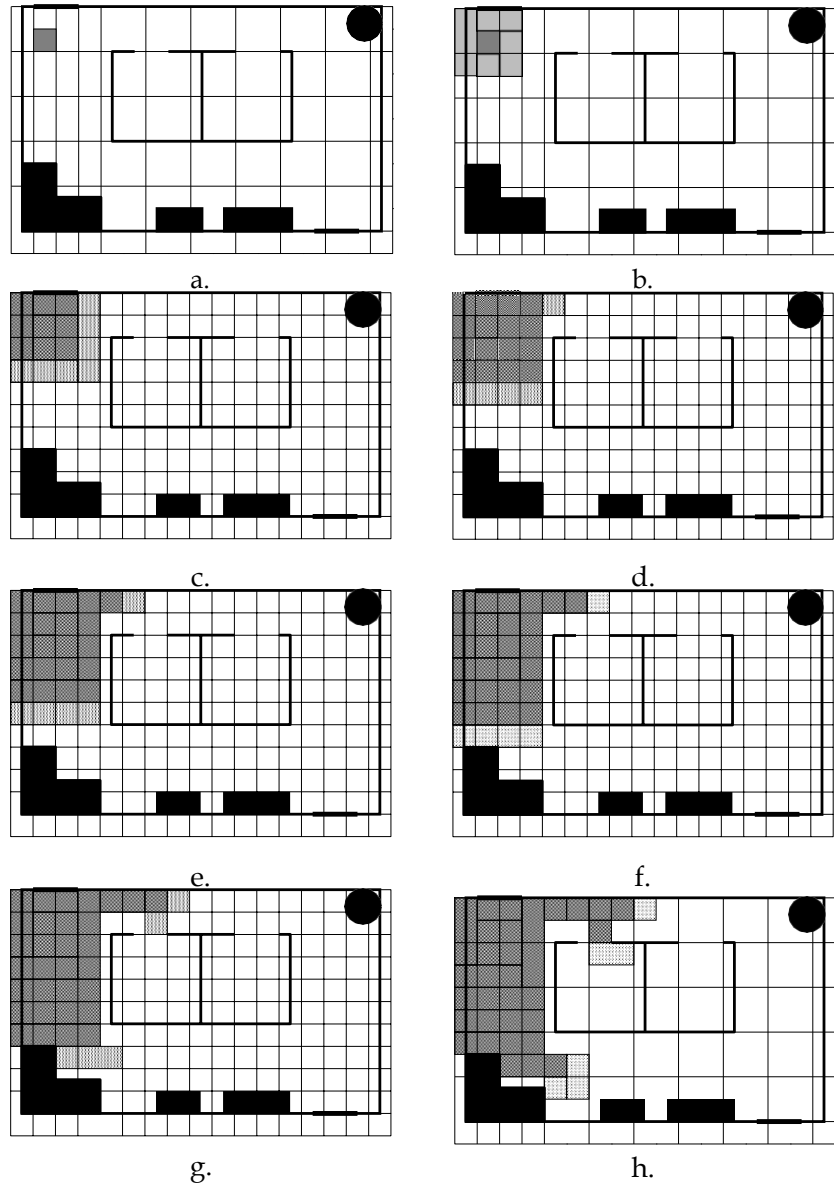


Figure 10.11 A wave propagating through a regular grid. Elements holding the current front are shown in gray, older elements are shown in dark gray.

An attractive feature of wavefront propagation is how it treats terrain types. A modeled obstacle can have a conductivity of 0.0 (no heat can be propagated through that grid element), while an open area has infinite conductivity. But undesirable terrains, which are traversable, but not desirable to go across, can be given a low value of conductivity. This means that heat will travel slower through undesirable regions. But it may turn out that this is the best path, even with the loss through the undesirable regions. A wavefront naturally handles the tradeoffs between a longer path through desirable terrain versus taking shortcuts through less desirable terrain.

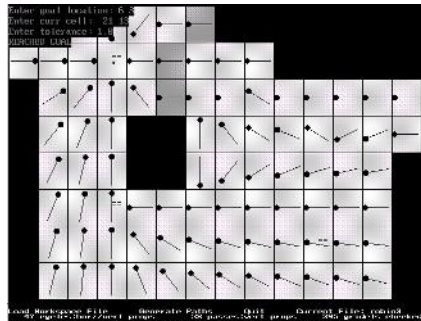
The examples in Fig. 10.12 use the Trulla planner. Undesirable terrain is shown on the Trulla display in gray, obstacles in black and open areas in white. The intensity of gray reflects the degree of undesirability of the terrain. In Fig. 10.12a, the robot Clementine can move over an extension cord but would prefer not to, as shown by the gray region. The path planner accordingly routes her around the cord. In Fig. 10.12 this isn't possible due to the placement of obstacles. Therefore, she has to cut through.

10.6 Interleaving Path Planning and Reactive Execution

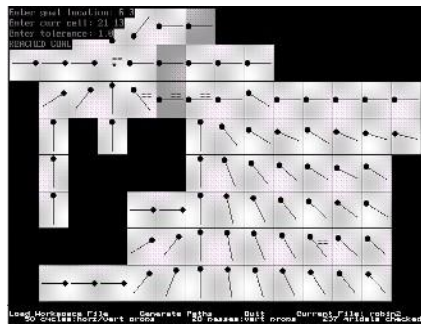
Most path planning algorithms are employed in a strictly plan once, then reactively execute style. Almost all techniques break a path into segments; even a wavefront planner actually produces a goal location (waypoint) for each directional vector. This is well suited for a Hybrid architecture with a Cartographer handing off path segments, or an entire path, for a Sequencer. The Sequencer can then employ a series of move-to-goal behaviors, deactivating and re-instantiating the behavior as a new subgoal is met. Unfortunately there are two problems with reactive execution of a metric path as described above: *subgoal obsession* and the lack of *opportunistic replanning*. Subgoal obsession is when the robot spends too much time and energy trying to reach the exact subgoal position, or more precisely, when the termination conditions are set with an unrealistic tolerance. The problem with termination conditions for subgoals is best defined by an example. Suppose the next waypoint is at location (35, 50). If the robot has shaft encoders or GPS, this should be straightforward to do in theory. In practice, it is very hard for a robot, even a holonomic robot, to reach any location exactly because it is hard

SUBGOAL OBSESSION
OPPORTUNISTIC
REPLANNING

to give the robot precise movement. The robot may reach (34.5, 50). The behavior sees the goal is now 0.5 meters ahead, and move again to attempt to reach (35, 50). On that move, it may overshoot and end up at (35.5, 50.5).



a.



b.

Figure 10.12 a.) Trulla output and photo of Clementine navigating around cord. b.) Trulla output and photo of Clementine navigating over the cord.

Now it has to turn and move again. A resulting see-sawing set of motions emerges that can go on for minutes. This wastes time and energy, as well as makes the robot appear to be unintelligent. The problem is exacerbated by non-holonomic vehicles which may have to back up in order to turn to reach a particular location. Backing up almost always introduces more errors in navigation.

To handle subgoal obsession, many roboticists program into their move-to-goal behaviors a tolerance on the terminating condition for reaching a goal. A common heuristic for holonomic robots is to place a tolerance of $\pm$ the width of the robot. So if a cylindrical, holonomic robot with a diameter of 1 meter is given a goal of (35, 50), it will stop when $34.5 < y < 35.5$ and 49.5

$< y < 50.5$. There is no common heuristic for non-holonomic robots, because the maneuverability of each platform is different. A more vexing aspect of subgoal obsession is when the goal is blocked and the robot can't reach the terminating condition. For example, consider a subgoal at the opposite end of a hall from a robot, but the hall is blocked and there is no way around. Because the robot is executing reactively, it doesn't necessarily realize that it isn't making progress. One solution is for the Sequencer to estimate a maximum allowable time for the robot to reach the goal. This can be implemented either as a parameter on the behavior (terminate with an error code after n seconds), or as an internal state releaser. The advantage of the latter solution is that the code can become part of a monitor, leading to some form of self-awareness.

Related to subgoal obsession is the fact that reactive execution of plans often lacks opportunistic improvements. Suppose that the robot is heading for Subgoal 2, when an unmodeled obstacle diverts the robot from its intended path. Now suppose that from its new position, the robot can perceive Subgoal 3. In a classic implementation, the robot would not be looking for Subgoal 3, so it would continue to move to Subgoal 2 even though it would be more optimal to head straight for Subgoal 3.

The need to opportunistically replan also arises when an *a priori* map turns out to be incorrect. What happens when the robot discovers it is being sent through a patch of muddy ground? Trying to reactively navigate around the mud patch seems unintelligent because choosing left or right may have serious consequences. Instead the robot should return control to the Cartographer, which will update its map and replan. The issue becomes how does a robot know it has deviated too far from its intended path and needs to replan?

D* ALGORITHM

Two different types of planners address the problems of subgoal obsession and replanning: the *D* algorithm* developed by Tony Stentz¹³⁶ which is a variant of the A* replanning algorithm, and an extension to the Trulla algorithm. Both planners begin with an *a priori* map and compute the optimal path from every location to the goal. D* does it by executing an A* search from each possible location to the goal in advance; this converts A* from being a single-source shortest path algorithm into an all-paths algorithm. This is computationally expensive and time-consuming, but that isn't a problem since the paths are computed when the robot starts the mission and is sitting still. Since Trulla is a wavefront type of planner,

it generates the optimal path between all pairs of points in Cspace as a side effect of computing the path from the starting location to the goal.

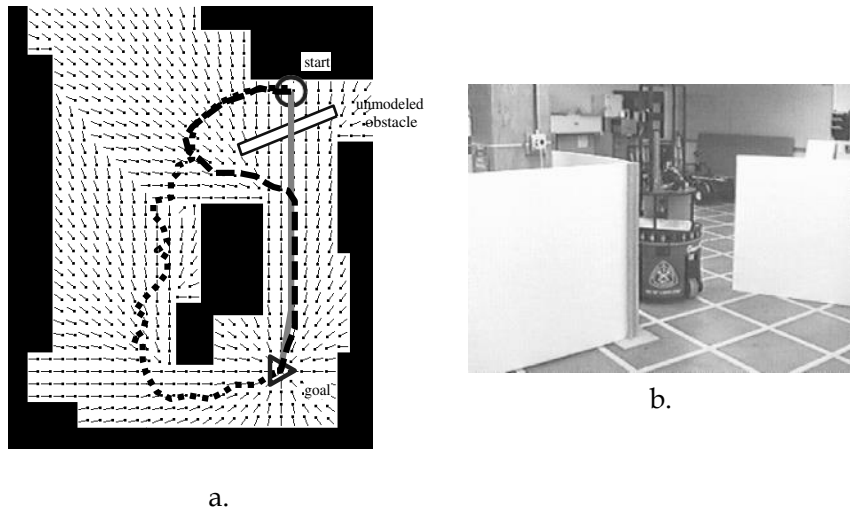


Figure 10.13 Layout showing unmodeled obstacle. a.) Gray line shows expected path, long dashed line the actual path with Trulla, and short dashed line shows purely reactive path. b.) *Clementine* opportunistically turning.

Computing the optimal path from every location to the goal actually helps with reactive execution of the path. It means that if the robot can localize itself on the *a priori* map, it can read the optimal subgoal for move-to-goal on each update. If the robot has to swing wide to avoid an unmodeled obstacle in Fig. 10.13, the robot automatically becomes redirected to the optimal path without having to replan. Note how the metric path becomes a virtual sensor, guiding the move-to-goal behavior replacing the direct sensor data. This is a rich mechanism for the deliberative and reactive components of Hybrid architectures to interact.

This approach eliminates subgoal obsession, since the robot can change “optimal” paths reactively and opportunistically move to a closer waypoint. As with most things in life, too much of a good thing is bad. At some point though, the sheer number of unmodeled obstacles might force the robot to get trapped or wander about, changing subgoals but making no real progress. The D^* solution to this problem is to continuously update the map and dynamically repair the A^* paths affected by the changes in the map. D^* represents one extreme on the replanning scale: *continuous replanning*.

CONTINUOUS

REPLANNING

Continuous replanning has two disadvantages. First, it may be too

370 compu- tationally expensive to be practical for a robot with an embedded processor

and memory limitations, such as a planetary rover. Second, continuous re-planning is highly dependent on the sensing quality. If the robot senses an unmodeled obstacle at time T_1 , it computes a new path and makes a large course correction. If it no longer senses that obstacle at time T_2 because the first reading was a phantom from sensor noise, it will recompute another large course correction. The result can be a robot which has a very jerky motion and actually takes longer to reach the goal.

EVENT-DRIVEN

In the cases of path planning with embedded processors and noisy sensors, it would be desirable to have some sort of *event-driven* scheme, where an

REPLANNING

event noticeable by the reactive system would trigger replanning. Trulla uses the dot-product of the intended path vector and the actual path vector. When the actual path deviates by 90° or more, the dot product of the path vector and the actual vector the robot is following becomes 0 or negative. Therefore the dot product acts as an affordance for triggering replanning: the robot doesn't have to know why it is drifting off-course, only that it has drifted noticeably off-course.

This is very good for situations that would interfere with making progress on the originally computed path, in effect, situations where the real world is less amenable to reaching the intended goal. But it doesn't handle the situation where the real world is actually friendlier. In Fig. 10.14, an obstacle thought to be there really isn't. The robot could achieve a significant savings in navigation by opportunistically going through the gap.

Such opportunism requires the robot to notice that the world is really more favorable than originally modeled. A continuous replanner such as D^* has a distinct advantage, since it will automatically notice the change in the world and respond appropriately, whereas Trulla will not notice the favorable change because it won't lead to a path deviation. It is an open research question whether there are affordances for noticing favorable changes in the world that allow the robot to opportunistically optimize its path.

10.7 Summary

Metric path planning converts the world space to a configuration space, or *Cspace*, representation that facilitates path planning. Cspace representations such as generalized Voronoi diagrams exploit interesting

geometric prop- erties of the environment. These representations can then be converted to graphs, suitable for an A* search. Since Voronoi diagrams tend to produce sparser graphs, they work particularly well with A*. Regular grids work

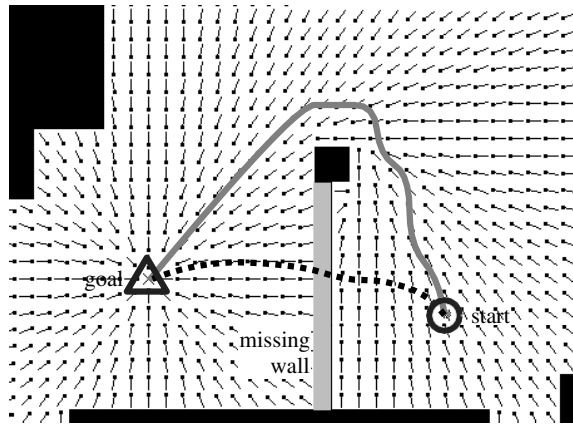


Figure 10.14 Opportunity to improve path. The gray line is the actual path, while the dashed line represents a more desirable path.

well with wavefront planners, which treat path planning as an expanding heat wave from the initial location.

Metric path planning tends to be expensive, both in computation and in storage. However, they can be interleaved with the reactive component of a Hybrid architecture, where the Cartographer gives the Sequencer a set of waypoints. Two problems in interleaving metric path planning with reactive execution are subgoal obsession and when to replan. Optimal path planning techniques for *a priori* fixed maps are well understood, but it is less clear how to update or repair the path(s) without starting over if the robot encounters a significant deviation from the *a priori* map. One solution is to continuously replan if resources permit and sensor reliability is high; another is event-driven replanning which uses affordances to detect when to replan.

Cspace representations and algorithms often do not consider how to represent and reason about terrain types, and special cases such as the robot actually conserving or generating energy going downhill are usually ignored. A possibly even more serious omission is that popular path planners are applicable for only holonomic vehicles.

1. Schueller A., Programming with Robots, Whitman College, 2011.
2. Murphy R., Introduction to AI Robotics, The MIT Press, Cambridge, Massachusetts, 2000.
3. Robotics From Wikipedia the free encyclopedia.
4. Poole D., Mackworth A. (2010), Artificial Intelligence: Foundation of Computational Agents, Cambridge University Press, New York.
5. Aamodt, A. and Plaza, E. (1994). Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1): 39-59.
6. Abelson, H. and DiSessa, A. (1981). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge, MA.
7. Abramson, H. and Rogers, M.H. (Eds.) (1989). *Meta-Programming in Logic Programming*. MIT Press, Cambridge, MA.
8. Agre, P.E. (1995). Computational research on interaction and agency. *Artificial Intelligence*, 72: 1-52.
9. Aha, D.W., Marling, C., and Watson, I. (Eds.) (2005). *The Knowledge Engineering Review, special edition on case-based reasoning*, volume 20 (3). Cambridge University Press.
<http://journals.cambridge.org/action/displayIssue?jid=KER&volumeId=20&issueId=03>.
10. Albus, J.S. (1981). *Brains, Behavior and Robotics*. BYTE Publications, Peterborough, NH.
11. Allais, M. and Hagen, O. (Eds.) (1979). *Expected Utility Hypothesis and the Allais Paradox*. Reidel, Boston, MA.
12. Allen, J., Hendler, J., and Tate, A. (Eds.) (1990). *Readings in Planning*. Morgan Kaufmann, San Mateo, CA.
13. Anderson, M. and Leigh Anderson, S.L. (2007). Machine ethics: Creating an ethical intelligent agent. *AI Magazine*, 28(4): 15-26.
14. Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M.I. (2003). An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2): 5-43.
15. Antoniou, G. and van Harmelen, F. (2008). *A Semantic Web Primer*. MIT Pres, Cambridge, MA, 2nd edition.
16. Apt, K. and Bol, R. (1994). Logic programming and negation: A survey. *Journal of Logic Programming*, 19,20: 9-71.
17. Aristotle (350 B.C.). *Categories*. Translated by E. M. Edghill,
<http://www.classicallibrary.org/Aristotle/categories/>.
18. Asimov, I. (1950). *I, Robot*. Doubleday, Garden City, NY.

19. Bacchus, F. and Grove, A. (1995). Graphical models for preference and utility. In *Uncertainty in Artificial Intelligence (UAI-95)*, pp. 3-10.
20. Bacchus, F., Grove, A.J., Halpern, J.Y., and Koller, D. (1996). From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87(1-2): 75-143.
<http://www.cs.toronto.edu/fbacchus/Papers/BGHKAIJ96.ps>.
21. Bacchus, F. and Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani (Eds.), *New Directions in AI Planning*, pp. 141-153. ISO Press, Amsterdam.
22. Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, NY.
23. Ballard, B.W. (1983). The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3): 327-350.
24. Baum, E.B. (2004). *What is Thought?* MIT Press.
25. Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53: 370-418. Reprinted in *Biometrika* 45, 298-315, 1958. Reprinted in S. J. Press, *Bayesian Statistics*, 189-217, Wiley, New York, 1989.
26. Beckett, D. and Berners-Lee, T. (2008). Turtle - terse RDF triple language.
<http://www.w3.org/TeamSubmission/turtle/>.
27. Bell, J.L. and Machover, M. (1977). *A Course in Mathematical Logic*. North-Holland, Amsterdam.
28. Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.
29. Bernardo, J.M. and Smith, A.F.M. (1994). *Bayesian Theory*. Wiley.
30. Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, May: 28-37. <http://www.sciam.com/article.cfm?id=the-semantic-web>.
31. Bertelè, U. and Brioschi, F. (1972). *Nonserial dynamic programming*, volume 91 of *Mathematics in Science and Engineering*. Academic Press.
32. Bertsekas, D.P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Massachusetts. Two volumes.
33. Bertsekas, D.P. and Tsitsiklis, J.N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts.
34. Besnard, P. and Hunter, A. (2008). *Elements Of Argumentation*. MIT Press, Cambridge, MA.
35. Bishop, C.M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, England.
36. Bishop, C.M. (2008). *Pattern Recognition and Machine Learning*. Springer-Verlag, New York.
37. Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90: 281-300.

38. Bobrow, D.G. (1993). Artificial intelligence in perspective: a retrospective on fifty volumes of Artificial Intelligence. *Artificial Intelligence*, 59: 5-20.
39. Bobrow, D.G. (1967). Natural language input for a computer problem solving system. In M. Minsky (Ed.), *Semantic Information Processing*, pp. 133-215. MIT Press, Cambridge MA.
40. Boddy, M. and Dean, T.L. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2): 245-285.
41. Bodlaender, H.L. (1993). A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2): 1-21.
42. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., and Poole, D. (2004). Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *Journal of Artificial Intelligence Research*, 21: 135-191.
<http://www.jair.org/abstracts/boutilier04a.html>.
43. Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11: 1-94.
44. Bowen, K.A. (1985). Meta-level programming and knowledge representation. *New Generation Computing*, 3(4): 359-383.
45. Bowling, M. and Veloso, M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2): 215-250.
46. Brachman, R.J. and Levesque, H.J. (Eds.) (1985). *Readings in Knowledge Representation*. Morgan Kaufmann, San Mateo, CA.
47. Brachman, R. and Levesque, H. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann.
48. Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.
49. Briscoe, G. and Caelli, T. (1996). *A Compendium of Machine Learning, Volume 1: Symbolic Machine Learning*. Ablex, Norwood, NJ.
50. Brooks, R.A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1): 14-23. Reprinted in [\[Shafer and Pearl \(1990\)\]](#).
51. Brooks, R.A. (1991). Intelligence without representation. *Artificial Intelligence*, 47: 139-159.
52. Brooks, R. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, 6: 3-15.
<http://people.csail.mit.edu/brooks/papers/elephants.pdf>.
53. Bryce, D. and Kambhampati, S. (2007). A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(47-83): 1.
54. Buchanan, B. and Shortliffe, E. (Eds.) (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA.
55. Buchanan, B.G. and Feigenbaum, E.A. (1978). Dendral and meta-dendral: Their applications dimension. *Artificial Intelligence*, 11: 5-24.
56. Buchanan, B.G. (2005). A (very) brief history of artificial intelligence. *AI Magazine*, 26(4): 53-60.
57. Buntine, W. (1992). Learning classification trees. *Statistics and Computing*, 2: 63-73.

58. Burch, R. (2008). Charles Sanders Peirce. *The Stanford Encyclopedia of Philosophy*.
<http://plato.stanford.edu/archives/spr2008/entries/peirce/>.
59. Campbell, M., Hoane Jr., A.J., and Hse, F.h. (2002). Deep blue. *Artificial Intelligence*, 134(1-2): 57-83. <http://www.sciencedirect.com/science/article/B6TYF-43PHC49-2/1/325879d3cbf078187ea49a232e421ea9>.
60. Castillo, E., Gutiérrez, J.M., and Hadi, A.S. (1996). *Expert Systems and Probabilistic Network Models*. Springer Verlag, New York.
61. Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32(3): 333-377.
62. Cheeseman, P. (1990). On finding the most probable model. In J. Shriringer and P. Langley (Eds.), *Computational Models of Scientific Discovery and Theory Formation*, chapter 3, pp. 73-95. Morgan Kaufmann, San Mateo, CA.
63. Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., and Freeman, D. (1988). Autoclass: A Bayesian classification system. In *Proc. Fifth International Conference on Machine Learning*, pp. 54-64. Ann Arbor, MI. Reprinted in [Shavlik and Dietterich (1990)].
64. Cheng, J. and Druzdzel, M. (2000). AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13: 155-188. <http://www.jair.org/papers/paper764.html>.
65. Chesnevar, C., Maguitman, A., and Loui, R. (2000). Logical models of argument. *ACM Comput. Surv.*, 32(4): 337-383.
66. Chomsky, N. (1957). *Syntactic Structures*. Mouton and Co., The Hague.
67. Chrisley, R. and Begeer, S. (2000). *Artificial intelligence: Critical Concepts in Cognitive Science*. Routledge, London and New York.
68. Clark, K.L. (1978). Negation as failure. In H. Gallaire and J. Minker (Eds.), *Logic and Databases*, pp. 293-322. Plenum Press, New York.
69. Cohen, P.R. (2005). If not Turing's test, then what? *AI Magazine*, 26(4): 61-67.
70. Colmerauer, A., Kanoui, H., Roussel, P., and Pasero, R. (1973). Un système de communication homme-machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille.
71. Colmerauer, A. and Roussel, P. (1996). The birth of Prolog. In T.J. Bergin and R.G. Gibson (Eds.), *History of Programming Languages*. ACM Press/Addison-Wesley.
<http://alain.colmerauer.free.fr/ArchivesPublications/HistoireProlog/19november92.pdf>.
72. Copi, I.M. (1982). *Introduction to Logic*. Macmillan, New York, sixth edition.
73. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition.
74. Cover, T.M. and Thomas, J.A. (1991). *Elements of information theory*. Wiley, New York.
75. Culberson, J. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14(3): 318-334.
76. Dahl, V. (1994). Natural language processing and logic programming. *Journal of Logic Programming*, 19,20: 681-714.

77. Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.
78. Dasarathy, B.V. (1991). NN concepts and techniques. In B.V. Dasarathy (Ed.), *Nearest Neighbour (NN) Norms: NN Pattern Classification Techniques*, pp. 1-30. IEEE Computer Society Press, New York.
79. Davis, E. (1990). *Representations of Commonsense Knowledge*. Morgan Kaufmann, San Mateo, CA.
80. Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine Learning*, pp. 233 - 240. http://www.icml2006.org/icml_documents/camera-ready/030_The_Relationship_Bet.pdf.
81. Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, 5(7): 394-397.
82. Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, 7(3): 201-215.
83. de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28(2): 127-162.
84. de Kleer, J., Mackworth, A.K., and Reiter, R. (1992). Characterizing diagnoses and systems. *Artificial Intelligence*, 56: 197-222.
85. De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S.H. (Eds.) (2008). *Probabilistic Inductive Logic Programming*. Springer.
86. Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3): 142-150.
87. Dean, T.L. and Wellman, M.P. (1991). *Planning and Control*. Morgan Kaufmann, San Mateo, CA.
88. Dechter, R. (1996). Bucket elimination: A unifying framework for probabilistic inference. In E. Horvitz and F. Jensen (Eds.), *Proc. Twelfth Conf. on Uncertainty in Artificial Intelligence (UAI-96)*, pp. 211-219. Portland, OR.
89. Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.
90. Dellaert, F., Fox, D., Burgard, W., and Thrun, S. (1999). Monte Carlo localization for mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA99)*. http://www.ri.cmu.edu/pub_files/pub1/dellaert_frank_1999_2/dellaert_frank_1999_2.pdf.
91. Dietterich, T.G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13: 227-303. <http://jair.org/papers/paper639.html>.
92. Dietterich, T.G. (2002). Ensemble learning. In M. Arbib (Ed.), *The Handbook of Brain Theory and Neural Networks*, pp. 405-408. MIT Press, Cambridge, MA, second edition. <http://web.engr.oregonstate.edu/tgd/publications/hbttnn-ensemble-learning.ps.gz>.
93. Dijkstra, E.W. (1976). *A discipline of programming*. Prentice-Hall, Englewood Cliffs, NJ.
94. Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269-271. <http://gdzdoc.sub.uni-goettingen.de/sub/digbib/loader?did=D196313>.

95. Doucet, A., de Freitas, N., and Gordon, N. (Eds.) (2001). *Sequential Monte Carlo in Practice*. Springer-Verlag.
96. Doyle, J. (1979). A truth maintenance system. AI Memo 521, MIT Artificial Intelligence Laboratory.
97. Dresner, K. and Stone, P. (2008). A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31: 591-656.
98. Duda, R.O., Hart, P.E., and Stork, D.G. (2001). *Pattern Classification*. Wiley-Interscience, 2nd edition.
99. Dung, P. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n -person games. *Artificial Intelligence*, 77(2): 321-357.
100. Dung, P., Mancarella, P., and Toni, F. (2007). Computing ideal sceptical argumentation. *Artificial Intelligence*, 171(10-15): 642-674.
101. Edwards, P. (Ed.) (1967). *The Encyclopedia of Philosophy*. Macmillan, New York.
102. Enderton, H.B. (1972). *A Mathematical Introduction to Logic*. Academic Press, Orlando, FL.
103. Felner, A., Korf, R.E., and Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22: 279-318.
104. Fikes, R.E. and Nilsson, N.J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4): 189-208.
105. Fisher, D. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2: 139-172. Reprinted in [Shavlik and Dietterich (1990)].
106. Forbus, K.D. (1996). Qualitative reasoning. In *CRC Hand-book of Computer Science and Engineering*. CRC Press. <http://www.qrg.northwestern.edu/papers/Files/crc7.pdf>.
107. Freuder, E.C. and Mackworth, A.K. (2006). Constraint satisfaction: An emerging paradigm. In P.V.B. F. Rossi and T. Walsh (Eds.), *Handbook of Constraint Programming*, pp. 13-28. Elsevier.
108. Friedman, N. and Goldszmidt, M. (1996a). Building classifiers using Bayesian networks. In *Proc. 13th National Conference on Artificial Intelligence*, pp. 1277-1284. Portland, OR.
109. Friedman, N. and Goldszmidt, M. (1996b). Learning Bayesian networks with local structure. In *Proc. Twelfth Conf. on Uncertainty in Artificial Intelligence (UAI-96)*, pp. 252-262. <http://www2.sis.pitt.edu/dsl/UAI/UAI96/Friedman1.UAI96.html>.
110. Friedman, N., Greiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29: 103-130.
111. Gabbay, D.M., Hogger, C.J., and Robinson, J.A. (Eds.) (1993). *Handbook of Logic in Artificial Intelligence and Logic Programming*. Clarendon Press, Oxford, England. 5 volumes.
112. Gangemi, A., Guarino, N., Masolo, C., and Oltramari, A. (2003). Sweetening wordnet with dolce. *AI Magazine*, 24(3): 13-24.
113. Garcia-Molina, H., Ullman, J.D., and Widom, J. (2009). *Database Systems: The Complete Book*. Prentice Hall, 2nd edition.
114. Gardner, H. (1985). *The Mind's New Science*. Basic Books, New York.

115. Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004). *Bayesian Data Analysis*. Chapman and Hall/CRC, 2nd edition.
116. Getoor, L. and Taskar, B. (Eds.) (2007). *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, MA.
117. Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA.
118. Goldberg, D.E. (2002). *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Addison-Wesley, Reading, MA.
119. Green, C. (1969). Application of theorem proving to problem solving. In *Proc. 1st International Joint Conf. on Artificial Intelligence*, pp. 219-237. Washington, DC. Reprinted in [\[Webber and Nilsson \(1981\)\]](#).
120. Grenon, P. and Smith, B. (2004). Snap and span: Towards dynamic spatial ontology. *Spatial Cognition and Computation*, 4(1): 69-103.
http://ontology.buffalo.edu/smith/articles/SNAP_SPAN.pdf.
121. Grünwald, P.D. (2007). *The Minimum Description Length Principle*. The MIT Press, Cambridge, MA.
122. Halpern, J. (1997). A logical approach to reasoning about uncertainty: A tutorial. In X. Arrazola, K. Kortha, and F. Pelletier (Eds.), *Discourse, Interaction and Communication*. Kluwer.
123. Hart, P.E., Nilsson, N.J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100-107.
124. Hart, T.P. and Edwards, D.J. (1961). The tree prune (TP) algorithm. Memo 30, MIT Artificial Intelligence Project, Cambridge MA.
125. Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, second edition.
126. Haugeland, J. (1985). *Artificial Intelligence: The Very Idea*. MIT Press, Cambridge, MA.
127. Haugeland, J. (Ed.) (1997). *Mind Design II: Philosophy, Psychology, Artificial Intelligence*. MIT Press, Cambridge, MA, revised and enlarged edition.
128. Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36(2): 177-221. Reprinted in [\[Shavlik and Dietterich \(1990\)\]](#).
129. Hayes, P.J. (1973). Computation and deduction. In *Proc. 2nd Symposium on Mathematical Foundations of Computer Science*, pp. 105-118. Czechoslovak Academy of Sciences.
130. Heckerman, D. (1999). A tutorial on learning with Bayesian networks. In M. Jordan (Ed.), *Learning in Graphical Models*. MIT press.
131. Hendler, J., Berners-Lee, T., and Miller, E. (2002). Integrating applications on the semantic web. *Journal of the Institute of Electrical Engineers of Japan*, 122(10): 676-680.
<http://www.w3.org/2002/07/swint>.

132. Henrion, M. (1988). Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In J.F. Lemmer and L.N. Kanal (Eds.), *Uncertainty in Artificial Intelligence 2*, pp. 149-163. Elsevier Science Publishers B.V.
133. Hertz, J., Krogh, A., and Palmer, R.G. (1991). *Introduction to the Theory of Neural Computation*. Lecture Notes, Volume I, Santa Fe Institute Studies in the Sciences of Complexity. Addison-Wesley, Reading, MA.
134. Hewitt, C. (1969). Planner: A language for proving theorems in robots. In *Proc. 1st International Joint Conf. on Artificial Intelligence*, pp. 295-301. Washington, DC.
135. Hillis, W.D. (2008). A forebrain for the world mind. *Edge: World Question Center*. www.edge.org/q2009/q09_12.html#hillis.
136. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., and Rudolph, S. (2009). *OWL 2 Web Ontology Language Primer*. W3C. <http://www.w3.org/TR/owl2-primer/>.
137. Hobbs, J.R., Stickel, M.E., Appelt, D.E., and Martin, P. (1993). Interpretation as abduction. *Artificial Intelligence*, 63(1-2): 69-142.
138. Holland, J.H. (1975). *Adaption in Natural and Artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, Ann Arbor, MI.
139. Hoos, H.H. and Stützle, T. (2004). *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier.
140. Horvitz, E.J. (1989). Reasoning about beliefs and actions under computational resource constraints. In L. Kanal, T. Levitt, and J. Lemmer (Eds.), *Uncertainty in Artificial Intelligence 3*, pp. 301-324. Elsevier, New York.
141. Horvitz, E. (2006). Eric Horvitz forecasts the future. *New Scientist*, 2578: 72. <http://www.newscientist.com/article/mg19225780.121-eric-horvitz-forecasts-the-future.html>.
142. Howard, R.A. and Matheson, J.E. (1984). Influence diagrams. In R.A. Howard and J.E. Matheson (Eds.), *The Principles and Applications of Decision Analysis*. Strategic Decisions Group, Menlo Park, CA.
143. Howson, C. and Urbach, P. (2006). *Scientific Reasoning: the Bayesian Approach*. Open Court, Chicago, Illinois, 3rd edition.
144. Jaynes, E.T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. <http://omega.albany.edu:8008/JaynesBook.html>.
145. Jensen, F.V. (1996). *An Introduction to Bayesian Networks*. Springer Verlag, New York.
146. Jordan, M. and Bishop, C. (1996). Neural networks. Memo 1562, MIT Artificial Intelligence Lab, Cambridge, MA. <ftp://psyche.mit.edu/pub/jordan/crc.ps.Z>.
147. Joy, B. (2000). Why the future doesn't need us. *Wired*. www.wired.com/wired/archive/8.04/joy.html.
148. Jurafsky, D. and Martin, J.H. (2008). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, second edition.

149. Kaelbling, L.P., Littman, M.L., and Moore, A.W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4: 237-285.
150. Kakas, A. and Denecker, M. (2002). Abduction in logic programming. In A. Kakas and F. Sadri (Eds.), *Computational Logic: Logic Programming and Beyond*, number 2407 in LNAI, pp. 402-436. Springer Verlag. http://www2.cs.kuleuven.be/cgi-bin/dtai/publ_info.pl?id=39495.
151. Kakas, A.C., Kowalski, R.A., and Toni, F. (1993). Abductive logic programming. *Journal of Logic and Computation*, 2(6): 719-770.
152. Kambhampati, S., Knoblock, C.A., and Yang, Q. (1995). Planning as refinement search: a unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76: 167-238. Special issue on Planning and Scheduling.
153. Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. 13th National Conference on Artificial Intelligence*, pp. 1194-1201. Portland, OR.
154. Kearns, M. and Vazirani, U. (1994). *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA.
155. Keeney, R.L. and Raiffa, H. (1976). *Decisions with Multiple Objectives*. John Wiley and Sons.
156. Kelly, K. (2008). A new kind of mind. *Edge: World Question Center*. www.edge.org/q2009/q09_1.html#kelly.
157. Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983). Optimization by simulated annealing. *Science*, 220: 671-680.
158. Kirsh, D. (1991a). Foundations of AI: the big issues. *Artificial Intelligence*, 47: 3-30.
159. Kirsh, D. (1991b). Today the earwig, tomorrow man? *Artificial Intelligence*, 47: 161-184.
160. Knuth, D.E. and Moore, R.W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4): 293-326.
161. Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
162. Koller, D. and Milch, B. (2003). Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, 45(1): 181-221. <http://people.csail.mit.edu/milch/papers/geb-maid.pdf>.
163. Kolodner, J. and Leake, D. (1996). A tutorial introduction to case-based reasoning. In D. Leake (Ed.), *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, pp. 31-65. AAAI Press/MIT Press.
164. Korf, K.E. (1985). Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1): 97-109.
165. Kowalski, R. (1979). *Logic for Problem Solving*. Artificial Intelligence Series. North-Holland, New York.
166. Kowalski, R. and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4(1): 67-95.

167. Kowalski, R.A. (1974). Predicate logic as a programming language. In *Information Processing 74*, pp. 569-574. North-Holland, Stockholm.
168. Kowalski, R.A. (1988). The early history of logic programming. *CACM*, 31(1): 38-43.
169. Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.
170. Kuipers, B. (2001). Qualitative simulation. In R.A. Meyers (Ed.), *Encyclopedia of Physical Science and Technology*, pp. 287-300. Academic Press, NY, third edition.
<http://www.cs.utexas.edu/users/qr/papers/Kuipers-epst-01.html>.
171. Langley, P., Iba, W., and Thompson, K. (1992). An analysis of Bayesian classifiers. In *Proc. 10th National Conference on Artificial Intelligence*, pp. 223-228. San Jose, CA.
172. Laplace, P. (1812). *Théorie Analytique de Probabilités*. Courcier, Paris.
173. Latombe, J.C. (1991). *Robot Motion Planning*. Kluwer Academic Publishers, Boston.
174. Lawler, E.L. and Wood, D.E. (1966). Branch-and-bound methods: A survey. *Operations Research*, 14(4): 699-719.
175. Leibniz, G.W. (1677). *The Method of Mathematics: Preface to the General Science*. Selections reprinted by [Chrisley and Begeer \(2000\)](#).
176. Lenat, D.B. and Feigenbaum, E.A. (1991). On the thresholds of knowledge. *Artificial Intelligence*, 47: 185-250.
177. Levesque, H.J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2): 155-212.
178. Liu, A.L., Hile, H., Kautz, H., Borriello, G., Brown, P.A., Harniss, M., and Johnson, K. (2006). Indoor wayfinding: Developing a functional interface for individuals with cognitive impairments. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, pp. 95-102. Association for Computing Machinery, New York.
179. Lloyd, J.W. (1987). *Foundations of Logic Programming*. Symbolic Computation Series. Springer-Verlag, Berlin, second edition.
180. Lopez, A. and Bacchus, F. (2003). Generalizing GraphPlan by formulating planning as a CSP. In *IJCAI-03*, pp. 954-960.
181. Lopez De Mantaras, R., Mcsherry, D., Bridge, D., Leake, D., Smyth, B., Craw, S., Faltings, B., Maher, M.L., Cox, M.T., Forbus, K., Keane, M., Aamodt, A., and Watson, I. (2005). Retrieval, reuse, revision and retention in case-based reasoning. *The Knowledge Engineering Review*, 20(3): 215-240. doi: 10.1017/S0269888906000646.
<http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=435263&fulltextType=RA&fileId=S0269888906000646>.
182. Loredó, T. (1990). From Laplace to supernova SN 1987A: Bayesian inference in astrophysics. In P. Fougère (Ed.), *Maximum Entropy and Bayesian Methods*, pp. 81-142. Kluwer Academic Press, Dordrecht, The Netherlands. <http://bayes.wustl.edu>.
183. Lowe, D.G. (1995). Similarity metric learning for a variable-kernel classifier. *Neural Computation*, 7: 72-85.

- 184.Luenberger, D.G. (1979). *Introduction to Dynamic Systems: Theory, Models and Applications*. Wiley, New York.
- 185.Mackworth, A.K. (1993). On seeing robots. In A. Basu and X. Li (Eds.), *Computer Vision: Systems, Theory, and Applications*, pp. 1-13. World Scientific Press, Singapore.
- 186.Mackworth, A.K. (2009). Agents, bodies, constraints, dynamics and evolution. *AI Magazine*.
- 187.Mackworth, A.K. (1977). On reading sketch maps. In *Proc. Fifth International Joint Conf. on Artificial Intelligence*, pp. 598-606. MIT, Cambridge, MA.
- 188.Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.
- 189.Mas-Colell, A., Whinston, M.D., and Green, J.R. (1995). *Microeconomic Theory*. Oxford University Press, New York, NY.
- 190.Matheson, J.E. (1990). Using influence diagrams to value information and control. In R.M. Oliver and J.Q. Smith (Eds.), *Influence Diagrams, Belief Nets and Decision Analysis*, chapter 1, pp. 25-48. Wiley.
- 191.McAllester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proc. 9th National Conference on Artificial Intelligence*, pp. 634-639.
- 192.McCarthy, J. (1986). Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28(1): 89-116.
- 193.McCarthy, J. and Hayes, P.J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In M. Meltzer and D. Michie (Eds.), *Machine Intelligence 4*, pp. 463-502. Edinburgh University Press.
- 194.McCulloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5: 115-133.
- 195.McDermott, D. and Hendler, J. (1995). Planning: What it is, what it could be, an introduction to the special issue on planning and scheduling. *Artificial Intelligence*, 76: 1-16.
- 196.McLuhan, M. (1964). *Understanding Media: The Extensions of Man*. New American Library, New York.
- 197.Meir, R. and Rätsch, G. (2003). An introduction to boosting and leveraging. In *In Advanced Lectures on Machine Learning (LNAI2600)*, pp. 119--184. Springer.
<http://www.boosting.org/papers/MeiRae03.pdf>.
- 198.Mendelson, E. (1987). *Introduction to Mathematical Logic*. Wadsworth and Brooks, Monterey, CA, third edition.
- 199.Michie, D., Spiegelhalter, D.J., and Taylor, C.C. (Eds.) (1994). *Machine Learning, Neural and Statistical Classification*. Series in Artificial Intelligence. Ellis Horwood, Hemel Hempstead, Hertfordshire, England.
- 200.Mihailidis, A., Boger, J., Candido, M., and Hoey, J. (2007). The use of an intelligent prompting system for people with dementia. *ACM Interactions*, 14(4): 34-37.
- 201.Minsky, M. (1961). Steps towards artificial intelligence. *Proceedings of the IEEE*, 49: 8-30.
<http://web.media.mit.edu/~minsky/papers/steps.html>.
- 202.Minsky, M. (1986). *The Society of Mind*. Simon and Schuster, New York.

203. Minsky, M. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, expanded edition.
204. Minsky, M.L. (1975). A framework for representing knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*, pp. 211-277. McGraw-Hill, New York. Alternative version is in [Haugeland (1997)], and reprinted in [Brachman and Levesque (1985)].
205. Minsky, M. (1952). A neural-analogue calculator based upon a probability model of reinforcement. Technical report, Harvard University Psychological Laboratories, Cambridge, MA.
206. Minton, S., Johnston, M.D., Philips, A.B., and Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3): 161-205. [http://dx.doi.org/10.1016/0004-3702\(92\)90007-K](http://dx.doi.org/10.1016/0004-3702(92)90007-K).
207. Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.
208. Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, New York.
209. Mitchell, T.M. (1977). Version spaces: A candidate elimination approach to rule learning. In *Proc. 5th International Joint Conf. on Artificial Intelligence*, pp. 305-310. Cambridge, MA.
210. Motik, B., Patel-Schneider, P.F., and Grau, B.C. (Eds.) (2009a). *OWL 2 Web Ontology Language Direct Semantics*. W3C. <http://www.w3.org/TR/owl2-semantics/>.
211. Motik, B., Patel-Schneider, P.F., and Parsia, B. (Eds.) (2009b). *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*. W3C. <http://www.w3.org/TR/owl2-syntax/>.
212. Muggleton, S. (1995). Inverse entailment and Prolog. *New Generation Computing*, 13(3,4): 245-286.
213. Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20: 629-679.
214. Muscettola, N., Nayak, P., Pell, B., and Williams, B. (1998). Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103: 5-47.
215. Nash, Jr., J.F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36: 48-49.
216. Nau, D.S. (2007). Current trends in automated planning. *AI Magazine*, 28(4): 43-58.
217. Neumann, J.V. and Morgenstern, O. (1953). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, third edition.
218. Newell, A. and Simon, H.A. (1976). Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, 19: 113-126. Reprinted in [Haugeland (1997)].
219. Newell, A. and Simon, H.A. (1956). The logic theory machine: A complex information processing system. Technical Report P-868, The Rand Corporation. <http://shelf1.library.cmu.edu/IMLS/MindModels/logictheorymachine.pdf>.
220. Niles, I. and Pease, A. (2001). Towards a standard upper ontology. In C. Welty and B. Smith (Eds.), *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*. Ogunquit, Maine. <http://www.ontologyportal.org/Pubs.html#FOIS>.

221. Nilsson, N. (2007). The physical symbol system hypothesis: Status and prospects. In e.a. M. Lungarella (Ed.), *50 Years of AI, Festschrift*, volume 4850 of *LNAI*, pp. 9-17. Springer. <http://ai.stanford.edu/pers/pssh.pdf>.
222. Nilsson, N.J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York.
223. Nilsson, N.J. (2009). *The Quest for Artificial Intelligence: A History of Ideas and Achievements*. Cambridge University Press, Cambridge, England.
224. Nisan, N. (2007). Introduction to mechanism design (for computer scientists). In N. Nisan et al. (Ed.), *Algorithmic Game Theory*, chapter 9, pp. 209-242. Cambridge University Press, Cambridge, England.
225. Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V.V. (Eds.) (2007). *Algorithmic Game Theory*. Cambridge University Press. http://www.cambridge.org/journals/nisan/downloads/Nisan_Non-printable.pdf.
226. Noy, N.F. and Hafner, C.D. (1997). The state of the art in ontology design: A survey and comparative review. *AI Magazine*, 18(3): 53-74. <http://www.aaai.org/Library/Magazine/vol18.php#Fall>.
227. Ordeshook, P.C. (1986). *Game theory and political theory: An introduction*. Cambridge University Press, New York.
228. Panton, K., Matuszek, C., Lenat, D., Schneider, D., Witbrock, M., Siegel, N., and Shepard, B. (2006). Common sense reasoning - from Cyc to intelligent assistant. In Y. Cai and J. Abascal (Eds.), *Ambient Intelligence in Everyday Life*, LNAI 3864, pp. 1-31. Springer.
229. Pearl, J. (1984). *Heuristics*. Addison-Wesley, Reading, MA.
230. Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA.
231. Pearl, J. (2000). *Causality: Models, Reasoning and Inference*. Cambridge University Press.
232. Peden, M.e.a. (Ed.) (2004). *World Report on Road Traffic Injury Prevention*. World Health Organization, Geneva.
233. Peng, Y. and Reggia, J.A. (1990). *Abductive Inference Models for Diagnostic Problem-Solving*. Symbolic Computation - AI Series. Springer-Verlag, New York.
234. Pereira, F.C.N. and Shieber, S.M. (2002). *Prolog and Natural-Language Analysis*. Microtome Publishing.
235. Pollack, M.E. (2005). Intelligent technology for an aging population: The use of ai to assist elders with cognitive impairment. *AI Magazine*, 26(2): 9-24.
236. Poole, D. (1993). Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1): 81-129.
237. Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94: 7-56. <http://www.cs.ubc.ca/spider/poole/abstracts/icl.html>. Special issue on economic principles of multi-agent systems.

238. Poole, D., Goebel, R., and Aleliunas, R. (1987). Theorist: A logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla (Eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge*, pp. 331-352. Springer-Verlag, New York, NY.
239. Poole, D., Mackworth, A., and Goebel, R. (1998). *Computational Intelligence: A Logical Approach*. Oxford University Press, New York.
240. Poole D., Mackworth A. (2010), *Artificial Intelligence: Foundation of Computational Agents*, Cambridge University Press, New York.
241. Posner, M.I. (Ed.) (1989). *Foundations of Cognitive Science*. MIT Press, Cambridge, MA.
242. Price, C.J., Travé-Massuyàs, L., Milne, R., Ironi, L., Forbus, K., Bredeweg, B., Lee, M.H., Struss, P., Snooke, N., Lucas, P., Cavazza, M., and Coghill, G.M. (2006). Qualitative futures. *The Knowledge Engineering Review*, 21(04): 317-334. doi: 10.1017/S026988890600097X. <http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=561496&fulltextType=RE&fileId=S026988890600097X>.
243. Puterman, M. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York.
244. Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning*, 1: 81-106. Reprinted in [\[Shavlik and Dietterich \(1990\)\]](#).
245. Quinlan, J.R. (1993). *C4.5 Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.
246. Quinlan, J.R. and Cameron-Jones, R.M. (1995). Induction of logic programs: FOIL and related systems. *New Generation Computing*, 13(3,4): 287-312.
247. Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2): 257-286.
248. Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359-380. Academic Press, San Diego, CA.
249. Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.
250. Riesbeck, C. and Schank, R. (1989). *Inside Case-Based Reasoning*. Lawrence Erlbaum, Hillsdale, NJ.
251. Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal ACM*, 12(1): 23-41.
252. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6): 386-408.
253. Rosenschein, S.J. and Kaelbling, L.P. (1995). A situated view of representation and control. *Artificial Intelligence*, 73: 149-173.
254. Rubinstein, R.Y. (1981). *Simulation and the Monte Carlo Method*. John Wiley and Sons.
255. Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. In D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed*

- Processing, chapter 8, pp. 318-362. MIT Press, Cambridge, MA. Reprinted in [\[Shavlik and Dietterich \(1990\)\]](#).
256. Russell, B. (1917). *Mysticism and Logic and Other Essays*. G. Allen and Unwin, London.
257. Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Series in Artificial Intelligence. Prentice-Hall, Englewood Cliffs, NJ, third edition.
<http://aima.cs.berkeley.edu/>.
258. Russell, S. (1997). Rationality and intelligence. *Artificial Intelligence*, 94: 57-77.
259. Sacerdoti, E.D. (1975). The nonlinear nature of plans. In *Proc. 4th International Joint Conf. on Artificial Intelligence*, pp. 206-214. Tbilisi, Georgia, USSR.
260. Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3(3): 210-229.
<http://www.research.ibm.com/journal/rd/033/ibmrd0303B.pdf>.
261. Sandholm, T. (2007). Expressive commerce and its application to sourcing: How we conducted \$35 billion of generalized combinatorial auctions. *AI Magazine*, 28(3): 45-58.
262. Savage, L.J. (1972). *The Foundation of Statistics*. Dover, New York, 2nd edition.
263. Schank, R.C. (1990). What is AI, anyway? In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence*, pp. 3-13. Cambridge University Press, Cambridge, England.
264. Schapire, R.E. (2002). The boosting approach to machine learning: An overview. In *MSRI Workshop on Nonlinear Estimation and Classification*. Springer Verlag.
<http://www.cs.princeton.edu/~schapire/boost.html>.
265. Schubert, L.K. (1990). Monotonic solutions to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H.E. Kyburg, R.P. Loui, and G.N. Carlson (Eds.), *Knowledge Representation and Defeasible Reasoning*, pp. 23-67. Kluwer Academic Press, Boston, MA.
266. Schueller A., *Programming with Robots*, Whitman College, 2011.
267. Shachter, R. and Peot, M.A. (1992). Decision making using probabilistic inference methods. In *Proc. Eighth Conf. on Uncertainty in Artificial Intelligence (UAI-92)*, pp. 276-283. Stanford, CA.
268. Shafer, G. and Pearl, J. (Eds.) (1990). *Readings in Uncertain Reasoning*. Morgan Kaufmann, San Mateo, CA.
269. Shanahan, M. (1989). Prediction is deduction, but explanation is abduction. In *Proc. 11th International Joint Conf. on Artificial Intelligence (IJCAI-89)*, pp. 1055-1060. Detroit, MI.
270. Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, Cambridge, MA.
271. Shapiro, S.C. (Ed.) (1992). *Encyclopedia of Artificial Intelligence*. Wiley, New York, second edition.
272. Sharkey, N. (2008). The ethical frontiers of robotics. *Science*, 322(5909): 1800 - 1801. [DOI: 10.1126/science.1164582](https://doi.org/10.1126/science.1164582).

273. Shavlik, J.W. and Dietterich, T.G. (Eds.) (1990). *Readings in Machine Learning*. Morgan Kaufmann, San Mateo, CA.
274. Shelley, M.W. (1818). *Frankenstein; or, The Modern Prometheus*. Lackington, Hughes, Harding, Mavor and Jones, London.
275. Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press.
276. Simon, H.A. (1995). Artificial intelligence: an empirical science. *Artificial Intelligence*, 77(1): 95-127.
277. Simon, H. (1996). *The Sciences of the Artificial*. MIT Press, Cambridge, MA, third edition.
278. Singer, P.W. (2009a). Robots at war: The new battlefield. *The Wilson Quarterly*.
www.wilsoncenter.org/index.cfm?fuseaction=wq.essay&essay_id=496613.
279. Singer, P.W. (2009b). *Wired for War: The Robotics Revolution and Conflict in the 21st Century*. Penguin, New York.
280. Smith, B. (2003). Ontology. In L. Floridi (Ed.), *Blackwell Guide to the Philosophy of Computing and Information*, pp. 155--166. Oxford: Blackwell.
http://ontology.buffalo.edu/smith/articles/ontology_pic.pdf.
281. Smith, B.C. (1991). The owl and the electric encyclopedia. *Artificial Intelligence*, 47: 251-288.
282. Smith, B.C. (1996). *On the Origin of Objects*. MIT Press, Cambridge, MA.
283. Somerville, M. (2006). *The Ethical Imagination: Journeys of the Human Spirit*. House of Anansi Press, Toronto.
284. Spall, J.C. (2003). *Introduction to Stochastic Search and Optimization: Estimation, Simulation*. Wiley.
285. Spiegelhalter, D.J., Franklin, R.C.G., and Bull, K. (1990). Assessment, criticism and improvement of imprecise subjective probabilities for a medical expert system. In M. Henrion, R.D. Shachter, L. Kanal, and J. Lemmer (Eds.), *Uncertainty in Artificial Intelligence 5*, pp. 285-294. North-Holland, Amsterdam, The Netherlands.
286. Spirtes, P., Glymour, C., and Scheines, R. (2000). *Causation, Prediction, and Search*. MIT Press, Cambridge MA, 2nd edition.
287. Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MIT Press, Cambridge, MA.
288. Stillings, N.A., Feinstein, M.H., Garfield, J.L., Rissland, E.L., Rosenbaum, D.A., Weisler, S.E., and Baker-Ward, L. (1987). *Cognitive Science: An Introduction*. MIT Press, Cambridge, MA.
289. Stone, P. (2007). Learning and multiagent reasoning for autonomous agents. In *The 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 13-30.
<http://www.cs.utexas.edu/~pstone/Papers/bib2html-links/IJCAI07-award.pdf>.
290. Stone, P. and Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8: 345-383.
291. Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

292. Tarski, A. (1956). *Logic, Semantics, Metamathematics*. Clarendon Press, Oxford, England. Papers from 1923 to 1938 collected and translated by J. H. Woodger.
293. Tate, A. (1977). Generating project networks. In *Proc. 5th International Joint Conf. on Artificial Intelligence*, pp. 888-893. Cambridge, MA.
294. Tharp, T. (2003). *The Creative Habit: Learn It and Use It for Life*. Simon and Schuster.
295. Thrun, S. (2006). Winning the darpa grand challenge. In *Innovative Applications of Artificial Intelligence Conference, (IAAI-06)*, pp. 16-20. Boston, MA.
296. Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics*. MIT Press, Cambridge, MA.
297. Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59: 433-460. Reprinted in [\[Haugeland \(1997\)\]](#).
298. Tversky, A. and Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, 185: 1124-1131.
299. Valiant, L.G. (1984). A theory of the learnable. *Communications of the ACM*, 27: 1134-1142. Reprinted in [\[Shavlik and Dietterich \(1990\)\]](#).
300. van Beek, P. and Chen, X. (1999). Cplan: A constraint programming approach to planning. In *AAAI-99*, pp. 585-590.
301. van Emden, M.H. and Kowalski, R.A. (1976). The semantics of predicate logic as a programming language. *Journal ACM*, 23(4): 733-742.
302. Visser, U. and Burkhard, H.D. (2007). Robocup: 10 years of achievements and challenges. *AI Magazine*, 28(2): 115-130.
303. Viswanathan, P., Mackworth, A.K., Little, J.J., and Mihailidis, A. (2007). Intelligent wheelchairs: Collision avoidance and navigation assistance for older adults with cognitive impairment. In *Proc. Workshop on Intelligent Systems for Assisted Cognition*. Rochester, NY,.
304. Waldinger, R. (1977). Achieving several goals simultaneously. In E. Elcock and D. Michie (Eds.), *Machine Intelligence 8: Machine Representations of Knowledge*, pp. 94-136. Ellis Horwood, Chichester, England.
305. Walsh, T. (2007). Representing and reasoning with preferences. *AI Magazine*, 28(4): 59-69.
306. Warren, D.H.D. and Pereira, F.C.N. (1982). An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8(3-4): 110 - 122. <http://portal.acm.org/citation.cfm?id=972944>.
307. Webber, B.L. and Nilsson, N.J. (Eds.) (1981). *Readings in Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.
308. Weiss, G. (Ed.) (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA.
309. Weiss, S. and Kulikowski, C. (1991). *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufmann, San Mateo, CA.

310. Weld, D. (1999). Recent advances in AI planning. *AI Magazine*, 20(2).
<http://www.cs.washington.edu/homes/weld/papers/pi2.pdf>.
311. Weld, D.S. (1992). Qualitative physics: Albatross or eagle? *Computational Intelligence*, 8(2): 175-186. Introduction to special issue on the future of qualitative physics.
312. Weld, D.S. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4): 27-61.
313. Weld, D. and de Kleer, J. (Eds.) (1990). *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, San Mateo, CA.
314. Whitley, D. (2001). An overview of evolutionary algorithms. *Journal of Information and Software Technology*, 43: 817-831. <http://www.cs.colostate.edu/genitor/2001/overview.pdf>.
315. Wilkins, D.E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA.
316. Winograd, T. (1990). Thinking machines: Can there be? Are we? In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence: A Sourcebook*, pp. 167-189. Cambridge University Press, Cambridge, England.
317. Winograd, T. (1972). *Understanding Natural Language*. Academic Press, New York.
318. Woods, W.A. (2007). Meaning and links. *AI Magazine*, 28(4): 71-92.
319. Wooldridge, M. (2002). *An Introduction to Multiagent Systems*. John Wiley and Sons, Chichester, England.
320. Yang, Q. (1997). *Intelligent Planning: A Decomposition and Abstraction-Based Approach*. Springer-Verlag, New York.
321. Yang, S. and Mackworth, A.K. (2007). Hierarchical shortest pathfinding applied to route-planning for wheelchair users. In *Proc. Canadian Conf. on Artificial Intelligence, AI-2007*. Montreal, PQ.
322. Zhang, N.L. and Poole, D. (1994). A simple approach to Bayesian network computations. In *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, pp. 171-178.
323. Zhang, N.L. (2004). Hierarchical latent class models for cluster analysis. *Journal of Machine Learning Research*, 5(6): 697-723.
324. Zhang, Y. and Mackworth, A.K. (1995). Constraint nets: A semantic model for hybrid dynamic systems. *Theoretical Computer Science*, 138: 211-239.
325. Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3): 73-83.
326. Albus, J., and Proctor, F.G., "A Reference Model Architecture for Intelligent Hybrid Control Systems," proceedings of the *International Federation of Automatic Control*, San Francisco, CA, June 30-July 5, 1996.
327. Allocca, J. A., and Stuart, A., *Transducers: Theory and Application*, Prentice-Hall, 1984.
328. Anderson, M.O., McKay, M.D., Richardson, B.S., "Multirobot Automated Indoor Floor Characterization Team," proceedings of the *1996 IEEE International Conference on*

329. Anderson, T.L., and Donath, M., “Animal Behavior as a Paradigm for Developing Robot Autonomy,” *Robotics and Autonomous Systems*, vol. 6, 1990, pp. 145–186.
330. Arbib, M., “Perceptual Structures and Distributed Motor Control,” *Handbook of Physiology—The Nervous System II*, ed. Brooks, pp. 1449–1465, 1981.
331. Arbib, M., “Schema Theory,” *The Handbook of Brain Theory and Neural Networks*,
332. Michael A. Arbib, ed., MIT Press, pp. 830–834.
333. Arbib, M., and Liaw, J.-S., “Sensorimotor transformations in the worlds of frogs and toads,” *Computational Theories of Interaction and Agency*, Philip Agre and Stanley J. Rosenschein, ed., MIT Press, 1996, pp. 53–80.
334. Arkin, R.C., “Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-Made Environments,” COINS Technical Report 87-80, *Computer and Information Science*, University of Massachusetts at Amherst, 1987.
335. Arkin, R., personal communication, 1988.
336. Arkin, R., *Behavior-Based Robotics*, MIT Press, 1998.
337. Arkin, R., and MacKenzie, D., “Temporal Coordination of Perceptual Algorithms for Mobile Robot Navigation,” *IEEE Transactions on Robotics and Automation*, vol. 10, no. 3, June, 1994, pp. 276–286.
338. Arkin, R.C. and Murphy, R.R., “Autonomous Navigation in a Manufacturing Environment,” *IEEE Transactions on Robotics and Automation*, vol. 6, no. 4, August, 1990, pp. 445–454.

- 339.Arkin, R. C., Riseman, E. M., and Hansen, A., "AuRA: An Architecture for Vision- Based Robot Navigation," proceedings of the *DARPA Image Understanding Work- shop*, Los Angeles, CA, February, 1987, pp. 417–413.*Artificial Intelligence and Mobile Robots*, D. Kortenkamp, P. Bonasso, R. Murphy, edi- tors, MIT Press, 1998.
- 340.Asimov, I., *I, Robot*, Bantam Books, 1994 (reprint).
- 341.Balch, T., "Social Entropy: a New Metric for Learning Multi-Robot Teams," proc.
- 342.*10th International FLAIRS Conference*, 1997.
- 343.Balch, T.R. and Arkin R.C., "Avoiding the Past: A Simple but Effective Strategy for Reactive Navigation," *IEEE International Conference on Robotics and Automation*, 1993.
- 344.Balch, T., and Arkin, R.C., "Behavior-Based Formation Control for Multirobot Teams," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, 1999, pp. 926– 939.
- 345.Balch, T., Boone, G., Collins, T., Forbes, H., MacKenzie, D., and Santamaria, J., "Io, Ganymede, and Callisto: A Multiagent Robot Trash-Collecting Team," *AI Maga- zine*, vol. 16, no. 2, 1995, pp. 39–51.
- 346.Barlow, J.B., "General Principles: The Senses Considered as Physical Instruments," *The Senses*, ed. Barlow, J.B., and Mollon, J.B., Cambridge University Press, 1982, pp. 1–34.
- 347.Bonasso, R.P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., and Slack, M., "A Proven Three tiered Architecture for Programming Autonomous Robots," *Journal of Exper- imental and Theoretical Artificial Intelligence*, vol. 9, no. 2, 1997.
- 348.Borenstein, J., and Koren, Y., "The Vector Field Histogram- Fast Obstacle Avoidance for Mobile Robots," *IEEE Transactions on Robotics and Automation*, vol.7, no.3, June, 1991, pp. 278–288.
- 349.Borenstein, J., and Koren, Y., "Histogramic In-Motion Mapping for Mobile Robot Obstacle Avoidance," *IEEE Transactions on Robotics and Automation*, vol.7, no.4, Au- gust, 1991, pp. 535–539.
- 350.Bower, T.G.R., "The Evolution of Sensory Systems," *Perception: Essays in Honor of James J. Gibson*, ed. by R.B. MacLeod and H.L. Pick, Jr., Cornell University Press, Ithaca, NY, 1974, pp. 141–153.
- 351.Braitenberg, V., *Vehicles: Experiments in Synthetic Psychology*, MIT Press, 1984.
- 352.Brooks, R., "Challenges for Complete Creature Architectures," proceedings of *First Intenational Conference on Simulation of Adaptive Behavior*, pp. 434–443.
- 353.Brooks, R.A., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation* (now *IEEE Transactions on Robotics and Automation*), vol. 1, no. 1, 1986, pp. 1–10.

354. Brooks, R., *Cambrian Intelligence: The Early History of the New AI*, MIT Press, 1999.
355. Brooks, R., Breazeal, C., Irie, R., Kemp, C., Marjanovic, M., Scassellati, B., Williamson, M., "Alternative Essences on Intelligence," proceedings *AAAI-98*, 1998, pp. 961–968.
356. Brooks, R., and Flynn, A., "Fast, Cheap and Out of Control," AI Memo 1182, MIT AI Laboratory, December, 1989.
357. Cai, A.; Fukuda, T.; Arai, F., "Information sharing among multiple robots for cooperation in cellular robotic system," proceedings *1997 IEEE/RSJ International Conference on Intelligent Robot and Systems*, vol. 3, pp. 1768–74.
358. Cattaneo, M., Cavalchini, A.G., Rogonoi, G.L., "Design and construction of a robotic milking system," Sixth International Conference on Computers in Agriculture, Cancun, Mexico, June 1996, pp. 155–60.
359. Cervantes-Perez, F., "Visuomotor Coordination in Frogs and Toads," *The Handbook of Brain Theory and Neural Networks*, Michael A. Arbib, ed., MIT Press, pp. 1036–1042.
360. Choset, H. and Burdick, J., "Sensor Based Motion Planning: Incremental Construction of the Hierarchical Generalized Voronoi Graph," To appear in *International Journal of Robotics Research*, 2000.
361. Choset, H. and Burdick, J., "Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph," to appear in *International Journal of Robotics Research*, 2000.
362. Choset, H. and Burdick, J., "Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph," to appear in the *International Journal of Robotics Research*.
363. *Concise International Encyclopedia of Robotics: Applications and Automation*, Richard C. Dorf, editor-in-chief, John Wiley and Sons, Inc., NY, 1990.
364. Congdon, C., Huber, M., Kortenkamp, D., Konolige, K., Myers, K., Saffiotti, A., and Ruspini, E., "Carmel vs. Flakey: A Comparison of Two Winners," *AI Magazine*, vol. 14, no. 1, Spring 1993, pp. 49–57.
365. Connell, J., *Minimalist Mobile Robotics: A Colony Architecture for an Artificial Creature*, Academic Press, 1990.
366. Connolly, C.I., and Grupen, R.A., "The applications of harmonic functions to robotics," *Journal of Robotic Systems*, vol. 10, October 1993, pp. 931–946.
367. Dean, T., Allen, J., and Aloimonis, Y., *Artificial Intelligence: Theory and Practice*, Benjamin/Cummings Publishing, 1995.
368. Dean, T., Bonasso, R.P., "1992 AAAI Robot Exhibition and Competition," *AI Magazine*,

vol. 14, no. 1, Spring 1993, pp. 35–48.

369. Dean, T., and Wellman, M., *Planning and Control*, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

- 370.deBerg, M., vanKreveld, M., Overmars, M., Schwarzkoph, O., *Computational Geometry: Theory and Applications*, Springer-Verlag, 1999.
- 371.Dickmanns, E.D. and Graefe, V., "Dynamic Monocular Machine Vision," *Int. J. Machine Vision and Applications*, vol. 1, 1988, pp. 223–240.
- 372.daaper. Draper, B., Collins, J., Brolio, A. Hanson and E. Riseman. "The Schema Sys- tem," *International Journal of Computer Vision*, vol. 2, 1989, pp. 209–250.
- 373.Drumheller, M., "Mobile Robot Localization Using Sonar", *IEEE Transactions on Pat- tern Analysis and Machine Intelligence*, vol. 9, no . 2, Mar 1987, pp. 325–332.
- 374.Duchon, A.P., and Warren, W.H., "Robot Navigation form a Gibsonian Viewpoint," *1994 IEEE International conference on Systems, Man, and Cybernetics*, vol. 3, pp. 2272– 7.
- 375.Edwards, G.R., Burnard, R.H., Bewley, W.L., and Bullock, B.L., "The Ground Vehicle Manager's Associate," *Tech Report AIAA-94-1249-CP*, 1994, pp. 520–526.
- 376.Engelson, S. P., and McDermott, D. V., "Passive Robot Map Building with Explo- ration Scripts," *Technical Report: YALEU/DCS/TR-898*, Yale University, Dept. of Computer Science, March, 1992.
- 377.Erickson, J., "Supervised Space Robots are Needed in Space Exploration," *Conference on Intelligent Robotics in Field, Factory, Service, and Space, NASA Conference Publica- tion 3251*, vol. 2, 1994, pp. 705–716.
- 378.Everett, H.R., *Sensors for Mobile Robots: Theory and Application*, A. K. Peters, Ltd, Wellesley, MA, 1995.
- 379.Firby, R.J., Kahn, R.E., Prokopwicz, P.N., and Swain, M.J., "An Architecture for Vision and Action," proceedings of *1995 International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Pub., San Mateo, CA, vol. 1, 1995, pp. 72–79.
- 380.Firby, R.J., Prokopwicz, P.N., Swain, M.J., Kahn, R.E., and Franklin, D., "Program- ming CHIP for the IJCAI-95 Robot Competition," *AI Magazine*, Spring 1996, pp. 71–81.
- 381.Fowler, M., Scot, K., Booch, G., *UML Distilled*, 2nd ed., Addison-Wesley, 1999.

382. Gardner, H., *The Mind's New Science*, Basic Books, Inc., New York, 1987.
383. Gat, E., "Reliable Goal-directed Reactive Control for Real-World Autonomous Mobile Robots," Ph.D. dissertation, Dept. of Computer Science, Virginia Polytechnic Institute and State University, 1991.
384. Gat, E., "Three-layer Architectures," *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonasson, R. Murphy, editors, MIT Press, 1998.
385. Gibson, J.J., *The Ecological Approach to Visual Perception*, Houghton Mifflin, Boston MA, 1979.

Additional References

386. Nocks, Lisa (2007). *The robot : the life story of a technology*. Westport, CT: Greenwood Publishing Group.
387. Zunt, Dominik. ["Who did actually invent the word "robot" and what does it mean?"](#). The Karel Čapek website. Archived from [the original](#) on 2013-01-23. Retrieved 2017-02-05.
388. [Asimov, Isaac](#) (1996) [1995]. "The Robot Chronicles". Gold. London: Voyager. pp. 224–225. [ISBN 0-00-648202-3](#).
389. [Asimov, Isaac](#) (1983). "4 The Word I Invented". [Counting the Eons](#). Doubleday. *Robotics has become a sufficiently well developed technology to warrant articles and books on its history and I have watched this in amazement, and in some disbelief, because I invented ... the word*
390. ["Robotics: About the Exhibition"](#). The Tech Museum of Innovation. Archived from [the original](#) on 2008-09-13. Retrieved 2008-09-15.
391. [Needham, Joseph](#) (1991). *Science and Civilisation in China: Volume 2, History of Scientific Thought*. Cambridge University Press. [ISBN 0-521-05800-7](#).
392. Fowler, Charles B. (October 1967). "The Museum of Music: A History of Mechanical Instruments". *Music Educators Journal*. **54** (2): 45–49. [doi:10.2307/3391092](#). [JSTOR 3391092](#).
393. Rosheim, Mark E. (1994). *Robot Evolution: The Development of Anthrobotics*. Wiley-IEEE. pp. 9–10. [ISBN 0-471-02622-0](#).
394. [al-Jazari \(Islamic artist\)](#), [Encyclopædia Britannica](#).
395. PhD, Renato M.E. Sabbatini, ["Sabbatini, RME: An Imitation of Life: The First Robots"](#).
396. Waurzyniak, Patrick (2006). ["Masters of Manufacturing: Joseph F. Engelberger"](#). Society of Manufacturing Engineers. **137** (1). Archived from [the original](#) on 2011-11-09.
397. ["Humanoid History -WABOT-"](#). [www.humanoid.waseda.ac.jp](#).

398. Zeghloul, Saïd; Laribi, Med Amine; Gazeau, Jean-Pierre (21 September 2015). [*Robotics and Mechatronics: Proceedings of the 4th IFToMM International Symposium on Robotics and Mechatronics*](#). Springer – via Google Books.
399. ["Historical Android Projects"](#). androidworld.com.
400. [Robots: From Science Fiction to Technological Revolution](#), page 130
401. Duffy, Vincent G. (19 April 2016). ["Handbook of Digital Human Modeling: Research for Applied Ergonomics and Human Factors Engineering"](#). CRC Press – via Google Books.
402. ["KUKA Industrial Robot FAMULUS"](#). Retrieved 2008-01-10.
403. ["History of Industrial Robots" \(PDF\)](#). Archived from [the original](#) (PDF) on 2012-12-24. Retrieved 2012-10-27.
404. ["Communist Robot ::"](#).
405. ["OUCL Robot Sheepdog Project"](#).
406. Pinto, Jim (October 1, 2003). ["Fully automated factories approach reality"](#). AutomationWorld.
407. ["パナソニック電工株式会社 – 住まいと暮らし、制御機器、電子材料の総合メーカー – Panasonic"](#). Archived from [the original](#) on 2007-09-28.
408. [\[1\]](#) ^[dead link]
409. ["At Hong Kong High-Tech Cafe, Everything Is Served With Microchips"](#).
410. ["Communist Robot ::"](#).
411. ["Robots may force chefs out of the kitchen"](#).
412. ["Meet Boris, the robot that can load a dishwasher"](#). 3 October 2014.
413. ["AI-driven robot makes 'perfect' flatbread"](#). IoT Hub.
414. One database, developed by the [United States Department of Energy](#) contains information on almost 500 existing robotic technologies and can be found on the [D&D Knowledge Management Information Tool](#).
415. ["UIUC Agricultural Engineering - Faculty and Staff"](#). Archived from [the original](#) on 2007-05-04.
416. ["service-robots.org – agriculture & harvesting"](#). Archived from [the original](#) on 2007-03-23.
417. Dowling, Kevin. ["Power Sources for Small Robots" \(PDF\)](#). Carnegie Mellon University. Retrieved 11 May 2012.
418. Bi-directional series-parallel elastic actuator and overlap of the actuation layers Raphaël Furnémont¹, Glenn Mathijssen^{1,2}, Tom Verstraten¹, Dirk Lefeber¹ and Bram Vanderborght¹ Published 26 January 2016 • © 2016 IOP Publishing Ltd
419. ["CiteSeerX — Series Elastic Actuators for legged robots"](#). Citeseerx.ist.psu.edu. Retrieved 2010-11-27.
420. www.imagesco.com, Images SI Inc -. ["Air Muscle actuators, going further, page 6"](#).
421. ["Air Muscles"](#). Shadow Robot. Archived from [the original](#) on 2007-09-27.
422. Tondy, Bertrand (2012). "Modelling of the McKibben artificial muscle: A review". *Journal of Intelligent Material Systems and Structures*. **23** (3): 225–253. doi:10.1177/1045389X11435435.
423. ["TALKING ELECTRONICS Nitinol Page-1"](#). Talkingelectronics.com. Retrieved 2010-11-27.
424. ["If205. Hardware: Building a Linux-controlled walking robot"](#). Ibiblio.org. 2001-11-01. Retrieved 2010-11-27.
425. ["WW-EAP and Artificial Muscles"](#). Eap.jpl.nasa.gov. Retrieved 2010-11-27.
426. ["Empa – a117-2-eap"](#). Empa.ch. Retrieved 2010-11-27.
427. ["Electroactive Polymers \(EAP\) as Artificial Muscles \(EPAM\) for Robot Applications"](#). Hizook. Retrieved 2010-11-27.
428. ["Piezo LEGS – -09-26"](#).

429. ["Squiggle Motors: Overview"](#). Retrieved 2007-10-08.
430. Nishibori; et al. (2003). ["Robot Hand with Fingers Using Vibration-Type Ultrasonic Motors \(Driving Characteristics\)"](#). *Journal of Robotics and Mechatronics*. Retrieved 2007-10-09.
431. Otake; et al. (2001). ["Shape Design of Gel Robots made of Electroactive Polymer trolol Gel" \(PDF\)](#). Retrieved 2007-10-16.
432. John D. Madden, 2007, /science.1146351
433. ["Syntouch LLC: BioTac\(R\) Biomimetic Tactile Sensor Array"](#). Retrieved 2009-08-10.
434. Wettels, N; Santos, VJ; Johansson, RS; Loeb, Gerald E.; et al. (2008). "Biomimetic tactile sensor array". *Advanced Robotics*. **22** (8): 829–849.
[doi:10.1163/156855308X314533](#).
435. ["What is The SmartHand?"](#). SmartHand Project. Retrieved 4 February 2011.
436. ["What is a robotic end-effector?"](#). ATI Industrial Automation. 2007. Retrieved 2007-10-16.
437. Crane, Carl D.; Joseph Duffy (1998). ["Kinematic Analysis of Robot Manipulators"](#). Cambridge University Press. [ISBN 0-521-57063-8](#). Retrieved 2007-10-16.
438. G.J. Monkman, S. Hesse, R. Steinmann & H. Schunk – Robot Grippers – Wiley, Berlin 2007
 - I. Ben-Gal, Y. Bukchin, O. Goldstain (2007). ["Remote learning for the manipulation and control of robotic cells" \(PDF\)](#). *European Journal of Engineering Education* 32 (4), 481–494, 2007.
439. ["Annotated Mythbusters: Episode 78: Ninja Myths – Walking on Water, Catching a Sword, Catching an Arrow"](#). (Discovery Channel's Mythbusters making mechanical gripper from chain and metal wire)
440. [Robonaut hand](#)
441. ["Delft hand"](#). [TU Delft](#).
442. M&C. ["TU Delft ontwikkelt goedkope, voorzichtige robohand"](#).
443. ["astrictive definition – English definition dictionary – Reverso"](#).
444. Tijssma, H. A.; Liefhebber, F.; Herder, J. L. (1 June 2005). ["Evaluation of new user interface features for the MANUS robot arm" \(PDF\)](#). pp. 258–263.
[doi:10.1109/ICORR.2005.1501097](#) – via IEEE Xplore.
445. Allcock, Andrew (2006). ["Anthropomorphic hand is almost human"](#). *Machinery*. Archived from [the original](#) on 2007-09-28. Retrieved 2007-10-17.
446. ["Welcome"](#).
447. ["T.O.B.B"](#). [Mtoussaint.de](#). Retrieved 2010-11-27.
448. ["nBot, a two wheel balancing robot"](#). [Geology.heroy.smu.edu](#). Retrieved 2010-11-27.
449. ["ROBONAUT Activity Report"](#). [NASA](#). 2004. Archived from [the original](#) on 2007-08-20. Retrieved 2007-10-20.
450. ["IEEE Spectrum: A Robot That Balances on a Ball"](#). [Spectrum.ieee.org](#). Retrieved 2010-11-27.
451. ["Carnegie Mellon Researchers Develop New Type of Mobile Robot That Balances and Moves on a Ball Instead of Legs or Wheels" \(Press release\)](#). Carnegie Mellon. 2006-08-09. Archived from [the original](#) on 2007-06-09. Retrieved 2007-10-20.
452. ["Spherical Robot Can Climb Over Obstacles"](#). [BotJunkie](#). Retrieved 2010-11-27.
453. ["Rotundus"](#). [Rotundus.se](#). Retrieved 2010-11-27.
454. ["OrbSwarm Gets A Brain"](#). [BotJunkie](#). 2007-07-11. Retrieved 2010-11-27.
455. ["Rolling Orbital Bluetooth Operated Thing"](#). [BotJunkie](#). Retrieved 2010-11-27.
456. ["Swarm"](#). [Orbswarm.com](#). Retrieved 2010-11-27.
457. ["The Ball Bot : Johnnytronic@Sun"](#). [Blogs.sun.com](#). Archived from [the original](#) on 2011-10-13. Retrieved 2010-11-27.

458. ["Senior Design Projects | College of Engineering & Applied Science| University of Colorado at Boulder"](#). Engineering.colorado.edu. 2008-04-30. Retrieved 2010-11-27.
459. ["JPL Robotics: System: Commercial Rovers"](#).
460. ["AMBER Lab"](#).
461. ["Micromagic Systems Robotics Lab"](#).
462. ["AMRU-5 hexapod robot"](#) (PDF).
463. ["Achieving Stable Walking"](#). Honda Worldwide. Retrieved 2007-10-22.
464. ["Funny Walk"](#). Pooter Geek. 2004-12-28. Retrieved 2007-10-22.
465. ["ASIMO's Pimp Shuffle"](#). Popular Science. 2007-01-09. Retrieved 2007-10-22.
466. ["The Temple of VTEC – Honda and Acura Enthusiasts Online Forums > Robot Shows Prime Minister How to Loosen Up > > A drunk robot?"](#).
467. ["3D One-Leg Hopper \(1983–1984\)"](#). MIT Leg Laboratory. Retrieved 2007-10-22.
468. ["3D Biped \(1989–1995\)"](#). MIT Leg Laboratory.
469. ["Quadruped \(1984–1987\)"](#). MIT Leg Laboratory.
470. ["MIT Leg Lab Robots- Main"](#).
471. ["About the robots"](#). Anybots. Archived from [the original](#) on 2007-09-09. Retrieved 2007-10-23.
472. ["Homepage"](#). Anybots. Retrieved 2007-10-23.
473. ["Dexter Jumps video"](#). YouTube. 2007-03-01. Retrieved 2007-10-23.
474. Collins, Steve; Wisse, Martijn; Ruina, Andy; Tedrake, Russ (2005-02-11). ["Efficient bipedal robots based on passive-dynamic Walkers"](#) (PDF). Science. **307** (5712): 1082–1085. doi:[10.1126/science.1107799](#). PMID [15718465](#). Archived from [the original](#) (PDF) on 2007-06-22. Retrieved 2007-09-11.
475. Collins, Steve; Ruina, Andy. ["A bipedal walking robot with efficient and human-like gait"](#) (PDF). Proc. IEEE International Conference on Robotics and Automation.
476. ["Testing the Limits"](#) (PDF). Boeing. p. 29. Retrieved 2008-04-09.
477. Miller, Gavin. ["Introduction"](#). snakerobots.com. Retrieved 2007-10-22.
478. ["ACM-R5"](#). Archived from [the original](#) on 2011-10-11.
479. ["Swimming snake robot \(commentary in Japanese\)"](#).
480. ["Commercialized Quadruped Walking Vehicle "TITAN VII"'"](#). Hirose Fukushima Robotics Lab. Archived from [the original](#) on 2007-11-06. Retrieved 2007-10-23.
481. ["Plen, the robot that skates across your desk"](#). SCI FI Tech. 2007-01-23. Archived from [the original](#) on 2007-10-11. Retrieved 2007-10-23.
482. [Capuchin](#) on [YouTube](#)
483. [Wallbot](#) on [YouTube](#)
484. [Stanford University: Stickybot](#) on [YouTube](#)
485. Sfakiotakis; et al. (1999). ["Review of Fish Swimming Modes for Aquatic Locomotion"](#) (PDF). IEEE Journal of Oceanic Engineering. Archived from [the original](#) (PDF) on 2007-09-26. Retrieved 2007-10-24.
486. Richard Mason. ["What is the market for robot fish?"](#). Archived from [the original](#) on 2009-07-04.
487. ["Robotic fish powered by Gumstix PC and PIC"](#). Human Centred Robotics Group at Essex University. Retrieved 2007-10-25.
488. Witoon Juwarahawong. ["Fish Robot"](#). Institute of Field Robotics. Archived from [the original](#) on 2007-11-04. Retrieved 2007-10-25.
489. ["YouTube"](#).
490. ["High-Speed Robotic Fish | iSplash"](#). isplash-robot. Retrieved 2017-01-07.
491. ["iSplash-II: Realizing Fast Carangiform Swimming to Outperform a Real Fish"](#) (PDF). Robotics Group at Essex University. Retrieved 2015-09-29.

492. ["iSplash-I: High Performance Swimming Motion of a Carangiform Robotic Fish with Full-Body Coordination"](#) (PDF). Robotics Group at Essex University. Retrieved 2015-09-29.
493. Jaulin, L.; Le Bars, F. (2012). ["An interval approach for stability analysis; Application to sailboat robotics"](#) (PDF). *IEEE Transaction on Robotics*. 27 (5).
494. J. Norberto Pires, (2005). "Robot-by-voice: experiments on commanding an industrial robot using the human voice", *Industrial Robot: An International Journal*, Vol. 32, Issue 6, pp. 505–511, [doi:10.1108/01439910510629244](#). Available: [online](#) and [pdf](#)
495. ["Survey of the State of the Art in Human Language Technology: 1.2: Speech Recognition"](#). Archived from [the original](#) on 2007-11-11.
496. Fournier, Randolph Scott., and B. June. Schmidt. "Voice Input Technology: Learning Style and Attitude Toward Its Use." *Delta Pi Epsilon Journal* 37 (1995): 1_12.
497. ["History of Speech & Voice Recognition and Transcription Software"](#). *Dragon Naturally Speaking*. Retrieved 2007-10-27.
498. M.L. Walters, D.S. Syrdal, K.L. Koay, K. Dautenhahn, R. te Boekhorst, (2008). *Human approach distances to a mechanical-looking robot with different robot voice styles*. In: Proceedings of the 17th IEEE International Symposium on Robot and Human Interactive Communication, 2008. RO-MAN 2008, Munich, 1–3 Aug. 2008, pp. 707–712, [doi:10.1109/ROMAN.2008.4600750](#). Available: [online](#) and [pdf Archived](#) 2011-07-18 at the [Wayback Machine](#).
499. Sandra Pauletto, Tristan Bowles, (2010). *Designing the emotional content of a robotic speech signal*. In: Proceedings of the 5th Audio Mostly Conference: A Conference on Interaction with Sound, New York, [ISBN 978-1-4503-0046-9](#), [doi:10.1145/1859799.1859804](#). Available: [online](#)
500. Tristan Bowles, Sandra Pauletto, (2010). *Emotions in the Voice: Humanising a Robotic Voice*. In: Proceedings of the 7th Sound and Music Computing Conference, Barcelona, Spain.
501. Waldherr, Romero & Thrun (2000). ["A Gesture Based Interface for Human-Robot Interaction"](#) (PDF). Kluwer Academic Publishers. Retrieved 2007-10-28.
502. Markus Kohler. ["Vision Based Hand Gesture Recognition Systems"](#). University of Dortmund. Archived from [the original](#) on 2012-07-11. Retrieved 2007-10-28.
503. ["Frubber facial expressions"](#). Archived from [the original](#) on 2009-02-07.
504. ["Best Inventions of 2008 – TIME"](#). 29 October 2008 – via [www.time.com](#).
505. ["Kismet: Robot at MIT's AI Lab Interacts With Humans"](#). Sam Ogden. Archived from [the original](#) on 2007-10-12. Retrieved 2007-10-28.
506. ["\(Park et al. 2005\) Synthetic Personality in Robots and its Effect on Human-Robot Relationship"](#) (PDF).
507. ["Robot Receptionist Dishes Directions and Attitude"](#).
508. ["New Scientist: A good robot has personality but not looks"](#) (PDF). Archived from [the original](#) (PDF) on 2006-09-29.
509. ["Playtime with Pleo, your robotic dinosaur friend"](#).
510. Jennifer Bogo (October 31, 2014). ["Meet a woman who trains robots for a living"](#).
511. ["A Ping-Pong-Playing Terminator"](#). *Popular Science*.
512. [NOVA](#) conversation with Professor Moravec, October, 1997. [NOVA Online](#)
513. Sandhana, Lakshmi (2002-09-05). [A Theory of Evolution, for Robots](#). *Wired Magazine*. Retrieved 2007-10-28.
514. [Experimental Evolution In Robots Probes The Emergence Of Biological Communication](#). *Science Daily*. 2007-02-24. Retrieved 2007-10-28.
515. Žlajpah, Leon (2008-12-15). "Simulation in robotics". *Mathematics and Computers in Simulation*. 79 (4): 879–897. [doi:10.1016/j.matcom.2008.02.017](#).
516. News, Technology Research. ["Evolution trains robot teams TRN 051904"](#). [www.trnmag.com](#).

517. Agarwal, P.K. *Elements of Physics XI*. Rastogi Publications. p. 2. [ISBN 978-81-7133-911-2](#).
518. ["Career: Robotics Engineer"](#). Princeton Review. 2012. Retrieved 2012-01-27.
519. Saad, Ashraf; Kroutil, Ryan (2012). *Hands-on Learning of Programming Concepts Using Robotics for Middle and High School Students*. *Proceedings of the 50th Annual Southeast Regional Conference of the Association for Computer Machinery*. ACM. pp. 361–362. [doi:10.1145/2184512.2184605](#).
520. ["Robotics Degree Programs at Worcester Polytechnic Institute"](#). Worcester Polytechnic Institute. 2013. Retrieved 2013-04-12.
521. ["B.E.S.T. Robotics"](#).
522. ["LEGO® Building & Robotics After School Programs"](#). Retrieved 5 November 2014.
523. Toy, Tommy (June 29, 2011). ["Outlook for robotics and Automation for 2011 and beyond are excellent says expert"](#). PBT Consulting. Retrieved 2012-01-27.
524. ["The future of employment: How susceptible are jobs to computerisation?"](#). *Technological Forecasting and Social Change*. **114**: 254–280. 2017-01-01. [doi:10.1016/j.techfore.2016.08.019](#). [ISSN 0040-1625](#).
525. E McGaughey, 'Will Robots Automate Your Job Away? Full Employment, Basic Income, and Economic Democracy' (2018) [SSRN, part 2\(3\)](#). DH Autor, 'Why Are There Still So Many Jobs? The History and Future of Workplace Automation' (2015) 29(3) *Journal of Economic Perspectives* 3.
526. ["Focal Points Seminar on review articles in the future of work – Safety and health at work – EU-OSHA"](#). [osha.europa.eu](#). Retrieved 2016-04-19.
527. ["Robotics: Redefining crime prevention, public safety and security"](#). [SourceSecurity.com](#).
528. [""Draft Standard for Intelligent Assist Devices — Personnel Safety Requirements"" \(PDF\)](#).
529. ["ISO/TS 15066:2016 – Robots and robotic devices – Collaborative robots"](#).