

6 CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Chapters 3 and 4 explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is atomic, or indivisible—a black box with no internal structure.

This chapter describes a way to solve a wide variety of problems more efficiently. We use a **factored representation** for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.

CSP search algorithms take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

CONSTRAINT
SATISFACTION
PROBLEM

6.1 DEFINING CONSTRAINT SATISFACTION PROBLEMS

A constraint satisfaction problem consists of three components, X , D , and C :

X is a set of variables, $\{X_1, \dots, X_n\}$.

D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.

C is a set of constraints that specify allowable combinations of values.

Each domain D_i consists of a set of allowable values, $\{v_1, \dots, v_k\}$ for variable X_i . Each constraint C_i consists of a pair $\langle \text{scope}, \text{rel} \rangle$, where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. For example, if X_1 and X_2 both have

ASSIGNMENT
CONSISTENT
COMPLETE
ASSIGNMENT
SOLUTION
PARTIAL
ASSIGNMENT

the domain $\{A, B\}$, then the constraint saying the two variables must have different values can be written as $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$.

To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \dots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

6.1.1 Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

The domain of each variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}.$$

There are many possible solutions to this problem, such as

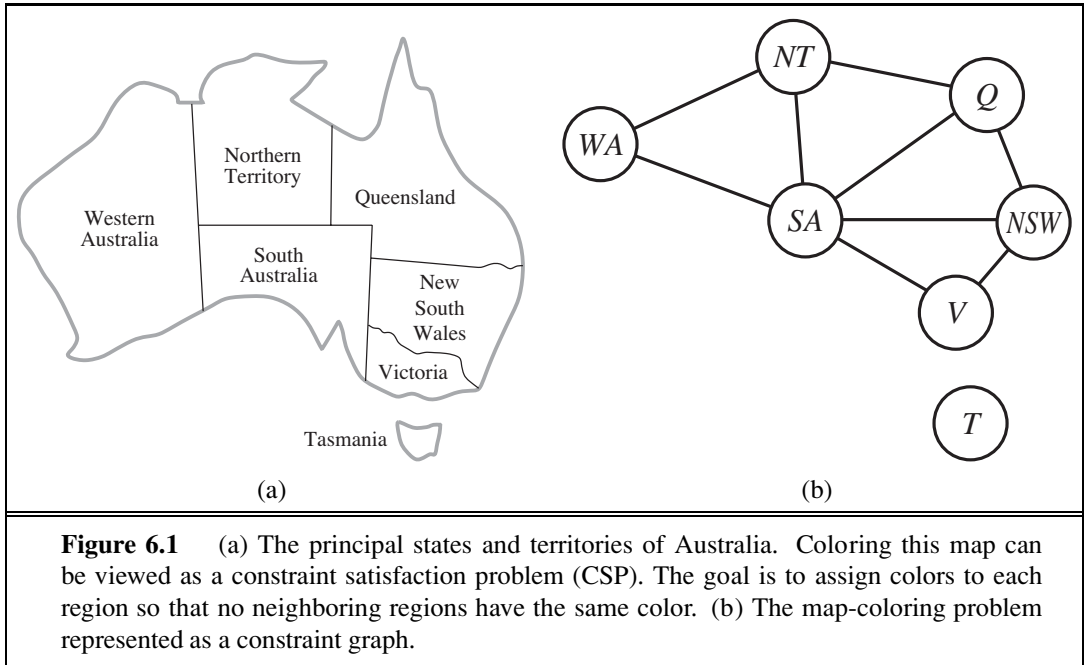
$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

CONSTRAINT GRAPH

It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique. In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space. For example, once we have chosen $\{SA = blue\}$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value *blue*. Without taking advantage of constraint propagation, a search procedure would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraint propagation we never have to consider *blue* as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.

In regular state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment is not a solution, we can



immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

6.1.2 Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

$$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}.$$

The value of each variable is the time that the task starts. Next we represent **precedence constraints** between individual tasks. Whenever a task T_1 must occur before task T_2 , and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form

$$T_1 + d_1 \leq T_2.$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$\begin{aligned} Axle_F + 10 &\leq Wheel_{RF}; & Axle_F + 10 &\leq Wheel_{LF}; \\ Axle_B + 10 &\leq Wheel_{RB}; & Axle_B + 10 &\leq Wheel_{LB}. \end{aligned}$$

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$\begin{aligned} Wheel_{RF} + 1 &\leq Nuts_{RF}; & Nuts_{RF} + 2 &\leq Cap_{RF}; \\ Wheel_{LF} + 1 &\leq Nuts_{LF}; & Nuts_{LF} + 2 &\leq Cap_{LF}; \\ Wheel_{RB} + 1 &\leq Nuts_{RB}; & Nuts_{RB} + 2 &\leq Cap_{RB}; \\ Wheel_{LB} + 1 &\leq Nuts_{LB}; & Nuts_{LB} + 2 &\leq Cap_{LB}. \end{aligned}$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:

$$(Axle_F + 10 \leq Axle_B) \quad \text{or} \quad (Axle_B + 10 \leq Axle_F).$$

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_B$ can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except *Inspect* we add a constraint of the form $X + d_X \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{1, 2, 3, \dots, 27\}.$$

This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used, as discussed in Chapter 11.

6.1.3 Variations on the CSP formalism

DISCRETE DOMAIN
FINITE DOMAIN

The simplest kind of CSP involves variables that have **discrete, finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables Q_1, \dots, Q_8 are the positions of each queen in columns $1, \dots, 8$ and each variable has the domain $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

INFINITE

A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn't put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used that understands constraints such as $T_1 + d_1 \leq T_2$ directly, without enumerating the set of pairs of allowable values for (T_1, T_2) . Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables.

CONSTRAINT
LANGUAGE

LINEAR
CONSTRAINTS

NONLINEAR
CONSTRAINTS

CONTINUOUS
DOMAINS

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

UNARY CONSTRAINT

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint $\langle (SA), SA \neq \text{green} \rangle$

BINARY CONSTRAINT

A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 6.1(b).

GLOBAL
CONSTRAINT

We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z , with the ternary constraint $Between(X, Y, Z)$.

CRYPTARITHMETIC

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because it need not involve *all* the variables in a problem). One of the most common global constraints is *Alldiff*, which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row or column must satisfy an *Alldiff* constraint. Another example is provided by **cryptarithmic** puzzles. (See Figure 6.2(a).) Each letter in a cryptarithmic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint $Alldiff(F, T, U, W, R, O)$. The addition constraints on the four columns of the puzzle can be written as the following n -ary constraints:

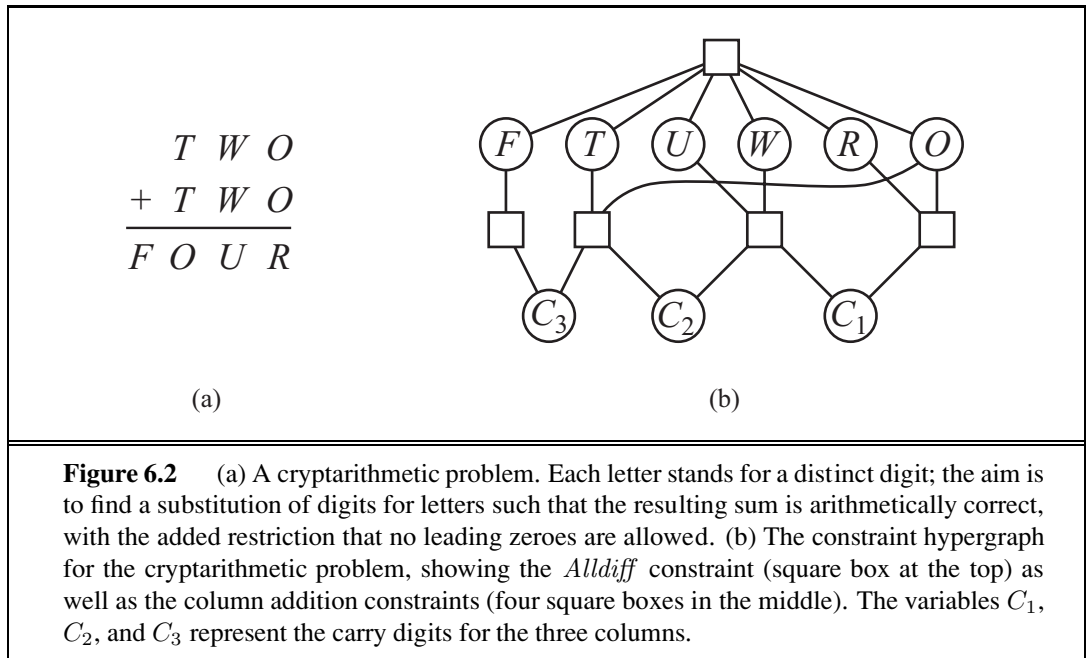
$$\begin{aligned} O + O &= R + 10 \cdot C_{10} \\ C_{10} + W + W &= U + 10 \cdot C_{100} \\ C_{100} + T + T &= O + 10 \cdot C_{1000} \\ C_{1000} &= F, \end{aligned}$$

CONSTRAINT
HYPERGRAPH

where C_{10} , C_{100} , and C_{1000} are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n -ary constraints.

DUAL GRAPH

Alternatively, as Exercise 6.6 asks you to prove, every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced, so we could transform any CSP into one with only binary constraints; this makes the algorithms simpler. Another way to convert an n -ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and



one binary constraint for each pair of constraints in the original graph that share variables. For example, if the original graph has variables $\{X, Y, Z\}$ and constraints $\langle (X, Y, Z), C_1 \rangle$ and $\langle (X, Y), C_2 \rangle$ then the dual graph would have variables $\{C_1, C_2\}$ with the binary constraint $\langle (X, Y), R_1 \rangle$, where (X, Y) are the shared variables and R_1 is a new relation that defines the constraint between the shared variables, as specified by the original C_1 and C_2 .

There are however two reasons why we might prefer a global constraint such as *AllDiff* rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using *AllDiff*. Second, it is possible to design special-purpose inference algorithms for global constraints that are not available for a set of more primitive constraints. We describe these inference algorithms in Section 6.2.5.

The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constraint optimization problem**, or COP. Linear programming problems do this kind of optimization.

6.2 CONSTRAINT PROPAGATION: INFERENCE IN CSPs

INFERENCE
CONSTRAINT
PROPAGATION

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

LOCAL
CONSISTENCY

The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 6.1(b)) and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

6.2.1 Node consistency

NODE CONSISTENCY

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable *SA* starts with domain $\{red, green, blue\}$, and we can make it node consistent by eliminating *green*, leaving *SA* with the reduced domain $\{red, blue\}$. We say that a network is node-consistent if every variable in the network is node-consistent.

It is always possible to eliminate all the unary constraints in a CSP by running node consistency. It is also possible to transform all n -ary constraints into binary ones (see Exercise 6.6). Because of this, it is common to define CSP solvers that work with only binary constraints; we make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

ARC CONSISTENCY

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j) . A network is arc-consistent if every variable is arc consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits. We can write this constraint explicitly as

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9)\} \rangle .$$

To make X arc-consistent with respect to Y , we reduce X 's domain to $\{0, 1, 2, 3\}$. If we also make Y arc-consistent with respect to X , then Y 's domain becomes $\{0, 1, 4, 9\}$ and the whole CSP is arc-consistent.

On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA, WA) :

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

```

function AC-3(csp) returns false if an inconsistency is found and true otherwise
inputs: csp, a binary CSP with components ( $X$ ,  $D$ ,  $C$ )
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
  ( $X_i$ ,  $X_j$ )  $\leftarrow$  REMOVE-FIRST(queue)
  if REVISE(csp,  $X_i$ ,  $X_j$ ) then
    if size of  $D_i$  = 0 then return false
    for each  $X_k$  in  $X_i$ .NEIGHBORS -  $\{X_j\}$  do
      add ( $X_k$ ,  $X_i$ ) to queue
return true

```

```

function REVISE(csp,  $X_i$ ,  $X_j$ ) returns true iff we revise the domain of  $X_i$ 
  revised  $\leftarrow$  false
  for each  $x$  in  $D_i$  do
    if no value  $y$  in  $D_j$  allows  $(x,y)$  to satisfy the constraint between  $X_i$  and  $X_j$  then
      delete  $x$  from  $D_i$ 
      revised  $\leftarrow$  true
  return revised

```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name “AC-3” was used by the algorithm’s inventor (Mackworth, 1977) because it’s the third version developed in the paper.

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

The most popular algorithm for arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. (Actually, the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue.) Initially, the queue contains all the arcs in the CSP. AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j . If this leaves D_i unchanged, the algorithm just moves on to the next arc. But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i . We need to do that because the change in D_i might enable further reductions in the domains of D_k , even if we have previously considered X_k . If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

The complexity of AC-3 can be analyzed as follows. Assume a CSP with n variables, each with domain size at most d , and with c binary constraints (arcs). Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete. Checking

consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.¹

GENERALIZED ARC
CONSISTENT

It is possible to extend the notion of arc consistency to handle n -ary rather than just binary constraints; this is called generalized arc consistency or sometimes hyperarc consistency, depending on the author. A variable X_i is **generalized arc consistent** with respect to an n -ary constraint if for every value v in the domain of X_i there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its X_i component equal to v . For example, if all variables have the domain $\{0, 1, 2, 3\}$, then to make the variable X consistent with the constraint $X < Y < Z$, we would have to eliminate 2 and 3 from the domain of X because the constraint cannot be satisfied when X is 2 or 3.

6.2.3 Path consistency

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences. Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone.

PATH CONSISTENCY

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. This is called path consistency because one can think of it as looking at a path from X_i to X_j with X_m in the middle.

Let's see how path consistency fares in coloring the Australia map with two colors. We will make the set $\{WA, SA\}$ path consistent with respect to NT . We start by enumerating the consistent assignments to the set. In this case, there are only two: $\{WA = red, SA = blue\}$ and $\{WA = blue, SA = red\}$. We can see that with both of these assignments NT can be neither *red* nor *blue* (because it would conflict with either WA or SA). Because there is no valid choice for NT , we eliminate both assignments, and we end up with no valid assignments for $\{WA, SA\}$. Therefore, we know that there can be no solution to this problem. The PC-2 algorithm (Mackworth, 1977) achieves path consistency in much the same way that AC-3 achieves arc consistency. Because it is so similar, we do not show it here.

¹ The AC-4 algorithm (Mohr and Henderson, 1986) runs in $O(cd^2)$ worst-case time but can be slower than AC-3 on average cases. See Exercise 6.13.

6.2.4 *K*-consistency

K-CONSISTENCY

Stronger forms of propagation can be defined with the notion of *k-consistency*. A CSP is *k-consistent* if, for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k th variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint networks, 3-consistency is the same as path consistency.

STRONGLY
K-CONSISTENT

A CSP is **strongly *k-consistent*** if it is *k-consistent* and is also $(k - 1)$ -consistent, $(k - 2)$ -consistent, \dots all the way down to 1-consistent. Now suppose we have a CSP with n nodes and make it strongly n -consistent (i.e., strongly *k-consistent* for $k = n$). We can then solve the problem as follows: First, we choose a consistent value for X_1 . We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i , we need only search through the d values in the domain to find a value consistent with X_1, \dots, X_{i-1} . We are guaranteed to find a solution in time $O(n^2d)$. Of course, there is no free lunch: any algorithm for establishing n -consistency must take time exponential in n in the worst case. Worse, n -consistency also requires space that is exponential in n . The memory issue is even more severe than the time. In practice, determining the appropriate level of consistency checking is mostly an empirical science. It can be said practitioners commonly compute 2-consistency and less commonly 3-consistency.

6.2.5 Global constraints

Remember that a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmic problem above and Sudoku puzzles below). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment $\{WA = red, NSW = red\}$ for Figure 6.1. Notice that the variables SA , NT , and Q are effectively connected by an *Alldiff* constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domain of each variable is reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints. There are more

RESOURCE
CONSTRAINT

complex inference algorithms for *Alldiff* (see van Hoeve and Katriel, 2006) that propagate more constraints but are more computationally expensive to run.

Another important higher-order constraint is the **resource constraint**, sometimes called the *atmost* constraint. For example, in a scheduling problem, let P_1, \dots, P_4 denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $Atmost(10, P_1, P_2, P_3, P_4)$. We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the *Atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

BOUNDS
PROPAGATION

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let's suppose there are two flights, F_1 and F_2 , for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

$$D_1 = [0, 165] \quad \text{and} \quad D_2 = [0, 385] .$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$. Propagating bounds constraints, we reduce the domains to

$$D_1 = [35, 165] \quad \text{and} \quad D_2 = [255, 385] .$$

BOUNDS
CONSISTENT

We say that a CSP is **bounds consistent** if for every variable X , and for both the lower-bound and upper-bound values of X , there exists some value of Y that satisfies the constraint between X and Y for every variable Y . This kind of bounds propagation is widely used in practical constraint problems.

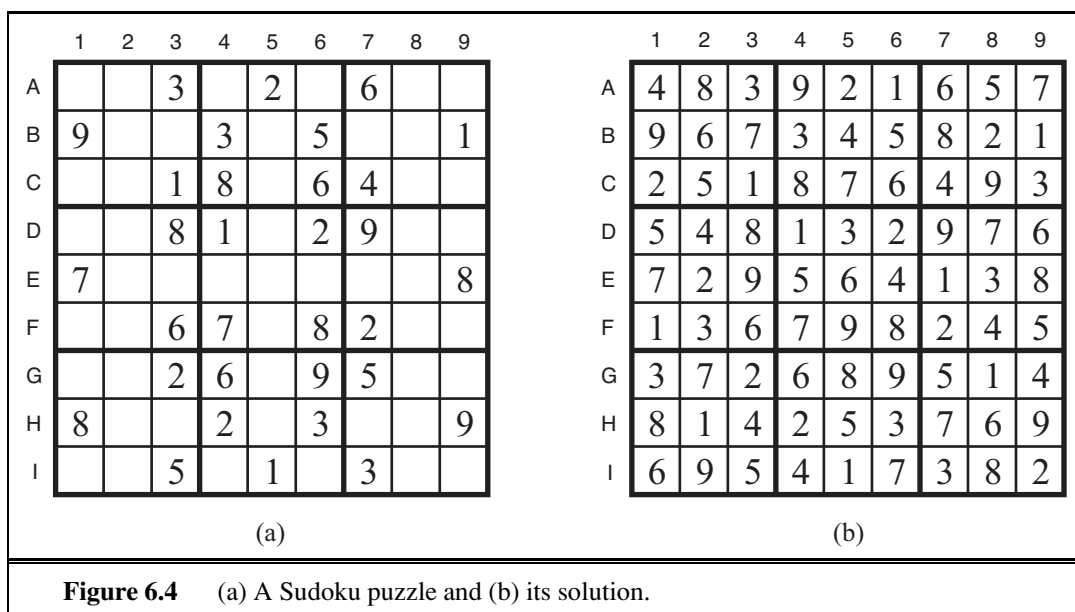
6.2.6 Sudoku example

SUDOKU

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names $A1$ through $A9$ for the top row (left to right), down to $I1$ through $I9$ for the bottom row. The empty squares have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different



Alldiff constraints: one for each row, column, and box of 9 squares.

$Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$
 $Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)$
 ...
 $Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)$
 $Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)$
 ...
 $Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)$
 $Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)$
 ...

Let us see how far arc consistency can take us. Assume that the *Alldiff* constraints have been expanded into binary constraints (such as $A1 \neq A2$) so that we can apply the AC-3 algorithm directly. Consider variable *E6* from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove not only 2 and 8 but also 1 and 7 from *E6*'s domain. From the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3. That leaves *E6* with a domain of {4}; in other words, we know the answer for *E6*. Now consider variable *I6*—the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know *E6* must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with *I5*, and we are left with only the value 7 in the domain of *I6*. Now there are 8 known values in column 6, so arc consistency can infer that *A6* must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).

Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a

mechanical application of AC-3, and indeed AC-3 works only for the easiest Sudoku puzzles. Slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle. To solve the hardest puzzles and to make efficient progress, we will have to be more clever.

Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as “naked triples.” That strategy works as follows: in any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. For example, the three domains might be $\{1, 8\}$, $\{3, 8\}$, and $\{1, 3, 8\}$. From that we don’t know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 *Alldiff* constraints. But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of *Alldiff* constraints and has nothing to do with Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

6.3 BACKTRACKING SEARCH FOR CSPs

Sudoku problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution. In this section we look at backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

We could apply a standard depth-limited search (from Chapter 3). A state would be a partial assignment, and an action would be adding $var = value$ to the assignment. But for a CSP with n variables of domain size d , we quickly notice something terrible: the branching factor at the top level is nd because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n - 1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider a *single* variable at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between $SA = red$, $SA = green$, and $SA = blue$, but we would never choose between $SA = red$ and $WA = blue$. With this restriction, the number of leaves is d^n , as we would hope.

```

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
      remove {var = value} and inferences from assignment
  return failure

```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k -consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

BACKTRACKING
SEARCH

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT, Q, \dots . Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones, as described on page 87.

In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?

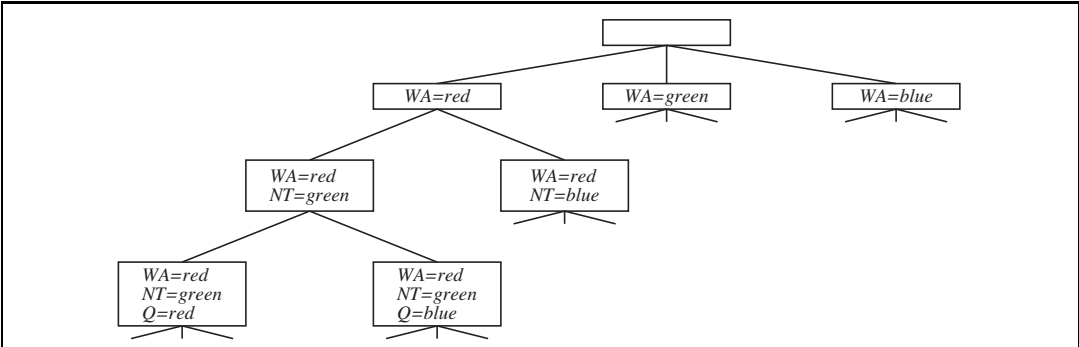


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

The subsections that follow answer each of these questions in turn.

6.3.1 Variable and value ordering

The backtracking algorithm contains the line

var ← SELECT-UNASSIGNED-VARIABLE(*csp*) .

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, $\{X_1, X_2, \dots\}$. This static variable ordering seldom results in the most efficient search. For example, after the assignments for $WA = red$ and $NT = green$ in Figure 6.6, there is only one possible value for SA , so it makes sense to assign $SA = blue$ next rather than assigning Q . In fact, after SA is assigned, the choices for Q , NSW , and V are all forced. This intuitive idea—choosing the variable with the fewest “legal” values—is called the **minimum-remaining-values** (MRV) heuristic. It also has been called the “most constrained variable” or “fail-first” heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

The MRV heuristic doesn’t help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 6.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T , which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose *any* consistent color at each choice point and still arrive at a solution with no backtracking. The minimum-remaining-

MINIMUM-
REMAINING-VALUES

DEGREE HEURISTIC

LEAST-
CONSTRAINING-
VALUE

values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 6.1 we have generated the partial assignment with $WA = red$ and $NT = green$ and that our next choice is for Q . Blue would be a bad choice because it eliminates the last legal value left for Q 's neighbor, SA . The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

6.3.2 Interleaving search and inference

So far we have seen how AC-3 and other algorithms can infer reductions in the domain of variables *before* we begin the search. But inference can be even more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

FORWARD
CHECKING

One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X . Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA = red$ and $Q = green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q . A second point to notice is that after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning $\{WA = red\}$. Intuitively, it seems that that assignment constrains its neighbors, NT and SA , so we should handle those

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA= <i>red</i>	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After Q= <i>green</i>	Ⓡ	B	Ⓢ	R B	R G B	B	R G B
After V= <i>blue</i>	Ⓡ	B	Ⓢ	R	Ⓟ		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. *WA = red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q = green* is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After *V = blue* is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

variables next, and then all the other variables will fall into place. That’s exactly what happens with MRV: *NT* and *SA* have two values, so one of them is chosen first, then the other, then *Q*, *NSW*, and *V* in order. Finally *T* still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it makes the current variable arc-consistent, but doesn’t look ahead and make all the other variables arc-consistent. For example, consider the third row of Figure 6.7. It shows that when *WA* is *red* and *Q* is *green*, both *NT* and *SA* are forced to be blue. Forward checking does not look far enough ahead to notice that this is an inconsistency: *NT* and *SA* are adjacent and so cannot have the same value.

MAINTAINING ARC
CONSISTENCY (MAC)

The algorithm called MAC (for **M**aintaining **A**rc **C**onsistency (**MAC**)) detects this inconsistency. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i . From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC’s queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 6.5 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited. In this subsection, we consider better possibilities.

CHRONOLOGICAL
BACKTRACKING

Consider what happens when we apply simple backtracking in Figure 6.1 with a fixed variable ordering *Q*, *NSW*, *V*, *T*, *SA*, *WA*, *NT*. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, *SA*, we see that every value violates a constraint. We back up to *T* and try a new color for

Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia.

CONFLICT SET
BACKJUMPING

A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of *SA* impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for *SA*. The set (in this case $\{Q = \text{red}, NSW = \text{green}, V = \text{blue}, \}$), is called the **conflict set** for *SA*. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for *V*. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment $X = x$ deletes a value from *Y*'s domain, it should add $X = x$ to *Y*'s conflict set. If the last value is deleted from *Y*'s domain, then the assignments in the conflict set of *Y* are added to the conflict set of *X*. Then, when we get to *Y*, we know immediately where to backtrack if needed.

The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every* branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = \text{red}, NSW = \text{red}\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = \text{red}$ next and then assign *NT*, *Q*, *V*, *SA*. We know that no assignment can work for these last four variables, so eventually we run out of values to try at *NT*. Now, the question is, where to backtrack? Backjumping cannot work, because *NT* *does* have values consistent with the preceding assigned variables—*NT* doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables *NT*, *Q*, *V*, and *SA*, *taken together*, failed because of a set of preceding variables, which must be those variables that directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as *NT*: it is that set of preceding variables that caused *NT*, *together with any subsequent variables*, to have no consistent solution. In this case, the set is *WA* and *NSW*, so the algorithm should backtrack to *NSW* and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

CONFLICT-DIRECTED
BACKJUMPING

We must now explain how these new conflict sets are computed. The method is in fact quite simple. The “terminal” failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, *SA* fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to *Q*, and *Q* *absorbs*

the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onward, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT , the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW , as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$, and set

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_j\}.$$

When we reach a contradiction, backjumping can tell us how far to back up, so we don't waste time changing variables that won't fix the problem. But we would also like to avoid running into the same problem again. When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

For example, consider the state $\{WA = red, NT = green, Q = blue\}$ in the bottom row of Figure 6.6. Forward checking can tell us this state is a no-good because there is no valid assignment to SA . In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again. But suppose that the search tree in Figure 6.6 were actually part of a larger search tree that started by first assigning values for V and T . Then it would be worthwhile to record $\{WA = red, NT = green, Q = blue\}$ as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T .

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

6.4 LOCAL SEARCH FOR CSPs

Local search algorithms (see Section 4.1) turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. For example, in the 8-queens problem (see Figure 4.3), the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column. Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints.²

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts**

² Local search can easily be extended to constraint optimization problems (COPs). In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

CONSTRAINT
LEARNING

NO-GOOD

MIN-CONFLICTS

```

function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max_steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure

```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

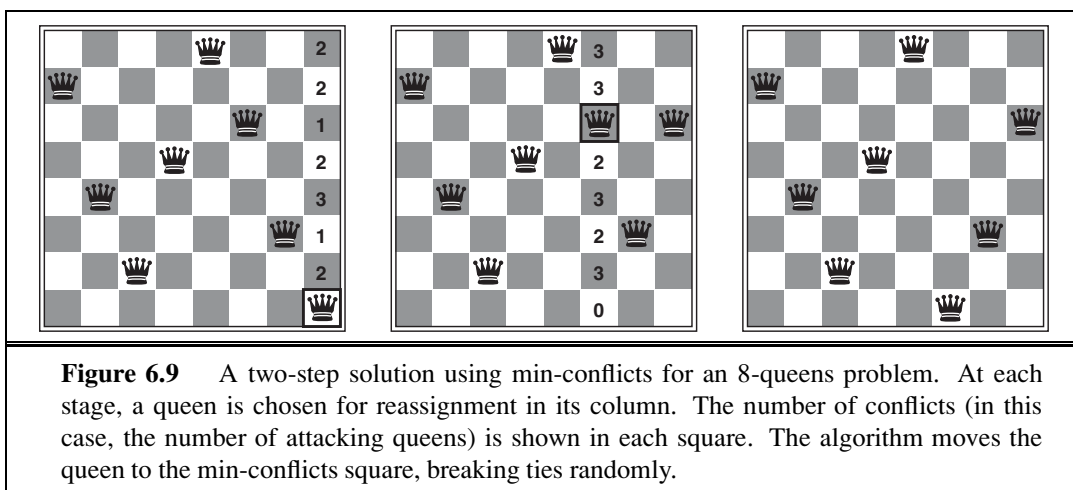


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

heuristic. The algorithm is shown in Figure 6.8 and its application to an 8-queens problem is diagrammed in Figure 6.9.

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n -queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Chapter 7. Roughly speaking, n -queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective. The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaux. There may be millions of variable assignments that are only one conflict away from a solution. Plateau search—allowing sideways moves to another state with the same score—can help local search find its way off this plateau. This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states. Simulated annealing can also be used to escape from plateaux.

CONSTRAINT
WEIGHTING

Another technique, called **constraint weighting**, can help concentrate the search on the important constraints. Each constraint is given a numeric weight, W_i , initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment. This has two benefits: it adds topography to plateaux, making sure that it is possible to improve from the current state, and it also, over time, adds weight to the constraints that are proving difficult to solve.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

6.5 THE STRUCTURE OF PROBLEMS

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many subproblems. Looking again at the constraint graph for Australia (Figure 6.1(b), repeated as Figure 6.12(a)), one fact stands out: Tasmania is not connected to the mainland.³ Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i . If assignment S_i is a solution of CSP_i , then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Consider the following: suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve,

INDEPENDENT
SUBPROBLEMS

CONNECTED
COMPONENT

³ A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

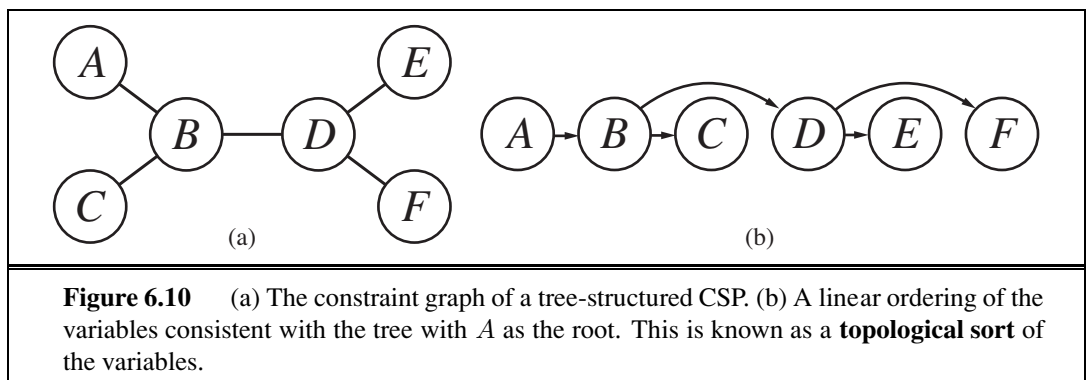
where d is the size of the domain. Hence, the total work is $O(d^c n/c)$, which is *linear* in n ; without the decomposition, the total work is $O(d^n)$, which is exponential in n . Let's make this more concrete: dividing a Boolean CSP with 80 variables into four subproblems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a **tree** when any two variables are connected by only one path. We show that *any tree-structured CSP can be solved in time linear in the number of variables*.⁴ The key is a new notion of consistency, called **directed arc consistency** or DAC. A CSP is defined to be directed arc-consistent under an ordering of variables X_1, X_2, \dots, X_n if and only if every X_i is arc-consistent with each X_j for $j > i$.

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort**. Figure 6.10(a) shows a sample tree and (b) shows one possible ordering. Any tree with n nodes has $n - 1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each link from a parent to its child is arc consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. That means we won't have to backtrack; we can move linearly through the variables. The complete algorithm is shown in Figure 6.11.



TOPOLOGICAL SORT



Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 6.12(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and

⁴ Sadly, very few regions of the world have tree-structured maps, although Sulawesi comes close.

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components  $X$ ,  $D$ ,  $C$ 

   $n \leftarrow$  number of variables in  $X$ 
  assignment  $\leftarrow$  an empty assignment
  root  $\leftarrow$  any variable in  $X$ 
   $X \leftarrow \text{TOPOLOGICALSORT}(X, \text{root})$ 
  for  $j = n$  down to 2 do
    MAKE-ARC-CONSISTENT(PARENT( $X_j$ ),  $X_j$ )
    if it cannot be made consistent then return failure
  for  $i = 1$  to  $n$  do
    assignment[ $X_i$ ]  $\leftarrow$  any consistent value from  $D_i$ 
    if there is no consistent value then return failure
  return assignment

```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

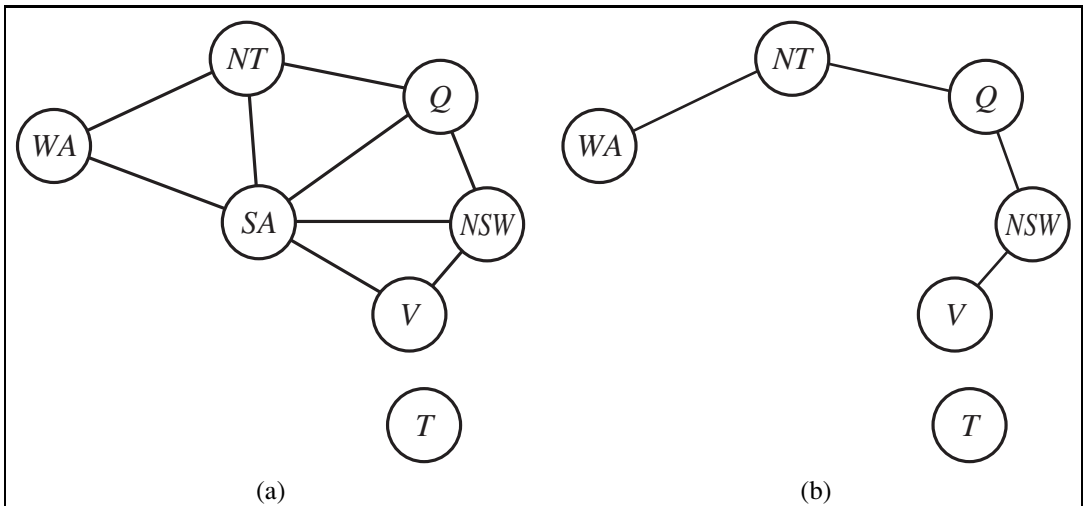


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of SA .

deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA .

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA . (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for SA could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

CYCLE CUTSET

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S . S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S ,
 - (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S , and
 - (b) If the remaining CSP has a solution, return it together with the assignment for S .

If the cycle cutset has size c , then the total run time is $O(d^c \cdot (n - c)d^2)$: we have to try each of the d^c combinations of values for the variables in S , and for each combination we must solve a tree problem of size $n - c$. If the graph is “nearly a tree,” then c will be small and the savings over straight backtracking will be huge. In the worst case, however, c can be as large as $(n - 2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known. The overall algorithmic approach is called **cutset conditioning**; it comes up again in Chapter 14, where it is used for reasoning about probabilities.

CUTSET
CONDITIONINGTREE
DECOMPOSITION

The second approach is based on constructing a **tree decomposition** of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 6.13 shows a tree decomposition of the map-coloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

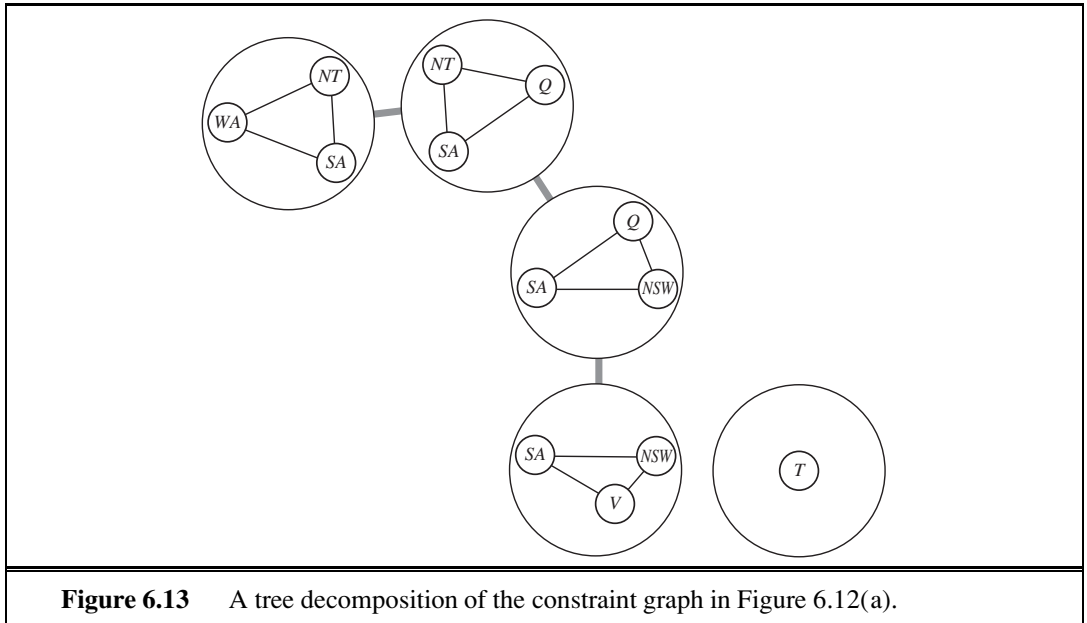
- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 6.13. You can verify from Figure 6.12 that this decomposition makes sense.

We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct a global solution as follows. First, we view each subproblem as a “mega-variable” whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblem in Figure 6.13 is a map-coloring problem with three variables and hence has six solutions—one is $\{WA = \text{red}, SA = \text{blue}, NT = \text{green}\}$. Then, we solve the constraints connecting the subproblems, using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = \text{red}, SA = \text{blue}, NT = \text{green}\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = \text{blue}, NT = \text{green}, Q = \text{red}\}$.

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree

TREE WIDTH



decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{w+1})$ time. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time*. Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

So far, we have looked at the structure of the constraint graph. There can be important structure in the *values* of variables as well. Consider the map-coloring problem with n colors. For every consistent solution, there is actually a set of $n!$ solutions formed by permuting the color names. For example, on the Australia map we know that WA , NT , and SA must all have different colors, but there are $3! = 6$ ways to assign the three colors to these three regions. This is called **value symmetry**. We would like to reduce the search space by a factor of $n!$ by breaking the symmetry. We do this by introducing a **symmetry-breaking constraint**. For our example, we might impose an arbitrary ordering constraint, $NT < SA < WA$, that requires the three values to be in alphabetical order. This constraint ensures that only one of the $n!$ solutions is possible: $\{NT = \text{blue}, SA = \text{green}, WA = \text{red}\}$.

For map coloring, it was easy to find a constraint that eliminates the symmetry, and in general it is possible to find constraints that eliminate all but one symmetric solution in polynomial time, but it is NP-hard to eliminate all symmetry among intermediate sets of values during search. In practice, breaking value symmetry has proved to be important and effective on a wide range of problems.



6.6 SUMMARY

- **Constraint satisfaction problems** (CSPs) represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.
- A number of inference techniques use the constraints to infer which variable/value pairs are consistent and which are not. These include node, arc, path, and k -consistency.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- The **minimum-remaining-values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in deciding which value to try first for a given variable. Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.
- Local search using the **min-conflicts** heuristic has also been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is quite efficient if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

DIOPHANTINE EQUATIONS

The earliest work related to constraint satisfaction dealt largely with numerical constraints. Equational constraints with integer domains were studied by the Indian mathematician Brahmagupta in the seventh century; they are often called **Diophantine equations**, after the Greek mathematician Diophantus (c. 200–284), who actually considered the domain of positive rationals. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

GRAPH COLORING

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. The four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of De Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised by Appel and Haken (1977) (see the book *Four Colors Suffice* (Wilson, 2004)). Purists were disappointed that part of the proof relied on a computer, so Georges Gonthier (2008), using the COQ theorem prover, derived a formal proof that Appel and Haken’s proof was correct.

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was the SKETCHPAD sys-

tem (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 6.6) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences. The bucket-elimination algorithm (Dechter, 1999) can also be applied to optimization problems.

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that, in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint networks and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient arc-consistency algorithm, was developed by Mohr and Henderson (1986). Soon after Mackworth's paper appeared, researchers began experimenting with the tradeoff between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliot (1980) favored the minimal forward-checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc-consistency checking after each variable assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that, on harder CSPs, full arc-consistency checking pays off. Freuder (1978, 1982) investigated the notion of k -consistency and its relationship to the complexity of solving CSPs. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed, and Bessière (2006) presents a current survey.

Special methods for handling higher-order or global constraints were developed first within the context of **constraint logic programming**. Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994), Stergiou and Walsh (1999), and van Hoeve (2001). Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998). A survey of global constraints is provided by van Hoeve and Katriel (2006).

Sudoku has become the most widely known CSP and was described as such by Simonis (2005). Agerbeck and Hansen (2008) describe some of the strategies and show that Sudoku on an $n^2 \times n^2$ board is in the class of *NP*-hard problems. Reeson *et al.* (2007) show an interactive solver based on CSP techniques.

The idea of backtracking search goes back to Golomb and Baumert (1965), and its application to constraint satisfaction is due to Bitner and Reingold (1975), although they trace the basic algorithm back to the 19th century. Bitner and Reingold also introduced the MRV heuristic, which they called the *most-constrained-variable* heuristic. Brelaz (1979) used the degree heuristic as a tiebreaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k -coloring arbitrary graphs. Haralick and Elliot (1980) proposed the least-constraining-value heuristic.

DEPENDENCY-
DIRECTED
BACKTRACKING

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). The most general and powerful form of intelligent backtracking was actually developed very early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** led to the development of **truth maintenance systems** (Doyle, 1979), which we discuss in Section 12.6.2. The connection between the two areas is analyzed by de Kleer (1989).

BACKMARKING

The work of Stallman and Sussman also introduced the idea of **constraint learning**, in which partial results obtained by search can be saved and reused later in the search. The idea was formalized Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately. The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success.

DYNAMIC
BACKTRACKING

Empirical studies of several randomized backtracking methods were done by Gomes *et al.* (2000) and Gomes and Selman (2001). Van Beek (2006) surveys backtracking.

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on simulated annealing (see Chapter 4), which is widely used for scheduling problems. The min-conflicts heuristic was first proposed by Gu (1989) and was developed independently by Minton *et al.* (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n -queens problem led to a reappraisal of the nature and prevalence of “easy” and “hard” problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find “hard” problem instances. We discuss this phenomenon further in Chapter 7. Konolige (1994) showed that local search is inferior to backtracking search on problems with a certain degree of local structure; this led to work that combined local search and inference, such as that by Pinkas and Dechter (1995). Hoos and Tsang (2006) survey local search techniques.

Work relating the structure and complexity of CSPs originates with Freuder (1985), who showed that search on arc consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et al.*, 1983). Bayardo and Miranker (1994) present an algorithm for tree-structured CSPs that runs in linear time without any preprocessing.

Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied a related notion (which they called **induced width**) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 6.5. Drawing on this work and on results

from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, **hypertree width**, that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of “width” in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

Interest in look-back approaches to backtracking was rekindled by the work of Bayardo and Schrag (1997), whose RELSAT algorithm combined constraint learning and backjumping and was shown to outperform many other algorithms of the time. This led to AND/OR search algorithms applicable to both CSPs and probabilistic reasoning (Dechter and Mateescu, 2007). Brown *et al.* (1988) introduce the idea of symmetry breaking in CSPs, and Gent *et al.* (2006) give a recent survey.

The field of **distributed constraint satisfaction** looks at solving CSPs when there is a collection of agents, each of which controls a subset of the constraint variables. There have been annual workshops on this problem since 2000, and good coverage elsewhere (Collin *et al.*, 1999; Pearce *et al.*, 2008; Shoham and Leyton-Brown, 2009).

Comparing CSP algorithms is mostly an empirical science: few theoretical results show that one algorithm dominates another on all problems; instead, we need to run experiments to see which algorithms perform better on typical instances of problems. As Hooker (1995) points out, we need to be careful to distinguish between competitive testing—as occurs in competitions among algorithms based on run time—and scientific testing, whose goal is to identify the properties of an algorithm that determine its efficacy on a class of problems.

The recent textbooks by Apt (2003) and Dechter (2003), and the collection by Rossi *et al.* (2006) are excellent resources on constraint processing. There are several good earlier surveys, including those by Kumar (1992), Dechter and Frost (2002), and Bartak (2001); and the encyclopedia articles by Dechter (1992) and Mackworth (1992). Pearson and Jeavons (1997) survey tractable classes of CSPs, covering both structural decomposition methods and methods that rely on properties of the domains or constraints themselves. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. Constraint programming is covered in the books by Apt (2003) and Fruhwirth and Abdennadher (2003). Several interesting applications are described in the collection edited by Freuder and Mackworth (1994). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal *Constraints*. The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called *CP*.

EXERCISES

6.1 How many solutions are there for the map-coloring problem in Figure 6.1? How many solutions if four colors are allowed? Two colors?

6.2 Consider the problem of placing k knights on an $n \times n$ chessboard such that no two knights are attacking each other, where k is given and $k \leq n^2$.

- a. Choose a CSP formulation. In your formulation, what are the variables?
- b. What are the possible values of each variable?
- c. What sets of variables are constrained, and how?
- d. Now consider the problem of putting *as many knights as possible* on the board without any attacks. Explain how to solve this with local search by defining appropriate ACTIONS and RESULT functions and a sensible objective function.

6.3 Consider the problem of constructing (not solving) crossword puzzles:⁵ fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares by using any subset of the list. Formulate this problem precisely in two ways:

- a. As a general search problem. Choose an appropriate search algorithm and specify a heuristic function. Is it better to fill in blanks one letter at a time or one word at a time?
- b. As a constraint satisfaction problem. Should the variables be words or letters?

Which formulation do you think will be better? Why?

6.4 Give precise formulations for each of the following as constraint satisfaction problems:

- a. Rectilinear floor-planning: find non-overlapping places in a large rectangle for a number of smaller rectangles.
- b. Class scheduling: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.
- c. Hamiltonian tour: given a network of cities connected by roads, choose an order to visit all cities in a country without repeating any.

6.5 Solve the cryptarithmic problem in Figure 6.2 by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics.

6.6 Show how a single ternary constraint such as “ $A + B = C$ ” can be turned into three binary constraints by using an auxiliary variable. You may assume finite domains. (*Hint:* Consider a new variable that takes on values that are pairs of other values, and consider constraints such as “ X is the first element of the pair Y .”) Next, show how constraints with more than three variables can be treated similarly. Finally, show how unary constraints can be eliminated by altering the domains of variables. This completes the demonstration that any CSP can be transformed into a CSP with only binary constraints.

6.7 Consider the following logic puzzle: In five houses, each with a different color, live five persons of different nationalities, each of whom prefers a different brand of candy, a different drink, and a different pet. Given the following facts, the questions to answer are “Where does the zebra live, and in which house do they drink water?”

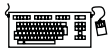
⁵ Ginsberg *et al.* (1990) discuss several methods for constructing crossword puzzles. Littman *et al.* (1999) tackle the harder problem of solving them.

The Englishman lives in the red house.
 The Spaniard owns the dog.
 The Norwegian lives in the first house on the left.
 The green house is immediately to the right of the ivory house.
 The man who eats Hershey bars lives in the house next to the man with the fox.
 Kit Kats are eaten in the yellow house.
 The Norwegian lives next to the blue house.
 The Smarties eater owns snails.
 The Snickers eater drinks orange juice.
 The Ukrainian drinks tea.
 The Japanese eats Milky Ways.
 Kit Kats are eaten in a house next to the house where the horse is kept.
 Coffee is drunk in the green house.
 Milk is drunk in the middle house.

Discuss different representations of this problem as a CSP. Why would one prefer one representation over another?

6.8 Consider the graph with 8 nodes $A_1, A_2, A_3, A_4, H, T, F_1, F_2$. A_i is connected to A_{i+1} for all i , each A_i is connected to H , H is connected to T , and T is connected to each F_i . Find a 3-coloring of this graph by hand using the following strategy: backtracking with conflict-directed backjumping, the variable order $A_1, H, A_4, F_1, A_2, F_2, A_3, T$, and the value order R, G, B .

6.9 Explain why it is a good heuristic to choose the variable that is *most* constrained but the value that is *least* constraining in a CSP search.



6.10 Generate random instances of map-coloring problems as follows: scatter n points on the unit square; select a point X at random, connect X by a straight line to the nearest point Y such that X is not already connected to Y and the line crosses no other line; repeat the previous step until no more connections are possible. The points represent regions on the map and the lines connect neighbors. Now try to find k -colorings of each map, for both $k = 3$ and $k = 4$, using min-conflicts, backtracking, backtracking with forward checking, and backtracking with MAC. Construct a table of average run times for each algorithm for values of n up to the largest you can manage. Comment on your results.

6.11 Use the AC-3 algorithm to show that arc consistency can detect the inconsistency of the partial assignment $\{WA = \text{green}, V = \text{red}\}$ for the problem shown in Figure 6.1.

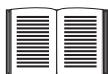
6.12 What is the worst-case complexity of running AC-3 on a tree-structured CSP?

6.13 AC-3 puts back on the queue *every* arc (X_k, X_i) whenever *any* value is deleted from the domain of X_i , even if each value of X_k is consistent with several remaining values of X_i . Suppose that, for every arc (X_k, X_i) , we keep track of the number of remaining values of X_i that are consistent with each value of X_k . Explain how to update these numbers efficiently and hence show that arc consistency can be enforced in total time $O(n^2d^2)$.

6.14 The TREE-CSP-SOLVER (Figure 6.10) makes arcs consistent starting at the leaves and working backwards towards the root. Why does it do that? What would happen if it went in the opposite direction?

6.15 We introduced Sudoku as a CSP to be solved by search over partial assignments because that is the way people generally undertake solving Sudoku problems. It is also possible, of course, to attack these problems with local search over complete assignments. How well would a local solver using the min-conflicts heuristic do on Sudoku problems?

6.16 Define in your own words the terms constraint, backtracking search, arc consistency, backjumping, min-conflicts, and cycle cutset.



6.17 Suppose that a graph is known to have a cycle cutset of no more than k nodes. Describe a simple algorithm for finding a minimal cycle cutset whose run time is not much more than $O(n^k)$ for a CSP with n variables. Search the literature for methods for finding approximately minimal cycle cutsets in time that is polynomial in the size of the cutset. Does the existence of such algorithms make the cycle cutset method practical?

7

LOGICAL AGENTS

In which we design agents that can form representations of a complex world, use a process of inference to derive new representations about the world, and use these new representations to deduce what to do.

REASONING
REPRESENTATION
KNOWLEDGE-BASED
AGENTS

Humans, it seems, know things; and what they know helps them do things. These are not empty statements. They make strong claims about how the intelligence of humans is achieved—not by purely reflex mechanisms but by processes of **reasoning** that operate on internal **representations** of knowledge. In AI, this approach to intelligence is embodied in **knowledge-based agents**.

The problem-solving agents of Chapters 3 and 4 know things, but only in a very limited, inflexible sense. For example, the transition model for the 8-puzzle—knowledge of what the actions do—is hidden inside the domain-specific code of the `RESULT` function. It can be used to predict the outcome of actions but not to deduce that two tiles cannot occupy the same space or that states with odd parity cannot be reached from states with even parity. The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, an agent’s only choice for representing what it knows about the current state is to list all possible concrete states—a hopeless prospect in large environments.

LOGIC

Chapter 6 introduced the idea of representing states as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. In this chapter and those that follow, we take this step to its logical conclusion, so to speak—we develop **logic** as a general class of representations to support knowledge-based agents. Such agents can combine and recombine information to suit myriad purposes. Often, this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth’s life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then we explain the general principles of **logic**

in Section 7.3 and the specifics of **propositional logic** in Section 7.4. While less expressive than **first-order logic** (Chapter 8), propositional logic illustrates all the basic concepts of logic; it also comes with well-developed inference technologies, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of knowledge-based agents with the technology of propositional logic to build some simple agents for the wumpus world.

7.1 KNOWLEDGE-BASED AGENTS

KNOWLEDGE BASE
SENTENCE

KNOWLEDGE
REPRESENTATION
LANGUAGE
AXIOM

The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here “sentence” is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

INFERENCE

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKS a question of the knowledge base, the answer should follow from what has been told (or TELLED) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word “follow.” For now, take it to mean that the inference process should not make things up as it goes along.

BACKGROUND
KNOWLEDGE

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, *KB*, which may initially contain some **background knowledge**.

Each time the agent program is called, it does three things. First, it TELLS the knowledge base what it perceives. Second, it ASKS the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLS the knowledge base which action was chosen, and the agent executes the action.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at

```

function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
               t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

KNOWLEDGE LEVEL

the **knowledge level**, where we need specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

IMPLEMENTATION LEVEL

DECLARATIVE

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 18, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

7.2 THE WUMPUS WORLD

WUMPUS WORLD

In this section we describe an environment in which knowledge-based agents can show their worth. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain

bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description:

- **Performance measure:** +1000 for climbing out of the cave with the gold, −1000 for falling into a pit or being eaten by the wumpus, −1 for each action taken and −10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment:** A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators:** The agent can move *Forward*, *TurnLeft* by 90° , or *TurnRight* by 90° . The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].
- **Sensors:** The agent has five sensors, each of which gives a single bit of information:
 - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
 - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
 - In the square where the gold is, the agent will perceive a *Glitter*.
 - When an agent walks into a wall, it will perceive a *Bump*.
 - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*, *Breeze*, *None*, *None*, *None*].

We can characterize the wumpus environment along the various dimensions given in Chapter 2. Clearly, it is discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state that happen to be immutable—in which case, the transition model for the environment is completely

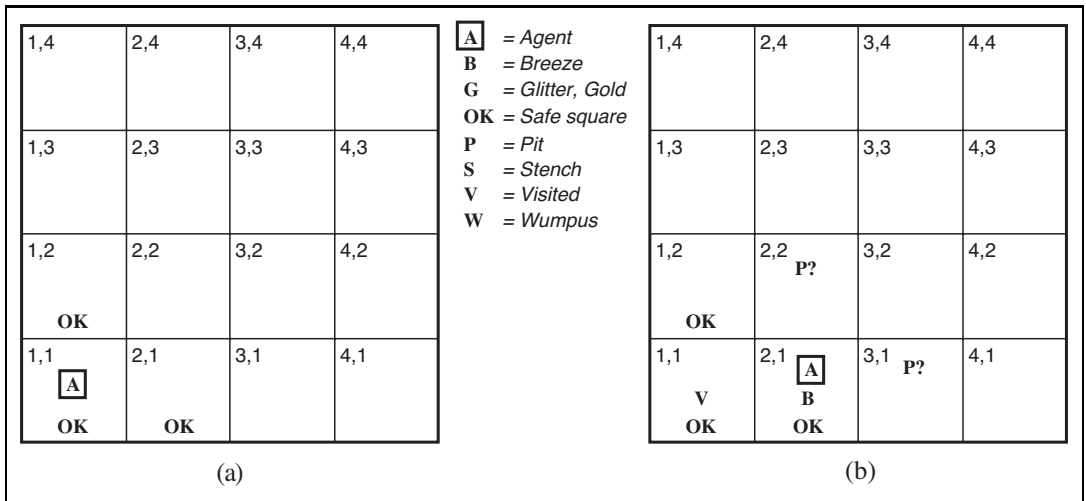


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept *[None, None, None, None, None]*. (b) After one move, with percept *[None, Breeze, None, None, None]*.

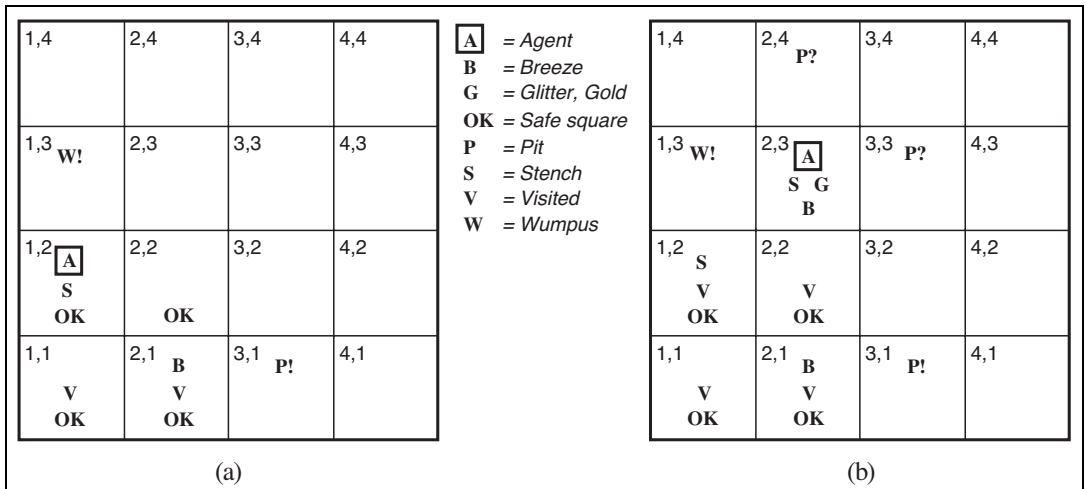


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept *[Stench, None, None, None, None]*. (b) After the fifth move, with percept *[Stench, Breeze, Glitter, None, None]*.

wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation **W!** indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions such as those described in the preceding paragraphs.

7.3 LOGIC

This section summarizes the fundamental concepts of logical representation and reasoning. These beautiful ideas are independent of any of logic's particular forms. We therefore postpone the technical details of those forms until the next section, using instead the familiar example of ordinary arithmetic.

SYNTAX

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: “ $x + y = 4$ ” is a well-formed sentence, whereas “ $x4y+ =$ ” is not.

SEMANTICS

TRUTH

POSSIBLE WORLD

A logic must also define the **semantics** or meaning of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence “ $x + y = 4$ ” is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no “in between.”¹

MODEL

When we need to be precise, we use the term **model** in place of “possible world.” Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence $x + y = 4$ is true when there are four people in total. Formally, the possible models are just all possible assignments of real numbers to the variables x and y . Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y . If a sentence α is true in model m , we say that m **satisfies** α or sometimes m **is a model of** α . We use the notation $M(\alpha)$ to mean the set of all models of α .

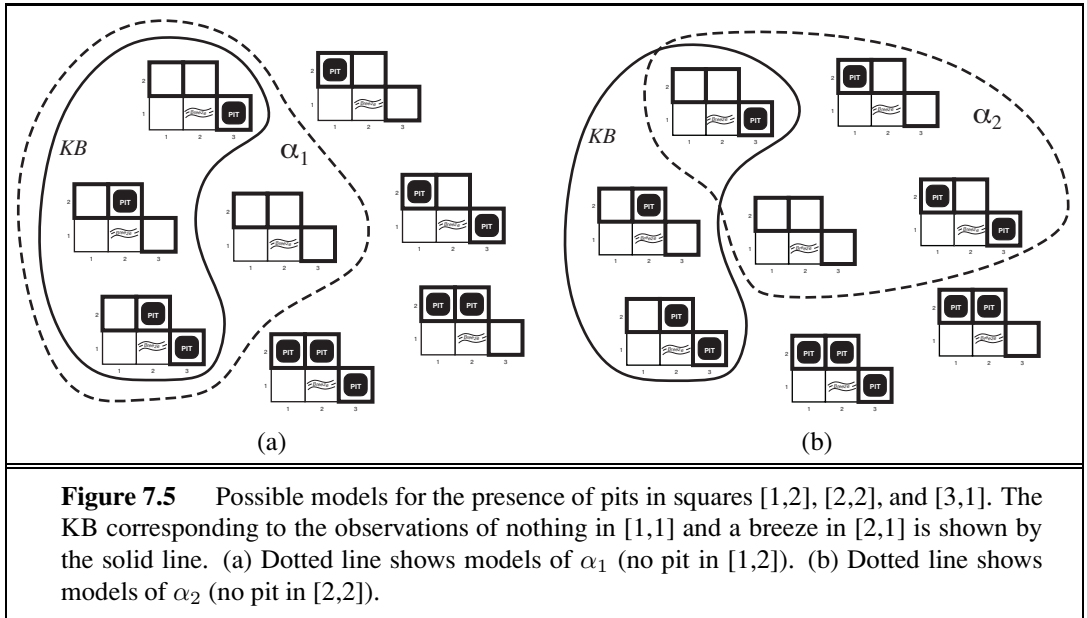
SATISFACTION

ENTAILMENT

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write

$$\alpha \models \beta$$

¹ **Fuzzy logic**, discussed in Chapter 14, allows for degrees of truth.



to mean that the sentence α entails the sentence β . The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta).$$

(Note the direction of the \subseteq here: if $\alpha \models \beta$, then α is a *stronger* assertion than β : it rules out *more* possible worlds.) The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence $x = 0$ entails the sentence $xy = 0$. Obviously, in any model where x is zero, it is the case that xy is zero (regardless of the value of y).

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These eight models are shown in Figure 7.5.²

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are

² Although the figure shows the models as partial wumpus worlds, they are really nothing more than assignments of *true* and *false* to the sentences “there is a pit in [1,2]” etc. Models, in the mathematical sense, do not need to have ‘orrible’ airy wumpuses in them.

shown surrounded by a solid line in Figure 7.5. Now let us consider two possible conclusions:

$\alpha_1 = \text{“There is no pit in [1,2].”}$

$\alpha_2 = \text{“There is no pit in [2,2].”}$

We have surrounded the models of α_1 and α_2 with dotted lines in Figures 7.5(a) and 7.5(b), respectively. By inspection, we see the following:

in every model in which KB is true, α_1 is also true.

Hence, $KB \models \alpha_1$: there is no pit in [1,2]. We can also see that

in some models in which KB is true, α_2 is false.

Hence, $KB \not\models \alpha_2$: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)³

LOGICAL INFERENCE

MODEL CHECKING

The preceding example not only illustrates entailment but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**. The inference algorithm illustrated in Figure 7.5 is called **model checking**, because it enumerates all possible models to check that α is true in all models in which KB is true, that is, that $M(KB) \subseteq M(\alpha)$.

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB , we write

$$KB \vdash_i \alpha,$$

which is pronounced “ α is derived from KB by i ” or “ i derives α from KB .”

SOUND

TRUTH-PRESERVING

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,⁴ is a sound procedure.

COMPLETENESS

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.⁵ Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

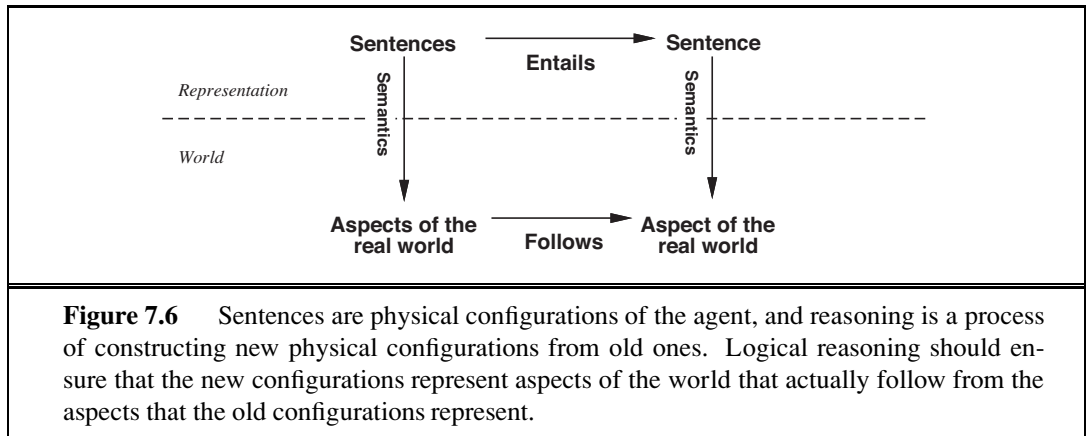


We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world*. So, while an inference process operates on “syntax”—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process *corresponds*

³ The agent can calculate the *probability* that there is a pit in [2,2]; Chapter 13 shows how.

⁴ Model checking works if the space of models is finite—for example, in wumpus worlds of fixed size. For arithmetic, on the other hand, the space of models is infinite: even if we restrict ourselves to the integers, there are infinitely many pairs of values for x and y in the sentence $x + y = 4$.

⁵ Compare with the case of infinite search spaces in Chapter 3, where depth-first search is not complete.



to the real-world relationship whereby some aspect of the real world is the case⁶ by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

GROUNDING



The final issue to consider is **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, *how do we know that KB is true in the real world?* (After all, *KB* is just “syntax” inside the agent’s head.) This is a philosophical question about which many, many books have been written. (See Chapter 26.) A simple answer is that the agent’s sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent’s knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called **learning**, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells *except on February 29 in leap years*, which is when they take their baths. Thus, *KB* may not be true in the real world, but with good learning procedures, there is reason for optimism.

7.4 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

PROPOSITIONAL LOGIC

We now present a simple but powerful logic called **propositional logic**. We cover the syntax of propositional logic and its semantics—the way in which the truth of sentences is determined. Then we look at **entailment**—the relation between a sentence and another sentence that follows from it—and see how this leads to a simple algorithm for logical inference. Everything takes place, of course, in the wumpus world.

⁶ As Wittgenstein (1922) put it in his famous *Tractatus*: “The world is everything that is the case.”

7.4.1 Syntax

ATOMIC SENTENCES
PROPOSITION
SYMBOL

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P , Q , R , $W_{1,3}$ and $North$. The names are arbitrary but are often chosen to have some mnemonic value—we use $W_{1,3}$ to stand for the proposition that the wumpus is in $[1,3]$. (Remember that symbols such as $W_{1,3}$ are *atomic*, i.e., W , 1 , and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: *True* is the always-true proposition and *False* is the always-false proposition. **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**. There are five connectives in common use:

COMPLEX
SENTENCES
LOGICAL
CONNECTIVES

NEGATION

\neg (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

LITERAL

CONJUNCTION

\wedge (and). A sentence whose main connective is \wedge , such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The \wedge looks like an “A” for “And.”)

DISJUNCTION

\vee (or). A sentence using \vee , such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$. (Historically, the \vee comes from the Latin “vel,” which means “or.” For most people, it is easier to remember \vee as an upside-down \wedge .)

IMPLICATION

\Rightarrow (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as \supset or \rightarrow .

PREMISE

CONCLUSION

RULES

BICONDITIONAL

\Leftrightarrow (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as \equiv .

$Sentence \rightarrow AtomicSentence \mid ComplexSentence$
 $AtomicSentence \rightarrow True \mid False \mid P \mid Q \mid R \mid \dots$
 $ComplexSentence \rightarrow (Sentence) \mid [Sentence]$
 $\mid \neg Sentence$
 $\mid Sentence \wedge Sentence$
 $\mid Sentence \vee Sentence$
 $\mid Sentence \Rightarrow Sentence$
 $\mid Sentence \Leftrightarrow Sentence$

OPERATOR PRECEDENCE : $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 7.7 gives a formal grammar of propositional logic; see page 1060 if you are not familiar with the BNF notation. The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. The “not” operator (\neg) has the highest precedence, which means that in the sentence $\neg A \wedge B$ the \neg binds most tightly, giving us the equivalent of $(\neg A) \wedge B$ rather than $\neg(A \wedge B)$. (The notation for ordinary arithmetic is the same: $-2 + 4$ is 2, not -6 .) When in doubt, use parentheses to make sure of the right interpretation. Square brackets mean the same thing as parentheses; the choice of square brackets or parentheses is solely to make it easier for a human to read a sentence.

7.4.2 Semantics

TRUTH VALUE

Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the **truth value**—*true* or *false*—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$$m_1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}.$$

With three proposition symbols, there are $2^3 = 8$ possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. $P_{1,2}$ is just a symbol; it might mean “there is a pit in [1,2]” or “I’m in Paris today and tomorrow.”

The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

- *True* is true in every model and *False* is false in every model.
- The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m_1 given earlier, $P_{1,2}$ is false.

For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here “iff” means “if and only if”):

- $\neg P$ is true iff P is false in m .
- $P \wedge Q$ is true iff both P and Q are true in m .
- $P \vee Q$ is true iff either P or Q is true in m .
- $P \Rightarrow Q$ is true unless P is true and Q is false in m .
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m .

TRUTH TABLE

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation. For example,

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is true and Q is false (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

the sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1 , gives $\text{true} \wedge (\text{false} \vee \text{true}) = \text{true} \wedge \text{true} = \text{true}$. Exercise 7.3 asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m .

The truth tables for “and,” “or,” and “not” are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \vee Q$ is true when P is true or Q is true *or both*. A different connective, called “exclusive or” (“xor” for short), yields false when both disjuncts are true.⁷ There is no consensus on the symbol for exclusive or; some choices are $\dot{\vee}$ or \neq or \oplus .

The truth table for \Rightarrow may not quite fit one’s intuitive understanding of “ P implies Q ” or “if P then Q .” For one thing, propositional logic does not require any relation of *causation* or *relevance* between P and Q . The sentence “5 is odd implies Tokyo is the capital of Japan” is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, “5 is even implies Sam is smart” is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of “ $P \Rightarrow Q$ ” as saying, “If P is true, then I am claiming that Q is true. Otherwise I am making no claim.” The only way for this sentence to be *false* is if P is true but Q is false.

The biconditional, $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as “ P if and only if Q .” Many of the rules of the wumpus world are best written using \Leftrightarrow . For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need a biconditional,

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

where $B_{1,1}$ means that there is a breeze in [1,1].

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each $[x, y]$ location:

⁷ Latin has a separate word, *aut*, for exclusive or.

$P_{x,y}$ is true if there is a pit in $[x, y]$.

$W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.

$B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.

$S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in $[1,2]$), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in $[1,1]$:

$$R_1 : \neg P_{1,1} .$$

- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) .$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).

$$R_4 : \neg B_{1,1} .$$

$$R_5 : B_{2,1} .$$

7.4.4 A simple inference procedure

Our goal now is to decide whether $KB \models \alpha$ for some sentence α . For example, is $\neg P_{1,2}$ entailed by our KB ? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of *true* or *false* to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true (Figure 7.9). In those three models, $\neg P_{1,2}$ is true, hence there is no pit in $[1,2]$. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in $[2,2]$.

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 215, TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates—there are only finitely many models to examine.

Of course, “finitely many” is not always the same as “few.” If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. (The space complexity is only $O(n)$ because the enumeration is depth-first.) Later in this chapter we show algorithms that are much more efficient in many cases. Unfortunately, propositional entailment is co-NP-complete (i.e., probably no easier than NP-complete—see Appendix A), so *every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input.*



$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	<u>true</u>
false	true	false	false	false	true	false	true	true	true	true	true	<u>true</u>
false	true	false	false	false	true	true	true	true	true	true	true	<u>true</u>
false	true	false	false	true	false	false	true	false	false	true	true	false
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
true	true	true	true	true	true	true	false	true	true	false	true	false

Figure 7.9 A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```

function TT-ENTAILS?( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $symbols \leftarrow$  a list of the proposition symbols in  $KB$  and  $\alpha$ 
  return TT-CHECK-ALL( $KB, \alpha, symbols, \{ \}$ )



---


function TT-CHECK-ALL( $KB, \alpha, symbols, model$ ) returns true or false
  if EMPTY?( $symbols$ ) then
    if PL-TRUE?( $KB, model$ ) then return PL-TRUE?( $\alpha, model$ )
    else return true // when  $KB$  is false, always return true
  else do
     $P \leftarrow$  FIRST( $symbols$ )
     $rest \leftarrow$  REST( $symbols$ )
    return (TT-CHECK-ALL( $KB, \alpha, rest, model \cup \{P = true\}$ )
      and
      TT-CHECK-ALL( $KB, \alpha, rest, model \cup \{P = false\}$ ))

```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable $model$ represents a partial model—an assignment to some of the symbols. The keyword “**and**” is used here as a logical operation on its two arguments, returning *true* or *false*.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Figure 7.11 Standard logical equivalences. The symbols α , β , and γ stand for arbitrary sentences of propositional logic.

7.5 PROPOSITIONAL THEOREM PROVING

THEOREM PROVING

So far, we have shown how to determine entailment by *model checking*: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

LOGICAL EQUIVALENCE

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

$$\alpha \equiv \beta \quad \text{if and only if} \quad \alpha \models \beta \text{ and } \beta \models \alpha.$$

VALIDITY

TAUTOLOGY

The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True*. What good are valid sentences? From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks:

DEDUCTION THEOREM



For any sentences α and β , $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

(Exercise 7.5 asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—

or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to *True*. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference.

SATISFIABILITY

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 6 ask whether the constraints are satisfiable by some assignment.

SAT

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:

$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

REDUCTIO AD
ABSURDUM

REFUTATION

CONTRADICTION

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum* (literally, “reduction to an absurd thing”). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α . This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.5.1 Inference and proofs

INFERENCE RULES

PROOF

MODUS PONENS

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for *mode that affirms*) and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}.$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then *Shoot* can be inferred.

AND-ELIMINATION

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from $(WumpusAhead \wedge WumpusAlive)$, *WumpusAlive* can be inferred.

By considering the possible truth values of α and β , one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β .

Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R_1 through R_5 and show how to prove $\neg P_{1,2}$, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R_2 to obtain

$$R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Logical equivalence for contrapositives gives

$$R_8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})) .$$

Now we can apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

$$R_9 : \neg(P_{1,2} \vee P_{2,1}) .$$

Finally, we apply De Morgan's rule, giving the conclusion

$$R_{10} : \neg P_{1,2} \wedge \neg P_{2,1} .$$

That is, neither [1,2] nor [2,1] contains a pit.

We found this proof by hand, but we can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.



Thus, searching for proofs is an alternative to enumerating models. In many practical cases *finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are*. For example, the proof given earlier leading to $\neg P_{1,2} \wedge \neg P_{2,1}$ does not mention the propositions $B_{2,1}$, $P_{1,1}$, $P_{2,2}$, or $P_{3,1}$. They can be ignored because the goal proposition, $P_{1,2}$, appears only in sentence R_2 ; the other propositions in R_2 appear only in R_4 and R_2 ; so R_1 , R_3 , and R_5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

One final property of logical systems is **monotonicity**, which says that the set of entailed sentences can only *increase* as information is added to the knowledge base.⁸ For any sentences α and β ,

$$\text{if } KB \models \alpha \text{ then } KB \wedge \beta \models \alpha .$$

⁸ **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind. They are discussed in Section 12.6.

For example, suppose the knowledge base contains the additional assertion β stating that there are exactly eight pits in the world. This knowledge might help the agent draw *additional* conclusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion that there is no pit in [1,2]. Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow *regardless of what else is in the knowledge base*.

7.5.2 Proof by resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed the question of *completeness* for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 89) are complete in the sense that they will find any reachable goal, but if the available inference rules are inadequate, then the goal is not reachable—no proof exists that uses only those inference rules. For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 7.4(a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

$$\begin{aligned} R_{11} : & \quad \neg B_{1,2} . \\ R_{12} : & \quad B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}) . \end{aligned}$$

By the same process that led to R_{10} earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

$$\begin{aligned} R_{13} : & \quad \neg P_{2,2} . \\ R_{14} : & \quad \neg P_{1,3} . \end{aligned}$$

We can also apply biconditional elimination to R_3 , followed by Modus Ponens with R_5 , to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$$R_{15} : \quad P_{1,1} \vee P_{2,2} \vee P_{3,1} .$$

Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R_{13} *resolves with* the literal $P_{2,2}$ in R_{15} to give the **resolvent**

$$R_{16} : \quad P_{1,1} \vee P_{3,1} .$$

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give

$$R_{17} : \quad P_{3,1} .$$

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the **unit resolution** inference rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k} ,$$

where each ℓ is a literal and ℓ_i and m are **complementary literals** (i.e., one is the negation

RESOLVENT

UNIT RESOLUTION

COMPLEMENTARY
LITERALS

CLAUSE

of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

UNIT CLAUSE

RESOLUTION

The unit resolution rule can be generalized to the full **resolution** rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n},$$

where ℓ_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

FACTORING

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.⁹ The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A .

The *soundness* of the resolution rule can be seen easily by considering the literal ℓ_i that is complementary to literal m_j in the other clause. If ℓ_i is true, then m_j is false, and hence $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ must be true, because $m_1 \vee \cdots \vee m_n$ is given. If ℓ_i is false, then $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ must be true because $\ell_1 \vee \cdots \vee \ell_k$ is given. Now ℓ_i is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.



What is more surprising about the resolution rule is that it forms the basis for a family of *complete* inference procedures. A *resolution-based theorem prover* can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$. The next two subsections explain how resolution accomplishes this.

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses*. A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.14). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:



1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$
2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}).$$

⁹ If a clause is viewed as a *set* of literals, then this restriction is automatically respected. Using set notation for clauses makes the resolution rule much cleaner, at the cost of introducing additional notation.

3. CNF requires \neg to appear only in literals, so we “move \neg inwards” by repeated application of the following equivalences from Figure 7.11:

$$\neg(\neg\alpha) \equiv \alpha \quad (\text{double-negation elimination})$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad (\text{De Morgan})$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad (\text{De Morgan})$$

In the example, we require just one application of the last rule:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

4. Now we have a sentence containing nested \wedge and \vee operators applied to literals. We apply the distributivity law from Figure 7.11, distributing \vee over \wedge wherever possible.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 250. That is, to show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

A resolution algorithm is shown in Figure 7.12. First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α ; or,
- two clauses resolve to yield the *empty* clause, in which case KB entails α .

The empty clause—a disjunction of no disjuncts—is equivalent to *False* because a disjunction is true only if at least one of its disjuncts is true. Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as P and $\neg P$.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

and we wish to prove α which is, say, $\neg P_{1,2}$. When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of Figure 7.13. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.13 reveals that many resolution steps are pointless. For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to $True \vee P_{1,2}$ which is equivalent to *True*. Deducing that *True* is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

```

function PL-RESOLUTION( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$ 
   $new \leftarrow \{ \}$ 
  loop do
    for each pair of clauses  $C_i, C_j$  in  $clauses$  do
       $resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if  $resolvents$  contains the empty clause then return true
       $new \leftarrow new \cup resolvents$ 
    if  $new \subseteq clauses$  then return false
     $clauses \leftarrow clauses \cup new$ 

```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

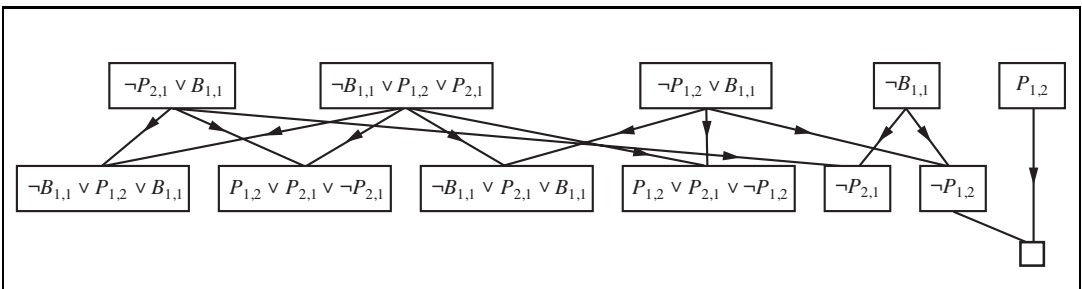


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

Completeness of resolution

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the **resolution closure** $RC(S)$ of a set of clauses S , which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable $clauses$. It is easy to see that $RC(S)$ must be finite, because there are only finitely many distinct clauses that can be constructed out of the symbols P_1, \dots, P_k that appear in S . (Notice that this would not be true without the factoring step that removes multiple copies of literals.) Hence, PL-RESOLUTION always terminates.

The completeness theorem for resolution in propositional logic is called the **ground resolution theorem**:

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

This theorem is proved by demonstrating its contrapositive: if the closure $RC(S)$ does *not*

RESOLUTION
CLOSURE

GROUND
RESOLUTION
THEOREM

contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for P_1, \dots, P_k . The construction procedure is as follows:

For i from 1 to k ,

- If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for P_1, \dots, P_{i-1} , then assign *false* to P_i .
- Otherwise, assign *true* to P_i .

This assignment to P_1, \dots, P_k is a model of S . To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the *other* literals in C must already have been falsified by assignments to P_1, \dots, P_{i-1} . Thus, C must now look like either $(false \vee false \vee \dots false \vee P_i)$ or like $(false \vee false \vee \dots false \vee \neg P_i)$. If just one of these two is in $RC(S)$, then the algorithm will assign the appropriate truth value to P_i to make C true, so C can only be falsified if *both* of these clauses are in $RC(S)$. Now, since $RC(S)$ is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to P_1, \dots, P_{i-1} . This contradicts our assumption that the first falsified clause appears at stage i . Hence, we have proved that the construction never falsifies a clause in $RC(S)$; that is, it produces a model of $RC(S)$ and thus a model of S itself (since S is contained in $RC(S)$).

7.5.3 Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

DEFINITE CLAUSE

One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ is a definite clause, whereas $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.

HORN CLAUSE

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at most one is positive*. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

GOAL CLAUSES

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.13.) For example, the definite clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $L_{1,1}$, is called a **fact**. It too can be written in implication form as $True \Rightarrow L_{1,1}$, but it is simpler to write just $L_{1,1}$.

BODY

HEAD

FACT

$$\begin{aligned}
\text{CNFSentence} &\rightarrow \text{Clause}_1 \wedge \cdots \wedge \text{Clause}_n \\
\text{Clause} &\rightarrow \text{Literal}_1 \vee \cdots \vee \text{Literal}_m \\
\text{Literal} &\rightarrow \text{Symbol} \mid \neg \text{Symbol} \\
\text{Symbol} &\rightarrow P \mid Q \mid R \mid \dots \\
\text{HornClauseForm} &\rightarrow \text{DefiniteClauseForm} \mid \text{GoalClauseForm} \\
\text{DefiniteClauseForm} &\rightarrow (\text{Symbol}_1 \wedge \cdots \wedge \text{Symbol}_l) \Rightarrow \text{Symbol} \\
\text{GoalClauseForm} &\rightarrow (\text{Symbol}_1 \wedge \cdots \wedge \text{Symbol}_l) \Rightarrow \text{False}
\end{aligned}$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the k -CNF sentence, which is a CNF sentence where each clause has at most k literals.

FORWARD-CHAINING
BACKWARD-
CHAINING

2. Inference with Horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**, which is discussed in Chapter 9.
3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

7.5.4 Forward and backward chaining

The forward-chaining algorithm $\text{PL-FC-ENTAILS?}(KB, q)$ determines if a single proposition symbol q —the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and *Breeze* are known and $(L_{1,1} \wedge \text{Breeze}) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an **AND-OR graph** (see Chapter 4). In AND-OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Whenever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.


```

function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
           q, the query, a proposition symbol
  count  $\leftarrow$  a table, where count[c] is the number of symbols in c's premise
  inferred  $\leftarrow$  a table, where inferred[s] is initially false for all symbols
  agenda  $\leftarrow$  a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p  $\leftarrow$  POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p]  $\leftarrow$  true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false

```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet “processed.” The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol *p* from the agenda is processed, the count is reduced by one for each implication in whose premise *p* appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

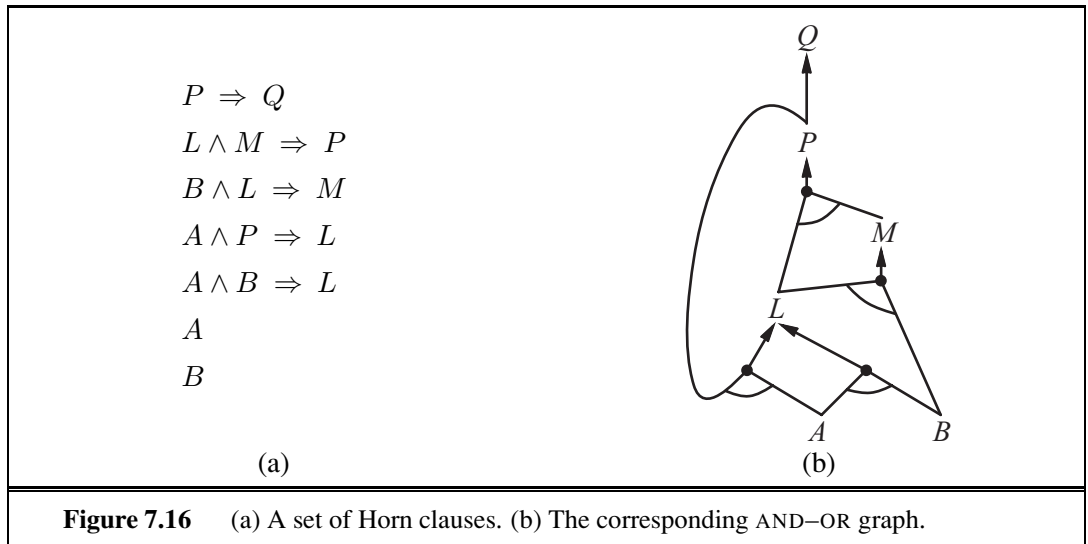
It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a **fixed point** where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model*. To see this, assume the opposite, namely that some clause $a_1 \wedge \dots \wedge a_k \Rightarrow b$ is false in the model. Then $a_1 \wedge \dots \wedge a_k$ must be true in the model and *b* must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence *q* that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence *q* must be inferred by the algorithm.

Forward chaining is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using

FIXED POINT



DATA-DRIVEN



an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor’s garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q . If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B , that forms the basis for a proof. The algorithm is essentially identical to the AND-OR-GRAPH-SEARCH algorithm in Figure 4.11. As with forward chaining, an efficient implementation runs in linear time.

GOAL-DIRECTED
REASONING

Backward chaining is a form of **goal-directed reasoning**. It is useful for answering specific questions such as “What shall I do now?” and “Where are my keys?” Often, the cost of backward chaining is *much less* than linear in the size of the knowledge base, because the process touches only relevant facts.

7.6 EFFECTIVE PROPOSITIONAL MODEL CHECKING

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the “technology” of propositional logic. This section can be skimmed on a first reading of the chapter.

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, $\alpha \models \beta$, can be done by testing *unsatisfiability* of $\alpha \wedge \neg\beta$.) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

DAVIS-PUTNAM
ALGORITHM

The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?:

- *Early termination*: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C . Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.
- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same “sign” in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals *true*, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = \text{false}$, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.
- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned *false* by the model. For example, if the model contains $B = \text{true}$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to *false*. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that

PURE SYMBOL

```

function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses  $\leftarrow$  the set of clauses in the CNF representation of s
  symbols  $\leftarrow$  a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

```

```

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value  $\leftarrow$  FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P, value  $\leftarrow$  FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P  $\leftarrow$  FIRST(symbols); rest  $\leftarrow$  REST(symbols)
  return DPLL(clauses, rest, model  $\cup$  {P=true}) or
    DPLL(clauses, rest, model  $\cup$  {P=false})

```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.23). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing *true* to be assigned to A . This “cascade” of forced assignments is called **unit propagation**. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining. (See Exercise 7.24.)

UNIT PROPAGATION

The DPLL algorithm is shown in Figure 7.17, which gives the the essential skeleton of the search process.

What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large problems. It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. **Component analysis** (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering** (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** (see page 216) suggests choosing the variable that appears most frequently over all remaining clauses.

3. **Intelligent backtracking** (as seen in Section 6.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts** (as seen on page 124 for hill-climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as “the set of clauses in which variable X_i appears as a positive literal.” This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laborious, hand-guided proofs.

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 122) and SIMULATED-ANNEALING (page 126). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 221). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.18). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state and (2) a “random walk” step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set $max_flips = \infty$ and $p > 0$, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit

```

function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
           p, the probability of choosing to do a “random walk” move, typically around 0.5
           max_flips, number of flips allowed before giving up

  model  $\leftarrow$  a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause  $\leftarrow$  a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure

```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

upon the solution. Alas, if *max_flips* is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

For this reason, WALKSAT is most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 6 usually have solutions. On the other hand, WALKSAT cannot always detect *unsatisfiability*, which is required for deciding entailment. For example, an agent cannot *reliably* use WALKSAT to prove that a square is safe in the wumpus world. Instead, it can say, “I thought about it for an hour and couldn’t come up with a possible world in which the square *isn’t* safe.” This may be a good empirical indicator that the square is safe, but it’s certainly not a proof.

7.6.3 The landscape of random SAT problems

Some SAT problems are harder than others. *Easy* problems can be solved by any old algorithm, but because we know that SAT is NP-complete, at least some problem instances must require exponential run time. In Chapter 6, we saw some surprising discoveries about certain kinds of problems. For example, the *n*-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, *n*-queens is easy because it is **underconstrained**.

UNDERCONSTRAINED

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively *few* clauses constraining the variables. For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \\ \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C).$$

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are

most such underconstrained problems. On the other hand, an *overconstrained* problem has many clauses relative to the number of variables and is likely to have no solutions.

To go beyond these basic intuitions, we must define exactly how random sentences are generated. The notation $CNF_k(m, n)$ denotes a k -CNF sentence with m clauses and n symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with k different literals, which are positive or negative at random. (A symbol may not appear twice in a clause, nor may a clause appear twice in a sentence.)

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m, 50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, m/n . As we expect, for small m/n the probability of satisfiability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the “cliff” stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases. Theoretically, the **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(n, rn)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. The conjecture remains unproven.

SATISFIABILITY
THRESHOLD
CONJECTURE

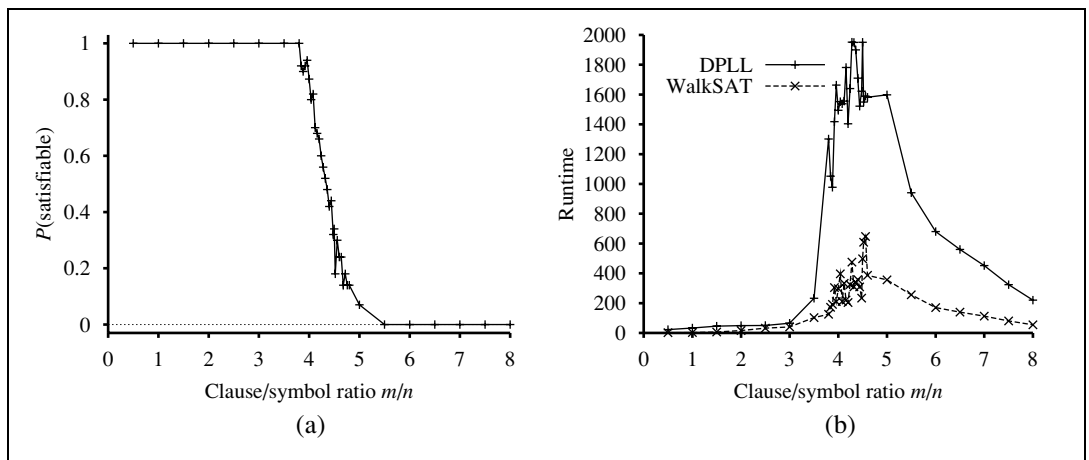


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n . (b) Graph of the median run time (measured in number of recursive calls to DPLL, a good proxy) on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

Now that we have a good idea where the satisfiable and unsatisfiable problems are, the next question is, where are the hard problems? It turns out that they are also often at the threshold value. Figure 7.19(b) shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at a ratio of 3.3. The underconstrained problems are easiest to solve (because it is so easy to guess a solution); the overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.

7.7 AGENTS BASED ON PROPOSITIONAL LOGIC

In this section, we bring together what we have learned so far in order to construct wumpus world agents that use propositional logic. The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. We also show how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals.

7.7.1 The current state of the world

As stated at the beginning of the chapter, a logical agent operates by deducing what to do from a knowledge base of sentences about the world. The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent’s experience in a particular world. In this section, we focus on the problem of deducing the current state of the wumpus world—where am I, is that square safe, and so on.

We began collecting axioms in Section 7.4.3. The agent knows that the starting square contains no pit ($\neg P_{1,1}$) and no wumpus ($\neg W_{1,1}$). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of sentences of the following form:

$$\begin{aligned} B_{1,1} &\Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ S_{1,1} &\Leftrightarrow (W_{1,2} \vee W_{2,1}) \\ &\dots \end{aligned}$$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is *at least one* wumpus:

$$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,3} \vee W_{4,4} .$$

Then, we have to say that there is *at most one* wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\begin{aligned} &\neg W_{1,1} \vee \neg W_{1,2} \\ &\neg W_{1,1} \vee \neg W_{1,3} \\ &\dots \\ &\neg W_{4,3} \vee \neg W_{4,4} . \end{aligned}$$

So far, so good. Now let’s consider the agent’s percepts. If there is currently a stench, one might suppose that a proposition *Stench* should be added to the knowledge base. This is not quite right, however: if there was no stench at the previous time step, then $\neg \text{Stench}$ would already be asserted, and the new assertion would simply result in a contradiction. The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step (as supplied to MAKE-PERCEPT-SENTENCE in Figure 7.1) is 4, then we add

$Stench^4$ to the knowledge base, rather than $Stench$ —neatly avoiding any contradiction with $\neg Stench^3$. The same goes for the breeze, bump, glitter, and scream percepts.

FLUENT

ATEMPORAL
VARIABLE

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. For example, the initial knowledge base includes $L_{1,1}^0$ —the agent is in square $[1, 1]$ at time 0—as well as $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$. We use the word **fluent** (from the Latin *fluens*, flowing) to refer an aspect of the world that changes. “Fluent” is a synonym for “state variable,” in the sense described in the discussion of factored representations in Section 2.4.7 on page 57. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced through the location fluent as follows.¹⁰ For any time step t and any square $[x, y]$, we assert

$$\begin{aligned} L_{x,y}^t &\Rightarrow (Breeze^t \Leftrightarrow B_{x,y}) \\ L_{x,y}^t &\Rightarrow (Stench^t \Leftrightarrow S_{x,y}). \end{aligned}$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L_{x,y}^t$. These fluents change as the result of actions taken by the agent, so, in the terminology of Chapter 3, we need to write down the **transition model** of the wumpus world as a set of logical sentences.

First, we need proposition symbols for the occurrences of actions. As with percepts, these symbols are indexed by time; thus, $Forward^0$ means that the agent executes the *Forward* action at time 0. By convention, the percept for a given time step happens first, followed by the action for that time step, followed by a transition to the next time step.

EFFECT AXIOM

To describe how the world changes, we can try writing **effect axioms** that specify the outcome of an action at the next time step. For example, if the agent is at location $[1, 1]$ facing east at time 0 and goes *Forward*, the result is that the agent is in square $[2, 1]$ and no longer is in $[1, 1]$:

$$L_{1,1}^0 \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L_{2,1}^1 \wedge \neg L_{1,1}^1). \quad (7.1)$$

We would need one such sentence for each possible time step, for each of the 16 squares, and each of the four orientations. We would also need similar sentences for the other actions: *Grab*, *Shoot*, *Climb*, *TurnLeft*, and *TurnRight*.

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation (7.1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2, 1]$. That is, $ASK(KB, L_{2,1}^1) = true$. So far, so good. Unfortunately, the news elsewhere is less good: if we $ASK(KB, HaveArrow^1)$, the answer is *false*, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**.¹¹ One possible solution to the frame problem would

FRAME PROBLEM

¹⁰ Section 7.4.3 conveniently glossed over this requirement.

¹¹ The name “frame problem” comes from “frame of reference” in physics—the assumed stationary background with respect to which motion is measured. It also has an analogy to the frames of a movie, in which normally most of the background stays constant while changes occur in the foreground.

FRAME AXIOM

be to add **frame axioms** explicitly asserting all the propositions that remain the same. For example, for each time t we would have

$$\begin{aligned} Forward^t &\Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1}) \\ Forward^t &\Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1}) \\ &\dots \end{aligned}$$

 REPRESENTATIONAL
FRAME PROBLEM

where we explicitly mention every proposition that stays unchanged from time t to time $t + 1$ under the action *Forward*. Although the agent now knows that it still has the arrow after moving forward and that the wumpus hasn't died or come back to life, the proliferation of frame axioms seems remarkably inefficient. In a world with m different actions and n fluents, the set of frame axioms will be of size $O(mn)$. This specific manifestation of the frame problem is sometimes called the **representational frame problem**. Historically, the problem was a significant one for AI researchers; we explore it further in the notes at the end of the chapter.

LOCALITY

 INFERENCE FRAME
PROBLEM

The representational frame problem is significant because the real world has very many fluents, to put it mildly. Fortunately for us humans, each action typically changes no more than some small number k of those fluents—the world exhibits **locality**. Solving the representational frame problem requires defining the transition model with a set of axioms of size $O(mk)$ rather than size $O(mn)$. There is also an **inferential frame problem**: the problem of projecting forward the results of a t step plan of action in time $O(kt)$ rather than $O(nt)$.

 SUCCESSOR-STATE
AXIOM

The solution to the problem involves changing one's focus from writing axioms about *actions* to writing axioms about *fluents*. Thus, for each fluent F , we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occurred at time t . Now, the truth value of F^{t+1} can be set in one of two ways: either the action at time t causes F to be true at $t + 1$, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a **successor-state axiom** and has this schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

One of the simplest successor-state axioms is the one for *HaveArrow*. Because there is no action for reloading, the *ActionCausesF^t* part goes away and we are left with

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t). \quad (7.2)$$

For the agent's location, the successor-state axioms are more elaborate. For example, $L_{1,1}^{t+1}$ is true if either (a) the agent moved *Forward* from $[1, 2]$ when facing south, or from $[2, 1]$ when facing west; or (b) $L_{1,1}^t$ was already true and the action did not cause movement (either because the action was not *Forward* or because the action bumped into a wall). Written out in propositional logic, this becomes

$$\begin{aligned} L_{1,1}^{t+1} &\Leftrightarrow (L_{1,1}^t \wedge (\neg Forward^t \vee Bump^{t+1})) \\ &\vee (L_{1,2}^t \wedge (South^t \wedge Forward^t)) \\ &\vee (L_{2,1}^t \wedge (West^t \wedge Forward^t)). \end{aligned} \quad (7.3)$$

Exercise 7.26 asks you to write out axioms for the remaining wumpus world fluents.

Given a complete set of successor-state axioms and the other axioms listed at the beginning of this section, the agent will be able to ASK and answer any answerable question about the current state of the world. For example, in Section 7.2 the initial sequence of percepts and actions is

$$\begin{aligned}
 &\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 ; Forward^0 \\
 &\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 ; TurnRight^1 \\
 &\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 ; TurnRight^2 \\
 &\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 ; Forward^3 \\
 &\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 ; TurnRight^4 \\
 &\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 ; Forward^5 \\
 &Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6
 \end{aligned}$$

At this point, we have $ASK(KB, L_{1,2}^6) = true$, so the agent knows where it is. Moreover, $ASK(KB, W_{1,3}) = true$ and $ASK(KB, P_{3,1}) = true$, so the agent has found the wumpus and one of the pits. The most important question for the agent is whether a square is OK to move into, that is, the square contains no pit nor live wumpus. It's convenient to add axioms for this, having the form

$$OK_{x,y}^t \Leftrightarrow \neg P_{x,y} \wedge \neg (W_{x,y} \wedge WumpusAlive^t).$$

Finally, $ASK(KB, OK_{2,2}^6) = true$, so the square $[2, 2]$ is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK—and can do so in just a few milliseconds for small-to-medium wumpus worlds.

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc. Specifying all these exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. We will see in Chapter 13 that probability theory allows us to summarize all the exceptions without explicitly naming them.

QUALIFICATION
PROBLEM

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules and with problem-solving algorithms from Chapters 3 and 4 to produce a **hybrid agent** for the wumpus world. Figure 7.20 shows one possible way to do this. The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t , such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t , such

HYBRID AGENT

as the successor-state axioms. (The next section explains why the agent doesn't need axioms for *future* time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares. Route planning is done with A* search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where $\text{ASK}(KB, \neg W_{x,y})$ is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which $\text{ASK}(KB, \neg OK_{x,y}^t)$ returns false. If there is no such square, then the mission is impossible and the agent retreats to [1, 1] and climbs out of the cave.

7.7.3 Logical state estimation

The agent program in Figure 7.20 works quite well, but it has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. Obviously, this is unsustainable—we cannot have an agent whose time to process each percept grows in proportion to the length of its life! What we really need is a *constant* update time—that is, independent of t . The obvious answer is to save, or **cache**, the results of inference, so that the inference process at the next time step can build on the results of earlier steps instead of having to start again from scratch.

As we saw in Section 4.4, the past history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states of the world.¹² The process of updating the belief state as new percepts arrive is called **state estimation**. Whereas in Section 4.4 the belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols. For example, the logical sentence

$$WumpusAlive^1 \wedge L_{2,1}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \quad (7.4)$$

represents the set of all states at time 1 in which the wumpus is alive, the agent is at [2, 1], that square is breezy, and there is a pit in [3, 1] or [2, 2] or both.

Maintaining an exact belief state as a logical formula turns out not to be easy. If there are n fluent symbols for time t , then there are 2^n possible states—that is, assignments of truth values to those symbols. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are 2^n physical states, hence 2^{2^n} belief states. Even if we used the most compact possible encoding of logical formulas, with each belief state represented

¹² We can think of the percept history itself as a representation of the belief state, but one that makes inference increasingly expensive as the history gets longer.

```

function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
               t, a counter, initially 0, indicating time
               plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal “physics” sentences for time t
  safe  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, OK_{x,y}^t) = \text{true}\}$ 
  if ASK(KB, Glittert) = true then
    plan  $\leftarrow [Grab] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [Climb]$ 
  if plan is empty then
    unvisited  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, L_{x,y}^{t'}) = \text{false} \text{ for all } t' \leq t\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{safe}, \text{safe})$ 
  if plan is empty and ASK(KB, HaveArrowt) = true then
    possible_wumpus  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \neg W_{x,y}) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-SHOT}(\text{current}, \text{possible\_wumpus}, \text{safe})$ 
  if plan is empty then // no choice but to take a risk
    not_unsafe  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \neg OK_{x,y}^t) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{not\_unsafe}, \text{safe})$ 
  if plan is empty then
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [Climb]$ 
  action  $\leftarrow \text{POP}(\text{plan})$ 
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow t + 1$ 
  return action

```

```

function PLAN-ROUTE(current, goals, allowed) returns an action sequence
  inputs: current, the agent’s current position
           goals, a set of squares; try to plan a route to one of them
           allowed, a set of squares that can form part of the route

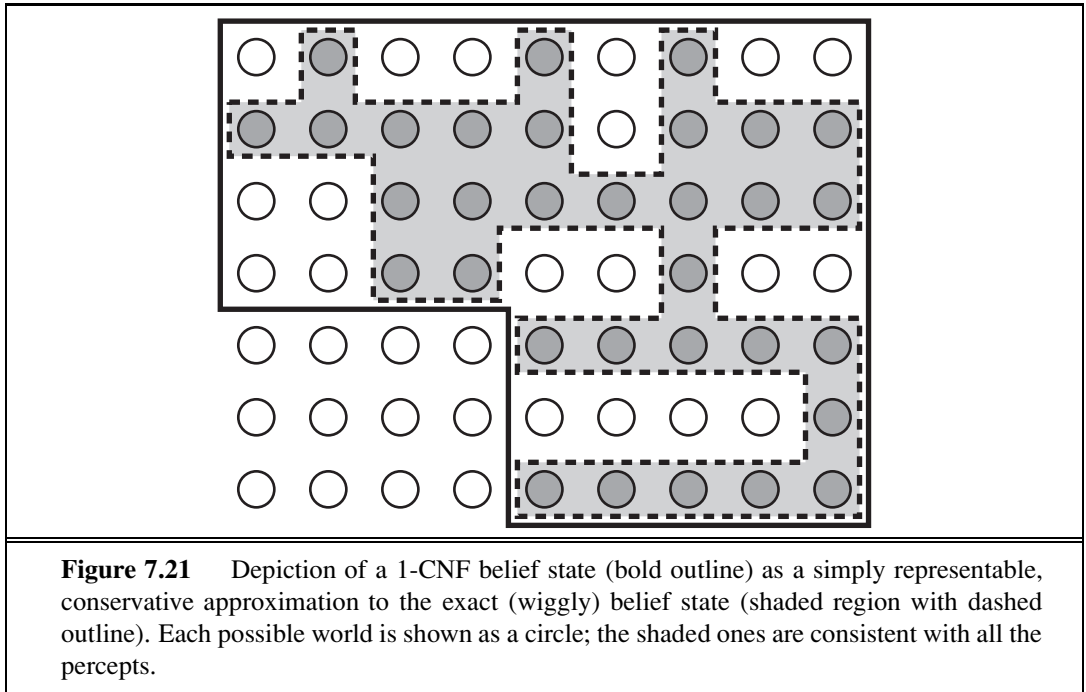
  problem  $\leftarrow \text{ROUTE-PROBLEM}(\text{current}, \text{goals}, \text{allowed})$ 
  return A*-GRAPH-SEARCH(problem)

```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

by a unique binary number, we would need numbers with $\log_2(2^{2^n}) = 2^n$ bits to label the current belief state. That is, exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One very common and natural scheme for *approximate* state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove X^t and $\neg X^t$ for each symbol X^t (as well as each atemporal symbol whose truth value is not yet known), given the belief state at $t - 1$. The conjunction of



provable literals becomes the new belief state, and the previous belief state is discarded.

It is important to understand that this scheme may lose some information as time goes along. For example, if the sentence in Equation (7.4) were the true belief state, then neither $P_{3,1}$ nor $P_{2,2}$ would be provable individually and neither would appear in the 1-CNF belief state. (Exercise 7.27 explores one possible solution to this problem.) On the other hand, because every literal in the 1-CNF belief state is proved from the previous belief state, and the initial belief state is a true assertion, we know that entire 1-CNF belief state must be true. Thus, *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history*. As illustrated in Figure 7.21, the 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**, around the exact belief state. We see this idea of conservative approximations to complicated sets as a recurring theme in many areas of AI.



7.7.4 Making plans by propositional inference

The agent in Figure 7.20 uses logical inference to determine which squares are safe, but uses A* search to make plans. In this section, we show how to make plans by logical inference. The basic idea is very simple:

1. Construct a sentence that includes
 - (a) $Init^0$, a collection of assertions about the initial state;
 - (b) $Transition^1, \dots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t ;
 - (c) the assertion that the goal is achieved at time t : $HaveGold^t \wedge ClimbedOut^t$.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the planning problem is impossible.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned *true*. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure 7.22. It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t , up to some maximum conceivable plan length T_{\max} . In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

```

function SATPLAN(init, transition, goal,  $T_{\max}$ ) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
            $T_{\max}$ , an upper limit for plan length

  for  $t = 0$  to  $T_{\max}$  do
     $cnf \leftarrow$  TRANSLATE-TO-SAT(init, transition, goal,  $t$ )
     $model \leftarrow$  SAT-SOLVER( $cnf$ )
    if  $model$  is not null then
      return EXTRACT-SOLUTION( $model$ )
  return failure

```

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t . If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

The key step in using SATPLAN is the construction of the knowledge base. It might seem, on casual inspection, that the wumpus world axioms in Section 7.7.1 suffice for steps 1(a) and 1(b) above. There is, however, a significant difference between the requirements for entailment (as tested by ASK) and those for satisfiability. Consider, for example, the agent's location, initially $[1, 1]$, and suppose the agent's unambitious goal is to be in $[2, 1]$ at time 1. The initial knowledge base contains $L_{1,1}^0$ and the goal is $L_{2,1}^1$. Using ASK, we can prove $L_{2,1}^1$ if $Forward^0$ is asserted, and, reassuringly, we cannot prove $L_{2,1}^1$ if, say, $Shoot^0$ is asserted instead. Now, SATPLAN will find the plan $[Forward^0]$; so far, so good. Unfortunately, SATPLAN also finds the plan $[Shoot^0]$. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L_{2,1}^0$, that is, the agent can be in $[2, 1]$ at time 1 by being there at time 0 and shooting. One might ask, "Didn't we say the agent is in $[1, 1]$ at time 0?" Yes, we did, but we didn't tell the agent that it can't be in two places at once! For entailment, $L_{2,1}^0$ is unknown and cannot, therefore, be used in a proof; for satisfiability,

on the other hand, $L_{2,1}^0$ is unknown and can, therefore, be set to whatever value helps to make the goal true. For this reason, SATPLAN is a good debugging tool for knowledge bases because it reveals places where knowledge is missing. In this particular case, we can fix the knowledge base by asserting that, at each time step, the agent is in exactly one location, using a collection of sentences similar to those used to assert the existence of exactly one wumpus. Alternatively, we can assert $\neg L_{x,y}^0$ for all locations other than $[1, 1]$; the successor-state axiom for location takes care of subsequent time steps. The same fixes also work to make sure the agent has only one orientation.

SATPLAN has more surprises in store, however. The first is that it finds models with impossible actions, such as shooting with no arrow. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (7.3)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 10.14), but they *do not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.¹³ For example, we need to say, for each time t , that

$$\text{Shoot}^t \Rightarrow \text{HaveArrow}^t.$$

This ensures that if a plan selects the *Shoot* action at any time, it must be the case that the agent has an arrow at that time.

SATPLAN's second surprise is the creation of plans with multiple simultaneous actions. For example, it may come up with a model in which both Forward^0 and Shoot^0 are true, which is not allowed. To eliminate this problem, we introduce **action exclusion axioms**: for every pair of actions A_i^t and A_j^t we add the axiom

$$\neg A_i^t \vee \neg A_j^t.$$

It might be pointed out that walking forward and shooting at the same time is not so hard to do, whereas, say, shooting and grabbing at the same time is rather impractical. By imposing action exclusion axioms only on pairs of actions that really do interfere with each other, we can allow for plans that include multiple simultaneous actions—and because SATPLAN finds the shortest legal plan, we can be sure that it will take advantage of this capability.

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious “solutions.” Any model satisfying the propositional sentence will be a valid plan for the original problem. Modern SAT-solving technology makes the approach quite practical. For example, a DPLL-style solver has no difficulty in generating the 11-step solution for the wumpus world instance shown in Figure 7.2.

This section has described a declarative approach to agent construction: the agent works by a combination of asserting sentences in the knowledge base and performing logical inference. This approach has some weaknesses hidden in phrases such as “for each time t ” and

¹³ Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

PRECONDITION
AXIOMS

ACTION EXCLUSION
AXIOM

“for each square $[x, y]$.” For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion into the knowledge base. For a wumpus world of reasonable size—one comparable to a smallish computer game—we might need a 100×100 board and 1000 time steps, leading to knowledge bases with tens or hundreds of millions of sentences. Not only does this become rather impractical, but it also illustrates a deeper problem: we know something about the wumpus world—namely, that the “physics” works the same way across all squares and all time steps—that we cannot express directly in the language of propositional logic. To solve this problem, we need a more expressive language, one in which phrases like “for each time t ” and “for each square $[x, y]$ ” can be written in a natural way. First-order logic, described in Chapter 8, is such a language; in first-order logic a wumpus world of any size and duration can be described in about ten sentences rather than ten million or ten trillion.

7.8 SUMMARY

We have introduced knowledge-based agents and have shown how to define a logic with which such agents can reason about the world. The main points are as follows:

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences** in a **knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using these sentences to decide what action to take.
- A representation language is defined by its **syntax**, which specifies the structure of sentences, and its **semantics**, which defines the **truth** of each sentence in each **possible world** or **model**.
- The relationship of **entailment** between sentences is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all worlds where α is true. Equivalent definitions include the **validity** of the sentence $\alpha \Rightarrow \beta$ and the **unsatisfiability** of the sentence $\alpha \wedge \neg\beta$.
- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known true, known false, or completely unknown.
- The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local search methods and can often solve large problems quickly.

- **Inference rules** are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.
- **Local search** methods such as WALKSAT can be used to find solutions. Such algorithms are sound but not complete.
- Logical **state estimation** involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.
- Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environments.
- Propositional logic does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space, and universal patterns of relationships among objects.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

John McCarthy's paper "Programs with Common Sense" (McCarthy, 1958, 1968) promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. It also raised the flag of declarativism, pointing out that telling an agent what it needs to know is an elegant way to build software. Allen Newell's (1982) article "The Knowledge Level" makes the case that rational agents can be described and analyzed at an abstract level defined by the knowledge they possess rather than the programs they run. The declarative and procedural approaches to AI are analyzed in depth by Boden (1977). The debate was revived by, among others, Brooks (1991) and Nilsson (1991), and continues to this day (Shaparaou *et al.*, 2008). Meanwhile, the declarative approach has spread into other areas of computer science such as networking (Loo *et al.*, 2006).

Logic itself had its origins in ancient Greek philosophy and mathematics. Various logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, with their meaning, or with the validity of arguments in which they figure—are scattered in the works of Plato. The first known systematic study of logic was carried out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a treatise called the *Organon*. Aristotle's **sylogisms** were what we would now call inference rules. Although the syllogisms included elements of both propositional and first-order logic, the system as a whole lacked the compositional properties required to handle sentences of arbitrary complexity.

The closely related Megarian and Stoic schools (originating in the fifth century B.C. and continuing for several centuries thereafter) began the systematic study of the basic logical connectives. The use of truth tables for defining connectives is due to Philo of Megara. The

Stoics took five basic inference rules as valid without proof, including the rule we now call Modus Ponens. They derived a number of other rules from these five, using, among other principles, the deduction theorem (page 249) and were much clearer about the notion of proof than was Aristotle. A good account of the history of Megarian and Stoic logic is given by Benson Mates (1953).

The idea of reducing logical inference to a purely mechanical process applied to a formal language is due to Wilhelm Leibniz (1646–1716), although he had limited success in implementing the ideas. George Boole (1847) introduced the first comprehensive and workable system of formal logic in his book *The Mathematical Analysis of Logic*. Boole’s logic was closely modeled on the ordinary algebra of real numbers and used substitution of logically equivalent expressions as its primary inference method. Although Boole’s system still fell short of full propositional logic, it was close enough that other mathematicians could quickly fill in the gaps. Schröder (1877) described conjunctive normal form, while Horn form was introduced much later by Alfred Horn (1951). The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege’s (1879) *Begriffsschrift* (“Concept Writing” or “Conceptual Notation”).

The first mechanical device to carry out logical inferences was constructed by the third Earl of Stanhope (1753–1816). The Stanhope Demonstrator could handle syllogisms and certain inferences in the theory of probability. William Stanley Jevons, one of those who improved upon and extended Boole’s work, constructed his “logical piano” in 1869 to perform inferences in Boolean logic. An entertaining and instructive history of these and other early mechanical devices for reasoning is given by Martin Gardner (1968). The first published computer program for logical inference was the Logic Theorist of Newell, Shaw, and Simon (1957). This program was intended to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist’s results were published slightly earlier.

Truth tables as a method of testing validity or unsatisfiability in propositional logic were introduced independently by Emil Post (1921) and Ludwig Wittgenstein (1922). In the 1930s, a great deal of progress was made on inference methods for first-order logic. In particular, Gödel (1930) showed that a complete procedure for inference in first-order logic could be obtained via a reduction to propositional logic, using Herbrand’s theorem (Herbrand, 1930). We take up this history again in Chapter 9; the important point here is that the development of efficient propositional algorithms in the 1960s was motivated largely by the interest of mathematicians in an effective theorem prover for first-order logic. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first effective algorithm for propositional resolution but was in most cases much less efficient than the DPLL backtracking algorithm introduced two years later (1962). The full resolution rule and a proof of its completeness appeared in a seminal paper by J. A. Robinson (1965), which also showed how to do first-order reasoning without resort to propositional techniques.

Stephen Cook (1971) showed that deciding satisfiability of a sentence in propositional logic (the SAT problem) is NP-complete. Since deciding entailment is equivalent to deciding unsatisfiability, it is co-NP-complete. Many subsets of propositional logic are known for which the satisfiability problem is polynomially solvable; Horn clauses are one such subset.

The linear-time forward-chaining algorithm for Horn clauses is due to Dowling and Gallier (1984), who describe their algorithm as a dataflow process similar to the propagation of signals in a circuit.

Early theoretical investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. This potentially exciting fact became less exciting when Franco and Paull (1983) showed that the same problems could be solved in constant time simply by guessing random assignments. The random-generation method described in the chapter produces much harder problems. Motivated by the empirical success of local search on these problems, Koutsoupias and Papadimitriou (1992) showed that a simple hill-climbing algorithm can solve *almost all* satisfiability problem instances very quickly, suggesting that hard problems are rare. Moreover, Schöning (1999) exhibited a randomized hill-climbing algorithm whose *worst-case* expected run time on 3-SAT problems (that is, satisfiability of 3-CNF sentences) is $O(1.333^n)$ —still exponential, but substantially faster than previous worst-case bounds. The current record is $O(1.324^n)$ (Iwama and Tamaki, 2004). Achlioptas *et al.* (2004) and Alekhovich *et al.* (2005) exhibit families of 3-SAT instances for which all known DPLL-like algorithms require exponential running time.

On the practical side, efficiency gains in propositional solvers have been marked. Given ten minutes of computing time, the original DPLL algorithm in 1962 could only solve problems with no more than 10 or 15 variables. By 1995 the SATZ solver (Li and Anbulagan, 1997) could handle 1,000 variables, thanks to optimized data structures for indexing variables. Two crucial contributions were the **watched literal** indexing technique of Zhang and Stickel (1996), which makes unit propagation very efficient, and the introduction of clause (i.e., constraint) learning techniques from the CSP community by Bayardo and Schrag (1997). Using these ideas, and spurred by the prospect of solving industrial-scale circuit verification problems, Moskewicz *et al.* (2001) developed the CHAFF solver, which could handle problems with millions of variables. Beginning in 2002, SAT competitions have been held regularly; most of the winning entries have either been descendants of CHAFF or have used the same general approach. RSAT (Pipatsrisawat and Darwiche, 2007), the 2007 winner, falls in the latter category. Also noteworthy is MINISAT (Een and Sörensson, 2003), an open-source implementation available at <http://minisat.se> that is designed to be easily modified and improved. The current landscape of solvers is surveyed by Gomes *et al.* (2008).

Local search algorithms for satisfiability were tried by various authors throughout the 1980s; all of the algorithms were based on the idea of minimizing the number of unsatisfied clauses (Hansen and Jaumard, 1990). A particularly effective algorithm was developed by Gu (1989) and independently by Selman *et al.* (1992), who called it GSAT and showed that it was capable of solving a wide range of very hard problems very quickly. The WALKSAT algorithm described in the chapter is due to Selman *et al.* (1996).

The “phase transition” in satisfiability of random k -SAT problems was first observed by Simon and Dubois (1989) and has given rise to a great deal of theoretical and empirical research—due, in part, to the obvious connection to phase transition phenomena in statistical physics. Cheeseman *et al.* (1991) observed phase transitions in several CSPs and conjecture that all NP-hard problems have a phase transition. Crawford and Auton (1993) located the 3-SAT transition at a clause/variable ratio of around 4.26, noting that this coincides with a

sharp peak in the run time of their SAT solver. Cook and Mitchell (1997) provide an excellent summary of the early literature on the problem.

The current state of theoretical understanding is summarized by Achlioptas (2009). The **satisfiability threshold conjecture** states that, for each k , there is a sharp satisfiability threshold r_k , such that as the number of variables $n \rightarrow \infty$, instances below the threshold are *satisfiable* with probability 1, while those above the threshold are *unsatisfiable* with probability 1. The conjecture was not quite proved by Friedgut (1999): a sharp threshold exists but its location might depend on n even as $n \rightarrow \infty$. Despite significant progress in asymptotic analysis of the threshold location for large k (Achlioptas and Peres, 2004; Achlioptas *et al.*, 2007), all that can be proved for $k = 3$ is that it lies in the range $[3.52, 4.51]$. Current theory suggests that a peak in the run time of a SAT solver is not necessarily related to the satisfiability threshold, but instead to a phase transition in the solution distribution and structure of SAT instances. Empirical results due to Coarfa *et al.* (2003) support this view. In fact, algorithms such as **survey propagation** (Parisi and Zecchina, 2002; Maneva *et al.*, 2007) take advantage of special properties of random SAT instances near the satisfiability threshold and greatly outperform general SAT solvers on such instances.

The best sources for information on satisfiability, both theoretical and practical, are the *Handbook of Satisfiability* (Biere *et al.*, 2009) and the regular *International Conferences on Theory and Applications of Satisfiability Testing*, known as SAT.

The idea of building agents with propositional logic can be traced back to the seminal paper of McCulloch and Pitts (1943), which initiated the field of neural networks. Contrary to popular supposition, the paper was concerned with the implementation of a Boolean circuit-based agent design in the brain. Circuit-based agents, which perform computation by propagating signals in hardware circuits rather than running algorithms in general-purpose computers, have received little attention in AI, however. The most notable exception is the work of Stan Rosenschein (Rosenstein, 1985; Kaelbling and Rosenschein, 1990), who developed ways to compile circuit-based agents from declarative descriptions of the task environment. (Rosenstein's approach is described at some length in the second edition of this book.) The work of Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots—a topic we take up in Chapter 25. Brooks (1991) argues that circuit-based designs are *all* that is needed for AI—that representation and reasoning are cumbersome, expensive, and unnecessary. In our view, neither approach is sufficient by itself. Williams *et al.* (2003) show how a hybrid agent design not too different from our wumpus agent has been used to control NASA spacecraft, planning sequences of actions and diagnosing and recovering from faults.

The general problem of keeping track of a partially observable environment was introduced for state-based representations in Chapter 4. Its instantiation for propositional representations was studied by Amir and Russell (2003), who identified several classes of environments that admit efficient state-estimation algorithms and showed that for several other classes the problem is intractable. The **temporal-projection** problem, which involves determining what propositions hold true after an action sequence is executed, can be seen as a special case of state estimation with empty percepts. Many authors have studied this problem because of its importance in planning; some important hardness results were established by

Liberatore (1997). The idea of representing a belief state with propositions can be traced to Wittgenstein (1922).

Logical state estimation, of course, requires a logical representation of the effects of actions—a key problem in AI since the late 1950s. The dominant proposal has been the **situation calculus** formalism (McCarthy, 1963), which is couched within first-order logic. We discuss situation calculus, and various extensions and alternatives, in Chapters 10 and 12. The approach taken in this chapter—using temporal indices on propositional variables—is more restrictive but has the benefit of simplicity. The general approach embodied in the SATPLAN algorithm was proposed by Kautz and Selman (1992). Later generations of SATPLAN were able to take advantage of the advances in SAT solvers, described earlier, and remain among the most effective ways of solving difficult problems (Kautz, 2006).

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem unsolvable within first-order logic, and it spurred a great deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett (1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The solution of the frame problem with successor-state axioms is due to Ray Reiter (1991). Thielscher (1999) identifies the inferential frame problem as a separate idea and provides a solution. In retrospect, one can see that Rosenschein's (1985) agents were using circuits that implemented successor-state axioms, but Rosenschein did not notice that the frame problem was thereby largely solved. Foo (2001) explains why the discrete-event control theory models typically used by engineers do not have to explicitly deal with the frame problem: because they are dealing with prediction and control, not with explanation and reasoning about counterfactual situations.

Modern propositional solvers have wide applicability in industrial applications. The application of propositional inference in the synthesis of computer hardware is now a standard technique having many large-scale deployments (Nowick *et al.*, 1993). The SATMC satisfiability checker was used to detect a previously unknown vulnerability in a Web browser user sign-on protocol (Armando *et al.*, 2008).

The wumpus world was invented by Gregory Yob (1975). Ironically, Yob developed it because he was bored with games played on a rectangular grid: the topology of his original wumpus world was a dodecahedron, and we put it back in the boring old grid. Michael Genesereth was the first to suggest that the wumpus world be used as an agent testbed.

EXERCISES

7.1 Suppose the agent has progressed to the point shown in Figure 7.4(a), page 239, having perceived nothing in [1,1], a breeze in [2,1], and a stench in [1,2], and is now concerned with the contents of [1,3], [2,2], and [3,1]. Each of these can contain a pit, and at most one can contain a wumpus. Following the example of Figure 7.5, construct the set of possible worlds. (You should find 32 of them.) Mark the worlds in which the KB is true and those in which

each of the following sentences is true:

$\alpha_2 = \text{"There is no pit in [2,2]."}'$

$\alpha_3 = \text{"There is a wumpus in [1,3]."}'$

Hence show that $KB \models \alpha_2$ and $KB \models \alpha_3$.

7.2 (Adapted from Barwise and Etchemendy (1993).) Given the following, can you prove that the unicorn is mythical? How about magical? Horned?

If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

7.3 Consider the problem of deciding whether a propositional logic sentence is true in a given model.

- Write a recursive algorithm $\text{PL-TRUE?}(s, m)$ that returns *true* if and only if the sentence s is true in the model m (where m assigns a truth value for every symbol in s). The algorithm should run in time linear in the size of the sentence. (Alternatively, use a version of this function from the online code repository.)
- Give three examples of sentences that can be determined to be true or false in a *partial* model that does not specify a truth value for some of the symbols.
- Show that the truth value (if any) of a sentence in a partial model cannot be determined efficiently in general.
- Modify your PL-TRUE? algorithm so that it can sometimes judge truth from partial models, while retaining its recursive structure and linear run time. Give three examples of sentences whose truth in a partial model is *not* detected by your algorithm.
- Investigate whether the modified algorithm makes TT-ENTAILS? more efficient.

7.4 Which of the following are correct?

- $\text{False} \models \text{True}$.
- $\text{True} \models \text{False}$.
- $(A \wedge B) \models (A \Leftrightarrow B)$.
- $A \Leftrightarrow B \models A \vee B$.
- $A \Leftrightarrow B \models \neg A \vee B$.
- $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
- $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
- $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$.
- $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
- $(A \vee B) \wedge \neg(A \Rightarrow B)$ is satisfiable.
- $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ is satisfiable.
- $(A \Leftrightarrow B) \Leftrightarrow C$ has the same number of models as $(A \Leftrightarrow B)$ for any fixed set of proposition symbols that includes A, B, C .

7.5 Prove each of the following assertions:

- a. α is valid if and only if $\text{True} \models \alpha$.
- b. For any α , $\text{False} \models \alpha$.
- c. $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.
- d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid.
- e. $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.6 Prove, or find a counterexample to, each of the following assertions:

- a. If $\alpha \models \gamma$ or $\beta \models \gamma$ (or both) then $(\alpha \wedge \beta) \models \gamma$
- b. If $\alpha \models (\beta \wedge \gamma)$ then $\alpha \models \beta$ and $\alpha \models \gamma$.
- c. If $\alpha \models (\beta \vee \gamma)$ then $\alpha \models \beta$ or $\alpha \models \gamma$ (or both).

7.7 Consider a vocabulary with only four propositions, A , B , C , and D . How many models are there for the following sentences?

- a. $B \vee C$.
- b. $\neg A \vee \neg B \vee \neg C \vee \neg D$.
- c. $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$.

7.8 We have defined four binary logical connectives.

- a. Are there any others that might be useful?
- b. How many binary connectives can there be?
- c. Why are some of them not very useful?

7.9 Using a method of your choice, verify each of the equivalences in Figure 7.11 (page 249).

7.10 Decide whether each of the following sentences is valid, unsatisfiable, or neither. Verify your decisions using truth tables or the equivalence rules of Figure 7.11 (page 249).

- a. $\text{Smoke} \Rightarrow \text{Smoke}$
- b. $\text{Smoke} \Rightarrow \text{Fire}$
- c. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow (\neg\text{Smoke} \Rightarrow \neg\text{Fire})$
- d. $\text{Smoke} \vee \text{Fire} \vee \neg\text{Fire}$
- e. $((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire}) \Leftrightarrow ((\text{Smoke} \Rightarrow \text{Fire}) \vee (\text{Heat} \Rightarrow \text{Fire}))$
- f. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow ((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire})$
- g. $\text{Big} \vee \text{Dumb} \vee (\text{Big} \Rightarrow \text{Dumb})$

7.11 Any propositional logic sentence is logically equivalent to the assertion that each possible world in which it would be false is not the case. From this observation, prove that any sentence can be written in CNF.

7.12 Use resolution to prove the sentence $\neg A \wedge \neg B$ from the clauses in Exercise 7.20.

7.13 This exercise looks into the relationship between clauses and implication sentences.

- a. Show that the clause $(\neg P_1 \vee \dots \vee \neg P_m \vee Q)$ is logically equivalent to the implication sentence $(P_1 \wedge \dots \wedge P_m) \Rightarrow Q$.
- b. Show that every clause (regardless of the number of positive literals) can be written in the form $(P_1 \wedge \dots \wedge P_m) \Rightarrow (Q_1 \vee \dots \vee Q_n)$, where the P s and Q s are proposition symbols. A knowledge base consisting of such sentences is in **implicative normal form** or **Kowalski form** (Kowalski, 1979).
- c. Write down the full resolution rule for sentences in implicative normal form.

7.14 According to some political pundits, a person who is radical (R) is electable (E) if he/she is conservative (C), but otherwise is not electable.

- a. Which of the following are correct representations of this assertion?

- (i) $(R \wedge E) \iff C$
- (ii) $R \Rightarrow (E \iff C)$
- (iii) $R \Rightarrow ((C \Rightarrow E) \vee \neg E)$

- b. Which of the sentences in (a) can be expressed in Horn form?

7.15 This question considers representing satisfiability (SAT) problems as CSPs.

- a. Draw the constraint graph corresponding to the SAT problem

$$(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \dots \wedge (\neg X_{n-1} \vee X_n)$$

for the particular case $n = 5$.

- b. How many solutions are there for this general SAT problem as a function of n ?
- c. Suppose we apply BACKTRACKING-SEARCH (page 215) to find *all* solutions to a SAT CSP of the type given in (a). (To find *all* solutions to a CSP, we simply modify the basic algorithm so it continues searching after each solution is found.) Assume that variables are ordered X_1, \dots, X_n and *false* is ordered before *true*. How much time will the algorithm take to terminate? (Write an $O(\cdot)$ expression as a function of n .)
- d. We know that SAT problems in Horn form can be solved in linear time by forward chaining (unit propagation). We also know that every tree-structured binary CSP with discrete, finite domains can be solved in time linear in the number of variables (Section 6.5). Are these two facts connected? Discuss.

7.16 Explain why every nonempty propositional clause, by itself, is satisfiable. Prove rigorously that every set of five 3-SAT clauses is satisfiable, provided that each clause mentions exactly three distinct variables. What is the smallest set of such clauses that is unsatisfiable? Construct such a set.

7.17 A propositional 2-CNF expression is a conjunction of clauses, each containing *exactly* 2 literals, e.g.,

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee D) \wedge (\neg C \vee G) \wedge (\neg D \vee G).$$

- a. Prove using resolution that the above sentence entails G .

- b. Two clauses are *semantically distinct* if they are not logically equivalent. How many semantically distinct 2-CNF clauses can be constructed from n proposition symbols?
- c. Using your answer to (b), prove that propositional resolution always terminates in time polynomial in n given a 2-CNF sentence containing no more than n distinct symbols.
- d. Explain why your argument in (c) does not apply to 3-CNF.

7.18 Consider the following sentence:

$$[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party] .$$

- a. Determine, using enumeration, whether this sentence is valid, satisfiable (but not valid), or unsatisfiable.
- b. Convert the left-hand and right-hand sides of the main implication into CNF, showing each step, and explain how the results confirm your answer to (a).
- c. Prove your answer to (a) using resolution.

DISJUNCTIVE
NORMAL FORM

7.19 A sentence is in **disjunctive normal form** (DNF) if it is the disjunction of conjunctions of literals. For example, the sentence $(A \wedge B \wedge \neg C) \vee (\neg A \wedge C) \vee (B \wedge \neg C)$ is in DNF.

- a. Any propositional logic sentence is logically equivalent to the assertion that some possible world in which it would be true is in fact the case. From this observation, prove that any sentence can be written in DNF.
- b. Construct an algorithm that converts any sentence in propositional logic into DNF. (*Hint:* The algorithm is similar to the algorithm for conversion to CNF given in Section 7.5.2.)
- c. Construct a simple algorithm that takes as input a sentence in DNF and returns a satisfying assignment if one exists, or reports that no satisfying assignment exists.
- d. Apply the algorithms in (b) and (c) to the following set of sentences:

$$\begin{aligned} A &\Rightarrow B \\ B &\Rightarrow C \\ C &\Rightarrow \neg A . \end{aligned}$$

- e. Since the algorithm in (b) is very similar to the algorithm for conversion to CNF, and since the algorithm in (c) is much simpler than any algorithm for solving a set of sentences in CNF, why is this technique not used in automated reasoning?

7.20 Convert the following set of sentences to clausal form.

$$\begin{aligned} \text{S1: } A &\Leftrightarrow (B \vee E). \\ \text{S2: } E &\Rightarrow D. \\ \text{S3: } C \wedge F &\Rightarrow \neg B. \\ \text{S4: } E &\Rightarrow B. \\ \text{S5: } B &\Rightarrow F. \\ \text{S6: } B &\Rightarrow C \end{aligned}$$

Give a trace of the execution of DPLL on the conjunction of these clauses.

7.21 Is a randomly generated 4-CNF sentence with n symbols and m clauses more or less likely to be solvable than a randomly generated 3-CNF sentence with n symbols and m clauses? Explain.

7.22 Minesweeper, the well-known computer game, is closely related to the wumpus world. A minesweeper world is a rectangular grid of N squares with M invisible mines scattered among them. Any square may be probed by the agent; instant death follows if a mine is probed. Minesweeper indicates the presence of mines by revealing, in each probed square, the *number* of mines that are directly or diagonally adjacent. The goal is to probe every unmined square.

- Let $X_{i,j}$ be true iff square $[i, j]$ contains a mine. Write down the assertion that exactly two mines are adjacent to $[1,1]$ as a sentence involving some logical combination of $X_{i,j}$ propositions.
- Generalize your assertion from (a) by explaining how to construct a CNF sentence asserting that k of n neighbors contain mines.
- Explain precisely how an agent can use DPLL to prove that a given square does (or does not) contain a mine, ignoring the global constraint that there are exactly M mines in all.
- Suppose that the global constraint is constructed from your method from part (b). How does the number of clauses depend on M and N ? Suggest a way to modify DPLL so that the global constraint does not need to be represented explicitly.
- Are any conclusions derived by the method in part (c) invalidated when the global constraint is taken into account?
- Give examples of configurations of probe values that induce *long-range dependencies* such that the contents of a given unprobed square would give information about the contents of a far-distant square. (*Hint*: consider an $N \times 1$ board.)

7.23 How long does it take to prove $KB \models \alpha$ using DPLL when α is a literal *already contained in KB* ? Explain.

7.24 Trace the behavior of DPLL on the knowledge base in Figure 7.16 when trying to prove Q , and compare this behavior with that of the forward-chaining algorithm.

7.25 Write a successor-state axiom for the *Locked* predicate, which applies to doors, assuming the only actions available are *Lock* and *Unlock*.

7.26 Section 7.7.1 provides some of the successor-state axioms required for the wumpus world. Write down axioms for all remaining fluent symbols.



7.27 Modify the HYBRID-WUMPUS-AGENT to use the 1-CNF logical state estimation method described on page 271. We noted on that page that such an agent will not be able to acquire, maintain, and use more complex beliefs such as the disjunction $P_{3,1} \vee P_{2,2}$. Suggest a method for overcoming this problem by defining additional proposition symbols, and try it out in the wumpus world. Does it improve the performance of the agent?

8

FIRST-ORDER LOGIC

In which we notice that the world is blessed with many objects, some of which are related to other objects, and in which we endeavor to reason about them.

FIRST-ORDER LOGIC

In Chapter 7, we showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. Unfortunately, propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine **first-order logic**,¹ which is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

8.1 REPRESENTATION REVISITED

In this section, we discuss the nature of representation languages. Our discussion motivates the development of first-order logic, a much more expressive language than the propositional logic introduced in Chapter 7. We look at propositional logic and at other kinds of languages to understand what works and what fails. Our discussion will be cursory, compressing centuries of thought, trial, and error into a few paragraphs.

Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent, in a direct sense, only computational processes. Data structures within programs can represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the programming language statement $World[2,2] \leftarrow Pit$ is a fairly natural way to assert that there is a pit in square [2,2]. (Such representations might be considered *ad hoc*; database systems were developed precisely to provide a more general, domain-independent way to store and

¹ Also called **first-order predicate calculus**, sometimes abbreviated as **FOL** or **FOPC**.

retrieve facts.) What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This procedural approach can be contrasted with the **declarative** nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent.

A second drawback of data structures in programs (and of databases, for that matter) is the lack of any easy way to say, for example, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].” Programs can store a single value for each variable, and some systems allow the value to be “unknown,” but they lack the expressiveness required to handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of “ $S_{1,4} \wedge S_{1,2}$ ” is related to the meanings of “ $S_{1,4}$ ” and “ $S_{1,2}$.” It would be very strange if “ $S_{1,4}$ ” meant that there is a stench in square [1,4] and “ $S_{1,2}$ ” meant that there is a stench in square [1,2], but “ $S_{1,4} \wedge S_{1,2}$ ” meant that France and Poland drew 1–1 in last week’s ice hockey qualifying match. Clearly, noncompositionality makes life much more difficult for the reasoning system.

As we saw in Chapter 7, however, propositional logic lacks the expressive power to *concisely* describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

In English, on the other hand, it seems easy enough to say, once and for all, “Squares adjacent to pits are breezy.” The syntax and semantics of English somehow make it possible to describe the environment concisely.

8.1.1 The language of thought

Natural languages (such as English or Spanish) are very expressive indeed. We managed to write almost this whole book in natural language, with only occasional lapses into other languages (including logic, mathematics, and the language of diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as a declarative knowledge representation language. If we could uncover the rules for natural language, we could use it in representation and reasoning systems and gain the benefit of the billions of pages that have been written in natural language.

The modern view of natural language is that it serves as a medium for **communication** rather than pure representation. When a speaker points and says, “Look!” the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence “Look!” represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the **context** in which the sentence was spoken. Clearly, one could not store a sentence such as “Look!” in a knowledge base and expect to

AMBIGUITY

recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented. Natural languages also suffer from **ambiguity**, a problem for a representation language. As Pinker (1995) puts it: “When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can’t be words.”

The famous **Sapir–Whorf hypothesis** claims that our understanding of the world *is* strongly influenced by the language we speak. Whorf (1956) wrote “We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement to organize it this way—an agreement that holds throughout our speech community and is codified in the patterns of our language.” It is certainly true that different speech communities divide up the world differently. The French have two words “chaise” and “fauteuil,” for a concept that English speakers cover with one: “chair.” But English speakers can easily recognize the category *fauteuil* and give it a name—roughly “open-arm chair”—so does language really make a difference? Whorf relied mainly on intuition and speculation, but in the intervening years we actually have real data from anthropological, psychological and neurological studies.

For example, can you remember which of the following two phrases formed the opening of Section 8.1?

“In this section, we discuss the nature of representation languages . . .”

“This section covers the topic of knowledge representation languages . . .”

Wanner (1974) did a similar experiment and found that subjects made the right choice at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people process the words to form some kind of *nonverbal* representation.

More interesting is the case in which a concept is completely absent in a language. Speakers of the Australian aboriginal language Guugu Yimithirr have no words for relative directions, such as front, back, right, or left. Instead they use absolute directions, saying, for example, the equivalent of “I have a pain in my north arm.” This difference in language makes a difference in behavior: Guugu Yimithirr speakers are better at navigating in open terrain, while English speakers are better at placing the fork to the right of the plate.

Language also seems to influence thought through seemingly arbitrary grammatical features such as the gender of nouns. For example, “bridge” is masculine in Spanish and feminine in German. Boroditsky (2003) asked subjects to choose English adjectives to describe a photograph of a particular bridge. Spanish speakers chose *big*, *dangerous*, *strong*, and *towering*, whereas German speakers chose *beautiful*, *elegant*, *fragile*, and *slender*. Words can serve as anchor points that affect how we perceive the world. Loftus and Palmer (1974) showed experimental subjects a movie of an auto accident. Subjects who were asked “How fast were the cars going when they contacted each other?” reported an average of 32 mph, while subjects who were asked the question with the word “smashed” instead of “contacted” reported 41mph for the same cars in the same movie.

In a first-order logic reasoning system that uses CNF, we can see that the linguistic form “ $\neg(A \vee B)$ ” and “ $\neg A \wedge \neg B$ ” are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. Can we do that with the human brain? Until recently the answer was “no,” but now it is “maybe.” Mitchell *et al.* (2008) put subjects in an fMRI (functional magnetic resonance imaging) machine, showed them words such as “celery,” and imaged their brains. The researchers were then able to train a computer program to predict, from a brain image, what word the subject had been presented with. Given two choices (e.g., “celery” or “airplane”), the system predicts correctly 77% of the time. The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin *et al.*, 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 18 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions**—relations in which there is only one “value” for a given “input.” It is easy to start listing examples of objects, relations, and functions:

OBJECT
RELATION
FUNCTION

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .
- Relations: these can be unary relations or **properties** such as red, round, bogus, prime, multistoried . . . , or more general *n*-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .
- Functions: father of, best friend, third inning of, one more than, beginning of . . .

PROPERTY

Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:

- “One plus two equals three.”
Objects: one, two, three, one plus two; Relation: equals; Function: plus. (“One plus two” is a name for the object that is obtained by applying the function “plus” to the objects “one” and “two.” “Three” is another name for this object.)
- “Squares neighboring the wumpus are smelly.”
Objects: wumpus, squares; Property: smelly; Relation: neighboring.
- “Evil King John ruled England in 1200.”
Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

The language of **first-order logic**, whose syntax and semantics we define in the next section, is built around objects and relations. It has been so important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations among them. First-order logic can also express facts about *some* or *all* of the objects in the universe. This enables one to represent general laws or rules, such as the statement “Squares neighboring the wumpus are smelly.”

ONTOLOGICAL
COMMITMENT

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*. Mathematically, this commitment is expressed through the nature of the formal **models** with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns *true* or *false* to each proposition symbol (see Section 7.4.2).² First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. The formal models are correspondingly more complicated than those for propositional logic. Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) “first class” status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

TEMPORAL LOGIC

HIGHER-ORDER
LOGIC

EPISTEMOLOGICAL
COMMITMENT

A logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact. In both propositional and first-order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using **probability theory**, on the other hand,

² In contrast, facts in **fuzzy logic** have a **degree of truth** between 0 and 1. For example, the sentence “Vienna is a large city” might be true in our world only to degree 0.6 in fuzzy logic.

can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).³ For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. The ontological and epistemological commitments of five different logics are summarized in Figure 8.1.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

In the next section, we will launch into the details of first-order logic. Just as a student of physics requires some familiarity with mathematics, a student of AI must develop a talent for working with logical notation. On the other hand, it is also important *not* to get too concerned with the *specifics* of logical notation—after all, there are dozens of different versions. The main things to keep hold of are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

8.2 SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along.

8.2.1 Models for first-order logic

Recall from Chapter 7 that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic are much more interesting. First, they have objects in them! The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. (See Exercise 8.7 for a discussion of empty worlds.) Mathematically speaking, it doesn’t matter *what* these objects are—all that matters is *how many* there are in each particular model—but for pedagogical purposes we’ll use a concrete example. Figure 8.2 shows a model with five

³ It is important not to confuse the degree of belief in probability theory with the degree of truth in fuzzy logic. Indeed, some fuzzy systems allow uncertainty (degree of belief) about degrees of truth.

DOMAIN
DOMAIN ELEMENTS

objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

TUPLE

The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{ \langle \text{Richard the Lionheart, King John} \rangle, \langle \text{King John, Richard the Lionheart} \rangle \} . \tag{8.1}$$

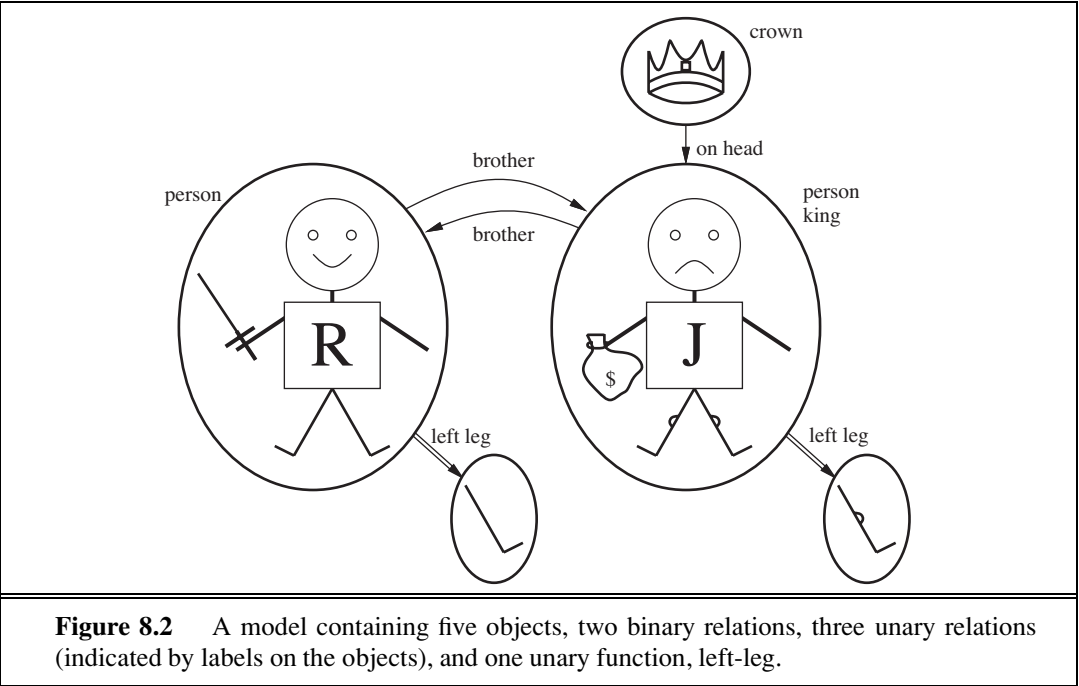
(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John’s head, so the “on head” relation contains just one tuple, $\langle \text{the crown, King John} \rangle$. The “brother” and “on head” relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the “person” property is true of both Richard and John; the “king” property is true only of John (presumably because Richard is dead at this point); and the “crown” property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary “left leg” function that includes the following mappings:

$$\begin{aligned} \langle \text{Richard the Lionheart} \rangle &\rightarrow \text{Richard’s left leg} \\ \langle \text{King John} \rangle &\rightarrow \text{John’s left leg} . \end{aligned} \tag{8.2}$$

TOTAL FUNCTIONS

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional “invisible”



object that is the left leg of everything that has no left leg, including itself. Fortunately, as long as one makes no assertions about the left legs of things that have no left legs, these technicalities are of no import.

So far, we have described the elements that populate models for first-order logic. The other essential part of a model is the link between those elements and the vocabulary of the logical sentences, which we explain next.

8.2.2 Symbols and interpretations

We turn now to the syntax of first-order logic. The impatient reader can obtain a complete description from the formal grammar in Figure 8.3.

CONSTANT SYMBOL
PREDICATE SYMBOL
FUNCTION SYMBOL

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols *Richard* and *John*; the predicate symbols *Brother*, *OnHead*, *Person*, *King*, and *Crown*; and the function symbol *LeftLeg*. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an **arity** that fixes the number of arguments.

ARITY

INTERPRETATION

As in propositional logic, every model must provide the information required to determine if any given sentence is true or false. Thus, in addition to its objects, relations, and functions, each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which a logician would call the **intended interpretation**—is as follows:

INTENDED
INTERPRETATION

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); *OnHead* refers to the “on head” relation that holds between the crown and King John; *Person*, *King*, and *Crown* refer to the sets of objects that are persons, kings, and crowns.
- *LeftLeg* refers to the “left leg” function, that is, the mapping given in Equation (8.2).

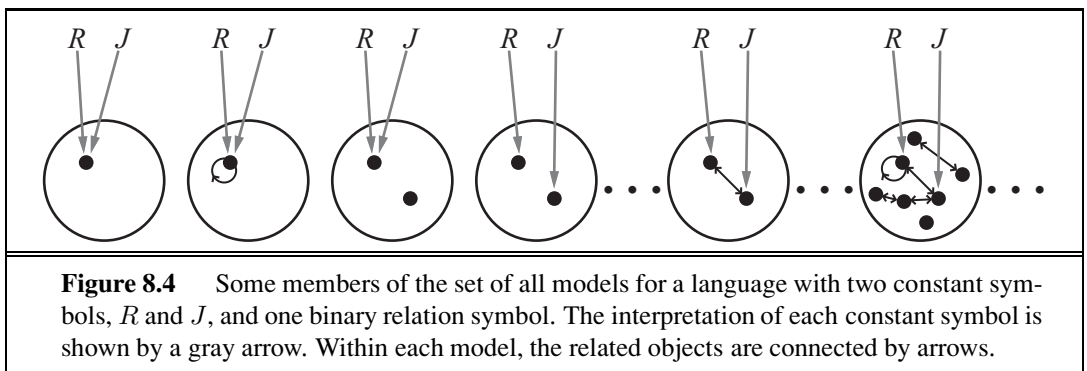
There are many other possible interpretations, of course. For example, one interpretation maps *Richard* to the crown and *John* to King John’s left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols *Richard* and *John*. Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown.⁴ If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

⁴ Later, in Section 8.2.8, we examine a semantics in which every object has exactly one name.

$$\begin{aligned}
 \text{Sentence} &\rightarrow \text{AtomicSentence} \mid \text{ComplexSentence} \\
 \text{AtomicSentence} &\rightarrow \text{Predicate} \mid \text{Predicate}(\text{Term}, \dots) \mid \text{Term} = \text{Term} \\
 \text{ComplexSentence} &\rightarrow (\text{Sentence}) \mid [\text{Sentence}] \\
 &\mid \neg \text{Sentence} \\
 &\mid \text{Sentence} \wedge \text{Sentence} \\
 &\mid \text{Sentence} \vee \text{Sentence} \\
 &\mid \text{Sentence} \Rightarrow \text{Sentence} \\
 &\mid \text{Sentence} \Leftrightarrow \text{Sentence} \\
 &\mid \text{Quantifier Variable}, \dots \text{Sentence} \\
 \\
 \text{Term} &\rightarrow \text{Function}(\text{Term}, \dots) \\
 &\mid \text{Constant} \\
 &\mid \text{Variable} \\
 \\
 \text{Quantifier} &\rightarrow \forall \mid \exists \\
 \text{Constant} &\rightarrow A \mid X_1 \mid \text{John} \mid \dots \\
 \text{Variable} &\rightarrow a \mid x \mid s \mid \dots \\
 \text{Predicate} &\rightarrow \text{True} \mid \text{False} \mid \text{After} \mid \text{Loves} \mid \text{Raining} \mid \dots \\
 \text{Function} &\rightarrow \text{Mother} \mid \text{LeftLeg} \mid \dots
 \end{aligned}$$

OPERATOR PRECEDENCE : $\neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.



In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one up to infinity—and in the way the constant symbols map to objects. If there are two constant symbols and one object, then both symbols must refer to the same object; but this can still happen even with more objects. When there are more objects than constant symbols, some of the objects will have no names. Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic (unlike propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. (See Exercise 8.5.) For the example in Figure 8.4, there are 137,506,194,466 models with six or fewer objects.

8.2.3 Terms

TERM

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression “King John’s left leg” rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use *LeftLeg(John)*. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a “subroutine call” that “returns a value.” There is no *LeftLeg* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of *LeftLeg*. This is something that cannot be done with subroutines in programming languages.⁵

The formal semantics of terms is straightforward. Consider a term $f(t_1, \dots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them d_1, \dots, d_n); and the term as a whole refers to the object that is the value of the function F applied to d_1, \dots, d_n . For example, suppose the *LeftLeg* function symbol refers to the function shown in Equation (8.2) and *John* refers to King John, then *LeftLeg(John)* refers to King John’s left leg. In this way, the interpretation fixes the referent of every term.

8.2.4 Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An **atomic**

⁵ **λ -expressions** provide a useful notation in which new function symbols are constructed “on the fly.” For example, the function that squares its argument can be written as $(\lambda x \ x \times x)$ and can be applied to arguments just like any other function symbol. A λ -expression can also be defined and used as a predicate symbol. (See Chapter 22.) The lambda operator in Lisp plays exactly the same role. Notice that the use of λ in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a λ -expression can be rewritten by “plugging in” its arguments to yield an equivalent sentence.

ATOMIC SENTENCE

ATOM

sentence (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

Brother(Richard, John).

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.⁶ Atomic sentences can have complex terms as arguments. Thus,

Married(Father(Richard), Mother(John))

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).



*An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

8.2.5 Complex sentences

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

$\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John})$
 $\text{Brother}(\text{Richard}, \text{John}) \wedge \text{Brother}(\text{John}, \text{Richard})$
 $\text{King}(\text{Richard}) \vee \text{King}(\text{John})$
 $\neg \text{King}(\text{Richard}) \Rightarrow \text{King}(\text{John}) .$

8.2.6 Quantifiers

QUANTIFIER

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification (\forall)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as “Squares neighboring the wumpus are smelly” and “All kings are persons” are the bread and butter of first-order logic. We deal with the first of these in Section 8.3. The second rule, “All kings are persons,” is written in first-order logic as

$\forall x \text{ King}(x) \Rightarrow \text{Person}(x) .$

\forall is usually pronounced “For all . . .”. (Remember that the upside-down A stands for “all.”) Thus, the sentence says, “For all x , if x is a king, then x is a person.” The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, *LeftLeg*(x). A term with no variables is called a **ground term**.

VARIABLE

GROUND TERM

Intuitively, the sentence $\forall x P$, where P is any logical expression, says that P is true for every object x . More precisely, $\forall x P$ is true in a given model if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each

EXTENDED
INTERPRETATION

⁶ We usually follow the argument-ordering convention that $P(x, y)$ is read as “ x is a P of y .”

extended interpretation specifies a domain element to which x refers.

This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

$x \rightarrow$ Richard the Lionheart,
 $x \rightarrow$ King John,
 $x \rightarrow$ Richard's left leg,
 $x \rightarrow$ John's left leg,
 $x \rightarrow$ the crown.

The universally quantified sentence $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$ is true in the original model if the sentence $\text{King}(x) \Rightarrow \text{Person}(x)$ is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king \Rightarrow Richard the Lionheart is a person.
 King John is a king \Rightarrow King John is a person.
 Richard's left leg is a king \Rightarrow Richard's left leg is a person.
 John's left leg is a king \Rightarrow John's left leg is a person.
 The crown is a king \Rightarrow the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of “All kings are persons”? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for \Rightarrow (Figure 7.8 on page 246), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table definition of \Rightarrow turns out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

$$\forall x \text{ King}(x) \wedge \text{Person}(x)$$

would be equivalent to asserting

Richard the Lionheart is a king \wedge Richard the Lionheart is a person,
 King John is a king \wedge King John is a person,
 Richard's left leg is a king \wedge Richard's left leg is a person,

and so on. Obviously, this does not capture what we want.

Existential quantification (\exists)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

$$\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John}) .$$

$\exists x$ is pronounced “There exists an x such that . . .” or “For some x . . .”.

Intuitively, the sentence $\exists x P$ says that P is true for at least one object x . More precisely, $\exists x P$ is true in a given model if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

Richard the Lionheart is a crown \wedge Richard the Lionheart is on John’s head;
 King John is a crown \wedge King John is on John’s head;
 Richard’s left leg is a crown \wedge Richard’s left leg is on John’s head;
 John’s left leg is a crown \wedge John’s left leg is on John’s head;
 The crown is a crown \wedge the crown is on John’s head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence “King John has a crown on his head.”⁷

Just as \Rightarrow appears to be the natural connective to use with \forall , \wedge is the natural connective to use with \exists . Using \wedge as the main connective with \forall led to an overly strong statement in the example in the previous section; using \Rightarrow with \exists usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists x \text{ Crown}(x) \Rightarrow \text{OnHead}(x, \text{John}) .$$

On the surface, this might look like a reasonable rendition of our sentence. Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown \Rightarrow Richard the Lionheart is on John’s head;
 King John is a crown \Rightarrow King John is on John’s head;
 Richard’s left leg is a crown \Rightarrow Richard’s left leg is on John’s head;

and so on. Now an implication is true if both premise and conclusion are true, *or if its premise is false*. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever *any* object fails to satisfy the premise; hence such sentences really do not say much at all.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as

$$\forall x \forall y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(x, y) .$$

⁷ There is a variant of the existential quantifier, usually written \exists^1 or $\exists!$, that means “There exists exactly one.” The same meaning can be expressed using equality statements.

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x) .$$

In other cases we will have mixtures. “Everybody loves somebody” means that for every person, there is someone that person loves:

$$\forall x \exists y \text{ Loves}(x, y) .$$

On the other hand, to say “There is someone who is loved by everyone,” we write

$$\exists y \forall x \text{ Loves}(x, y) .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x (\exists y \text{ Loves}(x, y))$ says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, $\exists y (\forall x \text{ Loves}(x, y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

$$\forall x (\text{Crown}(x) \vee (\exists x \text{ Brother}(\text{Richard}, x))) .$$

Here the x in $\text{Brother}(\text{Richard}, x)$ is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification. Another way to think of it is this: $\exists x \text{ Brother}(\text{Richard}, x)$ is a sentence about Richard (that he has a brother), not about x ; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z \text{ Brother}(\text{Richard}, z)$. Because this can be a source of confusion, we will always use different variable names with nested quantifiers.

Connections between \forall and \exists

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \neg \text{Likes}(x, \text{Parsnips}) \text{ is equivalent to } \neg \exists x \text{ Likes}(x, \text{Parsnips}) .$$

We can go one step further: “Everyone likes ice cream” means that there is no one who does not like ice cream:

$$\forall x \text{ Likes}(x, \text{IceCream}) \text{ is equivalent to } \neg \exists x \neg \text{Likes}(x, \text{IceCream}) .$$

Because \forall is really a conjunction over the universe of objects and \exists is a disjunction, it should not be surprising that they obey De Morgan’s rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{ll} \forall x \neg P & \equiv \neg \exists x P & \neg(P \vee Q) & \equiv \neg P \wedge \neg Q \\ \neg \forall x P & \equiv \exists x \neg P & \neg(P \wedge Q) & \equiv \neg P \vee \neg Q \\ \forall x P & \equiv \neg \exists x \neg P & P \wedge Q & \equiv \neg(\neg P \vee \neg Q) \\ \exists x P & \equiv \neg \forall x \neg P & P \vee Q & \equiv \neg(\neg P \wedge \neg Q) . \end{array}$$

Thus, we do not really need both \forall and \exists , just as we do not really need both \wedge and \vee . Still, readability is more important than parsimony, so we will keep both of the quantifiers.

8.2.7 Equality

EQUALITY SYMBOL

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to signify that two terms refer to the same object. For example,

$$\text{Father}(\text{John}) = \text{Henry}$$

says that the object referred to by $\text{Father}(\text{John})$ and the object referred to by Henry are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

$$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x = y) .$$

The sentence

$$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard})$$

does not have the intended meaning. In particular, it is true in the model of Figure 8.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x = y)$ rules out such models. The notation $x \neq y$ is sometimes used as an abbreviation for $\neg(x = y)$.

8.2.8 An alternative semantics?

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey.⁸ Can we capture this state of affairs by asserting

$$\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) ? \quad (8.3)$$

Not quite. First, this assertion is true in a model where Richard has only one brother—we need to add $\text{John} \neq \text{Geoffrey}$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

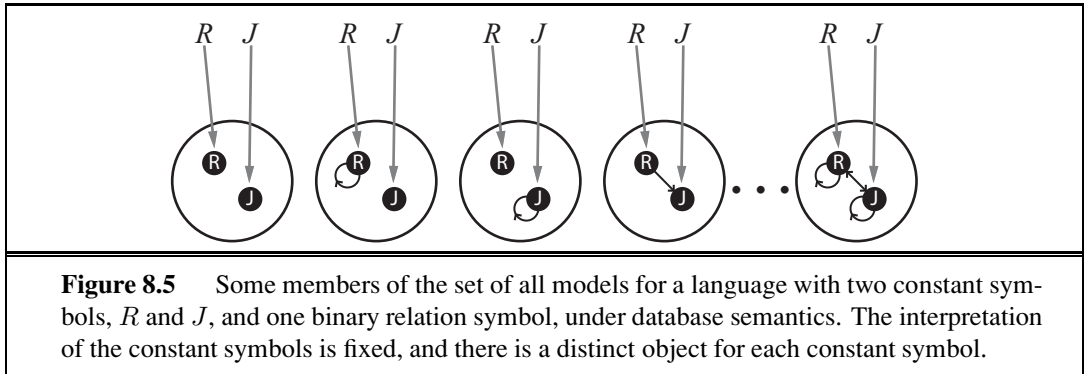
$$\begin{aligned} &\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) \wedge \text{John} \neq \text{Geoffrey} \\ &\wedge \forall x \text{ Brother}(x, \text{Richard}) \Rightarrow (x = \text{John} \vee x = \text{Geoffrey}) . \end{aligned}$$

For many purposes, this seems much more cumbersome than the corresponding natural-language expression. As a consequence, humans may make mistakes in translating their knowledge into first-order logic, resulting in unintuitive behaviors from logical reasoning systems that use the knowledge. Can we devise a semantics that allows a more straightforward logical expression?

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the so-called **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model

UNIQUE-NAMES
ASSUMPTION
CLOSED-WORLD
ASSUMPTION
DOMAIN CLOSURE

⁸ Actually he had four, the others being William and Henry.



DATABASE
SEMANTICS

contains no more domain elements than those named by the constant symbols. Under the resulting semantics, which we call **database semantics** to distinguish it from the standard semantics of first-order logic, the sentence Equation (8.3) does indeed state that Richard's two brothers are John and Geoffrey. Database semantics is also used in logic programming systems, as explained in Section 9.4.5.

It is instructive to consider the set of all possible models under database semantics for the same case as shown in Figure 8.4. Figure 8.5 shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are $2^4 = 16$ different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics. On the other hand, the database semantics requires definite knowledge of what the world contains.

This example brings up an important point: there is no one “correct” semantics for logic. The usefulness of any proposed semantics depends on how concise and intuitive it makes the expression of the kinds of knowledge we want to write down, and on how easy and natural it is to develop the corresponding rules of inference. Database semantics is most useful when we are certain about the identity of all the objects described in the knowledge base and when we have all the facts at hand; in other cases, it is quite awkward. For the rest of this chapter, we assume the standard semantics while noting instances in which this choice leads to cumbersome expressions.

8.3 USING FIRST-ORDER LOGIC

DOMAIN

Now that we have defined an expressive logical language, it is time to learn how to use it. The best way to do this is through examples. We have seen some simple sentences illustrating the various aspects of logical syntax; in this section, we provide more systematic representations of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at

the wumpus world. The next section contains a more substantial example (electronic circuits) and Chapter 12 covers everything in the universe.

8.3.1 Assertions and queries in first-order logic

ASSERTION

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

$$\begin{aligned} &\text{TELL}(KB, \text{King}(\text{John})) . \\ &\text{TELL}(KB, \text{Person}(\text{Richard})) . \\ &\text{TELL}(KB, \forall x \text{ King}(x) \Rightarrow \text{Person}(x)) . \end{aligned}$$

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK}(KB, \text{King}(\text{John}))$$

QUERY

returns *true*. Questions asked with ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two preceding assertions, the query

GOAL

$$\text{ASK}(KB, \text{Person}(\text{John}))$$

should also return *true*. We can ask quantified queries, such as

$$\text{ASK}(KB, \exists x \text{ Person}(x)) .$$

The answer is *true*, but this is perhaps not as helpful as we would like. It is rather like answering “Can you tell me the time?” with “Yes.” If we want to know what value of x makes the sentence true, we will need a different function, ASK VARS, which we call with

$$\text{ASK VARS}(KB, \text{Person}(x))$$

SUBSTITUTION

BINDING LIST

and which yields a stream of answers. In this case there will be two answers: $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. Such an answer is called a **substitution** or **binding list**. ASK VARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values. That is not the case with first-order logic; if KB has been told $\text{King}(\text{John}) \vee \text{King}(\text{Richard})$, then there is no binding to x for the query $\exists x \text{ King}(x)$, even though the query is true.

8.3.2 The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as “Elizabeth is the mother of Charles” and “Charles is the father of William” and rules such as “One’s grandmother is the mother of one’s parent.”

Clearly, the objects in our domain are people. We have two unary predicates, *Male* and *Female*. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, and *Uncle*. We use functions for *Mother* and *Father*, because every person has exactly one of each of these (at least according to nature’s design).

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's female parent:

$$\forall m, c \text{ Mother}(c) = m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m, c) .$$

One's husband is one's male spouse:

$$\forall w, h \text{ Husband}(h, w) \Leftrightarrow \text{Male}(h) \wedge \text{Spouse}(h, w) .$$

Male and female are disjoint categories:

$$\forall x \text{ Male}(x) \Leftrightarrow \neg \text{Female}(x) .$$

Parent and child are inverse relations:

$$\forall p, c \text{ Parent}(p, c) \Leftrightarrow \text{Child}(c, p) .$$

A grandparent is a parent of one's parent:

$$\forall g, c \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c) .$$

A sibling is another child of one's parents:

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow x \neq y \wedge \exists p \text{ Parent}(p, x) \wedge \text{Parent}(p, y) .$$

We could go on for several more pages like this, and Exercise 8.14 asks you to do just that.

Each of these sentences can be viewed as an **axiom** of the kinship domain, as explained in Section 7.1. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x, y \text{ } P(x, y) \Leftrightarrow \dots$. The axioms define the *Mother* function and the *Husband*, *Male*, *Parent*, *Grandparent*, and *Sibling* predicates in terms of other predicates. Our definitions “bottom out” at a basic set of predicates (*Child*, *Spouse*, and *Female*) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used *Parent*, *Spouse*, and *Male*. In some domains, as we show, there is no clearly identifiable basic set.

DEFINITION

THEOREM

Not all logical sentences about a domain are axioms. Some are **theorems**—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x) .$$

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return *true*.

From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time, rather like a physicist having to rederive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully. For example, there is no obvious definitive way to complete the sentence

$$\forall x \text{ Person}(x) \Leftrightarrow \dots$$

Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$$\begin{aligned} \forall x \text{ Person}(x) &\Rightarrow \dots \\ \forall x \dots &\Rightarrow \text{Person}(x). \end{aligned}$$

Axioms can also be “just plain facts,” such as *Male(Jim)* and *Spouse(Jim, Laura)*. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms. Often, one finds that the expected answers are not forthcoming—for example, from *Spouse(Jim, Laura)* one expects (under the laws of many countries) to be able to infer $\neg \text{Spouse}(\text{George}, \text{Laura})$; but this does not follow from the axioms given earlier—even after we add $\text{Jim} \neq \text{George}$ as suggested in Section 8.2.8. This is a sign that an axiom is missing. Exercise 8.8 asks the reader to supply it.

8.3.3 Numbers, sets, and lists

NATURAL NUMBERS

PEANO AXIOMS

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of **natural numbers** or non-negative integers. We need a predicate *NatNum* that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, *S* (successor). The **Peano axioms** define natural numbers and addition.⁹ Natural numbers are defined recursively:

$$\begin{aligned} \text{NatNum}(0) &. \\ \forall n \text{ NatNum}(n) &\Rightarrow \text{NatNum}(S(n)). \end{aligned}$$

That is, 0 is a natural number, and for every object *n*, if *n* is a natural number, then *S*(*n*) is a natural number. So the natural numbers are 0, *S*(0), *S*(*S*(0)), and so on. (After reading Section 8.2.8, you will notice that these axioms allow for other natural numbers besides the usual ones; see Exercise 8.12.) We also need axioms to constrain the successor function:

$$\begin{aligned} \forall n \quad 0 &\neq S(n). \\ \forall m, n \quad m &\neq n \Rightarrow S(m) \neq S(n). \end{aligned}$$

Now we can define addition in terms of the successor function:

$$\begin{aligned} \forall m \quad \text{NatNum}(m) &\Rightarrow + (0, m) = m. \\ \forall m, n \quad \text{NatNum}(m) \wedge \text{NatNum}(n) &\Rightarrow + (S(m), n) = S(+ (m, n)). \end{aligned}$$

The first of these axioms says that adding 0 to any natural number *m* gives *m* itself. Notice the use of the binary function symbol “+” in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order

INFIX

⁹ The Peano axioms also include the principle of induction, which is a sentence of second-order logic rather than of first-order logic. The importance of this distinction is explained in Chapter 9.

PREFIX

logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes

$$\forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow (m + 1) + n = (m + n) + 1.$$

This axiom reduces addition to repeated application of the successor function.

SYNTACTIC SUGAR

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be “desugared” to produce an equivalent sentence in ordinary first-order logic.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

SET

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, Set , which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:

$$\forall s \text{ Set}(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \text{ Set}(s_2) \wedge s = \{x|s_2\}).$$

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:

$$\neg \exists x, s \{x|s\} = \{\}.$$

3. Adjoining an element already in the set has no effect:

$$\forall x, s \ x \in s \Leftrightarrow s = \{x|s\}.$$

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y , where either y is the same as x or x is a member of s_2 :

$$\forall x, s \ x \in s \Leftrightarrow \exists y, s_2 (s = \{y|s_2\} \wedge (x = y \vee x \in s_2)).$$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

$$\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x \ x \in s_1 \Rightarrow x \in s_2).$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s_1, s_2 \quad x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2) .$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s_1, s_2 \quad x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2) .$$

LIST

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: *Nil* is the constant list with no elements; *Cons*, *Append*, *First*, and *Rest* are functions; and *Find* is the predicate that does for lists what *Member* does for sets. *List?* is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is []. The term *Cons*(*x*, *y*), where *y* is a nonempty list, is written [*x*|*y*]. The term *Cons*(*x*, *Nil*) (i.e., the list containing the element *x*) is written as [*x*]. A list of several elements, such as [*A*, *B*, *C*], corresponds to the nested term *Cons*(*A*, *Cons*(*B*, *Cons*(*C*, *Nil*))). Exercise 8.16 asks you to write out the axioms for lists.

8.3.4 The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The first-order axioms in this section are much more concise, capturing in a natural way exactly what we want to say.

Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

$$\text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5) .$$

Here, *Percept* is a binary predicate, and *Stench* and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$$\text{Turn}(\text{Right}), \text{Turn}(\text{Left}), \text{Forward}, \text{Shoot}, \text{Grab}, \text{Climb} .$$

To determine which is best, the agent program executes the query

$$\text{ASKVARS}(\exists a \text{ BestAction}(a, 5)) ,$$

which returns a binding list such as {*a*/Grab}. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:

$$\begin{aligned} \forall t, s, g, m, c \quad \text{Percept}([s, \text{Breeze}, g, m, c], t) &\Rightarrow \text{Breeze}(t) , \\ \forall t, s, b, m, c \quad \text{Percept}([s, b, \text{Glitter}, m, c], t) &\Rightarrow \text{Glitter}(t) , \end{aligned}$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 24. Notice the quantification over time *t*. In propositional logic, we would need copies of each sentence for each time step.

Simple “reflex” behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t \quad \text{Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t) .$$

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $BestAction(Grab, 5)$ —that is, *Grab* is the right thing to do.

We have represented the agent's inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square— $Square_{1,2}$ and so on—but then the fact that $Square_{1,2}$ and $Square_{1,3}$ are adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term $[1, 2]$. Adjacency of any two squares can be defined as

$$\forall x, y, a, b \text{ } Adjacent([x, y], [a, b]) \Leftrightarrow (x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)) .$$

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.¹⁰ It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant $Wumpus$ is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint).

The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t . We can fix the wumpus's location with $\forall t \text{ } At(Wumpus, [2, 2], t)$. We can then say that objects can only be at one location at a time:

$$\forall x, s_1, s_2, t \text{ } At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2 .$$

Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \text{ } At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s) .$$

It is useful to know that a *square* is breezy because we know that the pits cannot move about. Notice that $Breezy$ has no time argument.

Having discovered which places are breezy (or smelly) and, very important, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square (see R_2 and R_3 on page 247) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s \text{ } Breezy(s) \Leftrightarrow \exists r \text{ } Adjacent(r, s) \wedge Pit(r) . \quad (8.4)$$

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow (Equation (7.2) on page 267) becomes

$$\forall t \text{ } HaveArrow(t + 1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t)) .$$

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language description given in Chapter 7. The reader

¹⁰ Similarly, most of us do not name each bird that flies overhead as it migrates to warmer regions in winter. An ornithologist wishing to study migration patterns, survival rates, and so on *does* name each bird, by means of a ring on its leg, because individual birds must be tracked.

is invited to construct analogous axioms for the agent's location and orientation; in these cases, the axioms quantify over both space and time. As in the case of propositional state estimation, an agent can use logical inference with axioms of this kind to keep track of aspects of the world that are not directly observed. Chapter 10 goes into more depth on the subject of first-order successor-state axioms and their uses for constructing plans.

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

KNOWLEDGE ENGINEERING

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction—a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved. The approach we take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. *General-purpose* knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding, are discussed in Chapter 12.

8.4.1 The knowledge-engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the task.* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents in Chapter 2.
2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

KNOWLEDGE ACQUISITION

For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

ONTOLOGY

3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a “disembodied” knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.
6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7. *Debug the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.13) for finding the wumpus,

$$\forall s \text{ Smelly}(s) \Rightarrow \text{Adjacent}(\text{Home}(\text{Wumpus}), s),$$

instead of the biconditional, then the agent will never be able to prove the *absence* of wumpuses. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

$$\forall x \text{ NumOfLegs}(x, 4) \Rightarrow \text{Mammal}(x)$$

is false for reptiles, amphibians, and, more importantly, tables. *The falsehood of this sentence can be determined independently of the rest of the knowledge base.* In contrast,



a typical error in a program looks like this:

```
offset = position + 1.
```

It is impossible to tell whether this statement is correct without looking at the rest of the program to see whether, for example, `offset` is used to refer to the current position, or to one beyond the current position, or whether the value of `position` is changed by another statement and so `offset` should also be changed again.

To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

8.4.2 The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.6. We follow the seven-step process for knowledge engineering.

Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a

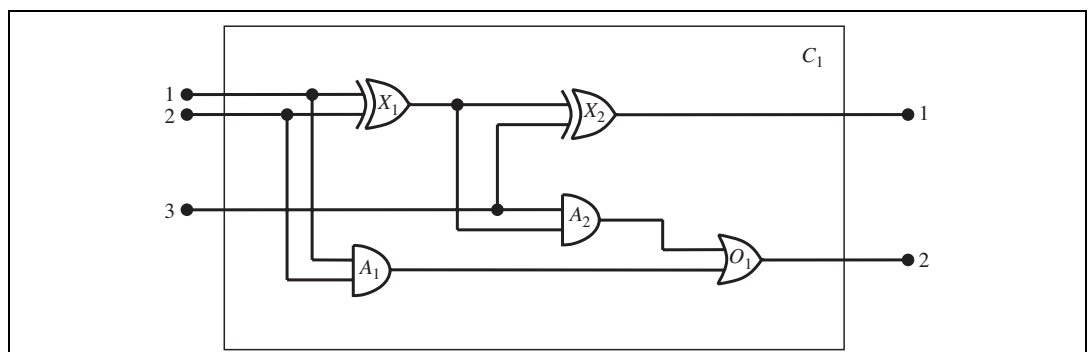


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

To reason about functionality and connectivity, we do not need to talk about the wires themselves, the paths they take, or the junctions where they come together. All that matters is the connections between terminals—we can say that one output terminal is connected to another input terminal without having to say what actually connects them. Other factors such as the size, shape, color, or cost of the various components are irrelevant to our analysis.

If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it. For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, $Gate(X_1)$. The behavior of each gate is determined by its type: one of the constants AND , OR , XOR , or NOT . Because a gate has exactly one type, a function is appropriate: $Type(X_1) = XOR$. Circuits, like gates, are identified by a predicate: $Circuit(C_1)$.

Next we consider terminals, which are identified by the predicate $Terminal(x)$. A gate or circuit can have one or more input terminals and one or more output terminals. We use the function $In(1, X_1)$ to denote the first input terminal for gate X_1 . A similar function Out is used for output terminals. The function $Arity(c, i, j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, $Connected$, which takes two terminals as arguments, as in $Connected(Out(1, X_1), In(1, X_2))$.

Finally, we need to know whether a signal is on or off. One possibility is to use a unary predicate, $On(t)$, which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as “What are all the possible values of the signals at the output terminals of circuit C_1 ?” We therefore introduce as objects two signal values, 1 and 0, and a function $Signal(t)$ that denotes the signal value for the terminal t .

Encode general knowledge of the domain

One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:

$$\forall t_1, t_2 \quad Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2) .$$

2. The signal at every terminal is either 1 or 0:

$$\forall t \text{ Terminal}(t) \Rightarrow \text{Signal}(t) = 1 \vee \text{Signal}(t) = 0 .$$
3. Connected is commutative:

$$\forall t_1, t_2 \text{ Connected}(t_1, t_2) \Leftrightarrow \text{Connected}(t_2, t_1) .$$
4. There are four types of gates:

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \Rightarrow k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR} \vee k = \text{NOT} .$$
5. An AND gate's output is 0 if and only if any of its inputs is 0:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{AND} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 0 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n, g)) = 0 .$$
6. An OR gate's output is 1 if and only if any of its inputs is 1:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{OR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 1 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n, g)) = 1 .$$
7. An XOR gate's output is 1 if and only if its inputs are different:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{XOR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2, g)) .$$
8. A NOT gate's output is different from its input:

$$\forall g \text{ Gate}(g) \wedge (\text{Type}(g) = \text{NOT}) \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) \neq \text{Signal}(\text{In}(1, g)) .$$
9. The gates (except for NOT) have two inputs and one output.

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) \neq \text{NOT} \Rightarrow \text{Arity}(g, 1, 1) .$$

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \wedge (k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR}) \Rightarrow$$

$$\text{Arity}(g, 2, 1)$$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:

$$\forall c, i, j \text{ Circuit}(c) \wedge \text{Arity}(c, i, j) \Rightarrow$$

$$\forall n (n \leq i \Rightarrow \text{Terminal}(\text{In}(c, n))) \wedge (n > i \Rightarrow \text{In}(c, n) = \text{Nothing}) \wedge$$

$$\forall n (n \leq j \Rightarrow \text{Terminal}(\text{Out}(c, n))) \wedge (n > j \Rightarrow \text{Out}(c, n) = \text{Nothing})$$
11. Gates, terminals, signals, gate types, and *Nothing* are all distinct.

$$\forall g, t \text{ Gate}(g) \wedge \text{Terminal}(t) \Rightarrow$$

$$g \neq t \neq 1 \neq 0 \neq \text{OR} \neq \text{AND} \neq \text{XOR} \neq \text{NOT} \neq \text{Nothing} .$$
12. Gates are circuits.

$$\forall g \text{ Gate}(g) \Rightarrow \text{Circuit}(g)$$

Encode the specific problem instance

The circuit shown in Figure 8.6 is encoded as circuit C_1 with the following description. First, we categorize the circuit and its component gates:

$$\begin{aligned} &\text{Circuit}(C_1) \wedge \text{Arity}(C_1, 3, 2) \\ &\text{Gate}(X_1) \wedge \text{Type}(X_1) = \text{XOR} \\ &\text{Gate}(X_2) \wedge \text{Type}(X_2) = \text{XOR} \\ &\text{Gate}(A_1) \wedge \text{Type}(A_1) = \text{AND} \\ &\text{Gate}(A_2) \wedge \text{Type}(A_2) = \text{AND} \\ &\text{Gate}(O_1) \wedge \text{Type}(O_1) = \text{OR} . \end{aligned}$$

Then, we show the connections between them:

$$\begin{array}{ll}
 \text{Connected}(\text{Out}(1, X_1), \text{In}(1, X_2)) & \text{Connected}(\text{In}(1, C_1), \text{In}(1, X_1)) \\
 \text{Connected}(\text{Out}(1, X_1), \text{In}(2, A_2)) & \text{Connected}(\text{In}(1, C_1), \text{In}(1, A_1)) \\
 \text{Connected}(\text{Out}(1, A_2), \text{In}(1, O_1)) & \text{Connected}(\text{In}(2, C_1), \text{In}(2, X_1)) \\
 \text{Connected}(\text{Out}(1, A_1), \text{In}(2, O_1)) & \text{Connected}(\text{In}(2, C_1), \text{In}(2, A_1)) \\
 \text{Connected}(\text{Out}(1, X_2), \text{Out}(1, C_1)) & \text{Connected}(\text{In}(3, C_1), \text{In}(2, X_2)) \\
 \text{Connected}(\text{Out}(1, O_1), \text{Out}(2, C_1)) & \text{Connected}(\text{In}(3, C_1), \text{In}(1, A_2)) .
 \end{array}$$

Pose queries to the inference procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

$$\begin{array}{l}
 \exists i_1, i_2, i_3 \text{ Signal}(\text{In}(1, C_1)) = i_1 \wedge \text{Signal}(\text{In}(2, C_1)) = i_2 \wedge \text{Signal}(\text{In}(3, C_1)) = i_3 \\
 \wedge \text{Signal}(\text{Out}(1, C_1)) = 0 \wedge \text{Signal}(\text{Out}(2, C_1)) = 1 .
 \end{array}$$

The answers are substitutions for the variables i_1 , i_2 , and i_3 such that the resulting sentence is entailed by the knowledge base. ASK VARS will give us three such substitutions:

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\} .$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\begin{array}{l}
 \exists i_1, i_2, i_3, o_1, o_2 \text{ Signal}(\text{In}(1, C_1)) = i_1 \wedge \text{Signal}(\text{In}(2, C_1)) = i_2 \\
 \wedge \text{Signal}(\text{In}(3, C_1)) = i_3 \wedge \text{Signal}(\text{Out}(1, C_1)) = o_1 \wedge \text{Signal}(\text{Out}(2, C_1)) = o_2 .
 \end{array}$$

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.26.) Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we fail to read Section 8.2.8 and hence forget to assert that $1 \neq 0$. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$$\exists i_1, i_2, o \text{ Signal}(\text{In}(1, C_1)) = i_1 \wedge \text{Signal}(\text{In}(2, C_1)) = i_2 \wedge \text{Signal}(\text{Out}(1, X_1)) ,$$

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1 :

$$\text{Signal}(\text{Out}(1, X_1)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, X_1)) \neq \text{Signal}(\text{In}(2, X_1)) .$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$\text{Signal}(\text{Out}(1, X_1)) = 1 \Leftrightarrow 1 \neq 0 .$$

Now the problem is apparent: the system is unable to infer that $\text{Signal}(\text{Out}(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

8.5 SUMMARY

This chapter has introduced **first-order logic**, a representation language that is far more powerful than propositional logic. The important points are as follows:

- Knowledge representation languages should be declarative, compositional, expressive, context independent, and unambiguous.
- Logics differ in their **ontological commitments** and **epistemological commitments**. While propositional logic commits only to the existence of facts, first-order logic commits to the existence of objects and relations and thereby gains expressive power.
- The syntax of first-order logic builds on that of propositional logic. It adds terms to represent objects, and has universal and existential quantifiers to construct assertions about all or some of the possible values of the quantified variables.
- A **possible world**, or **model**, for first-order logic includes a set of objects and an **interpretation** that maps constant symbols to objects, predicate symbols to relations among objects, and function symbols to functions on objects.
- An atomic sentence is true just when the relation named by the predicate holds between the objects named by the terms. **Extended interpretations**, which map quantifier variables to objects in the model, define the truth of quantified sentences.
- Developing a knowledge base in first-order logic requires a careful process of analyzing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although Aristotle's logic deals with generalizations over objects, it fell far short of the expressive power of first-order logic. A major barrier to its further development was its concentration on one-place predicates to the exclusion of many-place relational predicates. The first systematic treatment of relations was given by Augustus De Morgan (1864), who cited the following example to show the sorts of inferences that Aristotle's logic could not handle: "All horses are animals; therefore, the head of a horse is the head of an animal." This inference is inaccessible to Aristotle because any valid rule that can support this inference must first analyze the sentence using the two-place predicate " x is the head of y ." The logic of relations was studied in depth by Charles Sanders Peirce (1870, 2004).

True first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffsschrift* ("Concept Writing" or "Conceptual Notation"). Peirce (1883) also developed first-order logic independently of Frege, although slightly later. Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. The present notation for first-order logic is due substantially to Giuseppe Peano (1889), but the semantics is virtually identical to Frege's. Oddly enough, Peano's axioms were due in large measure to Grassmann (1861) and Dedekind (1888).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic, including the first proper treatment of the equality symbol. Löwenheim's results were further extended by Thoralf Skolem (1920). Alfred Tarski (1935, 1956) gave an explicit definition of truth and model-theoretic satisfaction in first-order logic, using set theory.

McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems. The prospects for logic-based AI were advanced significantly by Robinson's (1965) development of resolution, a complete procedure for first-order inference described in Chapter 9. The logicist approach took root at Stanford University. Cordell Green (1969a, 1969b) developed a first-order reasoning system, QA3, leading to the first attempts to build a logical robot at SRI (Fikes and Nilsson, 1971). First-order logic was applied by Zohar Manna and Richard Waldinger (1971) for reasoning about programs and later by Michael Genesereth (1984) for reasoning about circuits. In Europe, logic programming (a restricted form of first-order reasoning) was developed for linguistic analysis (Colmerauer *et al.*, 1973) and for general declarative systems (Kowalski, 1974). Computational logic was also well entrenched at Edinburgh through the LCF (Logic for Computable Functions) project (Gordon *et al.*, 1979). These developments are chronicled further in Chapters 9 and 12.

Practical applications built with first-order logic include a system for evaluating the manufacturing requirements for electronic products (Mannion, 2002), a system for reasoning about policies for file access and digital rights management (Halpern and Weissman, 2008), and a system for the automated composition of Web services (McIlraith and Zeng, 2001).

Reactions to the Whorf hypothesis (Whorf, 1956) and the problem of language and thought in general, appear in several recent books (Gumperz and Levinson, 1996; Bowerman and Levinson, 2001; Pinker, 2003; Gentner and Goldin-Meadow, 2003). The "theory" theory (Gopnik and Glymour, 2002; Tenenbaum *et al.*, 2007) views children's learning about the world as analogous to the construction of scientific theories. Just as the predictions of a machine learning algorithm depend strongly on the vocabulary supplied to it, so will the child's formulation of theories depend on the linguistic environment in which learning occurs.

There are a number of good introductory texts on first-order logic, including some by leading figures in the history of logic: Alfred Tarski (1941), Alonzo Church (1956), and W.V. Quine (1982) (which is one of the most readable). Enderton (1972) gives a more mathematically oriented perspective. A highly formal treatment of first-order logic, along with many more advanced topics in logic, is provided by Bell and Machover (1977). Manna and Waldinger (1985) give a readable introduction to logic from a computer science perspective, as do Huth and Ryan (2004), who concentrate on program verification. Barwise and Etchemendy (2002) take an approach similar to the one used here. Smullyan (1995) presents results concisely, using the tableau format. Gallier (1986) provides an extremely rigorous mathematical exposition of first-order logic, along with a great deal of material on its use in automated reasoning. *Logical Foundations of Artificial Intelligence* (Genesereth and Nilsson, 1987) is both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions, and there are two good handbooks: van Bentham and ter Meulen (1997) and Robinson and Voronkov (2001). The journal of record for the field of pure mathematical logic is the *Journal of Symbolic Logic*, whereas the *Journal of Applied Logic* deals with concerns closer to those of artificial intelligence.

EXERCISES

8.1 A logical knowledge base represents the world using a set of sentences with no explicit structure. An **analogical** representation, on the other hand, has physical structure that corresponds directly to the structure of the thing represented. Consider a road map of your country as an analogical representation of facts about the country—it represents facts with a map language. The two-dimensional structure of the map corresponds to the two-dimensional surface of the area.

- a. Give five examples of *symbols* in the map language.
- b. An *explicit* sentence is a sentence that the creator of the representation actually writes down. An *implicit* sentence is a sentence that results from explicit sentences because of properties of the analogical representation. Give three examples each of *implicit* and *explicit* sentences in the map language.
- c. Give three examples of facts about the physical structure of your country that cannot be represented in the map language.
- d. Give two examples of facts that are much easier to express in the map language than in first-order logic.
- e. Give two other examples of useful analogical representations. What are the advantages and disadvantages of each of these languages?

8.2 Consider a knowledge base containing just two sentences: $P(a)$ and $P(b)$. Does this knowledge base entail $\forall x P(x)$? Explain your answer in terms of models.

8.3 Is the sentence $\exists x, y \ x = y$ valid? Explain.

8.4 Write down a logical sentence such that every world in which it is true contains exactly one object.

8.5 Consider a symbol vocabulary that contains c constant symbols, p_k predicate symbols of each arity k , and f_k function symbols of each arity k , where $1 \leq k \leq A$. Let the domain size be fixed at D . For any given model, each predicate or function symbol is mapped onto a relation or function, respectively, of the same arity. You may assume that the functions in the model allow some input tuples to have no value for the function (i.e., the value is the invisible object). Derive a formula for the number of possible models for a domain with D elements. Don't worry about eliminating redundant combinations.

8.6 Which of the following are valid (necessarily true) sentences?

- a. $(\exists x \ x = x) \Rightarrow (\forall y \ \exists z \ y = z)$.
- b. $\forall x \ P(x) \vee \neg P(x)$.
- c. $\forall x \ \text{Smart}(x) \vee (x = x)$.

8.7 Consider a version of the semantics for first-order logic in which models with empty domains are allowed. Give at least two examples of sentences that are valid according to the

standard semantics but not according to the new semantics. Discuss which outcome makes more intuitive sense for your examples.

8.8 Does the fact $\neg \text{Spouse}(\text{George}, \text{Laura})$ follow from the facts $\text{Jim} \neq \text{George}$ and $\text{Spouse}(\text{Jim}, \text{Laura})$? If so, give a proof; if not, supply additional axioms as needed. What happens if we use Spouse as a unary function symbol instead of a binary predicate?

8.9 This exercise uses the function MapColor and predicates $\text{In}(x, y)$, $\text{Borders}(x, y)$, and $\text{Country}(x)$, whose arguments are geographical regions, along with constant symbols for various regions. In each of the following we give an English sentence and a number of candidate logical expressions. For each of the logical expressions, state whether it (1) correctly expresses the English sentence; (2) is syntactically invalid and therefore meaningless; or (3) is syntactically valid but does not express the meaning of the English sentence.

a. Paris and Marseilles are both in France.

- (i) $\text{In}(\text{Paris} \wedge \text{Marseilles}, \text{France})$.
- (ii) $\text{In}(\text{Paris}, \text{France}) \wedge \text{In}(\text{Marseilles}, \text{France})$.
- (iii) $\text{In}(\text{Paris}, \text{France}) \vee \text{In}(\text{Marseilles}, \text{France})$.

b. There is a country that borders both Iraq and Pakistan.

- (i) $\exists c \text{ Country}(c) \wedge \text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})$.
- (ii) $\exists c \text{ Country}(c) \Rightarrow [\text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})]$.
- (iii) $[\exists c \text{ Country}(c)] \Rightarrow [\text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})]$.
- (iv) $\exists c \text{ Border}(\text{Country}(c), \text{Iraq} \wedge \text{Pakistan})$.

c. All countries that border Ecuador are in South America.

- (i) $\forall c \text{ Country}(c) \wedge \text{Border}(c, \text{Ecuador}) \Rightarrow \text{In}(c, \text{SouthAmerica})$.
- (ii) $\forall c \text{ Country}(c) \Rightarrow [\text{Border}(c, \text{Ecuador}) \Rightarrow \text{In}(c, \text{SouthAmerica})]$.
- (iii) $\forall c [\text{Country}(c) \Rightarrow \text{Border}(c, \text{Ecuador})] \Rightarrow \text{In}(c, \text{SouthAmerica})$.
- (iv) $\forall c \text{ Country}(c) \wedge \text{Border}(c, \text{Ecuador}) \wedge \text{In}(c, \text{SouthAmerica})$.

d. No region in South America borders any region in Europe.

- (i) $\neg[\exists c, d \text{ In}(c, \text{SouthAmerica}) \wedge \text{In}(d, \text{Europe}) \wedge \text{Borders}(c, d)]$.
- (ii) $\forall c, d [\text{In}(c, \text{SouthAmerica}) \wedge \text{In}(d, \text{Europe})] \Rightarrow \neg \text{Borders}(c, d)$.
- (iii) $\neg \forall c \text{ In}(c, \text{SouthAmerica}) \Rightarrow \exists d \text{ In}(d, \text{Europe}) \wedge \neg \text{Borders}(c, d)$.
- (iv) $\forall c \text{ In}(c, \text{SouthAmerica}) \Rightarrow \forall d \text{ In}(d, \text{Europe}) \Rightarrow \neg \text{Borders}(c, d)$.

e. No two adjacent countries have the same map color.

- (i) $\forall x, y \neg \text{Country}(x) \vee \neg \text{Country}(y) \vee \neg \text{Borders}(x, y) \vee \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (ii) $\forall x, y (\text{Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y) \wedge \neg(x = y)) \Rightarrow \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (iii) $\forall x, y \text{ Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y) \wedge \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (iv) $\forall x, y (\text{Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y)) \Rightarrow \text{MapColor}(x \neq y)$.

8.10 Consider a vocabulary with the following symbols:

Occupation(p, o): Predicate. Person p has occupation o .

Customer($p1, p2$): Predicate. Person $p1$ is a customer of person $p2$.

Boss($p1, p2$): Predicate. Person $p1$ is a boss of person $p2$.

Doctor, Surgeon, Lawyer, Actor: Constants denoting occupations.

Emily, Joe: Constants denoting people.

Use these symbols to write the following assertions in first-order logic:

- a. Emily is either a surgeon or a lawyer.
- b. Joe is an actor, but he also holds another job.
- c. All surgeons are doctors.
- d. Joe does not have a lawyer (i.e., is not a customer of any lawyer).
- e. Emily has a boss who is a lawyer.
- f. There exists a lawyer all of whose customers are doctors.
- g. Every surgeon has a lawyer.

8.11 Complete the following exercises about logical sentences:

- a. Translate into *good, natural* English (no x s or y s!):

$$\begin{aligned} \forall x, y, l \text{ } \textit{SpeaksLanguage}(x, l) \wedge \textit{SpeaksLanguage}(y, l) \\ \Rightarrow \textit{Understands}(x, y) \wedge \textit{Understands}(y, x). \end{aligned}$$

- b. Explain why this sentence is entailed by the sentence

$$\begin{aligned} \forall x, y, l \text{ } \textit{SpeaksLanguage}(x, l) \wedge \textit{SpeaksLanguage}(y, l) \\ \Rightarrow \textit{Understands}(x, y). \end{aligned}$$

- c. Translate into first-order logic the following sentences:

- (i) Understanding leads to friendship.
- (ii) Friendship is transitive.

Remember to define all predicates, functions, and constants you use.

8.12 Rewrite the first two Peano axioms in Section 8.3.3 as a single axiom that defines *NatNum*(x) so as to exclude the possibility of natural numbers except for those generated by the successor function.

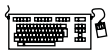
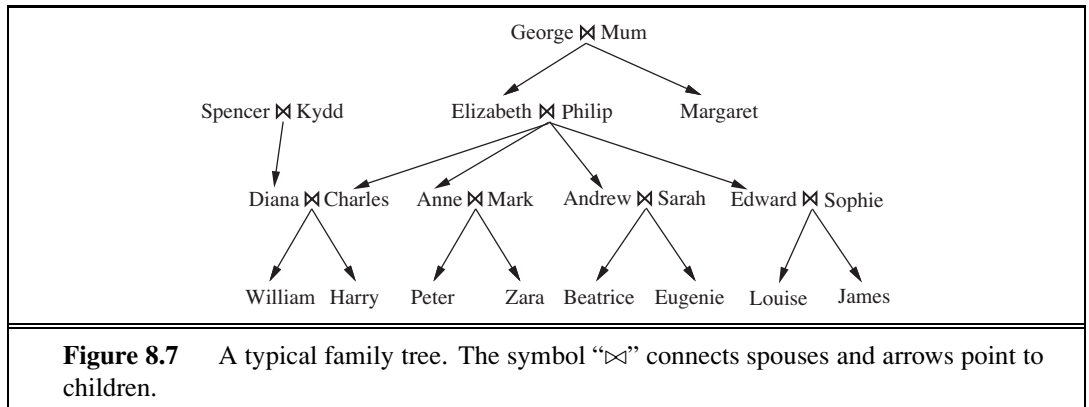
8.13 Equation (8.4) on page 306 defines the conditions under which a square is breezy. Here we consider two other ways to describe this aspect of the wumpus world.

DIAGNOSTIC RULE

- a. We can write **diagnostic rules** leading from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit; and if a square is not breezy, then no adjacent square contains a pit. Write these two rules in first-order logic and show that their conjunction is logically equivalent to Equation (8.4).

CAUSAL RULE

- b. We can write **causal rules** leading from cause to effect. One obvious causal rule is that a pit causes all adjacent squares to be breezy. Write this rule in first-order logic, explain why it is incomplete compared to Equation (8.4), and supply the missing axiom.



8.14 Write axioms describing the predicates *Grandchild*, *Greatgrandparent*, *Ancestor*, *Brother*, *Sister*, *Daughter*, *Son*, *FirstCousin*, *BrotherInLaw*, *SisterInLaw*, *Aunt*, and *Uncle*. Find out the proper definition of *m*th cousin *n* times removed, and write the definition in first-order logic. Now write down the basic facts depicted in the family tree in Figure 8.7. Using a suitable logical reasoning system, TELL it all the sentences you have written down, and ASK it who are Elizabeth’s grandchildren, Diana’s brothers-in-law, Zara’s great-grandparents, and Eugenie’s ancestors.

8.15 Explain what is wrong with the following proposed definition of the set membership predicate \in :

$$\begin{aligned} \forall x, s \quad x \in \{x|s\} \\ \forall x, s \quad x \in s \Rightarrow \forall y \quad x \in \{y|s\} . \end{aligned}$$

8.16 Using the set axioms as examples, write axioms for the list domain, including all the constants, functions, and predicates mentioned in the chapter.

8.17 Explain what is wrong with the following proposed definition of adjacent squares in the wumpus world:

$$\forall x, y \quad \text{Adjacent}([x, y], [x + 1, y]) \wedge \text{Adjacent}([x, y], [x, y + 1]) .$$

8.18 Write out the axioms required for reasoning about the wumpus’s location, using a constant symbol *Wumpus* and a binary predicate *At*(*Wumpus*, *Location*). Remember that there is only one wumpus.

8.19 Assuming predicates *Parent*(*p*, *q*) and *Female*(*p*) and constants *Joan* and *Kevin*, with the obvious meanings, express each of the following sentences in first-order logic. (You may use the abbreviation \exists^1 to mean “there exists exactly one.”)

- Joan has a daughter (possibly more than one, and possibly sons as well).
- Joan has exactly one daughter (but may have sons as well).
- Joan has exactly one child, a daughter.
- Joan and Kevin have exactly one child together.
- Joan has at least one child with Kevin, and no children with anyone else.

8.20 Arithmetic assertions can be written in first-order logic with the predicate symbol $<$, the function symbols $+$ and \times , and the constant symbols 0 and 1. Additional predicates can also be defined with biconditionals.

- Represent the property “ x is an even number.”
- Represent the property “ x is prime.”
- Goldbach’s conjecture is the conjecture (unproven as yet) that every even number is equal to the sum of two primes. Represent this conjecture as a logical sentence.

8.21 In Chapter 6, we used equality to indicate the relation between a variable and its value. For instance, we wrote $WA = red$ to mean that Western Australia is colored red. Representing this in first-order logic, we must write more verbosely $ColorOf(WA) = red$. What incorrect inference could be drawn if we wrote sentences such as $WA = red$ directly as logical assertions?

8.22 Write in first-order logic the assertion that every key and at least one of every pair of socks will eventually be lost forever, using only the following vocabulary: $Key(x)$, x is a key; $Sock(x)$, x is a sock; $Pair(x, y)$, x and y are a pair; Now , the current time; $Before(t_1, t_2)$, time t_1 comes before time t_2 ; $Lost(x, t)$, object x is lost at time t .

8.23 For each of the following sentences in English, decide if the accompanying first-order logic sentence is a good translation. If not, explain why not and correct it. (Some sentences may have more than one error!)

- No two people have the same social security number.

$$\neg \exists x, y, n \text{ Person}(x) \wedge \text{Person}(y) \Rightarrow [\text{HasSS}\#(x, n) \wedge \text{HasSS}\#(y, n)].$$

- John’s social security number is the same as Mary’s.

$$\exists n \text{ HasSS}\#(\text{John}, n) \wedge \text{HasSS}\#(\text{Mary}, n).$$

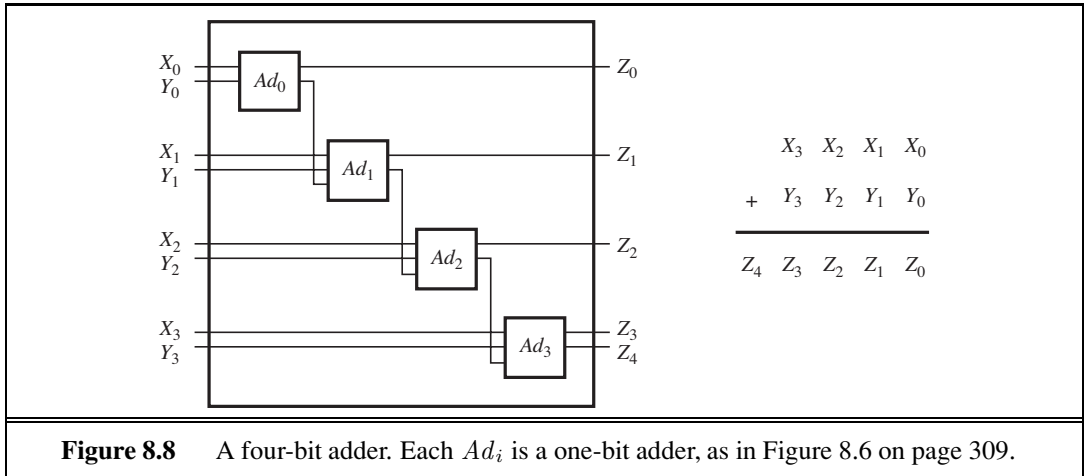
- Everyone’s social security number has nine digits.

$$\forall x, n \text{ Person}(x) \Rightarrow [\text{HasSS}\#(x, n) \wedge \text{Digits}(n, 9)].$$

- Rewrite each of the above (uncorrected) sentences using a function symbol $SS\#$ instead of the predicate $\text{HasSS}\#$.

8.24 Represent the following sentences in first-order logic, using a consistent vocabulary (which you must define):

- Some students took French in spring 2001.
- Every student who takes French passes it.
- Only one student took Greek in spring 2001.
- The best score in Greek is always higher than the best score in French.
- Every person who buys a policy is smart.
- No person buys an expensive policy.
- There is an agent who sells policies only to people who are not insured.



- h. There is a barber who shaves all men in town who do not shave themselves.
- i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.
- j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK citizen by descent.
- k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.
- l. All Greeks speak the same language. (Use $Speaks(x, l)$ to mean that person x speaks language l .)

8.25 Write a general set of facts and axioms to represent the assertion “Wellington heard about Napoleon’s death” and to correctly answer the question “Did Napoleon hear about Wellington’s death?”



8.26 Extend the vocabulary from Section 8.4 to define addition for n -bit binary numbers. Then encode the description of the four-bit adder in Figure 8.8, and pose the queries needed to verify that it is in fact correct.

8.27 Obtain a passport application for your country, identify the rules determining eligibility for a passport, and translate them into first-order logic, following the steps outlined in Section 8.4.

8.28 Consider a first-order logical knowledge base that describes worlds containing people, songs, albums (e.g., “Meet the Beatles”) and disks (i.e., particular physical instances of CDs). The vocabulary contains the following symbols:

$CopyOf(d, a)$: Predicate. Disk d is a copy of album a .

$Owns(p, d)$: Predicate. Person p owns disk d .

$Sings(p, s, a)$: Album a includes a recording of song s sung by person p .

$Wrote(p, s)$: Person p wrote song s .

$McCartney, Gershwin, BHoliday, Joe, EleanorRigby, TheManILove, Revolver$: Constants with the obvious meanings.

Express the following statements in first-order logic:

- a. Gershwin wrote “The Man I Love.”
- b. Gershwin did not write “Eleanor Rigby.”
- c. Either Gershwin or McCartney wrote “The Man I Love.”
- d. Joe has written at least one song.
- e. Joe owns a copy of *Revolver*.
- f. Every song that McCartney sings on *Revolver* was written by McCartney.
- g. Gershwin did not write any of the songs on *Revolver*.
- h. Every song that Gershwin wrote has been recorded on some album. (Possibly different songs are recorded on different albums.)
- i. There is a single album that contains every song that Joe has written.
- j. Joe owns a copy of an album that has Billie Holiday singing “The Man I Love.”
- k. Joe owns a copy of every album that has a song sung by McCartney. (Of course, each different album is instantiated in a different physical CD.)
- l. Joe owns a copy of every album on which all the songs are sung by Billie Holiday.

9

INFERENCE IN FIRST-ORDER LOGIC

In which we define effective procedures for answering questions posed in first-order logic.

Chapter 7 showed how sound and complete inference can be achieved for propositional logic. In this chapter, we extend those results to obtain algorithms that can answer any answerable question stated in first-order logic. Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at potentially great expense. Section 9.2 describes the idea of **unification**, showing how it can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms. **Forward chaining** and its applications to **deductive databases** and **production systems** are covered in Section 9.3; **backward chaining** and **logic programming** systems are developed in Section 9.4. Forward and backward chaining can be very efficient, but are applicable only to knowledge bases that can be expressed as sets of Horn clauses. General first-order sentences require resolution-based **theorem proving**, which is described in Section 9.5.

9.1 PROPOSITIONAL VS. FIRST-ORDER INFERENCE

This section and the next introduce the ideas underlying modern logical inference systems. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that *first-order* inference can be done by converting the knowledge base to *propositional* logic and using *propositional* inference, which we already know how to do. The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

9.1.1 Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x) .$$

Then it seems quite permissible to infer any of the following sentences:

$$\begin{aligned} & King(John) \wedge Greedy(John) \Rightarrow Evil(John) \\ & King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard) \\ & King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)) . \\ & \vdots \end{aligned}$$

UNIVERSAL
INSTANTIATION
GROUND TERM

The rule of **Universal Instantiation** (UI for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.¹ To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $SUBST(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α . Then the rule is written

$$\frac{\forall v \ \alpha}{SUBST(\{v/g\}, \alpha)}$$

for any variable v and ground term g . For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

EXISTENTIAL
INSTANTIATION

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α , variable v , and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \ \alpha}{SUBST(\{v/k\}, \alpha)} .$$

For example, from the sentence

$$\exists x \ Crown(x) \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(x^y)/dy = x^y$ for x . We can give this number a name, such as e , but it would be a mistake to give it the name of an existing object, such as π . In logic, the new name is called a **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**, which we cover in Section 9.5.

SKOLEM CONSTANT

Whereas Universal Instantiation can be applied many times to produce many different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need $\exists x \ Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$. Strictly speaking, the new knowledge base is not logically equivalent to the old, but it can be shown to be **inferentially equivalent** in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

INFERENTIAL
EQUIVALENCE

¹ Do not confuse these substitutions with the extended interpretations used to define the semantics of quantifiers. The substitution replaces a variable with a term (a piece of syntax) to produce a new sentence, whereas an interpretation maps a variable to an object in the domain.

9.1.2 Reduction to propositional inference

Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. In this section we give the main ideas; the details are given in Section 9.5.

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations. For example, suppose our knowledge base contains just the sentences

$$\begin{aligned} &\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x) \\ &\text{King}(\text{John}) \\ &\text{Greedy}(\text{John}) \\ &\text{Brother}(\text{Richard}, \text{John}) . \end{aligned} \tag{9.1}$$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. We obtain

$$\begin{aligned} &\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John}) \\ &\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard}) , \end{aligned}$$

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences— $\text{King}(\text{John})$, $\text{Greedy}(\text{John})$, and so on—as proposition symbols. Therefore, we can apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as $\text{Evil}(\text{John})$.

This technique of **propositionalization** can be made completely general, as we show in Section 9.5; that is, every first-order knowledge base and query can be propositionalized in such a way that entailment is preserved. Thus, we have a complete decision procedure for entailment . . . or perhaps not. There is a problem: when the knowledge base includes a function symbol, the set of possible ground-term substitutions is infinite! For example, if the knowledge base mentions the *Father* symbol, then infinitely many nested terms such as $\text{Father}(\text{Father}(\text{Father}(\text{John})))$ can be constructed. Our propositional algorithms will have difficulty with an infinitely large set of sentences.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a *finite* subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols (*Richard* and *John*), then all terms of depth 1 ($\text{Father}(\text{Richard})$ and $\text{Father}(\text{John})$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**—that is, any entailed sentence can be proved. This is a major achievement, given that the space of possible models is infinite. On the other hand, we do not know until the proof is done that the sentence *is* entailed! What happens when the sentence is *not* entailed? Can we tell? Well, for first-order logic, it turns out that we cannot. Our proof procedure can go on and on, generating more and more deeply nested terms, but we will not know whether it is stuck in a hopeless loop or whether the proof is just about to pop out. This is very much



like the halting problem for Turing machines. Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, the inevitability of this state of affairs. *The question of entailment for first-order logic is **semidecidable**—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

9.2 UNIFICATION AND LIFTING

The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the early 1960s) will have noticed that the propositionalization approach is rather inefficient. For example, given the query $Evil(x)$ and the knowledge base in Equation (9.1), it seems perverse to generate sentences such as $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$. Indeed, the inference of $Evil(John)$ from the sentences

$$\begin{aligned} &\forall x \text{ } King(x) \wedge Greedy(x) \Rightarrow Evil(x) \\ &King(John) \\ &Greedy(John) \end{aligned}$$

seems completely obvious to a human being. We now show how to make it completely obvious to a computer.

9.2.1 A first-order inference rule

The inference that John is evil—that is, that $\{x/John\}$ solves the query $Evil(x)$ —works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ . In this case, the substitution $\theta = \{x/John\}$ achieves that aim.

We can actually make the inference step do even more work. Suppose that instead of knowing $Greedy(John)$, we know that *everyone* is greedy:

$$\forall y \text{ } Greedy(y) . \tag{9.2}$$

Then we would still like to be able to conclude that $Evil(John)$, because we know that John is a king (given) and John is greedy (because everyone is greedy). What we need for this to work is to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge-base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the conclusion of the implication.

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**:² For atomic sentences p_i , p_i' , and q , where there is a substitution θ

such that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i ,

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}.$$

There are $n + 1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q . For our example:

$$\begin{array}{ll} p_1' \text{ is } King(John) & p_1 \text{ is } King(x) \\ p_2' \text{ is } Greedy(y) & p_2 \text{ is } Greedy(x) \\ \theta \text{ is } \{x/John, y/John\} & q \text{ is } Evil(x) \\ \text{SUBST}(\theta, q) \text{ is } Evil(John). \end{array}$$

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ ,

$$p \models \text{SUBST}(\theta, p)$$

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from p_1', \dots, p_n' we can infer

$$\text{SUBST}(\theta, p_1') \wedge \dots \wedge \text{SUBST}(\theta, p_n')$$

and from the implication $p_1 \wedge \dots \wedge p_n \Rightarrow q$ we can infer

$$\text{SUBST}(\theta, p_1) \wedge \dots \wedge \text{SUBST}(\theta, p_n) \Rightarrow \text{SUBST}(\theta, q).$$

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i ; therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

LIFTING

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. We will see in the rest of this chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

9.2.2 Unification

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query $\text{AskVars}(\text{Knows}(\text{John}, x))$: whom does John know? Answers to this query can be found

UNIFICATION

UNIFIER

² Generalized Modus Ponens is more general than Modus Ponens (page 249) in the sense that the known facts and the premise of the implication need match only up to a substitution, rather than exactly. On the other hand, Modus Ponens allows any sentence α as the premise, rather than just a conjunction of atomic sentences.

by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$$\begin{aligned} \text{UNIFY}(Knows(John, x), Knows(John, Jane)) &= \{x/Jane\} \\ \text{UNIFY}(Knows(John, x), Knows(y, Bill)) &= \{x/Bill, y/John\} \\ \text{UNIFY}(Knows(John, x), Knows(y, Mother(y))) &= \{y/John, x/Mother(John)\} \\ \text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) &= \text{fail} . \end{aligned}$$

The last unification fails because x cannot take on the values *John* and *Elizabeth* at the same time. Now, remember that $Knows(x, Elizabeth)$ means “Everyone knows Elizabeth,” so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x . The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

$$\text{UNIFY}(Knows(John, x), Knows(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\} .$$

Exercise 9.12 delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\text{UNIFY}(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives $Knows(John, John)$. The second result could be obtained from the first by an additional substitution $\{z/John\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or MGU) that is unique up to renaming and substitution of variables. (For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.) In this case it is $\{y/John, x/z\}$.

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously “side by side,” building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, $S(x)$ can’t unify with $S(S(x))$. This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

9.2.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some

STANDARDIZING
APART

MOST GENERAL
UNIFIER

OCCUR CHECK

```

function UNIFY( $x, y, \theta$ ) returns a substitution to make  $x$  and  $y$  identical
  inputs:  $x$ , a variable, constant, list, or compound expression
            $y$ , a variable, constant, list, or compound expression
            $\theta$ , the substitution built up so far (optional, defaults to empty)

  if  $\theta = \text{failure}$  then return failure
  else if  $x = y$  then return  $\theta$ 
  else if VARIABLE?( $x$ ) then return UNIFY-VAR( $x, y, \theta$ )
  else if VARIABLE?( $y$ ) then return UNIFY-VAR( $y, x, \theta$ )
  else if COMPOUND?( $x$ ) and COMPOUND?( $y$ ) then
    return UNIFY( $x$ .ARGS,  $y$ .ARGS, UNIFY( $x$ .OP,  $y$ .OP,  $\theta$ ))
  else if LIST?( $x$ ) and LIST?( $y$ ) then
    return UNIFY( $x$ .REST,  $y$ .REST, UNIFY( $x$ .FIRST,  $y$ .FIRST,  $\theta$ ))
  else return failure

```

```

function UNIFY-VAR( $var, x, \theta$ ) returns a substitution

  if  $\{var/val\} \in \theta$  then return UNIFY( $val, x, \theta$ )
  else if  $\{x/val\} \in \theta$  then return UNIFY( $var, val, \theta$ )
  else if OCCUR-CHECK?( $var, x$ ) then return failure
  else return add  $\{var/x\}$  to  $\theta$ 

```

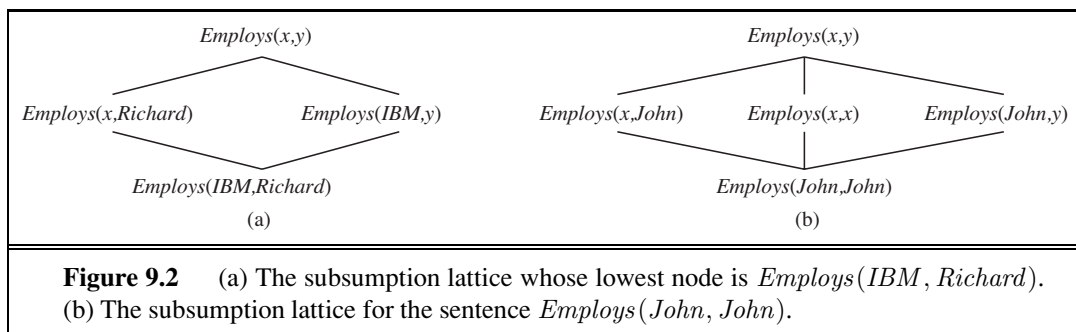
Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B) .

sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $Knows(John, x)$ —is an instance of FETCHing.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works, and it's all you need to understand the rest of the chapter. The remainder of this section outlines ways to make retrieval more efficient; it can be skipped on first reading.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying to unify $Knows(John, x)$ with $Brother(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the *Knows* facts in one bucket and all the *Brother* facts in another. The buckets can be stored in a hash table for efficient access.

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate $Employs(x, y)$. This would be a very large bucket with perhaps millions of employers



and tens of millions of employees. Answering a query such as $Employs(x, Richard)$ with predicate indexing would require scanning the entire bucket.

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as $Employs(IBM, y)$, we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact $Employs(IBM, Richard)$, the queries are

$Employs(IBM, Richard)$	Does IBM employ Richard?
$Employs(x, Richard)$	Who employs Richard?
$Employs(IBM, y)$	Whom does IBM employ?
$Employs(x, y)$	Who employs whom?

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the “highest” common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

The scheme we have described works very well whenever the lattice contains a small number of nodes. For a predicate with n arguments, however, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For most AI systems, the number of facts to be stored is small enough that efficient indexing is considered a solved problem. For commercial databases, where facts number in the billions, the problem has been the subject of intensive study and technology development..

9.3 FORWARD CHAINING

A forward-chaining algorithm for propositional definite clauses was given in Section 7.5. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses. Definite clauses such as *Situation* \Rightarrow *Response* are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

9.3.1 First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$\begin{aligned} &King(x) \wedge Greedy(x) \Rightarrow Evil(x) . \\ &King(John) . \\ &Greedy(y) . \end{aligned}$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem:

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

“... it is a crime for an American to sell weapons to hostile nations”:

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$$

“Nono ... has some missiles.” The sentence $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1 :

$$\text{Owns}(\text{Nono}, M_1) \quad (9.4)$$

$$\text{Missile}(M_1) \quad (9.5)$$

“All of its missiles were sold to it by Colonel West”:

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono}) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) \quad (9.7)$$

and we must know that an enemy of America counts as “hostile”:

$$\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x) . \quad (9.8)$$

“West, who is American . . .”:

$$\text{American}(\text{West}) . \quad (9.9)$$

“The country Nono, an enemy of America . . .”:

$$\text{Enemy}(\text{Nono}, \text{America}) . \quad (9.10)$$

DATALOG

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. We will see that the absence of function symbols makes inference much easier.

9.3.2 A simple forward-chaining algorithm

RENAMING

The first forward-chaining algorithm we consider is a simple one, shown in Figure 9.3. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not “new” if it is just a **renaming** of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, $\text{Likes}(x, \text{IceCream})$ and $\text{Likes}(y, \text{IceCream})$ are renamings of each other because they differ only in the choice of x or y ; their meanings are identical: everyone likes ice cream.

We use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
 Rule (9.6) is satisfied with $\{x/M_1\}$, and $\text{Sells}(\text{West}, M_1, \text{Nono})$ is added.
 Rule (9.7) is satisfied with $\{x/M_1\}$, and $\text{Weapon}(M_1)$ is added.
 Rule (9.8) is satisfied with $\{x/\text{Nono}\}$, and $\text{Hostile}(\text{Nono})$ is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/\text{West}, y/M_1, z/\text{Nono}\}$, and $\text{Criminal}(\text{West})$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining (page 258); the principal difference is that a first-order fixed point can include universally quantified atomic sentences.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of

```

function FOL-FC-ASK( $KB, \alpha$ ) returns a substitution or false
  inputs:  $KB$ , the knowledge base, a set of first-order definite clauses
            $\alpha$ , the query, an atomic sentence
  local variables:  $new$ , the new sentences inferred on each iteration

  repeat until  $new$  is empty
     $new \leftarrow \{ \}$ 
    for each  $rule$  in  $KB$  do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in  $KB$ 
         $q' \leftarrow \text{SUBST}(\theta, q)$ 
        if  $q'$  does not unify with some sentence already in  $KB$  or  $new$  then
          add  $q'$  to  $new$ 
           $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
          if  $\phi$  is not fail then return  $\phi$ 
    add  $new$  to  $KB$ 
  return false

```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB . The function `STANDARDIZE-VARIABLES` replaces all variables in its arguments with new ones that have not been used before.

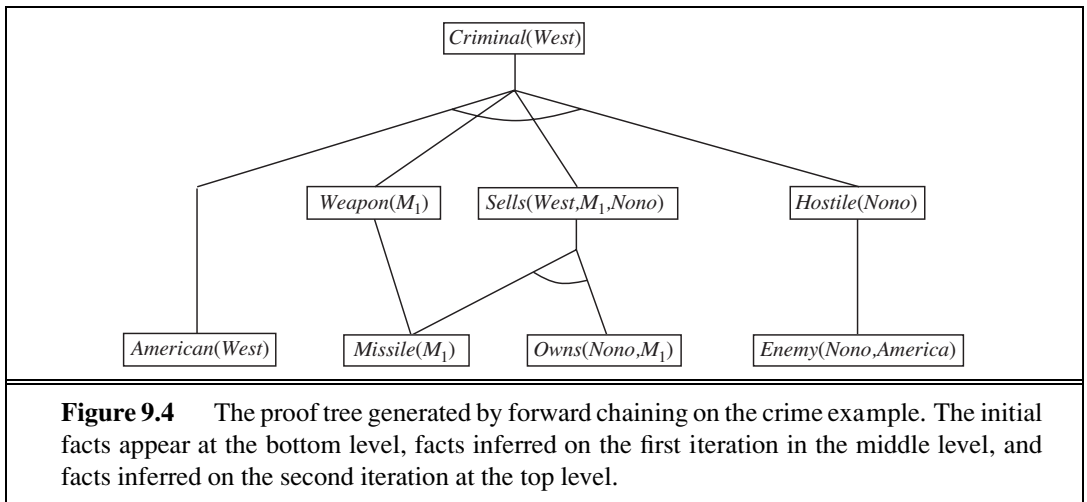


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum **arity** (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward

chaining. (See page 258.) The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

$$\begin{aligned} & \text{NatNum}(0) \\ & \forall n \text{ NatNum}(n) \Rightarrow \text{NatNum}(S(n)) , \end{aligned}$$

then forward chaining adds $\text{NatNum}(S(0))$, $\text{NatNum}(S(S(0)))$, $\text{NatNum}(S(S(S(0))))$, and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

9.3.3 Efficient forward chaining

The forward-chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of inefficiency. First, the “inner loop” of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal. We address each of these issues in turn.

PATTERN MATCHING

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

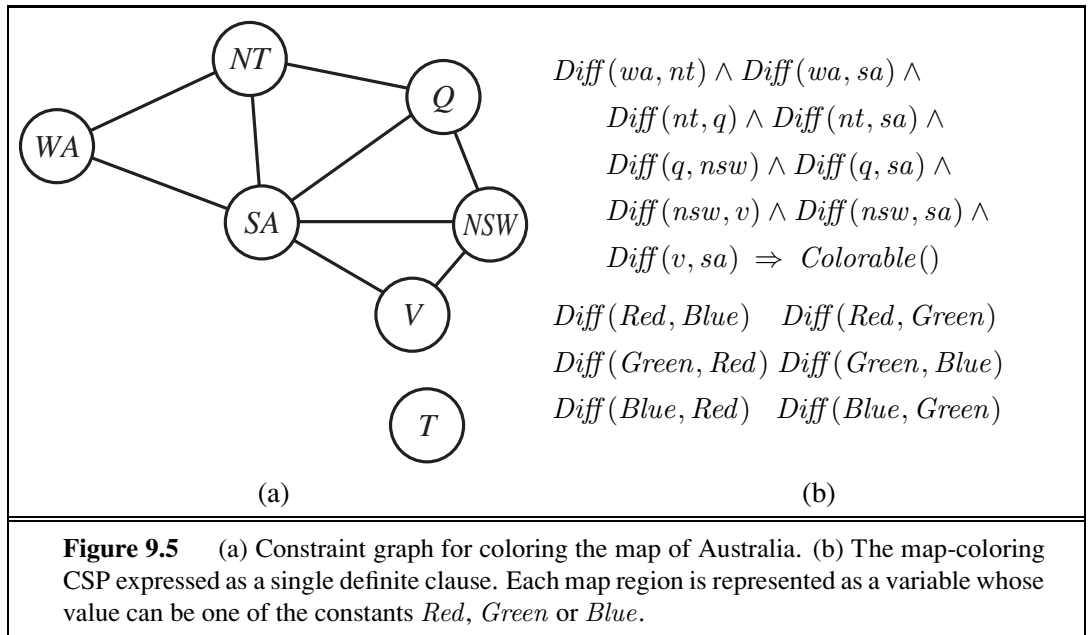
$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) .$$

Then we need to find all the facts that unify with $\text{Missile}(x)$; in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono}) .$$

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check whether it is a missile. If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is NP-hard, but good heuristics are available. For example, the **minimum-remaining-values** (MRV) heuristic used for CSPs in Chapter 6 would suggest ordering the conjuncts to look for missiles first if fewer missiles than objects are owned by Nono.

CONJUNCT ORDERING



The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, *Missile*(*x*) is a unary constraint on *x*. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts*. Consider the map-coloring problem from Figure 6.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion *Colorable*() can be inferred only if the CSP has a solution. Because CSPs in general include 3-SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard*.

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the knowledge base. It is easy to show that the data complexity of forward chaining is polynomial.
- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 6 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South

Australia from the map in Figure 9.5, the resulting clause is

$$\text{Diff}(wa, nt) \wedge \text{Diff}(nt, q) \wedge \text{Diff}(q, nsw) \wedge \text{Diff}(nsw, v) \Rightarrow \text{Colorable}()$$

which corresponds to the reduced CSP shown in Figure 6.12 on page 224. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.

- We can try to eliminate redundant rule-matching attempts in the forward-chaining algorithm, as described next.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x)$$

matches against $\text{Missile}(M_1)$ (again), and of course the conclusion $\text{Weapon}(M_1)$ is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t - 1$.* This is true because any inference that does not require a new fact from iteration $t - 1$ could have been done at iteration $t - 1$ already.

This observation leads naturally to an incremental forward-chaining algorithm where, at iteration t , we check a rule only if its premise includes a conjunct p_i that unifies with a fact p'_i newly inferred at iteration $t - 1$. The rule-matching step then fixes p_i to match with p'_i , but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and indeed many real systems operate in an “update” mode wherein forward chaining occurs in response to each new fact that is TElled to the system. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in repeatedly constructing partial matches that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

and the fact $\text{American}(\text{West})$. This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

The **rete** algorithm³ was the first to address this problem. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each

³ Rete is Latin for net. The English pronunciation rhymes with treaty.



node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, $Sells(x, y, z) \wedge Hostile(z)$ in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n -ary literal such as $Sells(x, y, z)$ might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

PRODUCTION
SYSTEM

Rete networks, and various improvements thereon, have been a key component of so-called **production systems**, which were among the earliest forward-chaining systems in widespread use.⁴ The XCON system (originally called R1; McDermott, 1982) was built with a production-system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built with the same underlying technology, which has been implemented in the general-purpose language OPS-5.

COGNITIVE
ARCHITECTURES

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the “working memory” of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with tens of millions of rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. Forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal at hand*. In our crime example, there were no rules capable of drawing irrelevant conclusions, so the lack of directedness was not a problem. In other cases (e.g., if many rules describe the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

DEDUCTIVE
DATABASES

MAGIC SET

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILS? (page 258). A third approach has emerged in the field of **deductive databases**, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is $Criminal(West)$, the rule that concludes $Criminal(x)$ will be rewritten to include an extra conjunct that constrains the value of x :

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x).$$

⁴ The word **production** in **production systems** denotes a condition–action rule.

The fact *Magic(West)* is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of “generic” backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.

9.4 BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach introduced in Section 7.5 for definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in **logic programming**, which is the most widely used form of automated reasoning. We also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we look at the close connection between logic programming and constraint satisfaction problems.

9.4.1 A backward-chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. $\text{FOL-BC-ASK}(KB, \text{goal})$ will be proved if the knowledge base contains a clause of the form $\text{lhs} \Rightarrow \text{goal}$, where *lhs* (left-hand side) is a list of conjuncts. An atomic fact like *American(West)* is considered as a clause whose *lhs* is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query *Person(x)* could be proved with the substitution $\{x/\text{John}\}$ as well as with $\{x/\text{Richard}\}$. So we implement FOL-BC-ASK as a **generator**—a function that returns multiple times, each time giving one possible result.

GENERATOR

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the *rhs* of the clause does indeed unify with the goal, proving every conjunct in the *lhs*, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving *Criminal(West)* from sentences (9.3) through (9.10).

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.


```

function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })



---


generator FOL-BC-OR(KB, goal,  $\theta$ ) yields a substitution
  for each rule (lhs  $\Rightarrow$  rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs)  $\leftarrow$  STANDARDIZE-VARIABLES((lhs, rhs))
    for each  $\theta'$  in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal,  $\theta$ )) do
      yield  $\theta'$ 



---


generator FOL-BC-AND(KB, goals,  $\theta$ ) yields a substitution
  if  $\theta = \text{failure}$  then return
  else if LENGTH(goals) = 0 then yield  $\theta$ 
  else do
    first, rest  $\leftarrow$  FIRST(goals), REST(goals)
    for each  $\theta'$  in FOL-BC-OR(KB, SUBST( $\theta$ , first),  $\theta$ ) do
      for each  $\theta''$  in FOL-BC-AND(KB, rest,  $\theta'$ ) do
        yield  $\theta''$ 

```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

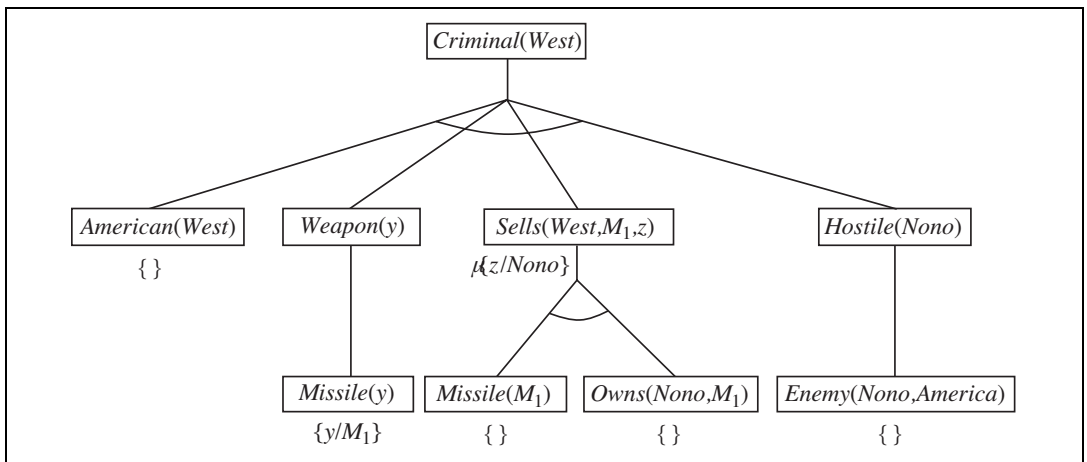


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*(*z*), *z* is already bound to *Nono*.

9.4.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$\textit{Algorithm} = \textit{Logic} + \textit{Control} .$$

PROLOG

Prolog is the most widely used logic programming language. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written “backwards” from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have $C :- A, B$. Here is a typical example:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

The notation $[E|L]$ denotes a list whose first element is E and whose rest is L . Here is a Prolog program for `append(X,Y,Z)`, which succeeds if list Z is the result of appending lists X and Y :

```
append([ ],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list Y produces the same list Y and (2) $[A|Z]$ is the result of appending $[A|X]$ onto Y , provided that Z is the result of appending X onto Y . In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments. For example, we can ask the query `append(X,Y,[1,2])`: what two lists can be appended to give $[1,2]$? We get back the solutions

```
X=[ ]      Y=[ 1,2 ];
X=[ 1 ]    Y=[ 2 ];
X=[ 1,2 ]  Y=[ ]
```

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.5).
- There is a set of built-in functions for arithmetic. Literals using these function symbols are “proved” by executing code rather than doing further inference. For example, the

goal “`X is 4+3`” succeeds with `X` bound to 7. On the other hand, the goal “`5 is X+Y`” fails, because the built-in functions do not do arbitrary equation solving.⁵

- There are built-in predicates that have side effects when executed. These include input-output predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The **occur check** is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

Prolog’s design represents a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed.

9.4.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 9.6, with the program as the knowledge base. We say “essentially” because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, our implementation had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure, a stack of **choice points**, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding. At any point in time, every variable in the program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the **trail**. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to *find* clauses that match the goal. A compiled Prolog

⁵ Note that if the Peano axioms are provided, such goals can be solved by inference within a Prolog program.

```

procedure APPEND(ax, y, az, continuation)

  trail ← GLOBAL-TRAIL-POINTER()
  if ax = [] and UNIFY(y, az) then CALL(continuation)
  RESET-TRAIL(trail)
  a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
  if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)

```

Figure 9.8 Pseudocode representing the result of compiling the **Append** predicate. The function **NEW-VARIABLE** returns a new variable, distinct from all other variables used so far. The procedure **CALL**(*continuation*) continues execution with the specified continuation.

program, on the other hand, is an inference procedure for a specific set of clauses, so it *knows* what clauses match the goal. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to **open-code** the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren *et al.* (1977).)

The instruction sets of today's computers give a poor match with Prolog's semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language. The most popular intermediate language is the Warren Abstract Machine, or WAM, named after David H. D. Warren, one of the implementers of the first Prolog compiler. The WAM is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language. Other compilers translate Prolog into a high-level language such as Lisp or C and then use that language's compiler to translate to machine language. For example, the definition of the **Append** predicate can be compiled into the code shown in Figure 9.8. Several points are worth mentioning:

- Rather than having to search the knowledge base for **Append** clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.
- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by **RESET-TRAIL** if the first clause fails. This will undo any bindings generated by the first call to **UNIFY**.
- The trickiest part is the use of **continuations** to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like **APPEND** when the goal succeeds, because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the **APPEND** code, if the first argument is empty and the second argument unifies with the third, then the **APPEND** predicate has succeeded. We then **CALL** the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to **APPEND** were at the top level, the continuation would print the bindings of the variables.

OPEN-CODE

CONTINUATION

Before Warren's work on the compilation of inference in Prolog, logic programming was too slow for general use. Compilers by Warren and others allowed Prolog code to achieve speeds that are competitive with C on a variety of standard benchmarks (Van Roy, 1990). Of course, the fact that one can write a planner or natural language parser in a few dozen lines of Prolog makes it somewhat more desirable than C for prototyping most small-scale AI research projects.

OR-PARALLELISM

AND-PARALLELISM

Parallelization can also provide substantial speedup. There are two principal sources of parallelism. The first, called **OR-parallelism**, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called **AND-parallelism**, comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution.

9.4.4 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 9.9(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 9.10(a). On the other hand, if we put the two clauses in the order

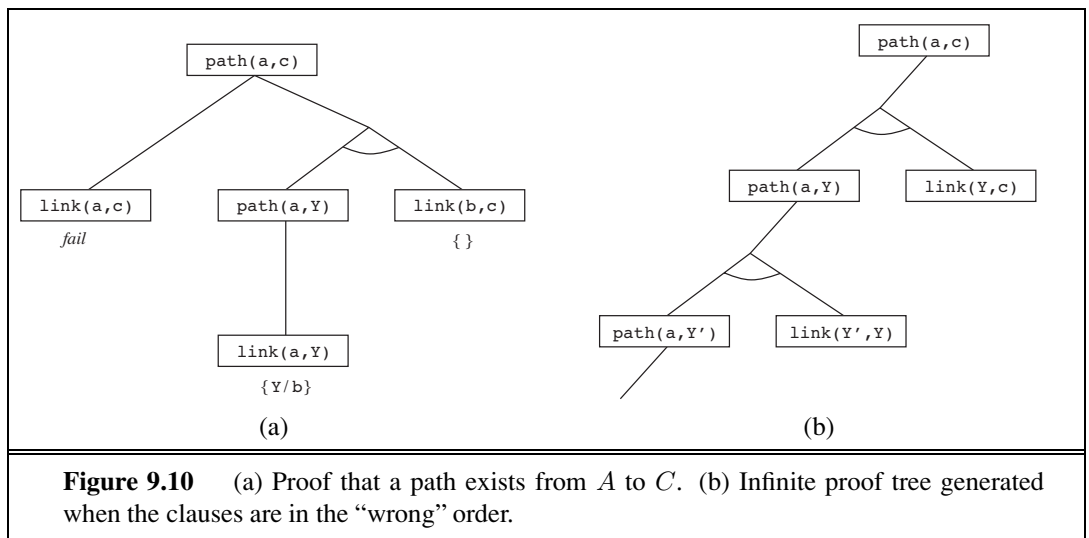
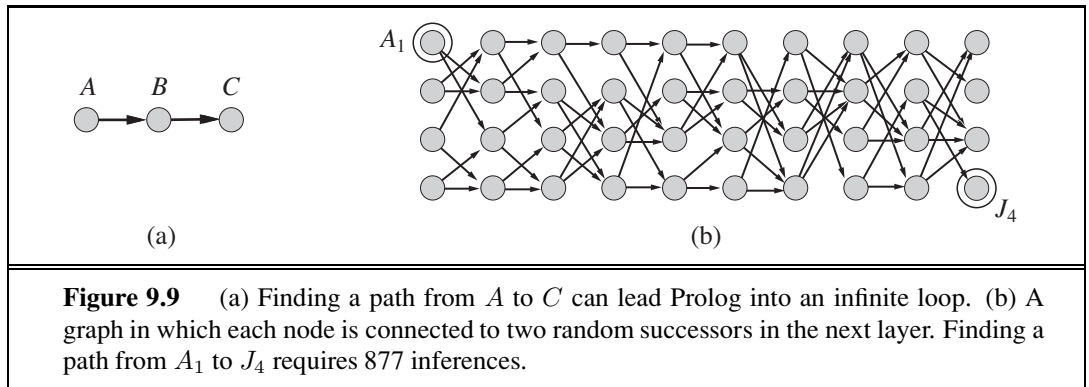
```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

then Prolog follows the infinite path shown in Figure 9.10(b). Prolog is therefore **incomplete** as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed. Notice that forward chaining does not suffer from this problem: once `path(a,b)`, `path(b,c)`, and `path(a,c)` are inferred, forward chaining halts.

Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A_1 to J_4 in Figure 9.9(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. This is similar to the repeated-state problem discussed in Chapter 3. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 9.9(b), only 62 inferences are needed.

DYNAMIC
PROGRAMMING

Forward chaining on graph search problems is an example of **dynamic programming**, in which the solutions to subproblems are constructed incrementally from those of smaller



subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using **memoization**—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by **tabled logic programming** systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic-programming efficiency of forward chaining. It is also complete for Datalog knowledge bases, which means that the programmer need worry less about infinite loops. (It is still possible to get an infinite loop with predicates like `father(X, Y)` that refer to a potentially unbounded number of objects.)

9.4.5 Database semantics of Prolog

Prolog uses database semantics, as discussed in Section 8.2.8. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed

by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following Prolog assertions about some course offerings:

$$\text{Course}(CS, 101), \text{Course}(CS, 102), \text{Course}(CS, 106), \text{Course}(EE, 101). \quad (9.11)$$

Under the unique names assumption, *CS* and *EE* are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed-world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in Prolog, then all we could say is that there are somewhere between one and infinity courses. That's because the assertions (in FOL) do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other. If we wanted to translate Equation (9.11) into FOL, we would get this:

$$\begin{aligned} \text{Course}(d, n) \quad \Leftrightarrow \quad & (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \\ & \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101). \end{aligned} \quad (9.12)$$

COMPLETION

This is called the **completion** of Equation (9.11). It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

$$\begin{aligned} x = y \quad \Leftrightarrow \quad & (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \\ & \vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106). \end{aligned}$$

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

9.4.6 Constraint logic programming

In our discussion of forward chaining (Section 9.3), we showed how constraint satisfaction problems (CSPs) can be encoded as definite clauses. Standard Prolog solves such problems in exactly the same way as the backtracking algorithm given in Figure 6.5.

Because backtracking enumerates the domains of the variables, it works only for **finite-domain** CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. (For example, the goal `diff(Q, SA)`, which says that Queensland and South Australia must be different colors, has six solutions if three colors are allowed.) Infinite-domain CSPs—for example, with integer or real-valued variables—require quite different algorithms, such as bounds propagation or linear programming.

Consider the following example. We define `triangle(X, Y, Z)` as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

```
triangle(X, Y, Z) :-
    X > 0, Y > 0, Z > 0, X + Y >= Z, Y + Z >= X, X + Z >= Y.
```

If we ask Prolog the query `triangle(3, 4, 5)`, it succeeds. On the other hand, if we ask `triangle(3, 4, Z)`, no solution will be found, because the subgoal `Z >= 0` cannot be handled by Prolog; we can't compare an unbound value to 0.

Constraint logic programming (CLP) allows variables to be *constrained* rather than *bound*. A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the `triangle(3, 4, Z)` query is the constraint $7 \geq Z \geq 1$. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

CLP systems incorporate various constraint-solving algorithms for the constraints allowed in the language. For example, a system that allows linear inequalities on real-valued variables might include a linear programming algorithm for solving those constraints. CLP systems also adopt a much more flexible approach to solving standard logic programming queries. For example, instead of depth-first, left-to-right backtracking, they might use any of the more efficient algorithms discussed in Chapter 6, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

Several systems that allow the programmer more control over the search order for inference have been defined. The MRS language (Genesereth and Smith, 1981; Russell, 1985) allows the programmer to write **metarules** to determine which conjuncts are tried first. The user could write a rule saying that the goal with the fewest variables should be tried first or could write domain-specific rules for particular predicates.

9.5 RESOLUTION

The last of our three families of logical systems is based on **resolution**. We saw on page 250 that propositional resolution using refutation is a complete inference procedure for propositional logic. In this section, we describe how to extend resolution to first-order logic.

9.5.1 Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.⁶ Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

becomes, in CNF,

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x).$$



Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

⁶ A clause can also be represented as an implication with a conjunction of atoms in the premise and a disjunction of atoms in the conclusion (Exercise 7.13). This is called **implicative normal form** or **Kowalski form** (especially when written with a right-to-left implication symbol (Kowalski, 1979)) and is often much easier to read.

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 253. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence “Everyone who loves all animals is loved by someone,” or

$$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)] .$$

The steps are as follows:

- **Eliminate implications:**

$$\forall x [\neg \forall y \neg \text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] .$$

- **Move \neg inwards:** In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\begin{array}{ll} \neg \forall x p & \text{becomes} \quad \exists x \neg p \\ \neg \exists x p & \text{becomes} \quad \forall x \neg p . \end{array}$$

Our sentence goes through the following transformations:

$$\begin{aligned} & \forall x [\exists y \neg(\neg \text{Animal}(y) \vee \text{Loves}(x, y))] \vee [\exists y \text{ Loves}(y, x)] . \\ & \forall x [\exists y \neg \neg \text{Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] . \\ & \forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] . \end{aligned}$$

Notice how a universal quantifier ($\forall y$) in the premise of the implication has become an existential quantifier. The sentence now reads “Either there is some animal that x doesn’t love, or (if this is not the case) someone loves x .” Clearly, the meaning of the original sentence has been preserved.

- **Standardize variables:** For sentences like $(\exists x P(x)) \vee (\exists x Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists z \text{ Loves}(z, x)] .$$

- **Skolemize: Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x P(x)$ into $P(A)$, where A is a new constant. However, we can’t apply Existential Instantiation to our sentence above because it doesn’t match the pattern $\exists v \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

$$\forall x [\text{Animal}(A) \wedge \neg \text{Loves}(x, A)] \vee \text{Loves}(B, x) ,$$

which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B . In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x and z :

$$\forall x [\text{Animal}(F(x)) \wedge \neg \text{Loves}(x, F(x))] \vee \text{Loves}(G(z), x) .$$

Here F and G are **Skolem functions**. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

SKOLEMIZATION

SKOLEM FUNCTION

- **Drop universal quantifiers:** At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(z), x) .$$

- **Distribute \vee over \wedge :**

$$[Animal(F(x)) \vee Loves(G(z), x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(z), x)] .$$

This step may also require flattening out nested conjunctions and disjunctions.

The sentence is now in CNF and consists of two clauses. It is quite unreadable. (It may help to explain that the Skolem function $F(x)$ refers to the animal potentially unloved by x , whereas $G(z)$ refers to someone who might love x .) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

9.5.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 253. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. Thus, we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

BINARY RESOLUTION

This rule is called the **binary resolution** rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are *identical*; first-order factoring reduces two literals to one if they are *unifiable*. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

9.5.3 Example proofs

Resolution proves that $KB \models \alpha$ by proving $KB \wedge \neg\alpha$ unsatisfiable, that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case, described in

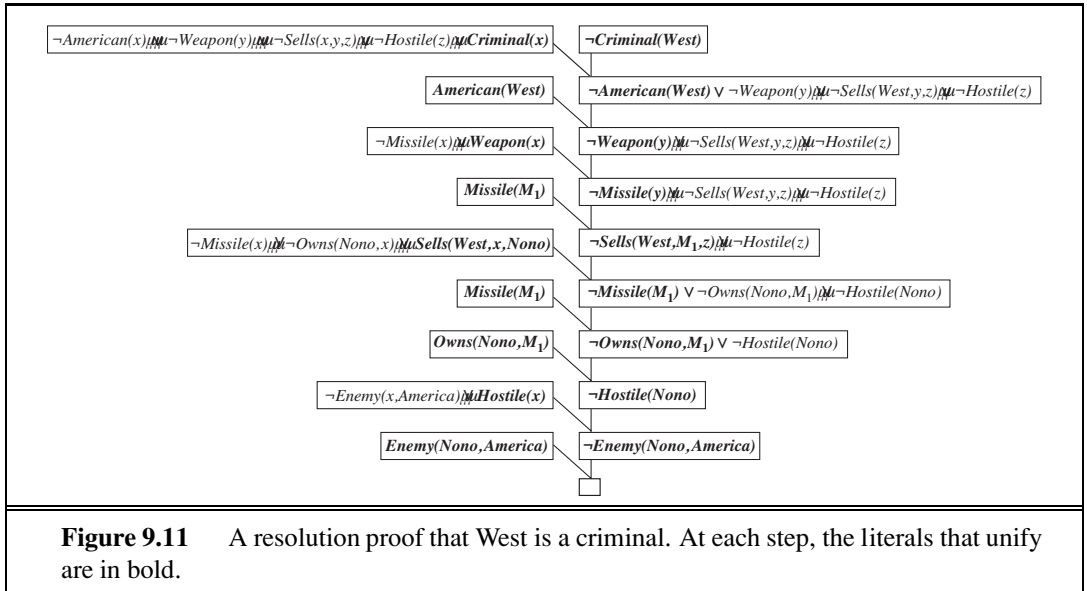


Figure 7.12, so we need not repeat it here. Instead, we give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

$$\begin{aligned}
 &\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x) \\
 &\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) \\
 &\neg Enemy(x, America) \vee Hostile(x) \\
 &\neg Missile(x) \vee Weapon(x) \\
 &Owns(Nono, M_1) \\
 &American(West) \\
 &Missile(M_1) \\
 &Enemy(Nono, America) .
 \end{aligned}$$

We also include the negated goal $\neg Criminal(West)$. The resolution proof is shown in Figure 9.11. Notice the structure: single “spine” beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond *exactly* to the consecutive values of the *goals* variable in the backward-chaining algorithm of Figure 9.6. This is because we always choose to resolve with a clause whose positive literal unified with the leftmost literal of the “current” clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

Everyone who loves all animals is loved by someone.
 Anyone who kills an animal is loved by no one.
 Jack loves all animals.
 Either Jack or Curiosity killed the cat, who is named Tuna.
 Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

- A. $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)]$
- B. $\forall x [\exists z \text{ Animal}(z) \wedge \text{Kills}(x, z)] \Rightarrow [\forall y \neg \text{Loves}(y, x)]$
- C. $\forall x \text{ Animal}(x) \Rightarrow \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\forall x \text{ Cat}(x) \Rightarrow \text{Animal}(x)$
- \neg G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

Now we apply the conversion procedure to convert each sentence to CNF:

- A1. $\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)$
- A2. $\neg \text{Loves}(x, F(x)) \vee \text{Loves}(G(x), x)$
- B. $\neg \text{Loves}(y, x) \vee \neg \text{Animal}(z) \vee \neg \text{Kills}(x, z)$
- C. $\neg \text{Animal}(x) \vee \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\neg \text{Cat}(x) \vee \text{Animal}(x)$
- \neg G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

The resolution proof that Curiosity killed the cat is given in Figure 9.12. In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

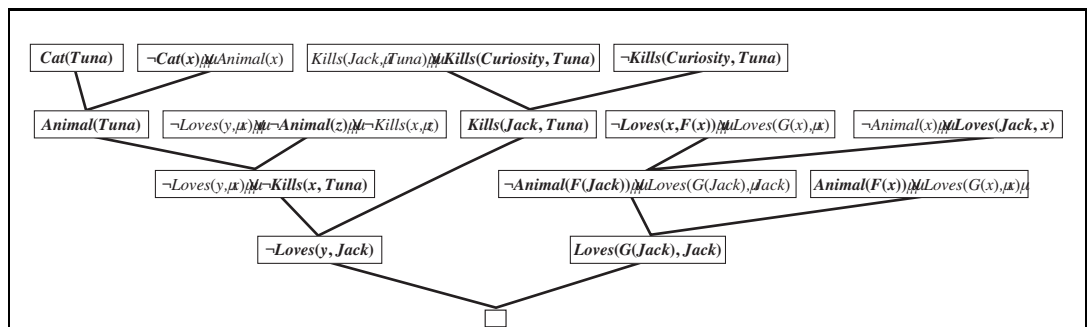


Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $\text{Loves}(G(\text{Jack}), \text{Jack})$. Notice also in the upper right, the unification of $\text{Loves}(x, F(x))$ and $\text{Loves}(\text{Jack}, x)$ can only succeed after the variables have been standardized apart.

The proof answers the question “Did Curiosity kill the cat?” but often we want to pose more general questions, such as “Who killed the cat?” Resolution can do this, but it takes a little more work to obtain the answer. The goal is $\exists w \text{ Kills}(w, \text{Tuna})$, which, when negated, becomes $\neg \text{Kills}(w, \text{Tuna})$ in CNF. Repeating the proof in Figure 9.12 with the new negated goal, we obtain a similar proof tree, but with the substitution $\{w/\text{Curiosity}\}$ in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

NONCONSTRUCTIVE
PROOF

Unfortunately, resolution can produce **nonconstructive proofs** for existential goals. For example, $\neg \text{Kills}(w, \text{Tuna})$ resolves with $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$ to give $\text{Kills}(\text{Jack}, \text{Tuna})$, which resolves again with $\neg \text{Kills}(w, \text{Tuna})$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna—either Jack or Curiosity. This is no great surprise! One solution is to restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to be able to backtrack over the possible bindings. Another solution is to add a special **answer literal** to the negated goal, which becomes $\neg \text{Kills}(w, \text{Tuna}) \vee \text{Answer}(w)$. Now, the resolution process generates an answer whenever a clause is generated containing just a *single* answer literal. For the proof in Figure 9.12, this is $\text{Answer}(\text{Curiosity})$. The nonconstructive proof would generate the clause $\text{Answer}(\text{Curiosity}) \vee \text{Answer}(\text{Jack})$, which does not constitute an answer.

ANSWER LITERAL

9.5.4 Completeness of resolution

This section gives a completeness proof of resolution. It can be safely skipped by those who are willing to take it on faith.

REFUTATION
COMPLETENESS

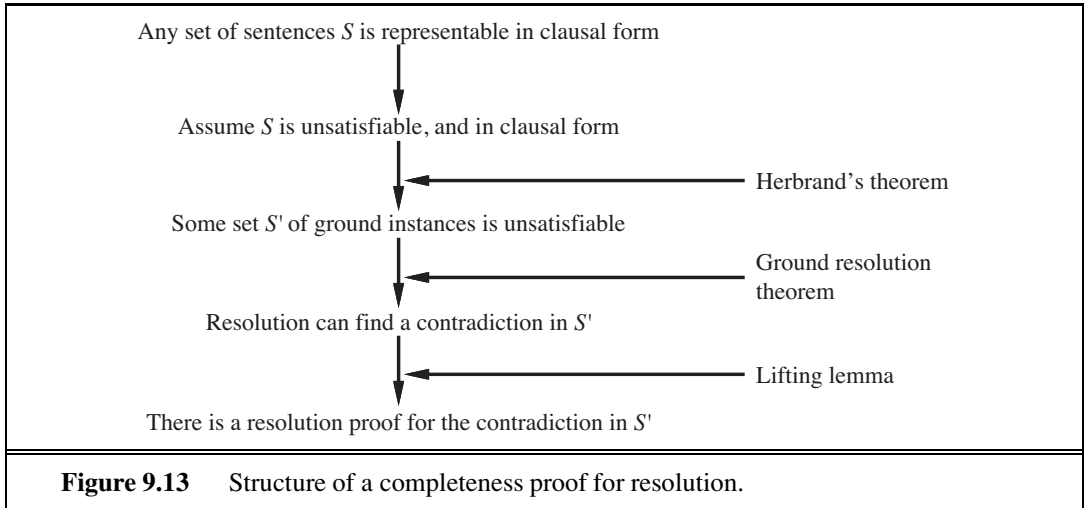
We show that resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \wedge \neg Q(x)$ is unsatisfiable.

We take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: *if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.*



Our proof sketch follows Robinson’s original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 9.13) is as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable (Herbrand’s theorem).
2. We then appeal to the **ground resolution theorem** given in Chapter 7, which states that propositional resolution is complete for ground sentences.
3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.



To carry out the first step, we need three new concepts:

HERBRAND
UNIVERSE

- **Herbrand universe:** If S is a set of clauses, then H_S , the Herbrand universe of S , is the set of all ground terms constructable from the following:

- The function symbols in S , if any.
- The constant symbols in S , if any; if none, then the constant symbol A .

For example, if S contains just the clause $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}.$$

SATURATION

- **Saturation:** If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P , is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S .

HERBRAND BASE

- **Herbrand base:** The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S , written as $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

$$\begin{aligned} &\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ &\quad \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ &\quad \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ &\quad \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots\} \end{aligned}$$

HERBRAND'S
THEOREM

These definitions allow us to state a form of **Herbrand's theorem** (Herbrand, 1930):

If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 255) to show that the **resolution closure** $RC(S')$ contains the empty clause. That is, running propositional resolution to completion on S' will derive a contradiction.

Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S , the next step is to show that there is a resolution

GÖDEL'S INCOMPLETENESS THEOREM

By slightly extending the language of first-order logic to allow for the **mathematical induction schema** in arithmetic, Kurt Gödel was able to show, in his **incompleteness theorem**, that there are true arithmetic sentences that cannot be proved.

The proof of the incompleteness theorem is somewhat beyond the scope of this book, occupying, as it does, at least 30 pages, but we can give a hint here. We begin with the logical theory of numbers. In this theory, there is a single constant, 0, and a single function, S (the successor function). In the intended model, $S(0)$ denotes 1, $S(S(0))$ denotes 2, and so on; the language therefore has names for all the natural numbers. The vocabulary also includes the function symbols $+$, \times , and $Expt$ (exponentiation) and the usual set of logical connectives and quantifiers. The first step is to notice that the set of sentences that we can write in this language can be enumerated. (Imagine defining an alphabetical order on the symbols and then arranging, in alphabetical order, each of the sets of sentences of length 1, 2, and so on.) We can then number each sentence α with a unique natural number $\# \alpha$ (the **Gödel number**). This is crucial: number theory contains a name for each of its own sentences. Similarly, we can number each possible proof P with a Gödel number $G(P)$, because a proof is simply a finite sequence of sentences.

Now suppose we have a recursively enumerable set A of sentences that are true statements about the natural numbers. Recalling that A can be named by a given set of integers, we can imagine writing in our language a sentence $\alpha(j, A)$ of the following sort:

$\forall i$ i is not the Gödel number of a proof of the sentence whose Gödel number is j , where the proof uses only premises in A .

Then let σ be the sentence $\alpha(\# \sigma, A)$, that is, a sentence that states its own unprovability from A . (That this sentence always exists is true but not entirely obvious.)

Now we make the following ingenious argument: Suppose that σ is provable from A ; then σ is false (because σ says it cannot be proved). But then we have a false sentence that is provable from A , so A cannot consist of only true sentences—a violation of our premise. Therefore, σ is *not* provable from A . But this is exactly what σ itself claims; hence σ is a true sentence.

So, we have shown (barring $29\frac{1}{2}$ pages) that for any set of true sentences of number theory, and in particular any set of basic axioms, there are other true sentences that *cannot* be proved from those axioms. This establishes, among other things, that we can never prove all the theorems of mathematics *within any given system of axioms*. Clearly, this was an important discovery for mathematics. Its significance for AI has been widely debated, beginning with speculations by Gödel himself. We take up the debate in Chapter 26.

proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson stated this lemma:

Let C_1 and C_2 be two clauses with no shared variables, and let C'_1 and C'_2 be ground instances of C_1 and C_2 . If C' is a resolvent of C'_1 and C'_2 , then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C .

LIFTING LEMMA

This is called a **lifting lemma**, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned} C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\ C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\ C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\ C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) . \end{aligned}$$

We see that indeed C' is a ground instance of C . In general, for C'_1 and C'_2 to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2 . From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C' .

From this fact, it follows that if the empty clause appears in the resolution closure of S' , it must also appear in the resolution closure of S . This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule.

The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

9.5.5 Equality

None of the inference methods described so far in this chapter handle an assertion of the form $x = y$. Three distinct approaches can be taken. The first approach is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one

for each predicate and function:

$$\begin{aligned}
 &\forall x \ x = x \\
 &\forall x, y \ x = y \Rightarrow y = x \\
 &\forall x, y, z \ x = y \wedge y = z \Rightarrow x = z \\
 &\forall x, y \ x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\
 &\forall x, y \ x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\
 &\vdots \\
 &\forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z)) \\
 &\forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z)) \\
 &\vdots
 \end{aligned}$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms. The simplest rule, **demodulation**, takes a unit clause $x = y$ and some clause α that contains the term x , and yields a new clause formed by substituting y for x within α . It works if the term within α unifies with x ; it need not be exactly equal to x . Note that demodulation is directional; given $x = y$, the x always gets replaced with y , never vice versa. That means that demodulation can be used for simplifying expressions using demodulators such as $x + 0 = x$ or $x^1 = x$. As another example, given

$$\begin{aligned}
 &Father(Father(x)) = PaternalGrandfather(x) \\
 &Birthdate(Father(Father(Bella)), 1926)
 \end{aligned}$$

we can conclude by demodulation

$$Birthdate(PaternalGrandfather(Bella), 1926) .$$

More formally, we have

DEMODULATION

- **Demodulation:** For any terms x , y , and z , where z appears somewhere in literal m_i and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{x = y, \quad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)} .$$

where SUBST is the usual substitution of a binding list, and $\text{SUB}(x, y, m)$ means to replace x with y everywhere that x occurs within m .

The rule can also be extended to handle non-unit clauses in which an equality literal appears:

PARAMODULATION

- **Paramodulation:** For any terms x , y , and z , where z appears somewhere in literal m_i , and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \quad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n))} .$$

For example, from

$$P(F(x, B), x) \vee Q(x) \quad \text{and} \quad F(A, y) = y \vee R(y)$$

we have $\theta = \text{UNIFY}(F(A, y), F(x, B)) = \{x/A, y/B\}$, and we can conclude by paramodulation the sentence

$$P(B, A) \vee Q(A) \vee R(B) .$$

Paramodulation yields a complete inference procedure for first-order logic with equality.

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are *provably* equal under some substitution, where “provably” allows for equality reasoning. For example, the terms $1 + 2$ and $2 + 1$ normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution. **Equational unification** of this kind can be done with efficient algorithms designed for the particular axioms used (commutativity, associativity, and so on) rather than through explicit inference with those axioms. Theorem provers using this technique are closely related to the CLP systems described in Section 9.4.

EQUATIONAL
UNIFICATION

9.5.6 Resolution strategies

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs *efficiently*.

UNIT PREFERENCE

Unit preference: This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a **unit clause**). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. **Unit resolution** is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

The OTTER theorem prover (Organized Techniques for Theorem-proving and Effective Research, McCune, 1992), uses a form of best-first search. Its heuristic function measures the “weight” of each clause, where lighter clauses are preferred. The exact choice of heuristic is up to the user, but generally, the weight of a clause should be correlated with its size or difficulty. Unit clauses are treated as light; the search can thus be seen as a generalization of the unit preference strategy.

SET OF SUPPORT

Set of support: Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the *set of support*. The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

We have to be careful with this approach because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support S so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution is complete. For example, one can use the negated query as the set of support, on the assumption that the

original knowledge base is consistent. (After all, if it is not consistent, then the fact that the query follows from it is vacuous.) The set-of-support strategy has the additional advantage of generating goal-directed proof trees that are often easy for humans to understand.

INPUT RESOLUTION

Input resolution: In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.11 on page 348 uses only input resolutions and has the characteristic shape of a single “spine” with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.

LINEAR RESOLUTION

SUBSUMPTION

Subsumption: The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

Practical uses of resolution theorem provers

SYNTHESIS

VERIFICATION

Theorem provers can be applied to the problems involved in the **synthesis** and **verification** of both hardware and software. Thus, theorem-proving research is carried out in the fields of hardware design, programming languages, and software engineering—not just in AI.

In the case of hardware, the axioms describe the interactions between signals and circuit elements. (See Section 8.4.2 on page 309 for an example.) Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivasa and Bickford, 1990). The AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983).

In the case of software, reasoning about programs is quite similar to reasoning about actions, as in Chapter 7: axioms describe the preconditions and effects of each statement. The formal synthesis of algorithms was one of the first uses of theorem provers, as outlined by Cordell Green (1969a), who built on earlier ideas by Herbert Simon (1963). The idea is to constructively prove a theorem to the effect that “there exists a program p satisfying a certain specification.” Although fully automated **deductive synthesis**, as it is called, has not yet become feasible for general-purpose programming, hand-guided deductive synthesis has been successful in designing several novel and sophisticated algorithms. Synthesis of special-purpose programs, such as scientific computing code, is also an active area of research.

DEDUCTIVE
SYNTHESIS

Similar techniques are now being applied to software verification by systems such as the SPIN model checker (Holzmann, 1997). For example, the Remote Agent spacecraft control program was verified before and after flight (Havelund *et al.*, 2000). The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984).

9.6 SUMMARY

We have presented an analysis of logical inference in first-order logic and a number of algorithms for doing it.

- A first approach uses inference rules (**universal instantiation** and **existential instantiation**) to **propositionalize** the inference problem. Typically, this approach is slow, unless the domain is small.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process more efficient in many cases.
- A lifted version of **Modus Ponens** uses unification to provide a natural and powerful inference rule, **generalized Modus Ponens**. The **forward-chaining** and **backward-chaining** algorithms apply this rule to sets of definite clauses.
- Generalized Modus Ponens is complete for definite clauses, although the entailment problem is **semidecidable**. For **Datalog** knowledge bases consisting of function-free definite clauses, entailment is decidable.
- Forward chaining is used in **deductive databases**, where it can be combined with relational database operations. It is also used in **production systems**, which perform efficient updates with very large rule sets. Forward chaining is complete for Datalog and runs in polynomial time.
- Backward chaining is used in **logic programming systems**, which employ sophisticated compiler technology to provide very fast inference. Backward chaining suffers from redundant inferences and infinite loops; these can be alleviated by **memoization**.
- Prolog, unlike first-order logic, uses a closed world with the unique names assumption and negation as failure. These make Prolog a more practical programming language, but bring it further from pure logic.
- The generalized **resolution** inference rule provides a complete proof system for first-order logic, using knowledge bases in conjunctive normal form.
- Several strategies exist for reducing the search space of a resolution system without compromising completeness. One of the most important issues is dealing with equality; we showed how **demodulation** and **paramodulation** can be used.
- Efficient resolution-based theorem provers have been used to prove interesting mathematical theorems and to verify and synthesize software and hardware.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a collection of valid schemas plus a single inference rule, Modus Ponens. Whitehead and Russell (1910) expounded the so-called *rules of passage* (the actual term is from Herbrand (1930)) that are used to move quantifiers to the front of formulas. Skolem constants

and Skolem functions were introduced, appropriately enough, by Thoralf Skolem (1920). Oddly enough, it was Skolem who introduced the Herbrand universe (Skolem, 1928).

Herbrand's theorem (Herbrand, 1930) has played a vital role in the development of automated reasoning. Herbrand is also the inventor of unification. Gödel (1930) built on the ideas of Skolem and Herbrand to show that first-order logic has a complete proof procedure. Alan Turing (1936) and Alonzo Church (1936) simultaneously showed, using very different proofs, that validity in first-order logic was not decidable. The excellent text by Enderton (1972) explains all of these results in a rigorous yet understandable fashion.

Abraham Robinson proposed that an automated reasoner could be built using propositionalization and Herbrand's theorem, and Paul Gilmore (1960) wrote the first program. Davis and Putnam (1960) introduced the propositionalization method of Section 9.1. Prawitz (1960) developed the key idea of letting the quest for propositional inconsistency drive the search, and generating terms from the Herbrand universe only when they were necessary to establish propositional inconsistency. After further development by other researchers, this idea led J. A. Robinson (no relation) to develop resolution (Robinson, 1965).

In AI, resolution was adopted for question-answering systems by Cordell Green and Bertram Raphael (1968). Early AI implementations put a good deal of effort into data structures that would allow efficient retrieval of facts; this work is covered in AI programming texts (Charniak *et al.*, 1987; Norvig, 1992; Forbus and de Kleer, 1993). By the early 1970s, **forward chaining** was well established in AI as an easily understandable alternative to resolution. AI applications typically involved large numbers of rules, so it was important to develop efficient rule-matching technology, particularly for incremental updates. The technology for **production systems** was developed to support such applications. The production system language OPS-5 (Forgy, 1981; Brownston *et al.*, 1985), incorporating the efficient **rete** match process (Forgy, 1982), was used for applications such as the R1 expert system for minicomputer configuration (McDermott, 1982).

RETE

The SOAR cognitive architecture (Laird *et al.*, 1987; Laird, 2008) was designed to handle very large rule sets—up to a million rules (Doorenbos, 1994). Example applications of SOAR include controlling simulated fighter aircraft (Jones *et al.*, 1998), airspace management (Taylor *et al.*, 2007), AI characters for computer games (Wintermute *et al.*, 2007), and training tools for soldiers (Wray and Jones, 2005).

The field of **deductive databases** began with a workshop in Toulouse in 1977 that brought together experts in logical inference and database systems (Gallaire and Minker, 1978). Influential work by Chandra and Harel (1980) and Ullman (1985) led to the adoption of Datalog as a standard language for deductive databases. The development of the **magic sets** technique for rule rewriting by Bancilhon *et al.* (1986) allowed forward chaining to borrow the advantage of goal-directedness from backward chaining. Current work includes the idea of integrating multiple databases into a consistent dataspace (Halevy, 2007).

Backward chaining for logical inference appeared first in Hewitt's PLANNER language (1969). Meanwhile, in 1972, Alain Colmerauer had developed and implemented **Prolog** for the purpose of parsing natural language—Prolog's clauses were intended initially as context-free grammar rules (Roussel, 1975; Colmerauer *et al.*, 1973). Much of the theoretical background for logic programming was developed by Robert Kowalski, working

with Colmerauer; see Kowalski (1988) and Colmerauer and Roussel (1993) for a historical overview. Efficient Prolog compilers are generally based on the Warren Abstract Machine (WAM) model of computation developed by David H. D. Warren (1983). Van Roy (1990) showed that Prolog programs can be competitive with C programs in terms of speed.

Methods for avoiding unnecessary looping in recursive logic programs were developed independently by Smith *et al.* (1986) and Tamaki and Sato (1986). The latter paper also included memoization for logic programs, a method developed extensively as **tabled logic programming** by David S. Warren. Swift and Warren (1994) show how to extend the WAM to handle tabling, enabling Datalog programs to execute an order of magnitude faster than forward-chaining deductive database systems.

Early work on constraint logic programming was done by Jaffar and Lassez (1987). Jaffar *et al.* (1992) developed the CLP(R) system for handling real-valued constraints. There are now commercial products for solving large-scale configuration and optimization problems with constraint programming; one of the best known is ILOG (Junker, 2003). Answer set programming (Gelfond, 2008) extends Prolog, allowing disjunction and negation.

Texts on logic programming and Prolog, including Shoham (1994), Bratko (2001), Clocksin (2003), and Clocksin and Mellish (2003). Prior to 2000, the *Journal of Logic Programming* was the journal of record; it has now been replaced by *Theory and Practice of Logic Programming*. Logic programming conferences include the International Conference on Logic Programming (ICLP) and the International Logic Programming Symposium (ILPS).

Research into **mathematical theorem proving** began even before the first complete first-order systems were developed. Herbert Gelemter's Geometry Theorem Prover (Gelemter, 1959) used heuristic search methods combined with diagrams for pruning false subgoals and was able to prove some quite intricate results in Euclidean geometry. The demodulation and paramodulation rules for equality reasoning were introduced by Wos *et al.* (1967) and Wos and Robinson (1968), respectively. These rules were also developed independently in the context of term-rewriting systems (Knuth and Bendix, 1970). The incorporation of equality reasoning into the unification algorithm is due to Gordon Plotkin (1972). Jouannaud and Kirchner (1991) survey equational unification from a term-rewriting perspective. An overview of unification is given by Baader and Snyder (2001).

A number of control strategies have been proposed for resolution, beginning with the unit preference strategy (Wos *et al.*, 1964). The set-of-support strategy was proposed by Wos *et al.* (1965) to provide a degree of goal-directedness in resolution. Linear resolution first appeared in Loveland (1970). Genesereth and Nilsson (1987, Chapter 5) provide a short but thorough analysis of a wide variety of control strategies.

A *Computational Logic* (Boyer and Moore, 1979) is the basic reference on the Boyer-Moore theorem prover. Stickel (1992) covers the Prolog Technology Theorem Prover (PTTP), which combines the advantages of Prolog compilation with the completeness of model elimination. SETHEO (Letz *et al.*, 1992) is another widely used theorem prover based on this approach. LEANTAP (Beckert and Posegga, 1995) is an efficient theorem prover implemented in only 25 lines of Prolog. Weidenbach (2001) describes SPASS, one of the strongest current theorem provers. The most successful theorem prover in recent annual competitions has been VAMPIRE (Riazanov and Voronkov, 2002). The COQ system (Bertot *et al.*, 2004) and the E

equational solver (Schulz, 2004) have also proven to be valuable tools for proving correctness. Theorem provers have been used to automatically synthesize and verify software for controlling spacecraft (Denney *et al.*, 2006), including NASA's new Orion capsule (Lowry, 2008). The design of the FM9001 32-bit microprocessor was proved correct by the NQTHM system (Hunt and Brock, 1992). The Conference on Automated Deduction (CADE) runs an annual contest for automated theorem provers. From 2002 through 2008, the most successful system has been VAMPIRE (Riazanov and Voronkov, 2002). Wiedijk (2003) compares the strength of 15 mathematical provers. TPTP (Thousands of Problems for Theorem Provers) is a library of theorem-proving problems, useful for comparing the performance of systems (Sutcliffe and Suttner, 1998; Sutcliffe *et al.*, 2006).

Theorem provers have come up with novel mathematical results that eluded human mathematicians for decades, as detailed in the book *Automated Reasoning and the Discovery of Missing Elegant Proofs* (Wos and Pieper, 2003). The SAM (Semi-Automated Mathematics) program was the first, proving a lemma in lattice theory (Guard *et al.*, 1969). The AURA program has also answered open questions in several areas of mathematics (Wos and Winker, 1983). The Boyer–Moore theorem prover (Boyer and Moore, 1979) was used by Natarajan Shankar to give the first fully rigorous formal proof of Gödel's Incompleteness Theorem (Shankar, 1986). The NUPRL system proved Girard's paradox (Howe, 1987) and Higman's Lemma (Murthy and Russell, 1990). In 1933, Herbert Robbins proposed a simple set of axioms—the **Robbins algebra**—that appeared to define Boolean algebra, but no proof could be found (despite serious work by Alfred Tarski and others). On October 10, 1996, after eight days of computation, EQP (a version of OTTER) found a proof (McCune, 1997).

Many early papers in mathematical logic are to be found in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). Textbooks geared toward automated deduction include the classic *Symbolic Logic and Mechanical Theorem Proving* (Chang and Lee, 1973), as well as more recent works by Duffy (1991), Wos *et al.* (1992), Bibel (1993), and Kaufmann *et al.* (2000). The principal journal for theorem proving is the *Journal of Automated Reasoning*; the main conferences are the annual Conference on Automated Deduction (CADE) and the International Joint Conference on Automated Reasoning (IJCAR). The *Handbook of Automated Reasoning* (Robinson and Voronkov, 2001) collects papers in the field. MacKenzie's *Mechanizing Proof* (2004) covers the history and technology of theorem proving for the popular audience.

ROBBINS ALGEBRA

EXERCISES

9.1 Prove that Universal Instantiation is sound and that Existential Instantiation produces an inferentially equivalent knowledge base.

9.2 From $Likes(Jerry, IceCream)$ it seems reasonable to infer $\exists x \text{ Likes}(x, IceCream)$. Write down a general inference rule, **Existential Introduction**, that sanctions this inference. State carefully the conditions that must be satisfied by the variables and terms involved.

EXISTENTIAL
INTRODUCTION

9.3 Suppose a knowledge base contains just one sentence, $\exists x \text{ AsHighAs}(x, \text{Everest})$. Which of the following are legitimate results of applying Existential Instantiation?

- a. $\text{AsHighAs}(\text{Everest}, \text{Everest})$.
- b. $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest})$.
- c. $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest}) \wedge \text{AsHighAs}(\text{BenNevis}, \text{Everest})$
(after two applications).

9.4 For each pair of atomic sentences, give the most general unifier if it exists:

- a. $P(A, B, B), P(x, y, z)$.
- b. $Q(y, G(A, B)), Q(G(x, x), y)$.
- c. $\text{Older}(\text{Father}(y), y), \text{Older}(\text{Father}(x), \text{John})$.
- d. $\text{Knows}(\text{Father}(y), y), \text{Knows}(x, x)$.

9.5 Consider the subsumption lattices shown in Figure 9.2 (page 329).

- a. Construct the lattice for the sentence $\text{Employs}(\text{Mother}(\text{John}), \text{Father}(\text{Richard}))$.
- b. Construct the lattice for the sentence $\text{Employs}(\text{IBM}, y)$ (“Everyone works for IBM”). Remember to include every kind of query that unifies with the sentence.
- c. Assume that STORE indexes each sentence under every node in its subsumption lattice. Explain how FETCH should work when some of these sentences contain variables; use as examples the sentences in (a) and (b) and the query $\text{Employs}(x, \text{Father}(x))$.

9.6 Write down logical representations for the following sentences, suitable for use with Generalized Modus Ponens:

- a. Horses, cows, and pigs are mammals.
- b. An offspring of a horse is a horse.
- c. Bluebeard is a horse.
- d. Bluebeard is Charlie’s parent.
- e. Offspring and parent are inverse relations.
- f. Every mammal has a parent.

9.7 These questions concern issues with substitution and Skolemization.

- a. Given the premise $\forall x \exists y P(x, y)$, it is not valid to conclude that $\exists q P(q, q)$. Give an example of a predicate P where the first is true but the second is false.
- b. Suppose that an inference engine is incorrectly written with the occurs check omitted, so that it allows a literal like $P(x, F(x))$ to be unified with $P(q, q)$. (As mentioned, most standard implementations of Prolog actually do allow this.) Show that such an inference engine will allow the conclusion $\exists y P(q, q)$ to be inferred from the premise $\forall x \exists y P(x, y)$.

- c. Suppose that a procedure that converts first-order logic to clausal form incorrectly Skolemizes $\forall x \exists y P(x, y)$ to $P(x, Sk0)$ —that is, it replaces y by a Skolem constant rather than by a Skolem function of x . Show that an inference engine that uses such a procedure will likewise allow $\exists q P(q, q)$ to be inferred from the premise $\forall x \exists y P(x, y)$.
- d. A common error among students is to suppose that, in unification, one is allowed to substitute a term for a Skolem constant instead of for a variable. For instance, they will say that the formulas $P(Sk1)$ and $P(A)$ can be unified under the substitution $\{Sk1/A\}$. Give an example where this leads to an invalid inference.

9.8 Explain how to write any given 3-SAT problem of arbitrary size using a single first-order definite clause and no more than 30 ground facts.

9.9 Suppose you are given the following axioms:

1. $0 \leq 3$.
2. $7 \leq 9$.
3. $\forall x \quad x \leq x$.
4. $\forall x \quad x \leq x + 0$.
5. $\forall x \quad x + 0 \leq x$.
6. $\forall x, y \quad x + y \leq y + x$.
7. $\forall w, x, y, z \quad w \leq y \wedge x \leq z \Rightarrow w + x \leq y + z$.
8. $\forall x, y, z \quad x \leq y \wedge y \leq z \Rightarrow x \leq z$.

- a. Give a backward-chaining proof of the sentence $7 \leq 3 + 9$. (Be sure, of course, to use only the axioms given here, not anything else you may know about arithmetic.) Show only the steps that leads to success, not the irrelevant steps.
- b. Give a forward-chaining proof of the sentence $7 \leq 3 + 9$. Again, show only the steps that lead to success.

9.10 A popular children's riddle is "Brothers and sisters have I none, but that man's father is my father's son." Use the rules of the family domain (Section 8.3.2 on page 301) to show who that man is. You may apply any of the inference methods described in this chapter. Why do you think that this riddle is difficult?

9.11 Suppose we put into a logical knowledge base a segment of the U.S. census data listing the age, city of residence, date of birth, and mother of every person, using social security numbers as identifying constants for each person. Thus, George's age is given by $Age(443-65-1282, 56)$. Which of the following indexing schemes S1–S5 enable an efficient solution for which of the queries Q1–Q4 (assuming normal backward chaining)?

- **S1:** an index for each atom in each position.
- **S2:** an index for each first argument.
- **S3:** an index for each predicate atom.
- **S4:** an index for each *combination* of predicate and first argument.

- **S5**: an index for each *combination* of predicate and second argument and an index for each first argument.
- **Q1**: $\text{Age}(443\text{-}44\text{-}4321, x)$
- **Q2**: $\text{ResidesIn}(x, \text{Houston})$
- **Q3**: $\text{Mother}(x, y)$
- **Q4**: $\text{Age}(x, 34) \wedge \text{ResidesIn}(x, \text{TinyTownUSA})$

9.12 One might suppose that we can avoid the problem of variable conflict in unification during backward chaining by standardizing apart all of the sentences in the knowledge base once and for all. Show that, for some sentences, this approach cannot work. (*Hint*: Consider a sentence in which one part unifies with another.)

9.13 In this exercise, use the sentences you wrote in Exercise 9.6 to answer a question by using a backward-chaining algorithm.

- a. Draw the proof tree generated by an exhaustive backward-chaining algorithm for the query $\exists h \text{ Horse}(h)$, where clauses are matched in the order given.
- b. What do you notice about this domain?
- c. How many solutions for h actually follow from your sentences?
- d. Can you think of a way to find all of them? (*Hint*: See Smith *et al.* (1986).)

9.14 Trace the execution of the backward-chaining algorithm in Figure 9.6 (page 338) when it is applied to solve the crime problem (page 330). Show the sequence of values taken on by the *goals* variable, and arrange them into a tree.

9.15 The following Prolog code defines a predicate *P*. (Remember that uppercase terms are variables, not constants, in Prolog.)

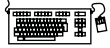
```
P(X, [X|Y]) .
P(X, [Y|Z]) :- P(X, Z) .
```

- a. Show proof trees and solutions for the queries $P(A, [2, 1, 3])$ and $P(2, [1, A, 3])$.
- b. What standard list operation does *P* represent?



9.16 This exercise looks at sorting in Prolog.

- a. Write Prolog clauses that define the predicate `sorted(L)`, which is true if and only if list *L* is sorted in ascending order.
- b. Write a Prolog definition for the predicate `perm(L, M)`, which is true if and only if *L* is a permutation of *M*.
- c. Define `sort(L, M)` (*M* is a sorted version of *L*) using `perm` and `sorted`.
- d. Run `sort` on longer and longer lists until you lose patience. What is the time complexity of your program?
- e. Write a faster sorting algorithm, such as insertion sort or quicksort, in Prolog.



9.17 This exercise looks at the recursive application of rewrite rules, using logic programming. A rewrite rule (or **demodulator** in OTTER terminology) is an equation with a specified direction. For example, the rewrite rule $x + 0 \rightarrow x$ suggests replacing any expression that matches $x + 0$ with the expression x . Rewrite rules are a key component of equational reasoning systems. Use the predicate `rewrite(X, Y)` to represent rewrite rules. For example, the earlier rewrite rule is written as `rewrite(X+0, X)`. Some terms are *primitive* and cannot be further simplified; thus, we write `primitive(0)` to say that 0 is a primitive term.

- Write a definition of a predicate `simplify(X, Y)`, that is true when Y is a simplified version of X —that is, when no further rewrite rules apply to any subexpression of Y .
- Write a collection of rules for the simplification of expressions involving arithmetic operators, and apply your simplification algorithm to some sample expressions.
- Write a collection of rewrite rules for symbolic differentiation, and use them along with your simplification rules to differentiate and simplify expressions involving arithmetic expressions, including exponentiation.

9.18 This exercise considers the implementation of search algorithms in Prolog. Suppose that `successor(X, Y)` is true when state Y is a successor of state X ; and that `goal(X)` is true when X is a goal state. Write a definition for `solve(X, P)`, which means that P is a path (list of states) beginning with X , ending in a goal state, and consisting of a sequence of legal steps as defined by `successor`. You will find that depth-first search is the easiest way to do this. How easy would it be to add heuristic search control?

9.19 Suppose a knowledge base contains just the following first-order Horn clauses:

$$\begin{aligned} & \text{Ancestor}(\text{Mother}(x), x) \\ & \text{Ancestor}(x, y) \wedge \text{Ancestor}(y, z) \Rightarrow \text{Ancestor}(x, z) \end{aligned}$$

Consider a forward chaining algorithm that, on the j th iteration, terminates if the KB contains a sentence that unifies with the query, else adds to the KB every atomic sentence that can be inferred from the sentences already in the KB after iteration $j - 1$.

- For each of the following queries, say whether the algorithm will (1) give an answer (if so, write down that answer); or (2) terminate with no answer; or (3) never terminate.
 - $\text{Ancestor}(\text{Mother}(y), \text{John})$
 - $\text{Ancestor}(\text{Mother}(\text{Mother}(y)), \text{John})$
 - $\text{Ancestor}(\text{Mother}(\text{Mother}(\text{Mother}(y))), \text{Mother}(y))$
 - $\text{Ancestor}(\text{Mother}(\text{John}), \text{Mother}(\text{Mother}(\text{John})))$
- Can a resolution algorithm prove the sentence $\neg \text{Ancestor}(\text{John}, \text{John})$ from the original knowledge base? Explain how, or why not.
- Suppose we add the assertion that $\neg(\text{Mother}(x) = x)$ and augment the resolution algorithm with inference rules for equality. Now what is the answer to (b)?

9.20 Let \mathcal{L} be the first-order language with a single predicate $S(p, q)$, meaning “ p shaves q .” Assume a domain of people.

- a. Consider the sentence “There exists a person P who shaves every one who does not shave themselves, and only people that do not shave themselves.” Express this in \mathcal{L} .
- b. Convert the sentence in (a) to clausal form.
- c. Construct a resolution proof to show that the clauses in (b) are inherently inconsistent. (Note: you do not need any additional axioms.)

9.21 How can resolution be used to show that a sentence is valid? Unsatisfiable?

9.22 Construct an example of two clauses that can be resolved together in two different ways giving two different outcomes.

9.23 From “Horses are animals,” it follows that “The head of a horse is the head of an animal.” Demonstrate that this inference is valid by carrying out the following steps:

- a. Translate the premise and the conclusion into the language of first-order logic. Use three predicates: $HeadOf(h, x)$ (meaning “ h is the head of x ”), $Horse(x)$, and $Animal(x)$.
- b. Negate the conclusion, and convert the premise and the negated conclusion into conjunctive normal form.
- c. Use resolution to show that the conclusion follows from the premise.

9.24 Here are two sentences in the language of first-order logic:

(A) $\forall x \exists y (x \geq y)$

(B) $\exists y \forall x (x \geq y)$

- a. Assume that the variables range over all the natural numbers $0, 1, 2, \dots, \infty$ and that the “ \geq ” predicate means “is greater than or equal to.” Under this interpretation, translate (A) and (B) into English.
- b. Is (A) true under this interpretation?
- c. Is (B) true under this interpretation?
- d. Does (A) logically entail (B)?
- e. Does (B) logically entail (A)?
- f. Using resolution, try to prove that (A) follows from (B). Do this even if you think that (B) does not logically entail (A); continue until the proof breaks down and you cannot proceed (if it does break down). Show the unifying substitution for each resolution step. If the proof fails, explain exactly where, how, and why it breaks down.
- g. Now try to prove that (B) follows from (A).

9.25 Resolution can produce nonconstructive proofs for queries with variables, so we had to introduce special mechanisms to extract definite answers. Explain why this issue does not arise with knowledge bases containing only definite clauses.

9.26 We said in this chapter that resolution cannot be used to generate all logical consequences of a set of sentences. Can any algorithm do this?

10 CLASSICAL PLANNING

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action.

We have defined AI as the study of rational action, which means that **planning**—devising a plan of action to achieve one’s goals—is a critical part of AI. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the hybrid logical agent of Chapter 7. In this chapter we introduce a representation for planning problems that scales up to problems that could not be handled by those earlier approaches.

Section 10.1 develops an expressive yet carefully constrained language for representing planning problems. Section 10.2 shows how forward and backward search algorithms can take advantage of this representation, primarily through accurate heuristics that can be derived automatically from the structure of the representation. (This is analogous to the way in which effective domain-independent heuristics were constructed for constraint satisfaction problems in Chapter 6.) Section 10.3 shows how a data structure called the planning graph can make the search for a plan more efficient. We then describe a few of the other approaches to planning, and conclude by comparing the various approaches.

This chapter covers fully observable, deterministic, static environments with single agents. Chapters 11 and 17 cover partially observable, stochastic, dynamic environments with multiple agents.

10.1 DEFINITION OF CLASSICAL PLANNING

The problem-solving agent of Chapter 3 can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well. The hybrid propositional logical agent of Chapter 7 can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

PDDL

In response to this, planning researchers have settled on a **factored representation**—one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language, that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL; we select a simple version and alter its syntax to be consistent with the rest of the book.¹ We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

SET SEMANTICS

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and a state in a package delivery problem might be $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$. **Database semantics** is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that $Truck_1$ and $Truck_2$ are distinct. The following fluents are *not* allowed in a state: $At(x, y)$ (because it is non-ground), $\neg Poor$ (because it is a negation), and $At(Father(Fred), Sydney)$ (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations. The **set semantics** is sometimes easier to deal with.

ACTION SCHEMA

Actions are described by a set of action schemas that implicitly define the $ACTIONS(s)$ and $RESULT(s, a)$ functions needed to do a problem-solving search. We saw in Chapter 7 that any system for action description needs to solve the frame problem—to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector Δ . A concise description of the action should mention only Δ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single **action schema**. The schema is a **lifted** representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

$$\begin{aligned} &Action(Fly(p, from, to), \\ &\quad PRECOND: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\ &\quad EFFECT: \neg At(p, from) \wedge At(p, to)) \end{aligned}$$

PRECONDITION

EFFECT

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground

¹ PDDL was derived from the original STRIPS planning language (Fikes and Nilsson, 1971), which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals.

action that results from substituting values for all the variables:

$Action(Fly(P_1, SFO, JFK),$
 $PRECOND: At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$
 $EFFECT: \neg At(P_1, SFO) \wedge At(P_1, JFK))$

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action a can be executed in state s if s entails the precondition of a . Entailment can also be expressed with the set semantics: $s \models q$ iff every positive literal in q is in s and every negated literal in q is not. In formal notation we say

$$(a \in ACTIONS(s)) \Leftrightarrow s \models PRECOND(a),$$

where any variables in a are universally quantified. For example,

$$\forall p, from, to \ (Fly(p, from, to) \in ACTIONS(s)) \Leftrightarrow$$

$$s \models (At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to))$$

APPLICABLE

We say that action a is **applicable** in state s if the preconditions are satisfied by s . When an action schema a contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the Fly action can be instantiated as $Fly(P_1, SFO, JFK)$ or as $Fly(P_2, JFK, SFO)$, both of which are applicable in the initial state. If an action a has v variables, then, in a domain with k unique names of objects, it takes $O(v^k)$ time in the worst case to find the applicable ground actions.

PROPOSITIONALIZE

Sometimes we want to **propositionalize** a PDDL problem—replace each action schema with a set of ground actions and then use a propositional solver such as SATPLAN to find a solution. However, this is impractical when v and k are large.

DELETE LIST

ADD LIST

The **result** of executing action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s , removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or $DEL(a)$), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or $ADD(a)$):

$$RESULT(s, a) = (s - DEL(a)) \cup ADD(a). \quad (10.1)$$

For example, with the action $Fly(P_1, SFO, JFK)$, we would remove $At(P_1, SFO)$ and add $At(P_1, JFK)$. It is a requirement of action schemas that any variable in the effect must also appear in the precondition. That way, when the precondition is matched against the state s , all the variables will be bound, and $RESULT(s, a)$ will therefore have only ground atoms. In other words, ground states are closed under the $RESULT$ operation.

Also note that the fluents do not explicitly refer to time, as they did in Chapter 7. There we needed superscripts for time, and successor-state axioms of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

In PDDL the times and states are implicit in the action schemas: the precondition always refers to time t and the effect to time $t + 1$.

A set of action schemas serves as a definition of a planning *domain*. A specific *problem* within the domain is defined with the addition of an initial state and a goal. The **initial**

```

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
    PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    EFFECT: ¬ At(p, from) ∧ At(p, to))

```

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

INITIAL STATE

GOAL

state is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.) The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as *At*(*p*, *SFO*) ∧ *Plane*(*p*). Any variables are treated as existentially quantified, so this goal is to have *any* plane at SFO. The problem is solved when we can find a sequence of actions that end in a state *s* that entails the goal. For example, the state *Rich* ∧ *Famous* ∧ *Miserable* entails the goal *Rich* ∧ *Famous*, and the state *Plane*(*Plane*₁) ∧ *At*(*Plane*₁, *SFO*) entails the goal *At*(*p*, *SFO*) ∧ *Plane*(*p*).

Now we have defined planning as a search problem: we have an initial state, an ACTIONS function, a RESULT function, and a goal test. We'll look at some example problems before investigating efficient search algorithms.

10.1.1 Example: Air cargo transport

Figure 10.1 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: *Load*, *Unload*, and *Fly*. The actions affect two predicates: *In*(*c*, *p*) means that cargo *c* is inside plane *p*, and *At*(*x*, *a*) means that object *x* (either plane or cargo) is at airport *a*. Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *At* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
 Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```


Finally, there is the problem of spurious actions such as $Fly(P_1, JFK, JFK)$, which should be a no-op, but which has contradictory effects (according to the definition, the effect would include $At(P_1, JFK) \wedge \neg At(P_1, JFK)$). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the *from* and *to* airports must be different; see another example of this in Figure 10.3.

10.1.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$.

```

Init(Tire(Flat)  $\wedge$  Tire(Spare)  $\wedge$  At(Flat, Axle)  $\wedge$  At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(obj, loc),
  PRECOND: At(obj, loc)
  EFFECT:  $\neg At(obj, loc) \wedge At(obj, Ground)$ )
Action(PutOn(t, Axle),
  PRECOND: Tire(t)  $\wedge At(t, Ground) \wedge \neg At(Flat, Axle)$ 
  EFFECT:  $\neg At(t, Ground) \wedge At(t, Axle)$ )
Action(LeaveOvernight,
  PRECOND:
  EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$ 
          $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

```

Figure 10.2 The simple spare tire problem.

10.1.3 Example: The blocks world

BLOCKS WORLD

One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table.² The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top

² The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

```

Init( $On(A, Table) \wedge On(B, Table) \wedge On(C, A)$ 
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)$ )
Goal( $On(A, B) \wedge On(B, C)$ )
Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$ 
 $(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$ ,
  EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ )
Action( $MoveToTable(b, x)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x)$ ,
  EFFECT:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$ )

```

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

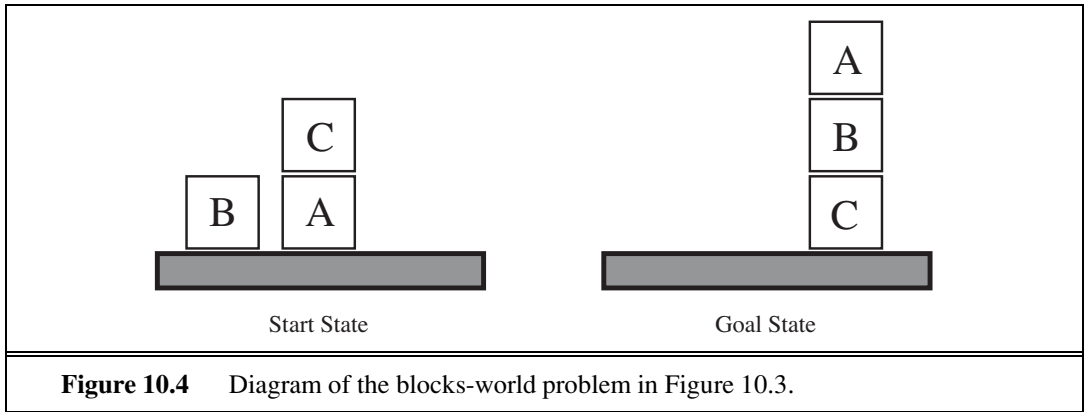


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

of what other blocks. For example, a goal might be to get block A on B and block B on C (see Figure 10.4).

We use $On(b, x)$ to indicate that block b is on x , where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b, x, y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg \exists x On(x, b)$ or, alternatively, $\forall x \neg On(x, b)$. Basic PDDL does not allow quantifiers, so instead we introduce a predicate $Clear(x)$ that is true when nothing is on x . (The complete problem description is in Figure 10.3.)

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the $Move$ schema is

```

Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y)$ ,
  EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ ) .

```

Unfortunately, this does not maintain $Clear$ properly when x or y is the table. When x is the $Table$, this action has the effect $Clear(Table)$, but the table should not become clear; and when $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear

for us to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

$$\begin{aligned} & \text{Action}(\text{MoveToTable}(b, x), \\ & \quad \text{PRECOND: } On(b, x) \wedge \text{Clear}(b), \\ & \quad \text{EFFECT: } On(b, \text{Table}) \wedge \text{Clear}(x) \wedge \neg On(b, x)) . \end{aligned}$$

Second, we take the interpretation of $\text{Clear}(x)$ to be “there is a clear space on x to hold a block.” Under this interpretation, $\text{Clear}(\text{Table})$ will always be true. The only problem is that nothing prevents the planner from using $\text{Move}(b, x, \text{Table})$ instead of $\text{MoveToTable}(b, x)$. We could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate Block and add $\text{Block}(b) \wedge \text{Block}(y)$ to the precondition of Move .

10.1.4 The complexity of classical planning

In this subsection we consider the theoretical complexity of planning and distinguish two decision problems. **PlanSAT** is the question of whether there exists any plan that solves a planning problem. **Bounded PlanSAT** asks whether there is a solution of length k or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols. For proofs of the assertions in this section, see Ghallab *et al.* (2004).

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult. For example, if we disallow negative effects, both problems are still NP-hard. However, if we also disallow negative preconditions, PlanSAT reduces to the class P.

These worst-case results may seem discouraging. We can take solace in the fact that agents are usually not asked to find plans for arbitrary worst-case problem instances, but rather are asked for plans in specific domains (such as blocks-world problems with n blocks), which can be much easier than the theoretical worst case. For many domains (including the blocks world and the air cargo world), Bounded PlanSAT is NP-complete while PlanSAT is in P; in other words, optimal planning is usually hard, but sub-optimal planning is sometimes easy. To do well on easier-than-worst-case problems, we will need good search heuristics. That’s the true advantage of the classical planning formalism: it has facilitated the development of very accurate domain-independent heuristics, whereas systems based on successor-state axioms in first-order logic have had less success in coming up with good heuristics.

PlanSAT

Bounded PlanSAT

10.2 ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.5 compares forward and backward searches.

10.2.1 Forward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms from Chapter 3 or a local search algorithm from Chapter 4 (provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until around 1998 it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why.

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an

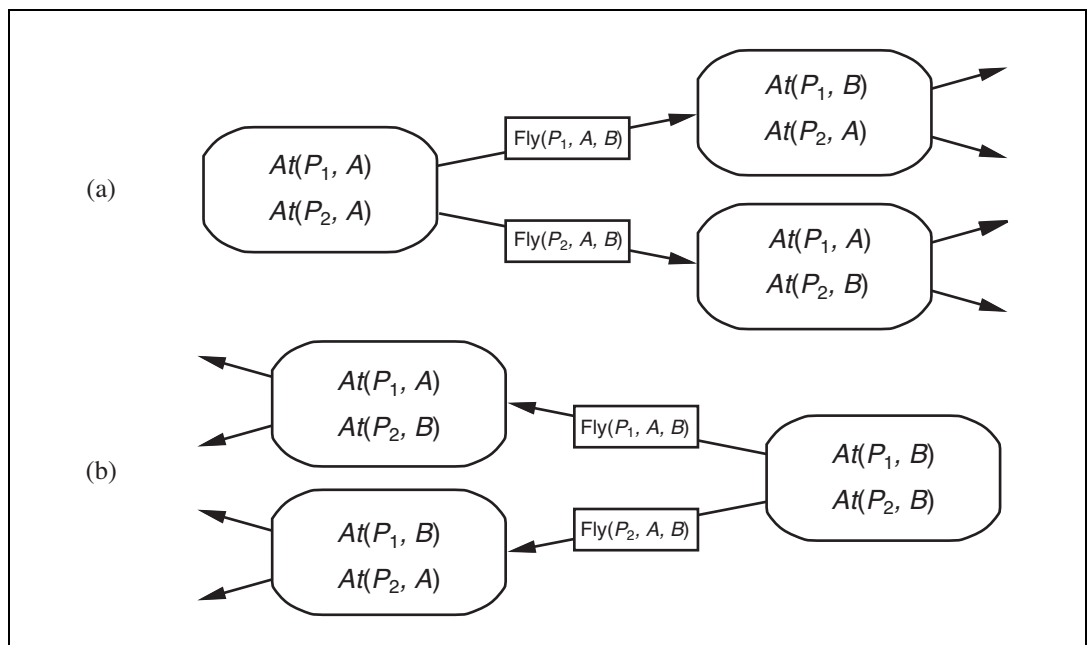


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

action schema $Buy(isbn)$ with effect $Own(isbn)$. ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B . There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A , fly the plane to B , and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 10.2.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

10.2.2 Backward (regression) relevant-states search

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). As in belief-state search (Section 4.4), there is a *set* of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal $\neg Poor \wedge Famous$ describes those states in which *Poor* is false, *Famous* is true, and any other fluent can have any value. If there are n ground fluents in a domain, then there are 2^n ground states (each fluent can be true or false), but 3^n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n -queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a , the regression from g over a gives us a state description g' defined by

$$g' = (g - \text{ADD}(a)) \cup \text{Precond}(a) .$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. Note that $\text{DEL}(a)$ does not appear in the formula; that's because while we know the fluents in $\text{DEL}(a)$ are no longer true after the action, we don't know whether or not they were true before, so there's nothing to be said about them.

To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $At(C_2, SFO)$. That suggests the action $Unload(C_2, p', SFO)$:

$Action(Unload(C_2, p', SFO),$
 PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$
 EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p')$.

(Note that we have **standardized** variable names (changing p to p' in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan. The same approach was used in Chapter 9 for first-order logical inference.) This represents unloading the package from an *unspecified* plane at SFO; any plane will do, but we need not say which one now. We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any* of the planes by implicitly quantifying over p' . The regressed state description is

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) .$$

The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the *last* step in a plan leading up to the current goal state.

RELEVANCE

For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal. Now, if the goal is $A \wedge B \wedge C$ and an action has the effect $A \wedge B \wedge \neg C$ then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there. But it is not relevant in the technical sense defined here, because this action could not be the *final* step of a solution—we would always need at least one more step to achieve C .

Given the goal $At(C_2, SFO)$, several instantiations of $Unload$ are relevant: we could choose any specific plane to unload from, or we could leave the plane unspecified by using the action $Unload(C_2, p', SFO)$. We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

As another example, consider the goal $Own(0136042597)$, given an initial state with 10 billion ISBNs, and the single action schema

$$A = Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i)) .$$

As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions. But with backward search, we would unify the goal $Own(0136042597)$ with the (standardized) effect $Own(i')$, yielding the substitution $\theta = \{i'/0136042597\}$. Then we would regress over the action $Subst(\theta, A')$ to yield the predecessor state description $ISBN(0136042597)$. This is part of, and thus entailed by, the initial state, so we are done.

We can make this more formal. Assume a goal description g which contains a goal literal g_i and an action schema A that is standardized to produce A' . If A' has an effect literal e'_j where $\text{Unify}(g_i, e'_j) = \theta$ and where we define $a' = \text{SUBST}(\theta, A')$ and if there is no effect in a' that is the negation of a literal in g , then a' is a relevant action towards g .

Backward search keeps the branching factor lower than forward search, for most problem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

10.2.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function $h(s)$ estimates the distance from a state s to the goal and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate—then we can use A^* search to find optimal solutions. An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

Think of a search problem as a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two ways we can relax this problem to make it easier: by adding more edges to the graph, making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of $\log n$ of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding-block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning

IGNORE
PRECONDITIONS
HEURISTIC

SET-COVER
PROBLEM

problem involving tiles with a single schema *Slide*:

$Action(Slide(t, s_1, s_2),$

$PRECOND: On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)$

$EFFECT: On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))$

As we saw in Section 3.6, if we remove the preconditions $Blank(s_2) \wedge Adjacent(s_1, s_2)$ then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove $Blank(s_2)$ then we get the Manhattan-distance heuristic. It is easy to see how these heuristics could be derived automatically from the action schema description. The ease of manipulating the schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

IGNORE DELETE
LISTS

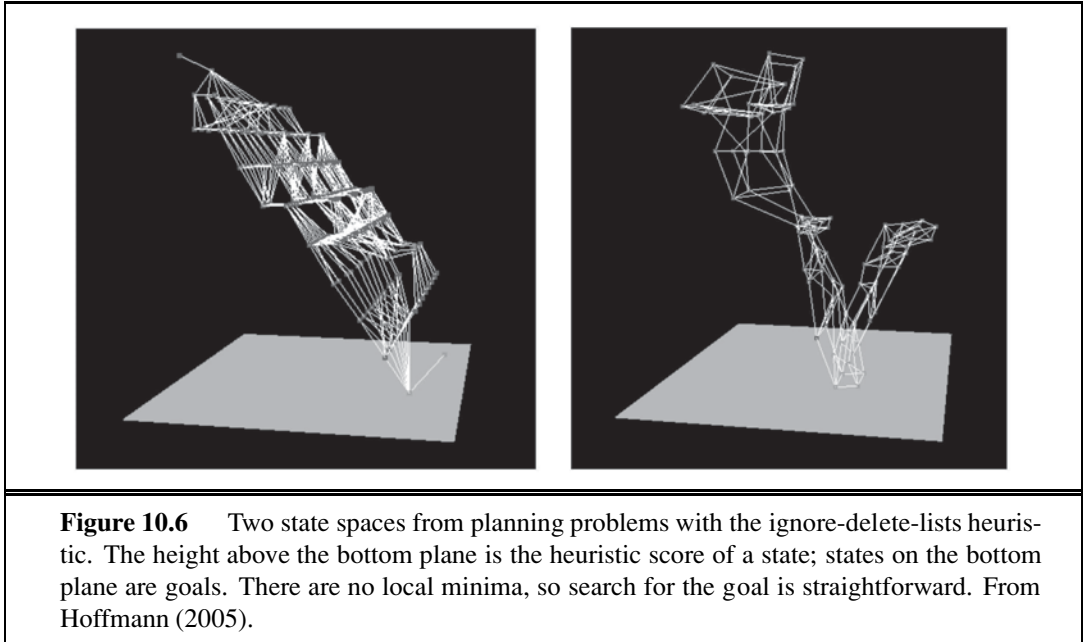
Another possibility is the **ignore delete lists** heuristic. Assume for a moment that all goals and preconditions contain only positive literals³ We want to create a relaxed version of the original problem that will be easier to solve, and where the length of the solution will serve as a good heuristic. We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill-climbing. Figure 10.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

STATE ABSTRACTION

The relaxed problems leave us with a simplified—but still expensive—planning problem just to calculate the value of the heuristic function. Many planning problems have 10^{100} states or more, and relaxing the *actions* does nothing to reduce the number of states. Therefore, we now look at relaxations that decrease the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

The easiest form of state abstraction is to ignore some fluents. For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. So there are $50^{10} \times 200^{50+10} \approx 10^{155}$ states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $5^{10} \times 5^{5+10} \approx 10^{17}$ states. A solution in this abstract state space will be shorter than a solution in the original space (and thus will be an admissible heuristic), and the abstract solution is easy to extend to a solution to the original problem (by adding additional *Load* and *Unload* actions).

³ Many problems are written with this convention. For problems that aren't, replace every negative literal $\neg P$ in a goal or precondition with a new positive literal, P' .



DECOMPOSITION
SUBGOAL
INDEPENDENCE

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently*. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

Suppose the goal is a set of fluents G , which we divide into disjoint subsets G_1, \dots, G_n . We then find plans P_1, \dots, P_n that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of G ? We can think of each $\text{Cost}(P_i)$ as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So $\max_i \text{COST}(P_i)$ is admissible, and sometimes it is exactly correct: it could be that P_1 serendipitously achieves all the G_i . But in most cases, in practice the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when we can determine that G_i and G_j are **independent**. If the effects of P_i leave all the preconditions and goals of P_j unchanged, then the estimate $\text{COST}(P_i) + \text{COST}(P_j)$ is admissible, and more accurate than the max estimate. We show in Section 10.3.1 that planning graphs can help provide better heuristic estimates.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The tech-

niques of **pattern databases** from Section 3.6.3 can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

An example of a system that makes use of effective heuristics is FF, or FASTFORWARD (Hoffmann, 2005), a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph (see Section 10.3). FF then uses hill-climbing search (modified to keep track of the plan) with the heuristic to find a solution. When it hits a plateau or local maximum—when no action leads to a state with better heuristic score—then FF uses iterative deepening search until it finds a state that is better, or it gives up and restarts hill-climbing.

10.3 PLANNING GRAPHS

PLANNING GRAPH

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a **planning graph** can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question “can we reach state G from state S_0 ” immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial-size approximation to this tree that can be constructed quickly. The planning graph can’t answer definitively whether G is reachable from S_0 , but it can *estimate* how many steps it takes to reach G . The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

LEVEL

A planning graph is a directed graph organized into **levels**: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0 ; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0 ; then alternating levels S_i followed by A_i ; until we reach a termination condition (to be discussed later).

Roughly speaking, S_i contains all the literals that *could* hold at time i , depending on the actions executed at preceding time steps. If it is possible that either P or $\neg P$ could hold, then both will be represented in S_i . Also roughly speaking, A_i contains all the actions that *could* have their preconditions satisfied at time i . We say “roughly speaking” because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level S_j when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of ac-

depending on the choice of actions in A_0 , either, but not both, could be the result. In other words, S_1 represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**. The graph in Figure 10.8 levels off at S_2 .

LEVELED OFF

What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i , along with constraints saying that two actions cannot both be executed at the same level. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1} , along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links.

We now define mutex links for both actions and literals. A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:

- *Inconsistent effects*: one action negates an effect of the other. For example, $Eat(Cake)$ and the persistence of $Have(Cake)$ have inconsistent effects because they disagree on the effect $Have(Cake)$.
- *Interference*: one of the effects of one action is the negation of a precondition of the other. For example $Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition.
- *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $Bake(Cake)$ and $Eat(Cake)$ are mutex because they compete on the value of the $Have(Cake)$ precondition.

A mutex relation holds between two *literals* at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called *inconsistent support*. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistence action, is mutex with the only way of achieving $Eaten(Cake)$, namely $Eat(Cake)$. In S_2 the two literals are not mutex, because there are new ways of achieving them, such as $Bake(Cake)$ and the persistence of $Eaten(Cake)$, that are not mutex.

A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including the no-ops), $(a + l)^2$ mutex links, and $2(al + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

10.3.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s . We call this the

LEVEL COST

level cost of g_i . In Figure 10.8, *Have*(*Cake*) has level cost 0 and *Eaten*(*Cake*) has level cost 1. It is easy to show (Exercise 10.10) that these estimates are admissible for the individual goals. The estimate might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a **serial planning graph** for computing heuristics. A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

SERIAL PLANNING GRAPH

MAX-LEVEL

To estimate the cost of a *conjunction* of goals, there are three simple approaches. The **max-level** heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily accurate.

LEVEL SUM

The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 10.2. For our problem, the level-sum heuristic estimate for the conjunctive goal *Have*(*Cake*) \wedge *Eaten*(*Cake*) will be $0 + 1 = 1$, whereas the correct answer is 2, achieved by the plan [*Eat*(*Cake*), *Bake*(*Cake*)]. That doesn't seem so bad. A more serious error is that if *Bake*(*Cake*) were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

SET-LEVEL

Finally, the **set-level** heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without *Bake*(*Cake*). It is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently solvable. To understand the nature of the relaxed problem, we need to understand exactly what it means for a literal g to appear at level S_i in the planning graph. Ideally, we would like it to be a guarantee that there exists a plan with i action levels that achieves g , and also that if g does not appear, there is no such plan. Unfortunately, making that guarantee is as difficult as solving the original planning problem. So the planning graph makes the second half of the guarantee (if g does not appear, there is no plan), but if g does appear, then all the planning graph promises is that there is a plan that *possibly* achieves g and has no “obvious” flaws. An obvious flaw is defined as a flaw that can be detected by considering two actions or two literals at a time—in other words, by looking at the mutex relations. There could be more subtle flaws involving three, four, or more actions, but experience has shown that it is not worth the computational effort to keep track of these possible flaws. This is similar to a lesson learned from constraint satisfaction problems—that it is often worthwhile to compute 2-consistency before searching for a solution, but less often worthwhile to compute 3-consistency or higher. (See page 211.)

One example of an unsolvable problem that cannot be recognized as such by a planning graph is the blocks-world problem where the goal is to get block A on B , B on C , and C on A . This is an impossible goal; a tower with the bottom on top of the top. But a planning graph

cannot detect the impossibility, because any two of the three subgoals are achievable. There are no mutexes between any pair of literals, only between the three as a whole. To detect that this problem is impossible, we would have to search over the planning graph.

10.3.2 The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm (Figure 10.9) repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

function GRAPHPLAN(problem) returns solution or failure
  graph  $\leftarrow$  INITIAL-PLANNING-GRAPH(problem)
  goals  $\leftarrow$  CONJUNCTS(problem.GOAL)
  nogoods  $\leftarrow$  an empty hash table
  for tl = 0 to  $\infty$  do
    if goals all non-mutex in  $S_t$  of graph then
      solution  $\leftarrow$  EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution  $\neq$  failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph  $\leftarrow$  EXPAND-GRAPH(graph, problem)

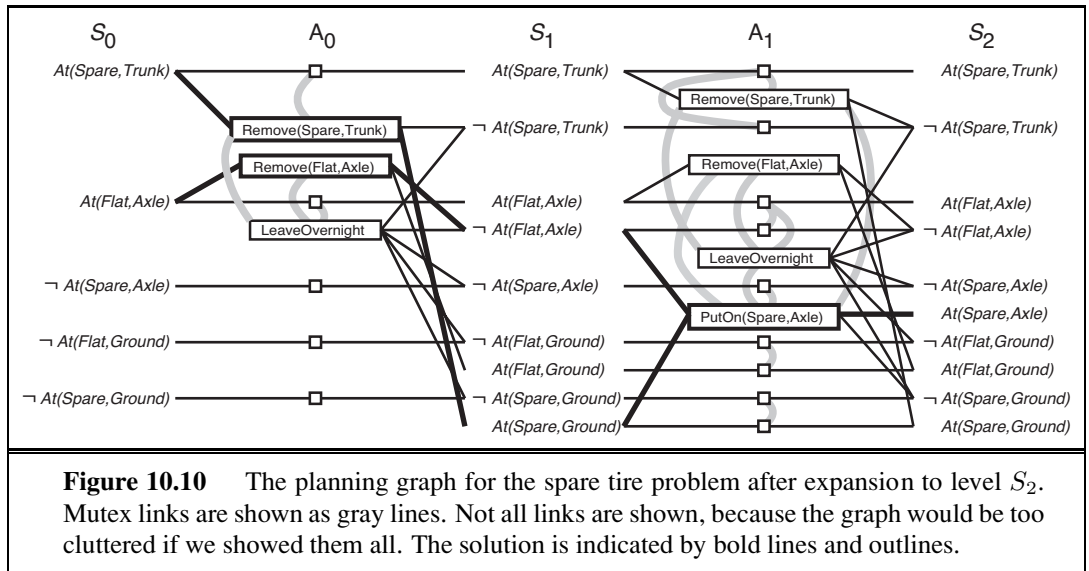
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph representing the initial state. The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as $Tire(Spare)$) and the irrelevant negative literals. The goal $At(Spare, Axle)$ is not present in S_0 , so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at level S_0 (i.e., all the actions except $PutOn(Spare, Axle)$), along with persistence actions for all the literals in S_0 . The effects of the actions are added at level S_1 . EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

$At(Spare, Axle)$ is still not present in S_1 , so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown in Figure 10.10. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:

- *Inconsistent effects*: $Remove(Spare, Trunk)$ is mutex with $LeaveOvernight$ because one has the effect $At(Spare, Ground)$ and the other has its negation.



- *Interference*: $Remove(Flat, Axle)$ is mutex with $LeaveOvernight$ because one has the precondition $At(Flat, Axle)$ and the other has its negation as an effect.
- *Competing needs*: $PutOn(Spare, Axle)$ is mutex with $Remove(Flat, Axle)$ because one has $At(Flat, Axle)$ as a precondition and the other has its negation.
- *Inconsistent support*: $At(Spare, Axle)$ is mutex with $At(Flat, Axle)$ in S_2 because the only way of achieving $At(Spare, Axle)$ is by $PutOn(Spare, Axle)$, and that is mutex with the persistence action that is the only way of achieving $At(Flat, Axle)$. Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

This time, when we go back to the start of the loop, all the literals from the goal are present in S_2 , and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n , along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By “conflict free,” we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.

- The goal is to reach a state at level S_0 such that all the goals are satisfied.
- The cost of each action is 1.

For this particular problem, we start at S_2 with the goal $At(Spare, Axle)$. The only choice we have for achieving the goal set is $PutOn(Spare, Axle)$. That brings us to a search state at S_1 with goals $At(Spare, Ground)$ and $\neg At(Flat, Axle)$. The former can be achieved only by $Remove(Spare, Trunk)$, and the latter by either $Remove(Flat, Axle)$ or $LeaveOvernight$. But $LeaveOvernight$ is mutex with $Remove(Spare, Trunk)$, so the only solution is to choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$. That brings us to a search state at S_0 with the goals $At(Spare, Trunk)$ and $At(Flat, Axle)$. Both of these are present in the state, so we have a solution: the actions $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0 , followed by $PutOn(Spare, Axle)$ in A_1 .

In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the $(level, goals)$ pair as a **no-good**, just as we did in constraint learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

10.3.3 Termination of GRAPHPLAN

So far, we have skated over the question of termination. Here we show that GRAPHPLAN will in fact terminate and return failure when there is no solution.

The first thing to understand is why we can't stop expanding the graph as soon as it has leveled off. Consider an air cargo domain with one plane and n pieces of cargo at airport A , all of which have airport B as their destination. In this version of the problem, only one piece of cargo can fit in the plane at a time. The graph will level off at level 4, reflecting the fact that for any single piece of cargo, we can load it, fly it, and unload it at the destination in three steps. But that does not mean that a solution can be extracted from the graph at level 4; in fact a solution will require $4n - 1$ steps: for each piece of cargo we load, fly, and unload, and for all but the last piece we need to fly back to airport A to get the next piece.

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution.

Now all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is that certain properties of planning graphs are monotonically increasing or decreasing. “X increases monotonically” means that the set of Xs at level $i + 1$ is a superset (not necessarily proper) of the set at level i . The properties are as follows:

- *Literals increase monotonically*: Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- *Actions increase monotonically*: Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- *Mutexes decrease monotonically*: If two actions are mutex at a given level A_i , then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level S_i nor actions that cannot be executed at level A_i . We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything.

The proof can be handled by cases: if actions A and B are mutex at level A_i , it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i , they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i : that level must contain a precondition of A that is mutex with a precondition of B . Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so, by induction, the mutexes must be decreasing.

- *No-goods decrease monotonically*: If a set of goals is not achievable at a given level, then they are not achievable in any *previous* level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.

Because the actions and literals increase monotonically and because there are only a finite number of actions and literals, there must come a level that has the same number of actions and literals as the previous level. Because mutexes and no-goods decrease, and because there can never be fewer than zero mutexes or no-goods, there must come a level that has the same number of mutexes and no-goods as the previous level. Once a graph has reached this state, then if one of the goals is missing or is mutex with another goal, then we can stop the GRAPHPLAN algorithm and return failure. That concludes a sketch of the proof; for more details see Ghallab *et al.* (2004).

Year	Track	Winning Systems (approaches)
2008	Optimal	GAMER (model checking, bidirectional search)
2008	Satisficing	LAMA (fast downward search with FF heuristic)
2006	Optimal	SATPLAN, MAXPLAN (Boolean satisfiability)
2006	Satisficing	SGPLAN (forward search; partitions into independent subproblems)
2004	Optimal	SATPLAN (Boolean satisfiability)
2004	Satisficing	FAST DIAGONALLY DOWNWARD (forward search with causal graph)
2002	Automated	LPG (local search, planning graphs converted to CSPs)
2002	Hand-coded	TLPLAN (temporal action logic with control rules for forward search)
2000	Automated	FF (forward search)
2000	Hand-coded	TALPLANNER (temporal action logic with control rules for forward search)
1998	Automated	IPP (planning graphs); HSP (forward search)

Figure 10.11 Some of the top-performing systems in the International Planning Competition. Each year there are various tracks: “Optimal” means the planners must produce the shortest possible plan, while “Satisficing” means nonoptimal solutions are accepted. “Hand-coded” means domain-specific heuristics are allowed; “Automated” means they are not.

10.4 OTHER CLASSICAL PLANNING APPROACHES

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics (Section 10.2)
- Search using a planning graph (Section 10.3)

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

10.4.1 Classical planning as Boolean satisfiability

In Section 7.7.4 we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
- Define the initial state: assert F^0 for every fluent F in the problem’s initial state, and $\neg F$ for every fluent not mentioned in the initial state.
- Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A

on another block, $On(A, x) \wedge Block(x)$ in a world with objects A, B and C , would be replaced by the goal

$$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C)).$$

- Add successor-state axioms: For each fluent F , add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t),$$

where $ActionCausesF$ is a disjunction of all the ground actions that have F in their add list, and $ActionCausesNotF$ is a disjunction of all the ground actions that have F in their delete list.

- Add precondition axioms: For each ground action A , add the axiom $A^t \Rightarrow PRE(A)^t$, that is, if an action is taken at time t , then the preconditions must have been true.
- Add action exclusion axioms: say that every action is distinct from every other action.

The resulting translation is in the form that we can hand to SATPLAN to find a solution.

10.4.2 Planning as first-order logical deduction: Situation calculus

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from A to B regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.

The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, $South^2$ means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching *situations*, using a representation called **situation calculus** that works like this:

- The initial state is called a **situation**. If s is a situation and a is an action, then $RESULT(s, a)$ is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same: $(RESULT(s, a) = RESULT(s', a')) \Leftrightarrow (s = s' \wedge a = a')$. Some examples of actions and situations are shown in Figure 10.12.
- A function or relation that can vary from one situation to the next is a **fluent**. By convention, the situation s is always the last argument to the fluent, for example $At(x, l, s)$ is a relational fluent that is true when object x is at location l in situation s , and $Location$ is a functional fluent such that $Location(x, s) = l$ holds in the same situations as $At(x, l, s)$.
- Each action's preconditions are described with a **possibility axiom** that says when the action can be taken. It has the form $\Phi(s) \Rightarrow Poss(a, s)$ where $\Phi(s)$ is some formula

SITUATION
CALCULUS

SITUATION

POSSIBILITY AXIOM

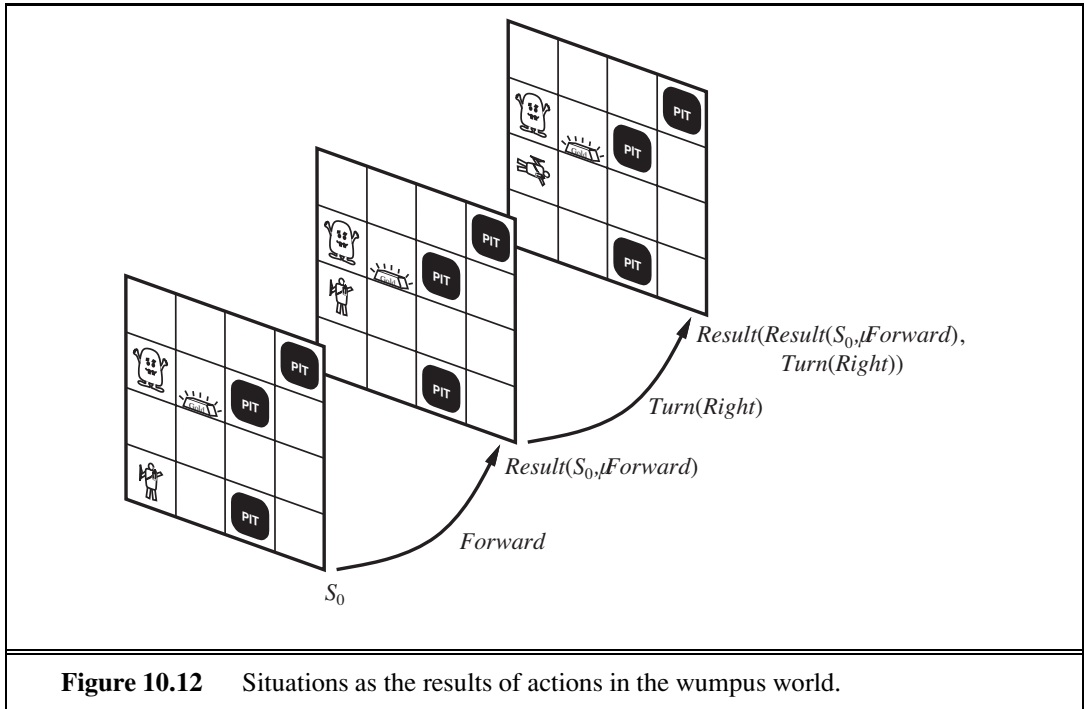


Figure 10.12 Situations as the results of actions in the wumpus world.

involving s that describes the preconditions. An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:

$$Alive(Agent, s) \wedge Have(Arrow, s) \Rightarrow Poss(Shoot, s)$$

- Each fluent is described with a **successor-state axiom** that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form

$$\begin{aligned} \text{Action is possible} &\Rightarrow \\ &(\text{Fluent is true in result state} \Leftrightarrow \text{Action's effect made it true} \\ &\quad \vee \text{It was true before and action left it alone}) . \end{aligned}$$

For example, the axiom for the relational fluent *Holding* says that the agent is holding some gold g after executing a possible action if and only if the action was a *Grab* of g or if the agent was already holding g and the action was not releasing it:

$$\begin{aligned} Poss(a, s) &\Rightarrow \\ &(\text{Holding}(Agent, g, Result(a, s)) \Leftrightarrow \\ &\quad a = Grab(g) \vee (\text{Holding}(Agent, g, s) \wedge a \neq Release(g))) . \end{aligned}$$

- We need **unique action axioms** so that the agent can deduce that, for example, $a \neq Release(g)$. For each distinct pair of action names A_i and A_j we have an axiom that says the actions are different:

$$A_i(x, \dots) \neq A_j(y, \dots)$$

and for each action name A_i we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

$$A_i(x_1, \dots, x_n) = A_i(y_1, \dots, y_n) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n.$$

- A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Work in situation calculus has done a lot to define the formal semantics of planning and to open up new areas of investigation. But so far there have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus.

10.4.3 Planning as constraint satisfaction

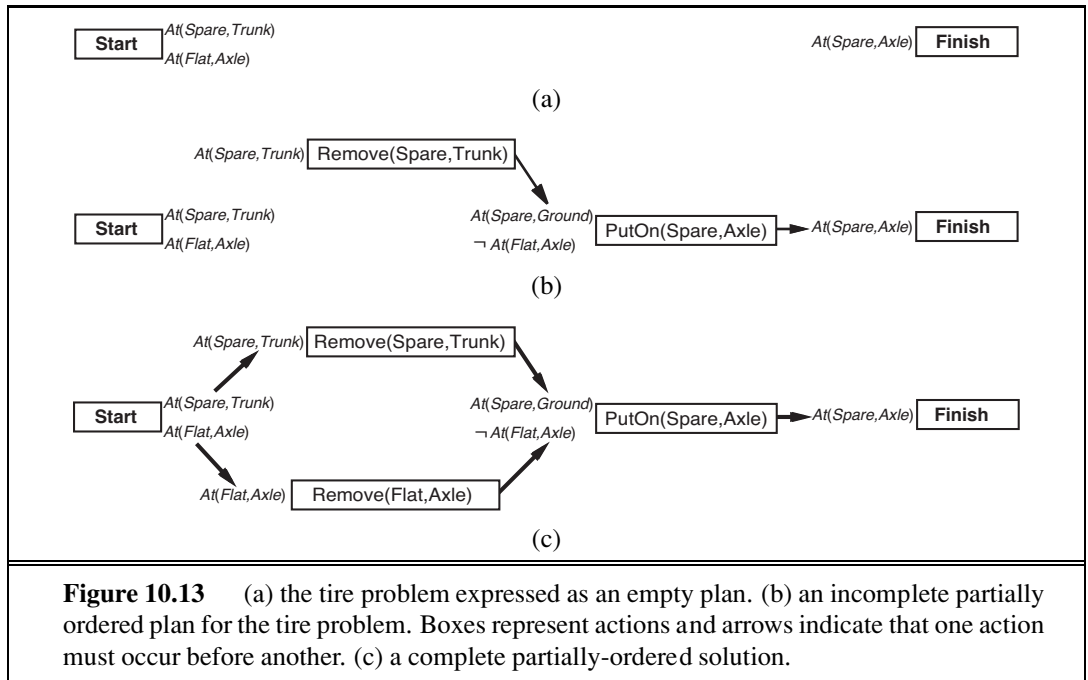
We have seen that constraint satisfaction has a lot in common with Boolean satisfiability, and we have seen that CSP techniques are effective for scheduling problems, so it is not surprising that it is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem (Section 10.4.1), with one important simplification: at each time step we need only a single variable, $Action^t$, whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms. It is also possible to encode a planning graph into a CSP. This is the approach taken by GP-CSP (Do and Kambhampati, 2003).

10.4.4 Planning as refinement of partially ordered plans

All the approaches we have seen so far construct *totally ordered* plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently.

An alternative is to represent plans as *partially ordered* structures: a plan is a set of actions and a set of constraints of the form $Before(a_i, a_j)$ saying that one action occurs before another. In the bottom of Figure 10.13, we see a partially ordered plan that is a solution to the spare tire problem. Actions are boxes and ordering constraints are arrows. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order as long as they are both completed before the $PutOn(Spare, Axle)$ action.

Partially ordered plans are created by a *search through the space of plans* rather than through the state space. We start with the empty plan consisting of just the initial state and the goal, with no actions in between, as in the top of Figure 10.13. The search procedure then looks for a **flaw** in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else). A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves $At(Spare, Axle)$. One way to correct the flaw is to insert into the plan



the action $PutOn(Spare, Axle)$. Of course that introduces some new flaws: the preconditions of the new action are not achieved. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13. At every step, we make the **least commitment** possible to fix the flaw. For example, in adding the action $Remove(Spare, Trunk)$ we need to commit to having it occur before $PutOn(Spare, Axle)$, but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

LEAST COMMITMENT

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial-order planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems.

However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

10.5 ANALYSIS OF PLANNING APPROACHES

Planning combines the two major areas of AI we have covered so far: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are n propositions in a domain, then there are 2^n states. As we have seen, planning is PSPACE-hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B , which in turn is on C , which in turn is on the *Table*, as in Figure 10.4 on page 371), then the subgoals are serializable bottom to top: if we first achieve C on *Table*, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner that commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is *designed* by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

SERIALIZABLE
SUBGOAL

10.6 SUMMARY

In this chapter, we defined the problem of planning in deterministic, fully observable, static environments. We described the PDDL representation for planning problems and several algorithmic approaches for solving them. The points to remember:

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.
- A **planning graph** can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the literals or actions that could occur at that time step and encodes mutual exclusion (mutex) relations among literals or actions that cannot co-occur. Planning graphs yield useful heuristics for state-space and partial-order planners and can be used directly in the GRAPHPLAN algorithm.
- Other approaches include first-order deduction over situation calculus axioms; encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.
- Each of the major approaches to planning has its adherents, and there is as yet no consensus on which is best. Competition and cross-fertilization among the approaches have resulted in significant gains in efficiency for planning systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI. Its overall control structure was modeled on that of GPS, the General Problem Solver (Newell and Simon, 1961), a state-space search system that used means–ends analysis. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and its relationship to more recent planning efforts.

The representation language used by STRIPS has been far more influential than its algorithmic approach; what we call the “classical” language is close to what STRIPS used.

The Action Description Language, or ADL (Pednault, 1986), relaxed some of the STRIPS restrictions and made it possible to encode more realistic problems. Nebel (2000) explores schemes for compiling ADL into STRIPS. The Problem Domain Description Language, or PDDL (Ghallab *et al.*, 1998), was introduced as a computer-parsable, standardized syntax for representing planning problems and has been used as the standard language for the International Planning Competition since 1998. There have been several extensions; the most recent version, PDDL 3.0, includes plan constraints and preferences (Gerevini and Long, 2005).

LINEAR PLANNING

Planners in the early 1970s generally considered totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then stringing the subplans together in some order. This approach, called **linear planning** by Sacerdoti (1975), was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly (see Exercise 10.7), found by Allen Brown during experimentation with the HACKER system (Sussman, 1975). A complete planner must allow for **interleaving** of actions from different subplans within a single sequence. The notion of serializable subgoals (Korf, 1987) corresponds exactly to the set of problems for which noninterleaved planners are complete.

INTERLEAVING

One solution to the interleaving problem was goal-regression planning, a technique in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals. This was introduced by Waldinger (1975) and also used by Warren's (1974) WARPLAN. WARPLAN is also notable in that it was the first planner to be written in a logic programming language (Prolog) and is one of the best examples of the remarkable economy that can sometimes be gained with logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time.

The ideas underlying partial-order planning include the detection of conflicts (Tate, 1975a) and the protection of achieved conditions from interference (Sussman, 1975). The construction of partially ordered plans (then called **task networks**) was pioneered by the NOAH planner (Sacerdoti, 1975, 1977) and by Tate's (1975b, 1977) NONLIN system.

Partial-order planning dominated the next 20 years of research, yet the first clear formal exposition was TWEAK (Chapman, 1987), a planner that was simple enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various planning problems. Chapman's work led to a straightforward description of a complete partial-order planner (McAllester and Rosenblitt, 1991), then to the widely distributed implementations SNLP (Soderland and Weld, 1991) and UCPOP (Penberthy and Weld, 1992). Partial-order planning fell out of favor in the late 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a reconsideration is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN in parallelizable domains and is competitive with the fastest state-space planners.

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest the ignore-delete-list heuristic. The name UNPOP was a reaction to the overwhelming concentration on partial-order planning at the time; McDermott suspected that other approaches were not getting the attention they deserved. Bonet and Geffner's Heuristic Search Planner (HSP) and its later derivatives (Bonet and Geffner, 1999; Haslum *et al.*, 2005; Haslum, 2006) were the first to make

state-space search practical for large planning problems. HSP searches in the forward direction while HSPR (Bonet and Geffner, 1999) searches backward. The most successful state-space searcher to date is FF (Hoffmann, 2001; Hoffmann and Nebel, 2001; Hoffmann, 2005), winner of the AIPS 2000 planning competition. FASTDOWNWARD (Helmert, 2006) is a forward state-space search planner that preprocesses the action schemas into an alternative representation which makes some of the constraints more explicit. FASTDOWNWARD (Helmert and Richter, 2004; Helmert, 2006) won the 2004 planning competition, and LAMA (Richter and Westphal, 2008), a planner based on FASTDOWNWARD with improved heuristics, won the 2008 competition.

Bylander (1994) and Ghallab *et al.* (2004) discuss the computational complexity of several variants of the planning problem. Helmert (2003) proves complexity bounds for many of the standard benchmark problems, and Hoffmann (2005) analyzes the search space of the ignore-delete-list heuristic. Heuristics for the set-covering problem are discussed by Caprara *et al.* (1995) for scheduling operations of the Italian railway. Edelkamp (2009) and Haslum *et al.* (2007) describe how to construct pattern databases for planning heuristics. As we mentioned in Chapter 3, Felner *et al.* (2004) show encouraging results using pattern databases for sliding blocks puzzles, which can be thought of as a planning domain, but Hoffmann *et al.* (2006) show some limitations of abstraction for classical planning problems.

Avrim Blum and Merrick Furst (1995, 1997) revitalized the field of planning with their GRAPHPLAN system, which was orders of magnitude faster than the partial-order planners of the time. Other graph-planning systems, such as IPP (Koehler *et al.*, 1997), STAN (Fox and Long, 1998), and SGP (Weld *et al.*, 1998), soon followed. A data structure closely resembling the planning graph had been developed slightly earlier by Ghallab and Laruelle (1994), whose IXTET partial-order planner used it to derive accurate heuristics to guide searches. Nguyen *et al.* (2001) thoroughly analyze heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes and articles by Subbarao Kambhampati (Bryce and Kambhampati, 2007). As mentioned in the chapter, a planning graph can be used in many different ways to guide the search for a solution. The winner of the 2002 AIPS planning competition, LPG (Gerevini and Serina, 2002, 2003), searched planning graphs using a local search technique inspired by WALKSAT.

The situation calculus approach to planning was introduced by John McCarthy (1963). The version we show here was proposed by Ray Reiter (1991, 2001).

Kautz *et al.* (1996) investigated various ways to propositionalize action schemas, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernst *et al.* (1997), who also developed an automatic “compiler” for generating propositional representations from PDDL problems. The BLACKBOX planner, which combines ideas from GRAPHPLAN and SATPLAN, was developed by Kautz and Selman (1998). CPLAN, a planner based on constraint satisfaction, was described by van Beek and Chen (1999).

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, compact data structures for Boolean expressions widely studied in the hardware verification community (Clarke and Grumberg, 1987; McMillan, 1993). There are techniques for proving properties of binary decision diagrams, including the property of being a solution

to a planning problem. Cimatti *et al.* (1998) present a planner based on this approach. Other representations have also been used; for example, Vossen *et al.* (2001) survey the use of integer programming for planning.

The jury is still out, but there are now some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches such as GRAPHPLAN and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. GRAPHPLAN and SATPLAN have trouble in domains with many objects because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

Readings in Planning (Allen *et al.*, 1990) is a comprehensive anthology of early work in the field. Weld (1994, 1999) provides two excellent surveys of planning algorithms of the 1990s. It is interesting to see the change in the five years between the two surveys: the first concentrates on partial-order planning, and the second introduces GRAPHPLAN and SATPLAN. *Automated Planning* (Ghallab *et al.*, 2004) is an excellent textbook on all aspects of planning. LaValle's text *Planning Algorithms* (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences. There are also specialized conferences such as the International Conference on AI Planning Systems, the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

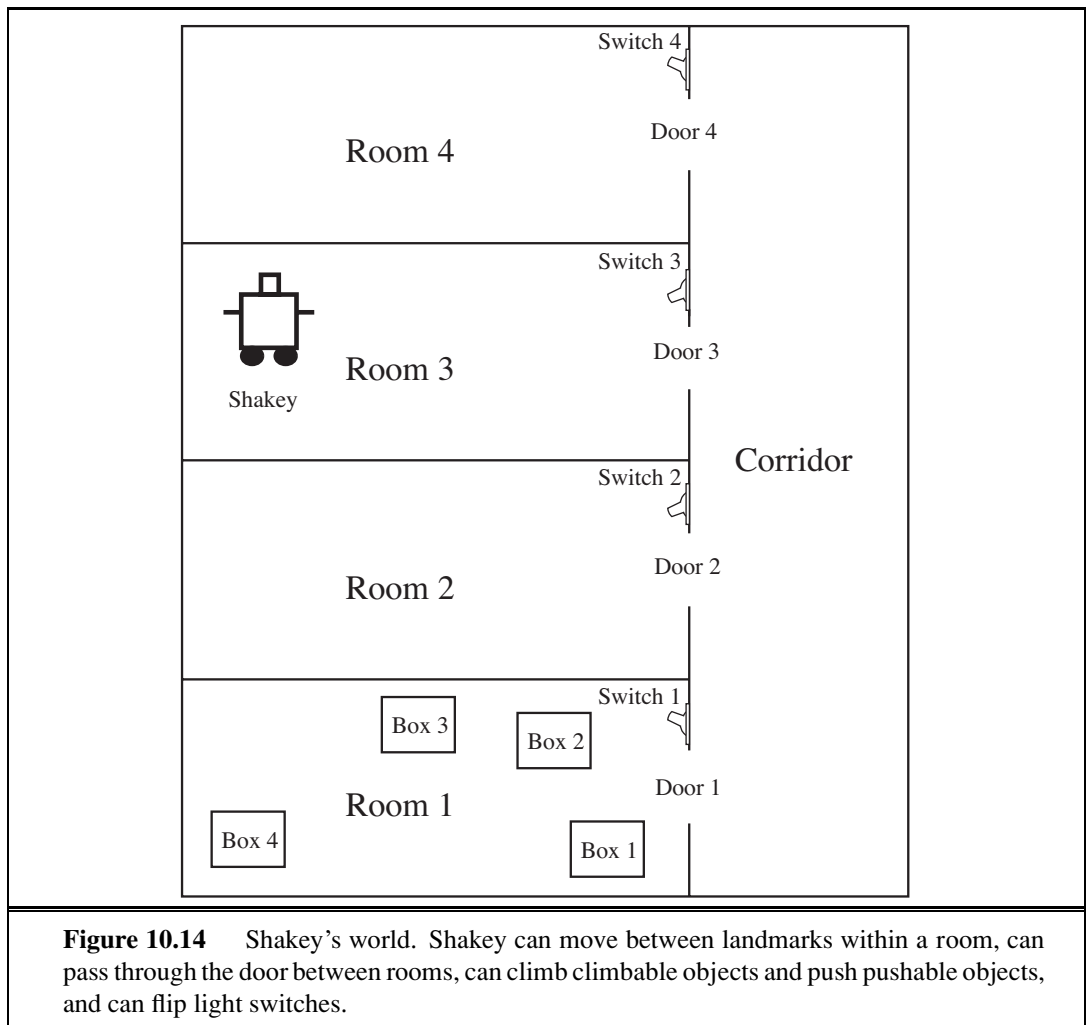
10.1 Describe the differences and similarities between problem solving and planning.

10.2 Given the action schemas and initial state from Figure 10.1, what are all the applicable concrete instances of $Fly(p, from, to)$ in the state described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) ?$$

10.3 The monkey-and-bananas problem is faced by a monkey in a laboratory with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it. Initially, the monkey is at A , the bananas at B , and the box at C . The monkey and box have height *Low*, but if the monkey climbs onto the box he will have height *High*, the same as the bananas. The actions available to the monkey include *Go* from one place to another, *Push* an object from one place to another, *ClimbUp* onto or *ClimbDown* from an object, and *Grasp* or *Ungrasp* an object. The result of a *Grasp* is that the monkey holds the object if the monkey and object are in the same place at the same height.

a. Write down the initial state description.



- b. Write the six action schemas.
- c. Suppose the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Write this as a general goal (i.e., not assuming that the box is necessarily at C) in the language of situation calculus. Can this goal be solved by a classical planning system?
- d. Your schema for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* schema is applied. Fix your action schema to account for heavy objects.

10.4 The original STRIPS planner was designed to control Shakey the robot. Figure 10.14 shows a version of Shakey's world consisting of four rooms lined up along a corridor, where each room has a door and a light switch. The actions in Shakey's world include moving from place to place, pushing movable objects (such as boxes), climbing onto and down from rigid

objects (such as boxes), and turning light switches on and off. The robot itself could not climb on a box or toggle a switch, but the planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y, r)$, which requires that Shakey be *At* x and that x and y are locations *In* the same room r . By convention a door between two rooms is in both of them.
- Push a box b from location x to location y within the same room: $Push(b, x, y, r)$. You will need the predicate *Box* and constants for the boxes.
- Climb onto a box from position x : $ClimbUp(x, b)$; climb down from a box to position x : $ClimbDown(b, x)$. We will need the predicate *On* and the constant *Floor*.
- Turn a light switch on or off: $TurnOn(s, b)$; $TurnOff(s, b)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.

Write PDDL sentences for Shakey's six actions and the initial state from Figure 10.14. Construct a plan for Shakey to get Box_2 into $Room_2$.

10.5 A finite Turing machine has a finite one-dimensional tape of cells, each cell containing one of a finite number of symbols. One cell has a read and write head above it. There is a finite set of states the machine can be in, one of which is the accept state. At each time step, depending on the symbol on the cell under the head and the machine's current state, there are a set of actions we can choose from. Each action involves writing a symbol to the cell under the head, transitioning the machine to a state, and optionally moving the head left or right. The mapping that determines which actions are allowed is the Turing machine's program. Your goal is to control the machine into the accept state.

Represent the Turing machine acceptance problem as a planning problem. If you can do this, it demonstrates that determining whether a planning problem has a solution is at least as hard as the Turing acceptance problem, which is PSPACE-hard.

10.6 Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem.

10.7 Figure 10.4 (page 371) shows a blocks-world problem that is known as the **Sussman anomaly**. The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Write a definition of the problem and solve it, either by hand or with a planning program. A noninterleaved planner is a planner that, when given two subgoals G_1 and G_2 , produces either a plan for G_1 concatenated with a plan for G_2 , or vice versa. Explain why a noninterleaved planner cannot solve this problem.

SUSSMAN ANOMALY

10.8 Prove that backward search with PDDL problems is complete.

10.9 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 10.1.

10.10 Prove the following assertions about planning graphs:

- a. A literal that does not appear in the final level of the graph cannot be achieved.

- b. The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

10.11 The set-level heuristic (see page 382) uses a planning graph to estimate the cost of achieving a conjunctive goal from the current state. What relaxed problem is the set-level heuristic the solution to?

10.12 Examine the definition of **bidirectional search** in Chapter 3.

- a. Would bidirectional state-space search be a good idea for planning?
- b. What about bidirectional search in the space of partial-order plans?
- c. Devise a version of partial-order planning in which an action can be added to a plan if its preconditions can be achieved by the effects of actions already in the plan. Explain how to deal with conflicts and ordering constraints. Is the algorithm essentially identical to forward state-space search?

10.13 We contrasted forward and backward state-space searchers with partial-order planners, saying that the latter is a plan-space searcher. Explain how forward and backward state-space search can also be considered plan-space searchers, and say what the plan refinement operators are.

10.14 Up to now we have assumed that the plans we create always make sure that an action's preconditions are satisfied. Let us now investigate what propositional successor-state axioms such as $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t)$ have to say about actions whose preconditions are not satisfied.

- a. Show that the axioms predict that nothing will happen when an action is executed in a state where its preconditions are not satisfied.
- b. Consider a plan p that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

$$initial\ state \wedge successor\text{-}state\ axioms \wedge p \models goal?$$

- c. With first-order successor-state axioms in situation calculus, is it possible to prove that a plan containing illegal actions will achieve the goal?

10.15 Consider how to translate a set of action schemas into the successor-state axioms of situation calculus.

- a. Consider the schema for $Fly(p, from, to)$. Write a logical definition for the predicate $Poss(Fly(p, from, to), s)$, which is true if the preconditions for $Fly(p, from, to)$ are satisfied in situation s .
- b. Next, assuming that $Fly(p, from, to)$ is the only action schema available to the agent, write down a successor-state axiom for $At(p, x, s)$ that captures the same information as the action schema.

- c. Now suppose there is an additional method of travel: $Teleport(p, from, to)$. It has the additional precondition $\neg Warped(p)$ and the additional effect $Warped(p)$. Explain how the situation calculus knowledge base must be modified.
- d. Finally, develop a general and precisely specified procedure for carrying out the translation from a set of action schemas to a set of successor-state axioms.

10.16 In the SATPLAN algorithm in Figure 7.22 (page 272), each call to the satisfiability algorithm asserts a goal g^T , where T ranges from 0 to T_{\max} . Suppose instead that the satisfiability algorithm is called only once, with the goal $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$.

- a. Will this always return a plan if one exists with length less than or equal to T_{\max} ?
- b. Does this approach introduce any new spurious “solutions”?
- c. Discuss how one might modify a satisfiability algorithm such as WALKSAT so that it finds short solutions (if they exist) when given a disjunctive goal of this form.

11 PLANNING AND ACTING IN THE REAL WORLD

In which we see how more expressive representations and more interactive agent architectures lead to planners that are useful in the real world.

The previous chapter introduced the most basic concepts, representations, and algorithms for planning. Planners that are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This chapter shows how. Section 11.1 extends the classical language for planning to talk about actions with durations and resource constraints. Section 11.2 describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Section 11.3 presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems. Section 11.4 shows how to plan when the environment contains other agents.

11.1 TIME, SCHEDULES, AND RESOURCES

The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is “plan first, schedule later”: that is, we divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

JOB

DURATION

CONSUMABLE

REUSABLE

MAKESPAN

$$\begin{aligned} &Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\}, \\ &\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\}) \\ &Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500)) \\ &Action(AddEngine1, DURATION:30, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddEngine2, DURATION:60, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddWheels1, DURATION:30, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(AddWheels2, DURATION:15, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(Inspect_i, DURATION:10, \\ &\quad USE:Inspectors(1)) \end{aligned}$$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B .

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

11.1.1 Representing temporal and resource constraints

A typical **job-shop scheduling problem**, as first introduced in Section 6.1.2, consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. Each constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be *produced* by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form $[AddEngine, AddWheels, Inspect]$. Then the

Resources statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

AGGREGATION

The representation of resources as numerical quantities, such as *Inspectors*(2), rather than as named entities, such as *Inspector*(I_1) and *Inspector*(I_2), is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent *Inspect* actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

11.1.2 Solving scheduling problems

CRITICAL PATH
METHOD

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*. (For example, there are two paths in the partial-order plan in Figure 11.2.)

CRITICAL PATH

The **critical path** is that path whose total duration is longest; the path is “critical” because it determines the duration of the entire plan—shortening other paths doesn’t shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, *ES*, and a latest possible start time, *LS*. The quantity $LS - ES$ is known as the **slack** of an action. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the *ES* and *LS* times for all the actions constitute a **schedule** for the problem.

SLACK

SCHEDULE

The following formulas serve as a definition for *ES* and *LS* and also as the outline of a dynamic-programming algorithm to compute them. *A* and *B* are actions, and $A \prec B$ means that *A* comes before *B*:

$$\begin{aligned} ES(\text{Start}) &= 0 \\ ES(B) &= \max_{A \prec B} ES(A) + \text{Duration}(A) \\ LS(\text{Finish}) &= ES(\text{Finish}) \\ LS(A) &= \min_{B \succ A} LS(B) - \text{Duration}(A) . \end{aligned}$$

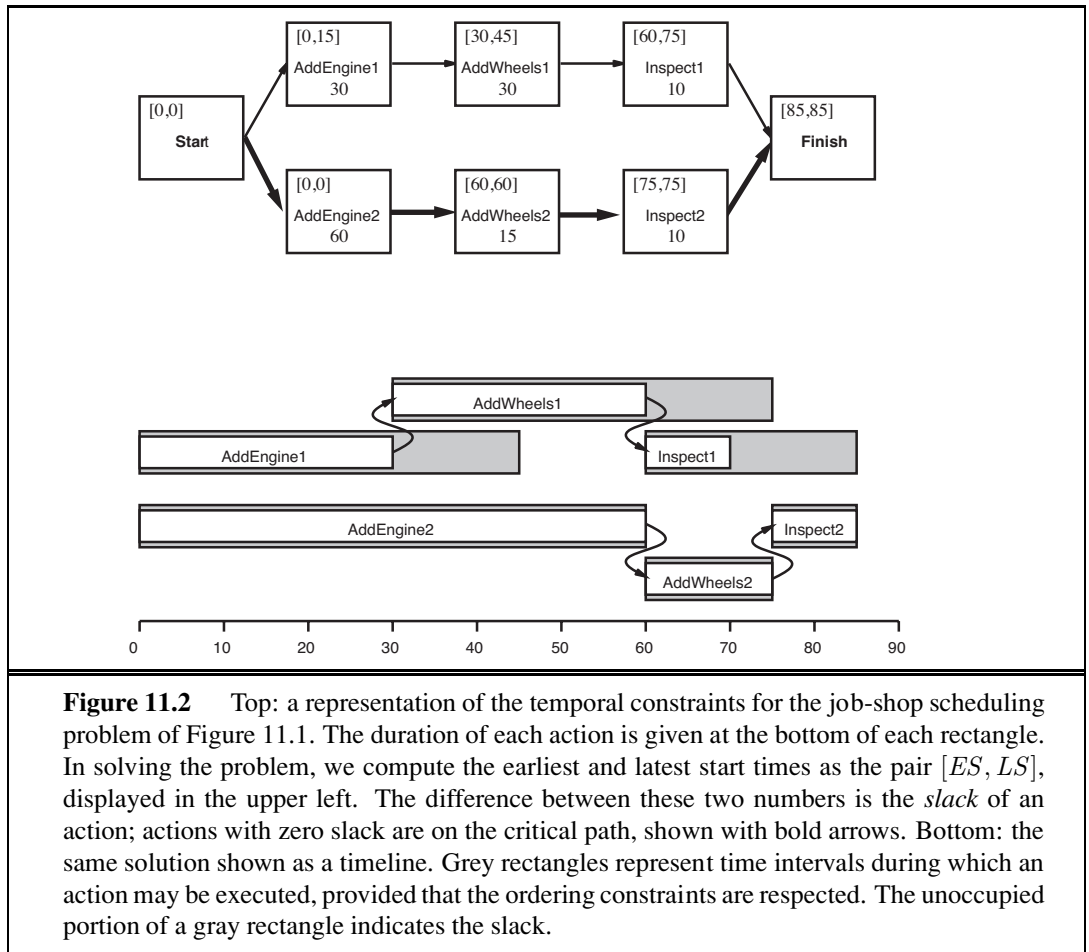


Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

The idea is that we start by assigning $ES(Start)$ to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the *Finish* action.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the *AddEngine* actions, which begin at the same time in Figure 11.2,

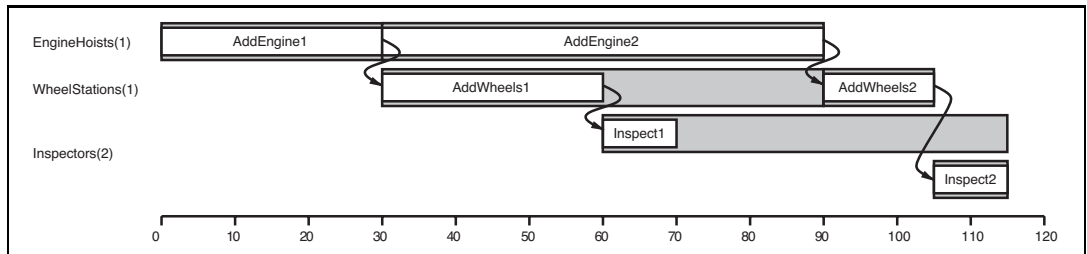


Figure 11.3 A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we’ve shown the shortest-duration solution, which takes 115 minutes.

require the same *EngineHoist* and so cannot overlap. The “cannot overlap” constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.*, 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Chapters 3 and 4. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the *ES* and *LS* times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130-minute solution, not the 115-minute solution of Figure 11.3.

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.

11.2 HIERARCHICAL PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

For plans executed by the human brain, atomic actions are muscle activations. In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all. Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude. Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be “Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.” Given such a plan, the action “Go to San Francisco airport” can be viewed as a planning task in itself, with a solution such as “Drive to the long-term parking lot; park; take the shuttle to the terminal.” Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. We defer discussion of this topic until Section 11.3. Here, we concentrate on the aspect of **hierarchical decomposition**, an idea that pervades almost all attempts to manage complexity. For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices. The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a *large* number of individual actions; for large-scale problems, this is completely impractical.

11.2.1 High-level actions

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. As in classical planning (Chapter 10), we assume full observability and determinism and the availability of a set of actions, now called **primitive actions**, with standard precondition–effect schemas. The key additional concept is the **high-level action** or HLA—for example, the action “Go to San Francisco airport” in the

HIERARCHICAL
DECOMPOSITION

HIERARCHICAL TASK
NETWORK

PRIMITIVE ACTION

HIGH-LEVEL ACTION

```

Refinement(Go(Home, SFO),
  STEPS: [Drive(Home, SFOLongTermParking),
         Shuttle(SFOLongTermParking, SFO)] )
Refinement(Go(Home, SFO),
  STEPS: [Taxi(Home, SFO)] )

Refinement(Navigate([a, b], [x, y]),
  PRECOND: a = x ∧ b = y
  STEPS: [] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a − 1, b])
  STEPS: [Left, Navigate([a − 1, b], [x, y])] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a + 1, b])
  STEPS: [Right, Navigate([a + 1, b], [x, y])] )
...

```

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

REFINEMENT

example given earlier. Each HLA has one or more possible **refinements**, into a sequence¹ of actions, each of which may be an HLA or a primitive action (which has no refinements by definition). For example, the action “Go to San Francisco airport,” represented formally as *Go*(*Home*, *SFO*), might have two possible refinements, as shown in Figure 11.4. The same figure shows a **recursive** refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

These examples show that high-level actions and their refinements embody knowledge about *how to do things*. For instance, the refinements for *Go*(*Home*, *SFO*) say that to get to the airport you can drive or take a taxi; buying milk, sitting down, and moving the knight to e4 are not to be considered.

IMPLEMENTATION

An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. For example, in the vacuum world, the sequences [*Right*, *Right*, *Down*] and [*Down*, *Right*, *Right*] both implement the HLA *Navigate*([1, 3], [3, 2]). An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal. We can say, then, that *a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state*. The “at least one” in this definition is crucial—not *all* implementations need to achieve the goal, because the agent gets



¹ HTN planners often allow refinement into partially ordered plans, and they allow the refinements of two different HLAs in a plan to *share* actions. We omit these important complications in the interest of understanding the basic concepts of hierarchical planning.

to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning. There, we required that a plan work for *all* outcomes because the agent doesn't get to choose the outcome; nature does.

The simplest case is an HLA that has exactly one implementation. In that case, we can compute the preconditions and effects of the HLA from those of the implementation (see Exercise 11.3) and then treat the HLA exactly as if it were a primitive action itself. It can be shown that the right collection of HLAs can result in the time complexity of blind search dropping from exponential in the solution depth to linear in the solution depth, although devising such a collection of HLAs may be a nontrivial task in itself. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works, as in Section 11.2.2; the other is to reason directly about the HLAs—despite the multiplicity of implementations—as explained in Section 11.2.3. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

11.2.2 Searching for primitive solutions

HTN planning is often formulated with a single “top level” action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action a_i , provide one refinement of *Act* with steps $[a_i, Act]$. That creates a recursive definition of *Act* that lets us add actions. But we need some way to stop the recursion; we do that by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal. One possible implementation based on breadth-first tree search is shown in Figure 11.5. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

In essence, this form of hierarchical search explores the space of sequences that conform to the knowledge contained in the HLA library about how things are to be done. A great deal of knowledge can be encoded, not just in the action sequences specified in each refinement but also in the preconditions for the refinements. For some domains, HTN planners have been able to generate huge plans with very little search. For example, O-PLAN (Bell and Tate, 1985), which combines HTN planning with scheduling, has been used to develop production plans for Hitachi. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner generates a 30-day schedule with three 8-hour shifts a day, involving tens of millions of steps. Another important aspect of HTN plans is that they are, by definition, hierarchically structured; usually this makes them easy for humans to understand.

```

function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier  $\leftarrow$  a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest plan in frontier */
    hla  $\leftarrow$  the first HLA in plan, or null if none
    prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
    outcome  $\leftarrow$  RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)

```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

The computational benefits of hierarchical search can be seen by examining an idealized case. Suppose that a planning problem has a solution with d primitive actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^d)$, as explained in Chapter 3. For an HTN planner, let us suppose a very regular refinement structure: each nonprimitive action has r possible refinements, each into k actions at the next lower level. We want to know how many different refinement trees there are with this structure. Now, if there are d actions at the primitive level, then the number of levels below the root is $\log_k d$, so the number of internal refinement nodes is $1 + k + k^2 + \dots + k^{\log_k d - 1} = (d - 1)/(k - 1)$. Each internal node has r possible refinements, so $r^{(d-1)/(k-1)}$ possible regular decomposition trees could be constructed. Examining this formula, we see that keeping r small and k large can result in huge savings: essentially we are taking the k th root of the nonhierarchical cost, if b and r are comparable. Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence (that nonetheless allows us to solve any problem). This is not always possible: long action sequences that are usable across a wide range of problems are extremely precious.

The key to HTN planning, then, is the construction of a plan library containing known methods for implementing complex, high-level actions. One method of constructing the library is to *learn* the methods from problem-solving experience. After the excruciating experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high-level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance (e.g., the name of

the builder or the address of the plot of land) and keeping just the key elements of the plan. Methods for achieving this kind of generalization are described in Chapter 19. It seems to us inconceivable that humans could be as competent as they are without some such mechanism.

11.2.3 Searching for abstract solutions

The hierarchical search algorithm in the preceding section refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan

$[Drive(Home, SFO_{LongTermParking}), Shuttle(SFO_{LongTermParking}, SFO)]$

gets one to the airport without having to determine a precise route, choice of parking spot, and so on. The solution seems obvious: write precondition–effect descriptions of the HLAs, just as we write down what the primitive actions do. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal. This is the holy grail, so to speak, of hierarchical planning because if we derive a high-level plan that provably achieves the goal, working in a small search space of high-level actions, then we can commit to that plan and work on the problem of refining each step of the plan. This gives us the exponential reduction we seek. For this to work, it has to be the case that every high-level plan that “claims” to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

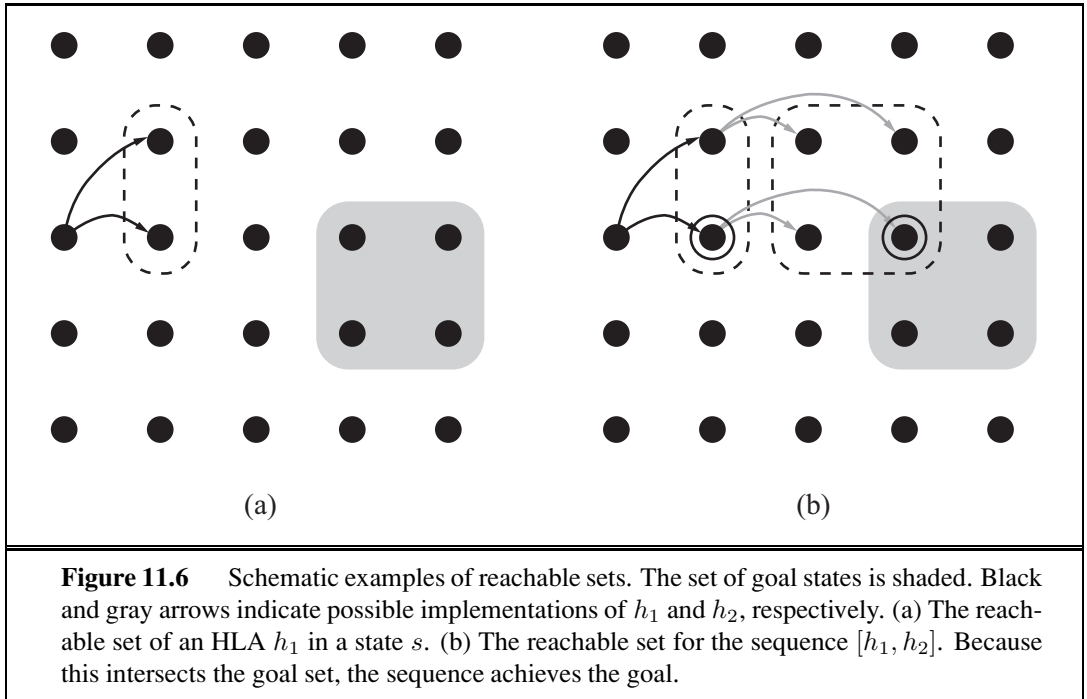
DOWNWARD
REFINEMENT
PROPERTY

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do. We have already seen how to write true descriptions for HLAs that have exactly one implementation (Exercise 11.3); a problem arises when the HLA has *multiple* implementations. How can we describe the effects of an action that can be implemented in many different ways?

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative. Consider again the HLA $Go(Home, SFO)$, which has two refinements, and suppose, for the sake of argument, a simple world in which one can always drive to the airport and park, but taking a taxi requires *Cash* as a precondition. In that case, $Go(Home, SFO)$ doesn’t always get you to the airport. In particular, it fails if *Cash* is false, and so we cannot assert $At(Agent, SFO)$ as an effect of the HLA. This makes no sense, however; if the agent didn’t have *Cash*, it would drive itself. Requiring that an effect hold for *every* implementation is equivalent to assuming that *someone else*—an adversary—will choose the implementation. It treats the HLA’s multiple outcomes exactly as if the HLA were a **nondeterministic** action, as in Section 4.3. For our case, the agent itself will choose the implementation.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nonde-**

DEMONIC
NONDETERMINISM



ANGELIC
NONDETERMINISM
ANGELIC SEMANTICS
REACHABLE SET

terminism, where the agent itself makes the choices. We borrow this term to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state s , the reachable set for an HLA h , written as $\text{REACH}(s, h)$, is the set of states reachable by any of the HLA's implementations. The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more “powerful” than the same HLA with fewer refinements. We can also define the reachable set of a sequences of HLAs. For example, the reachable set of a sequence $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1 :

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2).$$

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. (Compare this to the much stronger condition for demonic semantics, where every member of the reachable set has to be a goal state.) Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. Figure 11.6 illustrates these ideas.

The notion of reachable sets yields a straightforward algorithm: search among high-level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further. We will come back to the algorithmic issues later; first, we consider the question of how the effects of an HLA—the reachable set for each possible initial state—are represented. As with the classical action schemas of Chapter 10, we represent the *changes*

made to each fluent. Think of a fluent as a state variable. A primitive action can *add* or *delete* a variable or leave it *unchanged*. (With conditional effects (see Section 11.3.1) there is a fourth possibility: flipping a variable to its opposite.)

An HLA under angelic semantics can do more: it can *control* the value of a variable, setting it to true or false depending on which implementation is chosen. In fact, an HLA can have nine different effects on a variable: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the variable starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for each case can be combined arbitrarily, making nine. Notationally, this is a bit challenging. We'll use the \sim symbol to mean “possibly, if the agent so chooses.” Thus, an effect $\tilde{+}A$ means “possibly add A ,” that is, either leave A unchanged or make it true. Similarly, $\tilde{-}A$ means “possibly delete A ” and $\tilde{\pm}A$ means “possibly add or delete A .” For example, the HLA $Go(Home, SFO)$, with the two refinements shown in Figure 11.4, possibly deletes *Cash* (if the agent decides to take a taxi), so it should have the effect $\tilde{-}Cash$. Thus, we see that the descriptions of HLAs are *derivable*, in principle, from the descriptions of their refinements—in fact, this is required if we want true HLA descriptions, such that the downward refinement property holds. Now, suppose we have the following schemas for the HLAs h_1 and h_2 :

$$\begin{aligned} Action(h_1, PRECOND: \neg A, EFFECT: A \wedge \tilde{-}B) , \\ Action(h_2, PRECOND: \neg B, EFFECT: \tilde{+}A \wedge \tilde{\pm}C) . \end{aligned}$$

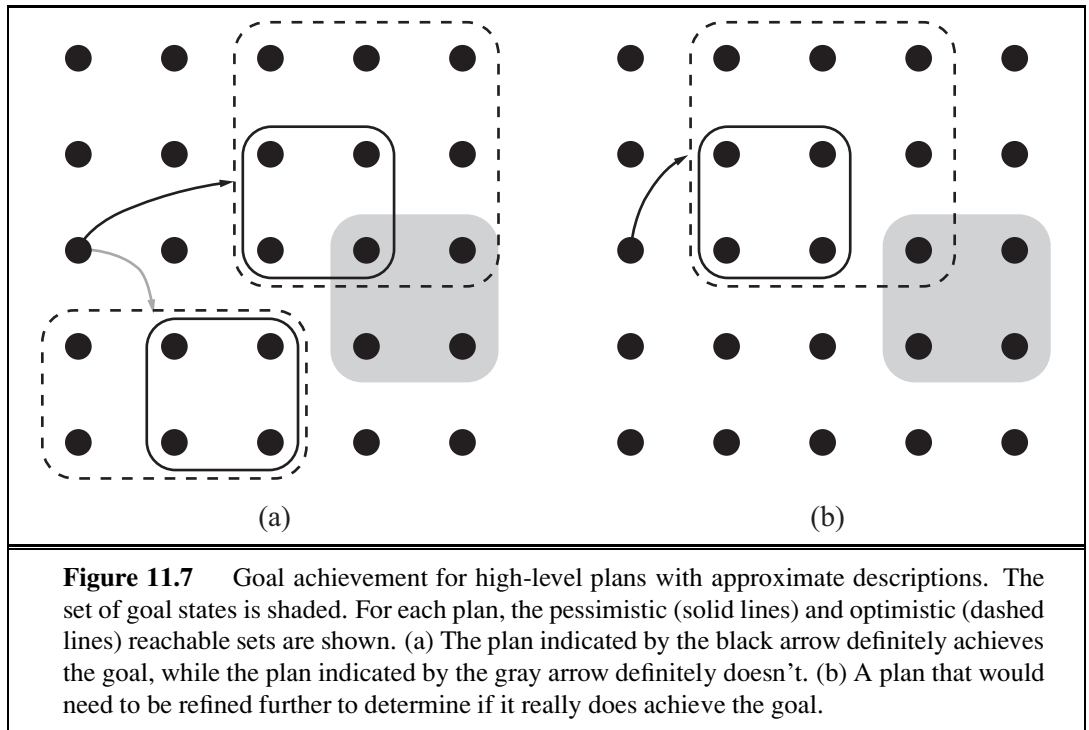
That is, h_1 adds A and possibly deletes B , while h_2 possibly adds A and has full control over C . Now, if only B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal: we choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

The preceding discussion assumes that the effects of an HLA—the reachable set for any given initial state—can be described exactly by describing the effect on each variable. It would be nice if this were always true, but in many cases we can only approximate the effects because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets—rather like the wiggly-belief-state problem illustrated in Figure 7.21 on page 271. For example, we said that $Go(Home, SFO)$ possibly deletes *Cash*; it also possibly adds $At(Car, SFOLongTermParking)$; but it cannot do both—in fact, it must do exactly one. As with belief states, we may need to write *approximate* descriptions. We will use two kinds of approximation: an **optimistic description** $REACH^+(s, h)$ of an HLA h may overstate the reachable set, while a **pessimistic description** $REACH^-(s, h)$ may understate the reachable set. Thus, we have

$$REACH^-(s, h) \subseteq REACH(s, h) \subseteq REACH^+(s, h) .$$

For example, an optimistic description of $Go(Home, SFO)$ says that it possibly deletes *Cash* and possibly adds $At(Car, SFOLongTermParking)$. Another good example arises in the 8-puzzle, half of whose states are unreachable from any given state (see Exercise 3.4 on page 113): the optimistic description of Act might well include the whole state space, since the exact reachable set is quite wiggly.

With approximate descriptions, the test for whether a plan achieves the goal needs to be modified slightly. If the optimistic reachable set for the plan doesn't intersect the goal,



then the plan doesn't work; if the pessimistic reachable set intersects the goal, then the plan does work (Figure 11.7(a)). With exact descriptions, a plan either works or it doesn't, but with approximate descriptions, there is a middle ground: if the optimistic set intersects the goal but the pessimistic set doesn't, then we cannot tell if the plan works (Figure 11.7(b)). When this circumstance arises, the uncertainty can be resolved by refining the plan. This is a very common situation in human reasoning. For example, in planning the aforementioned two-week Hawaii vacation, one might propose to spend two days on each of seven islands. Prudence would indicate that this ambitious plan needs to be refined by adding details of inter-island transportation.

An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure 11.8. For simplicity, we have kept to the same overall scheme used previously in Figure 11.5, that is, a breadth-first search in the space of refinements. As just explained, the algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. (The details of how to compute the reachable sets of a plan, given approximate descriptions of each step, are covered in Exercise 11.5.) When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. (See Section 10.2.2 for a discussion of how regression works.) Figure 11.6(b) illustrates the basic idea: the right-hand circled state is the guaranteed-reachable goal state, and the left-hand circled state is the intermediate goal obtained by regressing the

```

function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns solution or fail
  frontier  $\leftarrow$  a FIFO queue with initialPlan as the only element
  loop do
    if EMPTY?(frontier) then return fail
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    if REACH+(problem.INITIAL-STATE, plan) intersects problem.GOAL then
      if plan is primitive then return plan /* REACH+ is exact for primitive plans */
      guaranteed  $\leftarrow$  REACH-(problem.INITIAL-STATE, plan)  $\cap$  problem.GOAL
      if guaranteed  $\neq \{\}$  and MAKING-PROGRESS(plan, initialPlan) then
        finalState  $\leftarrow$  any element of guaranteed
        return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
      hla  $\leftarrow$  some HLA in plan
      prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
      for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)



---


function DECOMPOSE(hierarchy, s0, plan, sf) returns a solution
  solution  $\leftarrow$  an empty plan
  while plan is not empty do
    action  $\leftarrow$  REMOVE-LAST(plan)
    si  $\leftarrow$  a state in REACH-(s0, plan) such that sf  $\in$  REACH-(si, action)
    problem  $\leftarrow$  a problem with INITIAL-STATE = si and GOAL = sf
    solution  $\leftarrow$  APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf  $\leftarrow$  si
  return solution

```

Figure 11.8 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the *initialPlan*.

goal through the final action.

The ability to commit to or reject high-level plans can give ANGELIC-SEARCH a significant computational advantage over HIERARCHICAL-SEARCH, which in turn may have a large advantage over plain old BREADTH-FIRST-SEARCH. Consider, for example, cleaning up a large vacuum world consisting of rectangular rooms connected by narrow corridors. It makes sense to have an HLA for *Navigate* (as shown in Figure 11.4) and one for *CleanWholeRoom*. (Cleaning the room could be implemented with the repeated application of another HLA to clean each row.) Since there are five actions in this domain, the cost for BREADTH-FIRST-SEARCH grows as 5^d , where d is the length of the shortest solution (roughly twice the total number of squares); the algorithm cannot manage even two 2×2 rooms. HIERARCHICAL-SEARCH is more efficient, but still suffers from exponential growth because it tries all ways of cleaning that are consistent with the hierarchy. ANGELIC-SEARCH scales approximately linearly in the number of squares—it commits to a good high-level se-

quence and prunes away the other options. Notice that cleaning a set of rooms by cleaning each room in turn is hardly rocket science: it is easy for humans precisely because of the hierarchical structure of the task. When we consider how difficult humans find it to solve small puzzles such as the 8-puzzle, it seems likely that the human capacity for solving complex problems derives to a great extent from their skill in abstracting and decomposing the problem to eliminate combinatorics.

The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set. Instead of a state being reachable or not, it has a cost for the most efficient way to get there. (The cost is ∞ for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without considering their implementations. The same approach can be used to obtain effective **hierarchical lookahead** algorithms for online search, in the style of LRTA* (page 152). In some ways, such algorithms mirror aspects of human deliberation in tasks such as planning a vacation to Hawaii—consideration of alternatives is done initially at an abstract level over long time scales; some parts of the plan are left quite abstract until execution time, such as how to spend two lazy days on Molokai, while others parts are planned in detail, such as the flights to be taken and lodging to be reserved—without these refinements, there is no guarantee that the plan would be feasible.

HIERARCHICAL
LOOKAHEAD

11.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. Chapter 4 extended search in similar ways, and the methods here are also similar: **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments.

While the basic concepts are the same as in Chapter 4, there are also significant differences. These arise because planners deal with factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent **belief states**—the sets of possible physical states the agent might be in—for unobservable and partially observable environments. We can also take advantage of many of the domain-independent methods given in Chapter 10 for calculating search heuristics.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

$$\begin{aligned} &Init(Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2) \wedge InView(Table)) \\ &Goal(Color(Chair, c) \wedge Color(Table, c)) \end{aligned}$$

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can. The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable

list. That is, $Paint(x, can)$ does not mention the variable c , representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action $Paint(x, can, c)$. But in the partially observable case, we might or might not know what color is in the can. (The variable c is universally quantified, just like all the other variables in an action schema.)

$$\begin{aligned} &Action(RemoveLid(can), \\ &\quad PRECOND: Can(can) \\ &\quad EFFECT: Open(can)) \\ &Action(Paint(x, can), \\ &\quad PRECOND: Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can) \\ &\quad EFFECT: Color(x, c)) \end{aligned}$$

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, $PERCEPT(s)$. For planning, we augment PDDL with a new type of schema, the **percept schema**:

PERCEPT SCHEMA

$$\begin{aligned} &Percept(Color(x, c), \\ &\quad PRECOND: Object(x) \wedge InView(x)) \\ &Percept(Color(can, c), \\ &\quad PRECOND: Can(can) \wedge InView(can) \wedge Open(can)) \end{aligned}$$

The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object x , the agent will learn the truth value of $Color(x, c)$ for all c). The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$\begin{aligned} &Action(LookAt(x), \\ &\quad PRECOND: InView(y) \wedge (x \neq y) \\ &\quad EFFECT: InView(x) \wedge \neg InView(y)) \end{aligned}$$

For a fully observable environment, we would have a *Percept* axiom with no preconditions for each fluent. A sensorless agent, on the other hand, has no *Percept* axioms at all. Note that even a sensorless agent can solve the painting problem. One solution is to open any can of paint and apply it to both chair and table, thus **coercing** them to be the same color (even though the agent doesn't know what the color is).

A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done. If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal

with problems when they arise by replanning. It could also deal with incorrectness of its action schemas. Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes. On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents. In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening. Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning. We next look at each of the three approaches in more detail.

11.3.1 Sensorless planning

Section 4.4.1 (page 138) introduced the basic idea of searching in belief-state space to find a solution for sensorless problems. Conversion of a sensorless planning problem to a belief-state planning problem works much the same way as it did in Section 4.4.1; the main differences are that the underlying physical transition model is represented by a collection of action schemas and the belief state can be represented by a logical formula instead of an explicitly enumerated set of states. For simplicity, we assume that the underlying planning problem is deterministic.

The initial belief state for the sensorless painting problem can ignore *InView* fluents because the agent has no sensors. Furthermore, we take as given the unchanging facts $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$ because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: $\forall x \exists c Color(x, c)$. After Skolemizing, (see Section 9.5), we obtain the initial belief state:

$$b_0 = Color(x, C(x)) .$$

In classical planning, where the **closed-world assumption** is made, we would assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula. Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] .$$

We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

First, note that in a given belief state b , the agent can consider any action whose preconditions are satisfied by b . (The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.) According

to Equation (4.4) (page 139), the general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

where RESULT_P defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b' , we must consider what happens to each literal ℓ in each physical state s in b when action a is applied. For literals whose truth value is already known in b , the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b' .) What about a literal whose truth value is unknown in b ? There are three cases:

1. If the action adds ℓ , then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ , then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ , then ℓ will retain its initial value (which is unknown) and will not appear in b' .

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (10.1) on page 368:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a) .$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\text{RESULT}(b, a)$ is computed by starting with b , setting any atom that appears in $\text{DEL}(a)$ to false, and setting any atom that appears in $\text{ADD}(a)$ to true. For example, if we apply $\text{RemoveLid}(\text{Can}_1)$ to the initial belief state b_0 , we get

$$b_1 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) .$$

When we apply the action $\text{Paint}(\text{Chair}, \text{Can}_1)$, the precondition $\text{Color}(\text{Can}_1, c)$ is satisfied by the known literal $\text{Color}(x, C(x))$ with binding $\{x/\text{Can}_1, c/C(\text{Can}_1)\}$ and the new belief state is

$$b_2 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) .$$

Finally, we apply the action $\text{Paint}(\text{Table}, \text{Can}_1)$ to obtain

$$b_3 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) \\ \wedge \text{Color}(\text{Table}, C(\text{Can}_1)) .$$

The final belief state satisfies the goal, $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{Chair}, c)$, with the variable c bound to $C(\text{Can}_1)$.



The preceding analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas*. That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with n fluents, any belief state can be represented by a conjunction of size $O(n)$. This is a very comforting result, considering that there are 2^n states in the world. It says we can compactly represent all the subsets of those 2^n states that we will ever need. Moreover, the process of checking for belief

states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

The fly in the ointment of this pleasant picture is that it only works for action schemas that have the *same effects* for all states in which their preconditions are satisfied. It is this property that enables the preservation of the 1-CNF belief-state representation. As soon as the effect can depend on the state, dependencies are introduced between fluents and the 1-CNF property is lost. Consider, for example, the simple vacuum world defined in Section 3.2.1. Let the fluents be *AtL* and *AtR* for the location of the robot and *CleanL* and *CleanR* for the state of the squares. According to the definition of the problem, the *Suck* action has no precondition—it can always be done. The difficulty is that its effect depends on the robot’s location: when the robot is *AtL*, the result is *CleanL*, but when it is *AtR*, the result is *CleanR*. For such actions, our action schemas will need something new: a **conditional effect**. These have the syntax “**when** *condition*: *effect*,” where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state. For the vacuum world, we have

Action(*Suck*,
EFFECT:**when** *AtL*: *CleanL* \wedge **when** *AtR*: *CleanR*) .

When applied to the initial belief state *True*, the resulting belief state is $(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$, which is no longer in 1-CNF. (This transition can be seen in Figure 4.14 on page 141.) In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states of exponential size in the worst case.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action. For example, we could split *Suck* into two actions with unconditional effects as follows:

Action(*SuckL*,
PRECOND:*AtL*; EFFECT:*CleanL*)
Action(*SuckR*,
PRECOND:*AtR*; EFFECT:*CleanR*) .

Now we have only unconditional schemas, so the belief states all remain in 1-CNF; unfortunately, we cannot determine the applicability of *SuckL* and *SuckR* in the initial belief state.

It seems inevitable, then, that nontrivial problems will involve wiggly belief states, just like those encountered when we considered the problem of state estimation for the wumpus world (see Figure 7.21 on page 271). The solution suggested then was to use a **conservative approximation** to the exact belief state; for example, the belief state can remain in 1-CNF if it contains all literals whose truth values can be determined and treats all other literals as unknown. While this approach is *sound*, in that it never generates an incorrect plan, it is *incomplete* because it may be unable to find solutions to problems that necessarily involve interactions among literals. To give a trivial example, if the goal is for the robot to be on

a clean square, then $[Suck]$ is a solution but a sensorless agent that insists on 1-CNF belief states will not find it.

Perhaps a better solution is to look for action sequences that keep the belief state as simple as possible. For example, in the sensorless vacuum world, the action sequence $[Right, Suck, Left, Suck]$ generates the following sequence of belief states:

$$\begin{aligned} b_0 &= True \\ b_1 &= AtR \\ b_2 &= AtR \wedge CleanR \\ b_3 &= AtL \wedge CleanR \\ b_4 &= AtL \wedge CleanR \wedge CleanL \end{aligned}$$

That is, the agent *can* solve the problem while retaining a 1-CNF belief state, even though some sequences (e.g., those beginning with *Suck*) go outside 1-CNF. The general lesson is not lost on humans: we are always performing little actions (checking the time, patting our pockets to make sure we have the car keys, reading street signs as we navigate through a city) to eliminate uncertainty and keep our belief state manageable.

There is another, quite different approach to the problem of unmanageably wiggly belief states: don't bother computing them at all. Suppose the initial belief state is b_0 and we would like to know the belief state resulting from the action sequence $[a_1, \dots, a_m]$. Instead of computing it explicitly, just represent it as " b_0 then $[a_1, \dots, a_m]$." This is a lazy but unambiguous representation of the belief state, and it's quite concise— $O(n + m)$ where n is the size of the initial belief state (assumed to be in 1-CNF) and m is the maximum length of an action sequence. As a belief-state representation, it suffers from one drawback, however: determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

The computation can be implemented as an entailment test: if A_m represents the collection of successor-state axioms required to define occurrences of the actions a_1, \dots, a_m —as explained for SATPLAN in Section 10.4.1—and G_m asserts that the goal is true after m steps, then the plan achieves the goal if $b_0 \wedge A_m \models G_m$, that is, if $b_0 \wedge A_m \wedge \neg G_m$ is unsatisfiable. Given a modern SAT solver, it may be possible to do this much more quickly than computing the full belief state. For example, if none of the actions in the sequence has a particular goal fluent in its add list, the solver will detect this immediately. It also helps if partial results about the belief state—for example, fluents known to be true or false—are cached to simplify subsequent computations.

The final piece of the sensorless planning puzzle is a heuristic function to guide the search. The meaning of the heuristic function is the same as for classical planning: an estimate (perhaps admissible) of the cost of achieving the goal from the given belief state. With belief states, we have one additional fact: solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2) .$$

Hence, any admissible heuristic computed for a subset is admissible for the belief state itself. The most obvious candidates are the singleton subsets, that is, individual physical states. We

can take any random collection of states s_1, \dots, s_N that are in the belief state b , apply any admissible heuristic h from Chapter 10, and return

$$H(b) = \max\{h(s_1), \dots, h(s_N)\}$$

as the heuristic estimate for solving b . We could also use a planning graph directly on b itself: if it is a conjunction of literals (1-CNF), simply set those literals to be the initial state layer of the graph. If b is not in 1-CNF, it may be possible to find sets of literals that together entail b . For example, if b is in disjunctive normal form (DNF), each term of the DNF formula is a conjunction of literals that entails b and can form the initial layer of a planning graph. As before, we can take the maximum of the heuristics obtained from each set of literals. We can also use inadmissible heuristics such as the ignore-delete-lists heuristic (page 377), which seems to work quite well in practice.

11.3.2 Contingent planning

We saw in Chapter 4 that contingent planning—the generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, non-determinism, or both. For the partially observable painting problem with the percept axioms given earlier, one possible contingent solution is as follows:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1), RemoveLid(Can2), LookAt(Can2),
    if Color(Table, c) ∧ Color(can, c) then Paint(Chair, can)
    else if Color(Chair, c) ∧ Color(can, c) then Paint(Table, can)
    else [Paint(Chair, Can1), Paint(Table, Can1)]]]
```

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation. (It is up to the contingent-planning algorithm to make sure that the agent will never end up in a belief state where the condition formula's truth value is unknown.) Note that with first-order conditions, the formula may be satisfied in more than one way; for example, the condition $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{can}, c)$ might be satisfied by $\{\text{can}/\text{Can}_1\}$ and by $\{\text{can}/\text{Can}_2\}$ if both cans are the same color as the table. In that case, the agent can choose any satisfying substitution to apply to the rest of the plan.

As shown in Section 4.4.2, calculating the new belief state after an action and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:

$$\hat{b} = (b - \text{DEL}(a)) \cup \text{ADD}(a)$$

where, as before, we have assumed a belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals p_1, \dots, p_k are received. One might think that we simply need to add these into the belief state; in fact, we can also infer

that the preconditions for sensing are satisfied. Now, if a percept p has exactly one percept axiom, $Percept(p, PRECOND:c)$, where c is a conjunction of literals, then those literals can be thrown into the belief state along with p . On the other hand, if p has more than one percept axiom whose preconditions might hold according to the predicted belief state \hat{b} , then we have to add in the *disjunction* of the preconditions. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, we can generate contingent plans with an extension of the AND–OR forward search over belief states used in Section 4.4. Actions with nondeterministic effects—which are defined simply by using a disjunction in the EFFECT of the action schema—can be accommodated with minor changes to the belief-state update calculation and no change to the search algorithm.² For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

11.3.3 Online replanning

Imagine watching a spot-welding robot in a car plant. The robot’s fast, accurate motions are repeated over and over again as each car passes down the line. Although technically impressive, the robot probably does not seem at all *intelligent* because the motion is a fixed, preprogrammed sequence; the robot obviously doesn’t “know what it’s doing” in any meaningful sense. Now suppose that a poorly attached door falls off the car just as the robot is about to apply a spot-weld. The robot quickly replaces its welding actuator with a gripper, picks up the door, checks it for scratches, reattaches it to the car, sends an email to the floor supervisor, switches back to the welding actuator, and resumes its work. All of a sudden, the robot’s behavior seems *purposive* rather than rote; we assume it results not from a vast, precomputed contingent plan but from an online replanning process—which means that the robot *does* need to know what it’s trying to do.

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency, such as whether the sky might fall on its head.³ Some branches of a partially constructed contingent plan can simply say *Replan*; if such a branch is reached during execution, the agent reverts to planning mode. As we mentioned earlier, the decision as to how much of the problem to solve in advance and how much to leave to replanning is one that involves tradeoffs among possible events with different costs and probabilities of occurring. Nobody wants to have their car break down in the middle of the Sahara desert and only then think about having enough water.

EXECUTION
MONITORING

² If cyclic solutions are required for a nondeterministic problem, AND–OR search must be generalized to a loopy version such as LAO* (Hansen and Zilberstein, 2001).

³ In 1954, a Mrs. Hodges of Alabama was hit by meteorite that crashed through her roof. In 1992, a piece of the Mbale meteorite hit a small boy on the head; fortunately, its descent was slowed by banana leaves (Jenniskens *et al.*, 1994). And in 2009, a German boy claimed to have been hit in the hand by a pea-sized meteorite. No serious injuries resulted from any of these incidents, suggesting that the need for preplanning against such contingencies is sometimes overstated.

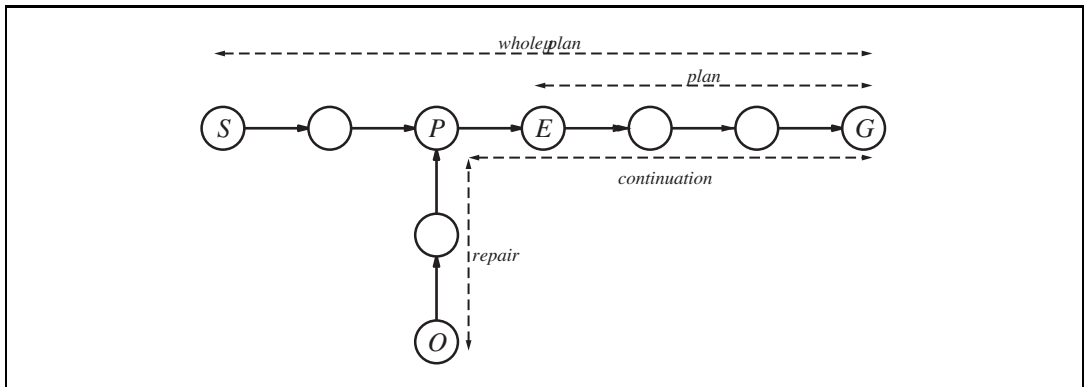


Figure 11.9 Before execution, the planner comes up with a plan, here called *whole plan*, to get from *S* to *G*. The agent executes steps of the plan until it expects to be in state *E*, but observes it is actually in *O*. The agent then replans for the minimal repair plus continuation to reach *G*.

MISSING
PRECONDITION

MISSING EFFECT
MISSING STATE
VARIABLE

EXOGENOUS EVENT

Replanning may also be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver; the model may have a **missing effect**—for example, painting an object may get paint on the floor as well; or the model may have a **missing state variable**—for example, the model given earlier has no notion of the amount of paint in a can, of how its actions affect this amount, or of the need for the amount to be nonzero. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be extremely fragile if it relies on absolute correctness of its model.

The online agent has a choice of how carefully to monitor the environment. We distinguish three levels:

ACTION MONITORING

- **Action monitoring:** before executing an action, the agent verifies that all the preconditions still hold.

PLAN MONITORING

- **Plan monitoring:** before executing an action, the agent verifies that the remaining plan will still succeed.

GOAL MONITORING

- **Goal monitoring:** before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

In Figure 11.9 we see a schematic of action monitoring. The agent keeps track of both its original plan, *wholeplan*, and the part of the plan that has not been executed yet, which is denoted by *plan*. After executing the first few steps of the plan, the agent expects to be in state *E*. But the agent observes it is actually in state *O*. It then needs to repair the plan by finding some point *P* on the original plan that it can get back to. (It may be that *P* is the goal state, *G*.) The agent tries to minimize the total cost of the plan: the repair part (from *O* to *P*) plus the continuation (from *P* to *G*).

Now let's return to the example problem of achieving a chair and table of matching color. Suppose the agent comes up with this plan:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1),
    if Color(Table, c) ∧ Color(Can1, c) then Paint(Chair, Can1)
    else REPLAN]] .
```

Now the agent is ready to execute the plan. Suppose the agent observes that the table and can of paint are white and the chair is black. It then executes *Paint(Chair, Can₁)*. At this point a classical planner would declare victory; the plan has been executed. But an online execution monitoring agent needs to check the preconditions of the remaining empty plan—that the table and chair are the same color. Suppose the agent perceives that they do not have the same color—in fact, the chair is now a mottled gray because the black paint is showing through. The agent then needs to figure out a position in *whole plan* to aim for and a repair action sequence to get there. The agent notices that the current state is identical to the precondition before the *Paint(Chair, Can₁)* action, so the agent chooses the empty sequence for *repair* and makes its *plan* be the same [*Paint*] sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the *Paint* action is retried. This behavior will loop until the chair is perceived to be completely painted. But notice that the loop is created by a process of plan–execute–replan, rather than by an explicit loop in a plan. Note also that the original plan need not cover every contingency. If the agent reaches the step marked REPLAN, it can then generate a new plan (perhaps involving *Can₂*).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red. Plan monitoring achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs.⁴ Plan monitoring also allows for **serendipity**—accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly

⁴ Plan monitoring means that finally, after 424 pages, we have an agent that is smarter than a dung beetle (see page 39). A plan-monitoring agent would notice that the dung ball was missing from its grasp and would replan to get another ball and plug its hole.

more complex to enable plan monitoring. Partial-order and planning-graph planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

Now that we have described a method for monitoring and replanning, we need to ask, “Does it work?” This is a surprisingly tricky question. If we mean, “Can we guarantee that the agent will always achieve the goal?” then the answer is no, because the agent could inadvertently arrive at a dead end from which there is no repair. For example, the vacuum agent might have a faulty model of itself and not know that its batteries can run out. Once they do, it cannot repair any plans. If we rule out dead ends—assume that there exists a plan to reach the goal from *any* state in the environment—and assume that the environment is really nondeterministic, in the sense that such a plan always has *some* chance of success on any given execution attempt, then the agent will eventually reach the goal.

Trouble occurs when an action is actually not nondeterministic, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this.⁵ One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair. This kind of learning is described in Chapters 18 and 19.

11.4 MULTIAGENT PLANNING

So far, we have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a **multiagent planning problem** in which it tries to achieve its own goals with the help or hindrance of others.

Between the purely single-agent and truly multiagent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do **multieffector planning** to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory—multieffector planning becomes **multibody planning**. A multibody problem is still a “standard” single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body. When communication constraints make this impossible, we have

MULTIAGENT
PLANNING PROBLEM

MULTIEFFECTOR
PLANNING

MULTIBODY
PLANNING

⁵ Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp (page 39).

DECENTRALIZED
PLANNING

what is sometimes called a **decentralized planning** problem; this is perhaps a misnomer, because the planning phase is centralized but the execution phase is at least partially decoupled. In this case, the subplan constructed for each body may need to include explicit communicative actions with other bodies. For example, multiple reconnaissance robots covering a wide area may often be out of radio contact with each other and should share their findings during times when communication is feasible.

COORDINATION

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multiagent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multiagent doubles team, on the other hand, each agent decides what to do; without some method for **coordination**, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

The clearest case of a multiagent problem, of course, is when the agents have different goals. In tennis, the goals of two opposing teams are in direct conflict, leading to the zero-sum situation of Chapter 5. Spectators could be viewed as agents if their support or disdain is a significant factor and can be influenced by the players' conduct; otherwise, they can be treated as an aspect of nature—just like the weather—that is assumed to be indifferent to the players' intentions.⁶

INCENTIVE

Finally, some systems are a mixture of centralized and multiagent planning. For example, a delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.

The issues involved in multiagent planning can be divided roughly into two sets. The first, covered in Section 11.4.1, involves issues of representing and planning for multiple simultaneous actions; these issues occur in all settings from multieffector to multiagent planning. The second, covered in Section 11.4.2, involves issues of cooperation, coordination, and competition arising in true multiagent settings.

11.4.1 Planning with multiple simultaneous actions

MULTIACTOR
ACTOR

For the time being, we will treat the multieffector, multibody, and multiagent settings in the same way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting. A correct plan is one that, if executed by the actors, achieves the goal. (In the true multiagent setting, of course, the agents may not agree to execute any particular plan, but at least they

⁶ We apologize to residents of the United Kingdom, where the mere act of contemplating a game of tennis guarantees rain.

```

Actors( $A, B$ )
Init( $At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$ 
     $Approaching(Ball, RightBaseline) \wedge Partner(A, B) \wedge Partner(B, A)$ 
     $Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet)))$ 
    Action( $Hit(actor, Ball)$ ,
        PRECOND: $Approaching(Ball, loc) \wedge At(actor, loc)$ 
        EFFECT: $Returned(Ball)$ )
    Action( $Go(actor, to)$ ,
        PRECOND: $At(actor, loc) \wedge to \neq loc$ ,
        EFFECT: $At(actor, to) \wedge \neg At(actor, loc)$ )

```

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: *LeftBaseline*, *RightBaseline*, *LeftNet*, and *RightNet*. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

will know what plans *would* work if they *did* agree to execute them.) For simplicity, we assume perfect **synchronization**: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

We begin with the transition model; for the deterministic case, this is the function $RESULT(s, a)$. In the single-agent setting, there might be b different choices for the action; b can be quite large, especially for first-order representations with many objects to act on, but action schemas provide a concise representation nonetheless. In the multiactor setting with n actors, the single action a is replaced by a **joint action** $\langle a_1, \dots, a_n \rangle$, where a_i is the action taken by the i th actor. Immediately, we see two problems: first, we have to describe the transition model for b^n different joint actions; second, we have a joint planning problem with a branching factor of b^n .

Having put the actors together into a multiactor system with a huge branching factor, the principal focus of research on multiactor planning has been to *decouple* the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are **loosely coupled**, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI. We have seen it explicitly in the context of CSPs, where “tree like” constraint graphs yielded efficient solution methods (see page 225), as well as in the context of disjoint pattern databases (page 106) and additive heuristics for planning (page 378).

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let’s see how this works for the doubles tennis problem. Let’s suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

A first pass at a multiactor definition might look like Figure 11.10. With this definition, it is easy to see that the following **joint plan** works:

JOINT PLAN

PLAN 1:

$$\begin{aligned} A : & [Go(A, RightBaseline), Hit(A, Ball)] \\ B : & [NoOp(B), NoOp(B)] . \end{aligned}$$

Problems arise, however, when a plan has both agents hitting the ball at the same time. In the real world, this won't work, but the action schema for *Hit* says that the ball will be returned successfully. Technically, the difficulty is that preconditions constrain the *state* in which an action can be executed successfully, but do not constrain other actions that might mess it up.

CONCURRENT ACTION LIST

We solve this by augmenting action schemas with one new feature: a **concurrent action list** stating which actions must or must not be executed concurrently. For example, the *Hit* action could be described as follows:

$$\begin{aligned} & Action(Hit(a, Ball), \\ & \quad CONCURRENT: b \neq a \Rightarrow \neg Hit(b, Ball) \\ & \quad PRECOND: Approaching(Ball, loc) \wedge At(a, loc) \\ & \quad EFFECT: Returned(Ball)) . \end{aligned}$$

In other words, the *Hit* action has its stated effect only if no other *Hit* action by another agent occurs at the same time. (In the SATPLAN approach, this would be handled by a partial **action exclusion axiom**.) For some actions, the desired effect is achieved *only* when another action occurs concurrently. For example, two agents are needed to carry a cooler full of beverages to the tennis court:

$$\begin{aligned} & Action(Carry(a, cooler, here, there), \\ & \quad CONCURRENT: b \neq a \wedge Carry(b, cooler, here, there) \\ & \quad PRECOND: At(a, here) \wedge At(cooler, here) \wedge Cooler(cooler) \\ & \quad EFFECT: At(a, there) \wedge At(cooler, there) \wedge \neg At(a, here) \wedge \neg At(cooler, here)) . \end{aligned}$$

With these kinds of action schemas, any of the planning algorithms described in Chapter 10 can be adapted with only minor modifications to generate multiactor plans. To the extent that the coupling among subplans is loose—meaning that concurrency constraints come into play only rarely during plan search—one would expect the various heuristics derived for single-agent planning to also be effective in the multiactor context. We could extend this approach with the refinements of the last two chapters—HTNs, partial observability, conditionals, execution monitoring, and replanning—but that is beyond the scope of this book.

11.4.2 Planning with multiple agents: Cooperation and coordination

Now let us consider the true multiagent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the “*the*” in “*the joint solution*” is misleading. For our doubles team, more than one joint solution exists:

PLAN 2:

$$\begin{aligned} A : & [Go(A, LeftNet), NoOp(A)] \\ B : & [Go(B, RightBaseline), Hit(B, Ball)] . \end{aligned}$$

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if *A* chooses plan 2 and *B* chooses plan 1, then nobody will return the ball. Conversely, if *A* chooses 1 and *B* chooses 2, then they will both try to hit the ball. The agents may realize this, but how can they coordinate to make sure they agree on the plan?

CONVENTION

One option is to adopt a **convention** before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention “stick to your side of the court” would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention “stay on the right side of the road” in most countries; the alternative, “stay on the left side,” works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called **social laws**.

SOCIAL LAWS

In the absence of a convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a tennis player could shout “Mine!” or “Yours!” to indicate a preferred joint plan. We cover mechanisms for communication in more depth in Chapter 22, where we observe that communication does not necessarily involve a verbal exchange. For example, one player can communicate a preferred joint plan to the other simply by executing the first part of it. If agent *A* heads for the net, then agent *B* is obliged to go back to the baseline to hit the ball, because plan 2 is the only joint plan that begins with *A*’s heading for the net. This approach to coordination, sometimes called **plan recognition**, works when a single action (or short sequence of actions) is enough to determine a joint plan unambiguously. Note that communication can work as well with competitive agents as with cooperative ones.

PLAN RECOGNITION

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen’s job is to reproduce, not to do centralized planning—and with very limited computation, communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role according to the local conditions it observes. For example, foragers travel away from the nest, search for a seed, and when they find one, bring it back immediately. Thus, the rate at which foragers return to the nest is an approximation of the availability of food today. When the rate is high, other ants abandon their current role and take on the role of scavenger. The ants appear to have a convention on the importance of roles—foraging is the most important—and ants will easily switch into the more important roles, but not into the less important. There is some learning mechanism: a colony learns to make more successful and prudent actions over the course of its decades-long life, even though individual ants live only about a year.

One final example of cooperative multiagent behavior appears in the flocking behavior of birds. We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a **boid**) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components:

BOID

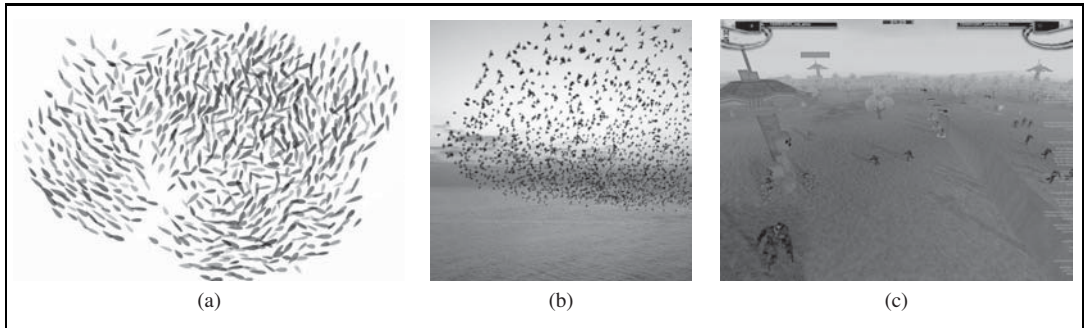


Figure 11.11 (a) A simulated flock of birds, using Reynolds's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors

EMERGENT
BEHAVIOR

If all the boids execute this policy, the flock exhibits the **emergent behavior** of flying as a pseudorigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents.

The most difficult multiagent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control. We see this in games such as robotic soccer or the NERO game shown in Figure 11.11(c), in which two teams of software agents compete to capture the control towers. As yet, methods for efficient planning in these kinds of environments—for example, taking advantage of loose coupling—are in their infancy.

11.5 SUMMARY

This chapter has addressed some of the complications of planning and acting in the real world. The main points:

- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these resources as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Actions can generate and consume resources, and it is usually cheap and effective to check partial plans for satisfaction of resource constraints before attempting further refinements.
- Time is one of the most important resources. It can be handled by specialized scheduling algorithms, or scheduling can be integrated with planning.

- **Hierarchical task network** (HTN) planning allows the agent to take advice from the domain designer in the form of **high-level actions** (HLAs) that can be implemented in various ways by lower-level action sequences. The effects of HLAs can be defined with **angelic semantics**, allowing provably correct high-level plans to be derived without consideration of lower-level implementations. HTN methods can create the very large plans required by many real-world applications.
- Standard planning algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption.
- **Contingent plans** allow the agent to sense the world during execution to decide what branch of the plan to follow. In some cases, **sensorless** or **conformant planning** can be used to construct a plan that works without the need for perception. Both conformant and contingent plans can be constructed by search in the space of **belief states**. Efficient representation or computation of belief states is a key problem.
- An **online planning agent** uses execution monitoring and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.
- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate or compete. Joint plans can be constructed, but must be augmented with some form of coordination if two agents are to agree on which joint plan to execute.
- This chapter extends classic planning to cover nondeterministic environments (where outcomes of actions are uncertain), but it is not the last word on planning. Chapter 17 describes techniques for stochastic environments (in which outcomes of actions have probabilities associated with them): Markov decision processes, partially observable Markov decision processes, and game theory. In Chapter 21 we show that reinforcement learning allows an agent to learn how to behave from past successes and failures.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with time constraints was first dealt with by DEVISER (Vere, 1983). The representation of time in plans was addressed by Allen (1984) and by Dean *et al.* (1990) in the FORBIN system. NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1988, 1990) could reason about the allocation of limited resources to various plan steps. O-PLAN (Bell and Tate, 1985), an HTN planner, had a uniform, general representation for constraints on time and resources. In addition to the Hitachi application mentioned in the text, O-PLAN has been applied to software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars.

The two planners SAPA (Do and Kambhampati, 2001) and T4 (Haslum and Geffner, 2001) both used forward state-space search with sophisticated heuristics to handle actions with durations and resources. An alternative is to use very expressive action languages, but guide them by human-written domain-specific heuristics, as is done by ASPEN (Fukunaga *et al.*, 1997), HSTS (Jonsson *et al.*, 2000), and IxTeT (Ghallab and Laruelle, 1994).

A number of hybrid planning-and-scheduling systems have been deployed: ISIS (Fox *et al.*, 1982; Fox, 1990) has been used for job shop scheduling at Westinghouse, GARI (Desctotte and Latombe, 1985) planned the machining and construction of mechanical parts, FORBIN was used for factory control, and NONLIN+ was used for naval logistics planning. We chose to present planning and scheduling as two separate problems; (Cushing *et al.*, 2007) show that this can lead to incompleteness on certain problems. There is a long history of scheduling in aerospace. T-SCHED (Drabble, 1990) was used to schedule mission-command sequences for the UOSAT-II satellite. OPTIMUM-AIV (Aarup *et al.*, 1994) and PLAN-ERS1 (Fuchs *et al.*, 1990), both based on O-PLAN, were used for spacecraft assembly and observation planning, respectively, at the European Space Agency. SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble Space Telescope, while the Space Shuttle Ground Processing Scheduling System (Deale *et al.*, 1994) does job-shop scheduling of up to 16,000 worker-shifts. Remote Agent (Muscettola *et al.*, 1998) became the first autonomous planner-scheduler to control a spacecraft when it flew onboard the Deep Space One probe in 1999. Space applications have driven the development of algorithms for resource allocations; see Laborie (2003) and Muscettola (2002). The literature on scheduling is presented in a classic survey article (Lawler *et al.*, 1993), a recent book (Pinedo, 2008), and an edited handbook (Blazewicz *et al.*, 2007).

MACROPS

ABSTRACTION
HIERARCHY

The facility in the STRIPS program for learning **macrops**—“macro-operators” consisting of a sequence of primitive steps—could be considered the first mechanism for hierarchical planning (Fikes *et al.*, 1972). Hierarchy was also used in the LAWALY system (Siklossy and Dreussi, 1973). The ABSTRIPS system (Sacerdoti, 1974) introduced the idea of an **abstraction hierarchy**, whereby planning at higher levels was permitted to ignore lower-level preconditions of actions in order to derive the general structure of a working plan. Austin Tate’s Ph.D. thesis (1975b) and work by Earl Sacerdoti (1977) developed the basic ideas of HTN planning in its modern form. Many practical planners, including O-PLAN and SIPE, are HTN planners. Yang (1990) discusses properties of actions that make HTN planning efficient. Erol, Hendler, and Nau (1994, 1996) present a complete hierarchical decomposition planner as well as a range of complexity results for pure HTN planners. Our presentation of HLAs and angelic semantics is due to Marthi *et al.* (2007, 2008). Kambhampati *et al.* (1998) have proposed an approach in which decompositions are just another form of plan refinement, similar to the refinements for non-hierarchical partial-order planning.

CASE-BASED
PLANNING

Beginning with the work on macro-operators in STRIPS, one of the goals of hierarchical planning has been the reuse of previous planning experience in the form of generalized plans. The technique of **explanation-based learning**, described in depth in Chapter 19, has been applied in several systems as a means of generalizing previously computed plans, including SOAR (Laird *et al.*, 1986) and PRODIGY (Carbonell *et al.*, 1989). An alternative approach is to store previously computed plans in their original form and then reuse them to solve new, similar problems by analogy to the original problem. This is the approach taken by the field called **case-based planning** (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) argues that case-based planning should be analyzed as a form of refinement planning and provides a formal foundation for case-based partial-order planning.

Early planners lacked conditionals and loops, but some could use coercion to form conformant plans. Sacerdoti's NOAH solved the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state, using coercion. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and described a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions, *regardless* of the initial position.

Goldman and Boddy (1996) introduced the term **conformant planning**, noting that sensorless plans are often effective even if the agent has sensors. The first moderately efficient conformant planner was Smith and Weld's (1998) Conformant Graphplan or CGP. Ferraris and Giunchiglia (2000) and Rintanen (1999) independently developed SATPLAN-based conformant planners. Bonet and Geffner (2000) describe a conformant planner based on heuristic search in the space of belief states, drawing on ideas first developed in the 1960s for partially observable Markov decision processes, or POMDPs (see Chapter 17).

Currently, there are three main approaches to conformant planning. The first two use heuristic search in belief-state space: HSCP (Bertoli *et al.*, 2001a) uses binary decision diagrams (BDDs) to represent belief states, whereas Hoffmann and Brafman (2006) adopt the lazy approach of computing precondition and goal tests on demand using a SAT solver. The third approach, championed primarily by Jussi Rintanen (2007), formulates the entire sensorless planning problem as a quantified Boolean formula (QBF) and solves it using a general-purpose QBF solver. Current conformant planners are five orders of magnitude faster than CGP. The winner of the 2006 conformant-planning track at the International Planning Competition was T_0 (Palacios and Geffner, 2007), which uses heuristic search in belief-state space while keeping the belief-state representation simple by defining derived literals that cover conditional effects. Bryce and Kambhampati (2007) discuss how a planning graph can be generalized to generate good heuristics for conformant and contingent planning.

There has been some confusion in the literature between the terms "conditional" and "contingent" planning. Following Majercik and Littman (2003), we use "conditional" to mean a plan (or action) that has different effects depending on the actual state of the world, and "contingent" to mean a plan in which the agent can choose different actions depending on the results of sensing. The problem of contingent planning received more attention after the publication of Drew McDermott's (1978a) influential article, *Planning and Acting*.

The contingent-planning approach described in the chapter is based on Hoffmann and Brafman (2005), and was influenced by the efficient search algorithms for cyclic AND-OR graphs developed by Jimenez and Torras (2000) and Hansen and Zilberstein (2001). Bertoli *et al.* (2001b) describe MBP (Model-Based Planner), which uses binary decision diagrams to do conformant and contingent planning.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for uncertain domains. Fast-forward heuristic search through state space led to forward search in belief space (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 2003) and to planning with quantified Boolean logic (Rintanen, 2007); partial order planning led to UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992); GRAPHPLAN led to Sensory Graphplan or SGP (Weld *et al.*, 1998).

The first online planner with execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. The NASL planner (McDermott, 1978a) treated a planning problem simply as a specification for carrying out a complex action, so that execution and planning were completely unified. SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988, 1990) was the first planner to deal systematically with the problem of replanning. It has been used in demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier, job-shop scheduling for an Australian beer factory, and planning the construction of multistory buildings (Kartam and Levitt, 1990).

REACTIVE PLANNING

In the mid-1980s, pessimism about the slow run times of planning systems led to the proposal of reflex agents called **reactive planning** systems (Brooks, 1986; Agre and Chapman, 1987). PENG1 (Agre and Chapman, 1987) could play a (fully observable) video game by using Boolean circuits combined with a “visual” representation of current goals and the agent’s internal state. “Universal plans” (Schoppers, 1987, 1989) were developed as a lookup-table method for reactive planning, but turned out to be a rediscovery of the idea of **policies** that had long been used in Markov decision processes (see Chapter 17). A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Koenig (2001) surveys online planning techniques, under the name *Agent-Centered Search*.

POLICY

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) formalizes multiagent planning in first-order logic, while Pednault (1986) gives a STRIPS-style description. The notion of joint intention, which is essential if agents are to execute a joint plan, comes from work on communicative acts (Cohen and Levesque, 1990; Cohen *et al.*, 1990). Boutilier and Brafman (2001) show how to adapt partial-order planning to a multiactor setting. Brafman and Domshlak (2008) devise a multiactor planning algorithm whose complexity grows only linearly with the number of actors, provided that the degree of coupling (measured partly by the **tree width** of the graph of interactions among agents) is bounded. Petrik and Zilberstein (2009) show that an approach based on bilinear programming outperforms the cover-set approach we outlined in the chapter.

We have barely skimmed the surface of work on negotiation in multiagent planning. Durfee and Lesser (1989) discuss how tasks can be shared out among agents by negotiation. Kraus *et al.* (1991) describe a system for playing Diplomacy, a board game requiring negotiation, coalition formation, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. In a later article, Stone (2003) analyzes two competitive multiagent environments—RoboCup, a robotic soccer competition, and TAC, the auction-based Trading Agents Competition—and finds that the computational intractability of our current theoretically well-founded approaches has led to many multiagent systems being designed by *ad hoc* methods.

In his highly influential *Society of Mind* theory, Marvin Minsky (1986, 2007) proposes that human minds are constructed from an ensemble of agents. Livnat and Pippenger (2006) prove that, for the problem of optimal path-finding, and given a limitation on the total amount of computing resources, the best architecture for an agent is an ensemble of subagents, each of which tries to optimize its own objective, and all of which are in conflict with one another.

The boid model on page 429 is due to Reynolds (1987), who won an Academy Award for its application to swarms of penguins in *Batman Returns*. The NERO game and the methods for learning strategies are described by Bryant and Miikkulainen (2007).

Recent book on multiagent systems include those by Weiss (2000a), Young (2004), Vlassis (2008), and Shoham and Leyton-Brown (2009). There is an annual conference on autonomous agents and multiagent systems (AAMAS).

EXERCISES

11.1 The goals we have considered so far all ask the planner to make the world satisfy the goal at just one time step. Not all goals can be expressed this way: you do not achieve the goal of suspending a chandelier above the ground by throwing it in the air. More seriously, you wouldn't want your spacecraft life-support system to supply oxygen one day but not the next. A *maintenance goal* is achieved when the agent's plan causes a condition to hold continuously from a given state onward. Describe how to extend the formalism of this chapter to support maintenance goals.

11.2 You have a number of trucks with which to deliver a set of packages. Each package starts at some location on a grid map, and has a destination somewhere else. Each truck is directly controlled by moving forward and turning. Construct a hierarchy of high-level actions for this problem. What knowledge about the solution does your hierarchy encode?

11.3 Suppose that a high-level action has exactly one implementation as a sequence of primitive actions. Give an algorithm for computing its preconditions and effects, given the complete refinement hierarchy and schemas for the primitive actions.

11.4 Suppose that the optimistic reachable set of a high-level plan is a superset of the goal set; can anything be concluded about whether the plan achieves the goal? What if the pessimistic reachable set doesn't intersect the goal set? Explain.

11.5 Write an algorithm that takes an initial state (specified by a set of propositional literals) and a sequence of HLAs (each defined by preconditions and angelic specifications of optimistic and pessimistic reachable sets) and computes optimistic and pessimistic descriptions of the reachable set of the sequence.

11.6 In Figure 11.2 we showed how to describe actions in a scheduling problem by using separate fields for DURATION, USE, and CONSUME. Now suppose we wanted to combine scheduling with nondeterministic planning, which requires nondeterministic and conditional effects. Consider each of the three fields and explain if they should remain separate fields, or if they should become effects of the action. Give an example for each of the three.

11.7 Some of the operations in standard programming languages can be modeled as actions that change the state of the world. For example, the assignment operation changes the contents of a memory location, and the print operation changes the state of the output stream. A program consisting of these operations can also be considered as a plan, whose goal is given

by the specification of the program. Therefore, planning algorithms can be used to construct programs that achieve a given specification.

- a. Write an action schema for the assignment operator (assigning the value of one variable to another). Remember that the original value will be overwritten!
- b. Show how object creation can be used by a planner to produce a plan for exchanging the values of two variables by using a temporary variable.

11.8 Suppose the *Flip* action always changes the truth value of variable *L*. Show how to define its effects by using an action schema with conditional effects. Show that, despite the use of conditional effects, a 1-CNF belief state representation remains in 1-CNF after a *Flip*.

11.9 In the blocks world we were forced to introduce two action schemas, *Move* and *MoveToTable*, in order to maintain the *Clear* predicate properly. Show how conditional effects can be used to represent both of these cases with a single action.

11.10 Conditional effects were illustrated for the *Suck* action in the vacuum world—which square becomes clean depends on which square the robot is in. Can you think of a new set of propositional variables to define states of the vacuum world, such that *Suck* has an *unconditional* description? Write out the descriptions of *Suck*, *Left*, and *Right*, using your propositions, and demonstrate that they suffice to describe all possible states of the world.

11.11 Find a suitably dirty carpet, free of obstacles, and vacuum it. Draw the path taken by the vacuum cleaner as accurately as you can. Explain it, with reference to the forms of planning discussed in this chapter.

11.12 To the medication problem in the previous exercise, add a *Test* action that has the conditional effect *CultureGrowth* when *Disease* is true and in any case has the perceptual effect *Known(CultureGrowth)*. Diagram a conditional plan that solves the problem and minimizes the use of the *Medicate* action.