

# 5

## ADVERSARIAL SEARCH

*In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.*

### 5.1 GAMES

GAME

Chapter 2 introduced **multiagent environments**, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce **contingencies** into the agent’s problem-solving process, as discussed in Chapter 4. In this chapter we cover **competitive environments**, in which the agents’ goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

ZERO-SUM GAMES  
PERFECT  
INFORMATION

Mathematical **game theory**, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive.<sup>1</sup> In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information** (such as chess). In our terminology, this means **deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite**. For example, if one player wins a game of chess, the other player necessarily loses. It is this **opposition between the agents’ utility functions that makes the situation adversarial**.

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules. Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

<sup>1</sup> Environments with very many agents are often viewed as **economies** rather than games.

Games, unlike most of the toy problems studied in Chapter 3, are interesting *because* they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about  $35^{100}$  or  $10^{154}$  nodes (although the search graph has “only” about  $10^{40}$  distinct nodes). Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A\* search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time.

PRUNING

We begin with a definition of the optimal move and an algorithm for finding it. We then look at techniques for choosing a good move when time is limited. **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice, and **heuristic evaluation functions** allow us to approximate the true utility of a state without doing a complete search. Section 5.5 discusses games such as backgammon that include an element of chance; we also discuss bridge, which includes elements of **imperfect information** because not all cards are visible to each player. Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.

IMPERFECT  
INFORMATION

We first consider games with **two players**, whom we call **MAX** and **MIN** for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:

TERMINAL TEST

TERMINAL STATES

- $S_0$ : The **initial state**, which specifies how the game is set up at the start.
- $\text{PLAYER}(s)$ : Defines which player has the move in a state.
- $\text{ACTIONS}(s)$ : Returns the set of legal moves in a state.
- $\text{RESULT}(s, a)$ : The **transition model**, which defines the result of a move.
- $\text{TERMINAL-TEST}(s)$ : A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- $\text{UTILITY}(s, p)$ : A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state  $s$  for a player  $p$ . In chess, the outcome is a win, loss, or draw, with values  $+1$ ,  $0$ , or  $\frac{1}{2}$ . Some games have a wider variety of possible outcomes; the payoffs in backgammon range from  $0$  to  $+192$ . A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either  $0 + 1$ ,  $1 + 0$  or  $\frac{1}{2} + \frac{1}{2}$ . “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of  $\frac{1}{2}$ .

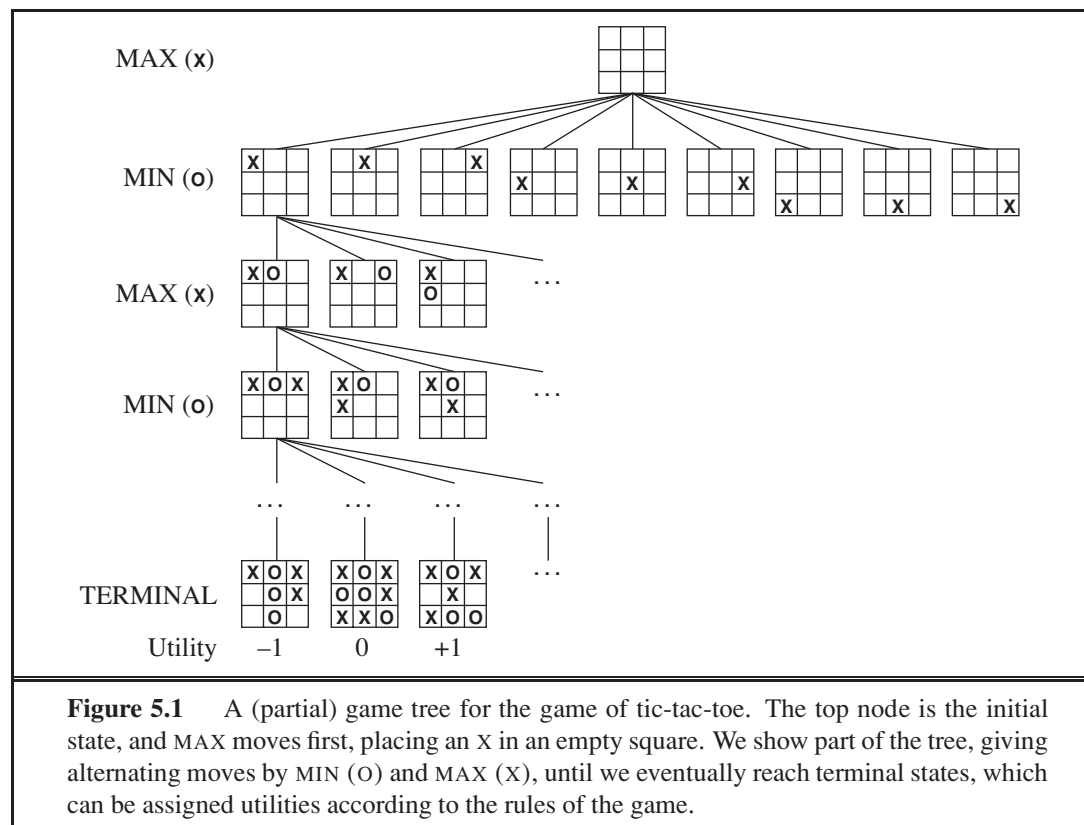
GAME TREE

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves. Figure 5.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX’s placing an X and MIN’s placing an O

until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

For tic-tac-toe the game tree is relatively small—fewer than  $9! = 362,880$  terminal nodes. But for chess there are over  $10^{40}$  nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the size of the game tree, it is MAX's job to search for a good move. We use the term **search tree** for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

SEARCH TREE

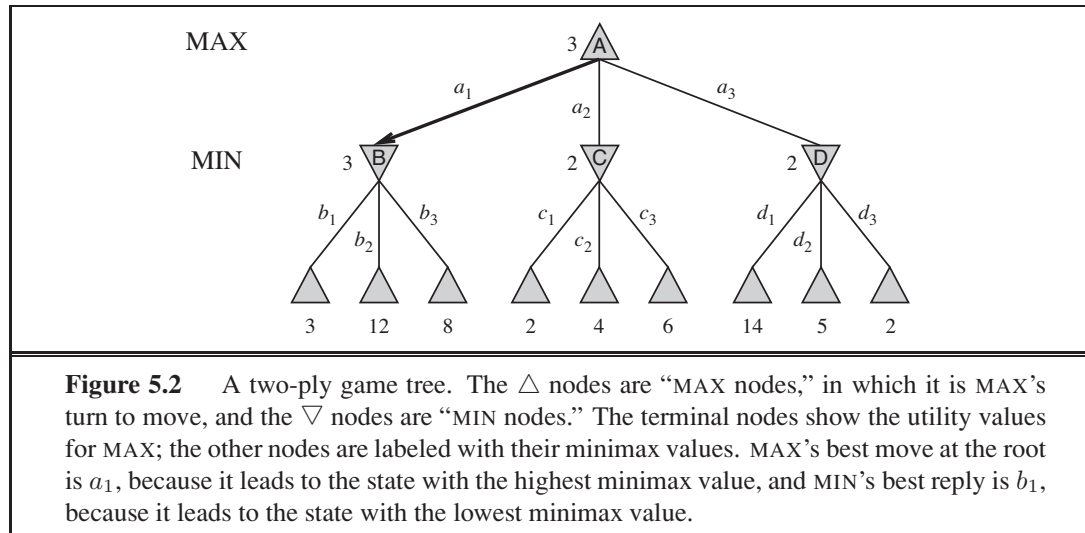


**Figure 5.1** A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

## 5.2 OPTIMAL DECISIONS IN GAMES

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by

STRATEGY



MIN, then MAX’s moves in the states resulting from every possible response by MIN to those moves, and so on. This is exactly analogous to the AND–OR search algorithm (Figure 4.11) with MAX playing the role of OR and MIN equivalent to AND. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We begin by showing how to find this optimal strategy.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure 5.2. The possible moves for MAX at the root node are labeled  $a_1$ ,  $a_2$ , and  $a_3$ . The possible replies to  $a_1$  for MIN are  $b_1$ ,  $b_2$ ,  $b_3$ , and so on. This particular game ends after one move each by MAX and MIN. (In game parlance, we say that this tree is one move deep, consisting of two half-moves, each of which is called a ply.) The utilities of the terminal states in this game range from 2 to 14.

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as  $\text{MINIMAX}(n)$ . The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

Let us apply these definitions to the game tree in Figure 5.2. The terminal nodes on the bottom level get their utility values from the game’s  $\text{UTILITY}$  function. The first MIN node, labeled  $B$ , has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify

PLY  
MINIMAX VALUE

## MINIMAX DECISION

the **minimax decision** at the root: action  $a_1$  is the optimal choice for MAX because it leads to the state with the highest minimax value.

This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the *worst-case* outcome for MAX. What if MIN does not play optimally? Then it is easy to show (Exercise 5.7) that MAX will do even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

### 5.2.1 The minimax algorithm

## MINIMAX ALGORITHM

The **minimax algorithm** (Figure 5.3) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 5.2, the algorithm first recurses down to the three bottom-left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node  $B$ . A similar process gives the backed-up values of 2 for  $C$  and 2 for  $D$ . Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is  $m$  and there are  $b$  legal moves at each point, then the time complexity of the minimax algorithm is  $O(b^m)$ . The space complexity is  $O(bm)$  for an algorithm that generates all actions at once, or  $O(m)$  for an algorithm that generates actions one at a time (see page 87). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

### 5.2.2 Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players  $A$ ,  $B$ , and  $C$ , a vector  $\langle v_A, v_B, v_C \rangle$  is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked  $X$  in the game tree shown in Figure 5.4. In that state, player  $C$  chooses what to do. The two choices lead to terminal states with utility vectors  $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$  and  $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$ . Since 6 is bigger than 3,  $C$  should choose the first move. This means that if state  $X$  is reached, subsequent play will lead to a terminal state with utilities  $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ . Hence, the backed-up value of  $X$  is this vector. The backed-up value of a node  $n$  is always the utility

---

```

function MINIMAX-DECISION(state) returns an action
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 

```

---

```

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
  return v

```

---

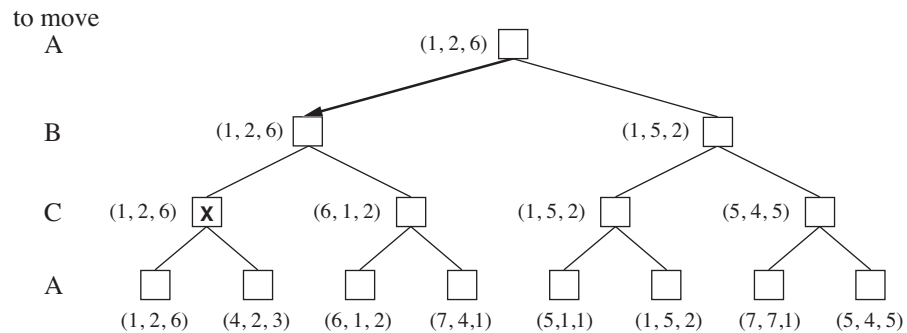
```

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
  return v

```

---

**Figure 5.3** An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation  $\arg \max_{a \in S} f(a)$  computes the element *a* of set *S* that has the maximum value of *f*(*a*).



**Figure 5.4** The first three plies of a game tree with three players (*A*, *B*, *C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

vector of the successor state with the highest value for the player choosing at *n*. Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be. For example,

suppose  $A$  and  $B$  are in weak positions and  $C$  is in a stronger position. Then it is often optimal for both  $A$  and  $B$  to attack  $C$  rather than each other, lest  $C$  destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as  $C$  weakens under the joint onslaught, the alliance loses its value, and either  $A$  or  $B$  could violate the agreement. In some cases, explicit alliances merely make concrete what would have happened anyway. In other cases, a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy. See Section 17.5 for more on these complications.

If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities  $\langle v_A = 1000, v_B = 1000 \rangle$  and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

### 5.3 ALPHA–BETA PRUNING

ALPHA–BETA  
PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from Chapter 3 to eliminate large parts of the tree from consideration. The particular technique we examine is called **alpha–beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider again the two-ply game tree from Figure 5.2. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 5.5. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

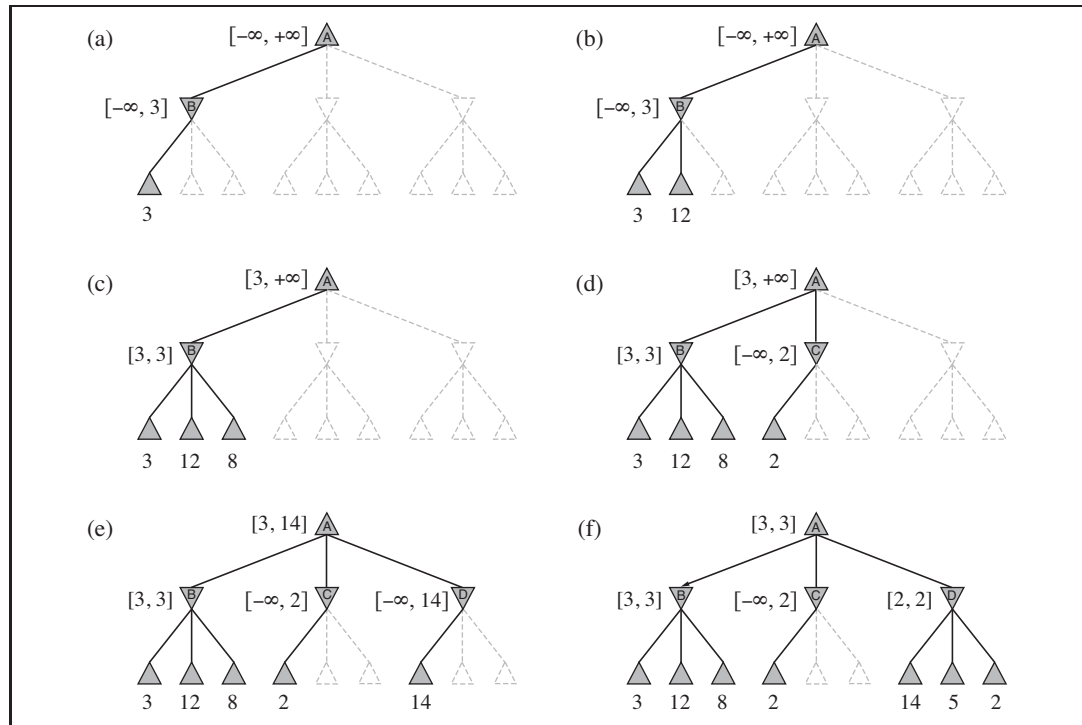
Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node  $C$  in Figure 5.5 have values  $x$  and  $y$ . Then the value of the root node is given by

$$\begin{aligned} \text{MINIMAX}(\text{root}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \quad \text{where } z = \min(2, x, y) \leq 2 \\ &= 3. \end{aligned}$$

In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves  $x$  and  $y$ .

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node  $n$





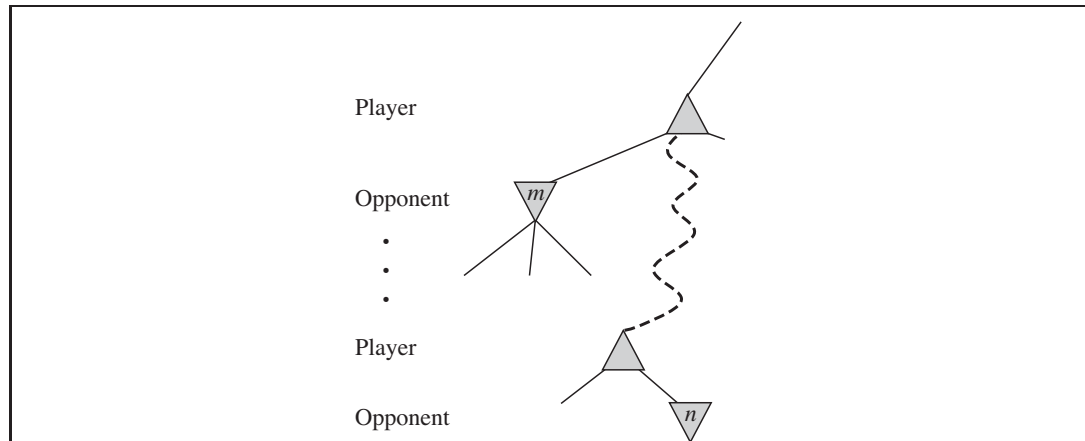
**Figure 5.5** Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below  $B$  has the value 3. Hence,  $B$ , which is a MIN node, has a value of *at most* 3. (b) The second leaf below  $B$  has a value of 12; MIN would avoid this move, so the value of  $B$  is still at most 3. (c) The third leaf below  $B$  has a value of 8; we have seen all  $B$ 's successor states, so the value of  $B$  is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below  $C$  has the value 2. Hence,  $C$ , which is a MIN node, has a value of *at most* 2. But we know that  $B$  is worth 3, so MAX would never choose  $C$ . Therefore, there is no point in looking at the other successor states of  $C$ . This is an example of alpha-beta pruning. (e) The first leaf below  $D$  has the value 14, so  $D$  is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring  $D$ 's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of  $D$  is worth 5, so again we need to keep exploring. The third successor is worth 2, so now  $D$  is worth exactly 2. MAX's decision at the root is to move to  $B$ , giving a value of 3.



somewhere in the tree (see Figure 5.6), such that Player has a choice of moving to that node. If Player has a better choice  $m$  either at the parent node of  $n$  or at any choice point further up, then  $n$  will never be reached in actual play. So once we have found out enough about  $n$  (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:





**Figure 5.6** The general case for alpha–beta pruning. If  $m$  is better than  $n$  for Player, we will never get to  $n$  in play.

$\alpha$  = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta$  = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of  $\alpha$  and  $\beta$  as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current  $\alpha$  or  $\beta$  value for MAX or MIN, respectively. The complete algorithm is given in Figure 5.7. We encourage you to trace its behavior when applied to the tree in Figure 5.5.

### 5.3.1 Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(e) and (f), we could not prune any successors of  $D$  at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of  $D$  had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

If this can be done,<sup>2</sup> then it turns out that alpha–beta needs to examine only  $O(b^{m/2})$  nodes to pick the best move, instead of  $O(b^m)$  for minimax. This means that the effective branching factor becomes  $\sqrt{b}$  instead of  $b$ —for chess, about 6 instead of 35. Put another way, alpha–beta can solve a tree roughly twice as deep as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly  $O(b^{3m/4})$  for moderate  $b$ . For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case  $O(b^{m/2})$  result.

<sup>2</sup> Obviously, it cannot be done perfectly; otherwise, the ordering function could be used to play a perfect game!

---

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action

$v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$

**return** the *action* in ACTIONS(*state*) with value *v*

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

**for each** *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

**if**  $v \geq \beta$  **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

**return** *v*

---

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow +\infty$

**for each** *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

**if**  $v \leq \alpha$  **then return** *v*

$\beta \leftarrow \text{MIN}(\beta, v)$

**return** *v*

---

**Figure 5.7** The alpha-beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain  $\alpha$  and  $\beta$  (and the bookkeeping to pass these parameters along).

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move. One way to gain information from the current move is with iterative deepening search. First, search 1 ply deep and record the best path of moves. Then search 1 ply deeper, but use the recorded path to inform move ordering. As we saw in Chapter 3, iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering. The best moves are often called **killer moves** and to try them first is called the killer move heuristic.

In Chapter 3, we noted that repeated states in the search tree can cause an exponential increase in search cost. In many games, repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position. For example, if White has one move,  $a_1$ , that can be answered by Black with  $b_1$  and an unrelated move  $a_2$  on the other side of the board that can be answered by  $b_2$ , then the sequences  $[a_1, b_1, a_2, b_2]$  and  $[a_2, b_2, a_1, b_1]$  both end up in the same position. It is worthwhile to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences. The hash table of previously seen positions is traditionally called a **transposition table**; it is essentially identical to the *explored*

KILLER MOVES

TRANSPPOSITION

TRANSPPOSITION  
TABLE

list in GRAPH-SEARCH (Section 3.3). Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, at some point it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

## 5.4 IMPERFECT REAL-TIME DECISIONS

EVALUATION  
FUNCTION

CUTOFF TEST

The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Claude Shannon’s paper *Programming a Computer for Playing Chess* (1950) proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha-beta in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position’s utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL. That gives us the following for heuristic minimax for state  $s$  and maximum depth  $d$ :

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

### 5.4.1 Evaluation functions

An evaluation function returns an *estimate* of the expected utility of the game from a given position, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. The idea of an estimator was not new when Shannon proposed it. For centuries, chess players (and aficionados of other games) have developed ways of judging the value of a position because humans are even more limited in the amount of search they can do than are computer programs. It should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost. How exactly do we design good evaluation functions?

First, the evaluation function should order the *terminal* states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game. Second, the computation must not take too long! (The whole point is to search faster.) Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

One might well wonder about the phrase “chances of winning.” After all, chess is not a game of chance: we know the current state with certainty, and no dice are involved. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes of those states. This type of uncertainty is induced by computational, rather than informational, limitations. Given the limited amount of computation that the evaluation function is allowed to do for a given state, the best it can do is make a guess about the final outcome.

EXPECTED VALUE

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features. For example, one category contains all two-pawn vs. one-pawn endgames. Any given category, generally speaking, will contain some states that lead to wins, some that lead to draws, and some that lead to losses. The evaluation function cannot know which states are which, but it can return a single value that reflects the *proportion* of states with each outcome. For example, suppose our experience suggests that 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility +1); 20% to a loss (0), and 8% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**:  $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$ . In principle, the expected value can be determined for each category, resulting in an evaluation function that works for any state. As with terminal states, the evaluation function need not return actual expected values as long as the *ordering* of the states is the same.

MATERIAL VALUE

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For example, introductory chess books give an approximate **material value** for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as “good pawn structure” and “king safety” might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position.

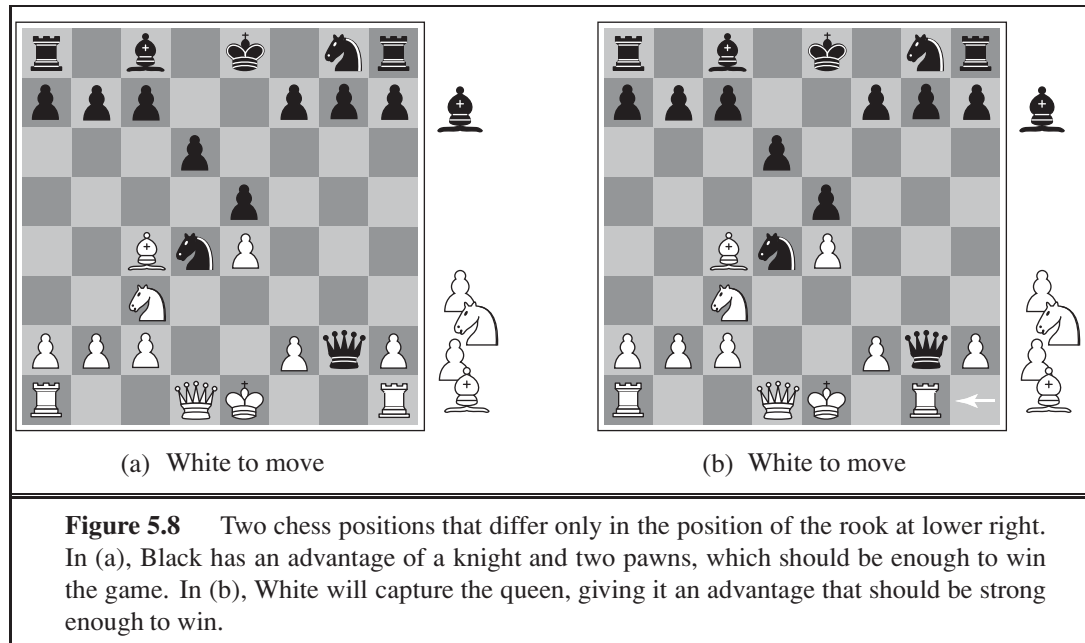
WEIGHTED LINEAR  
FUNCTION

A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 5.8(a). Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s),$$

where each  $w_i$  is a weight and each  $f_i$  is a feature of the position. For chess, the  $f_i$  could be the numbers of each kind of piece on the board, and the  $w_i$  could be the values of the pieces (1 for pawn, 3 for bishop, etc.).

Adding up the values of features seems like a reasonable thing to do, but in fact it involves a strong assumption: that the contribution of each feature is *independent* of the values of the other features. For example, assigning the value 3 to a bishop ignores the fact that bishops are more powerful in the endgame, when they have a lot of space to maneuver.



For this reason, current programs for chess and other games also use *nonlinear* combinations of features. For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame (that is, when the *move number* feature is high or the *number of remaining pieces* feature is low).

The astute reader will have noticed that the features and weights are *not* part of the rules of chess! They come from centuries of human chess-playing experience. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques of Chapter 18. Reassuringly, applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns.

#### 5.4.2 Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the **heuristic EVAL function when it is appropriate to cut off the search**. We replace the two lines in Figure 5.7 that mention TERMINAL-TEST with the following line:

```
if CUTOFF-TEST(state, depth) then return EVAL(state)
```

We also must arrange for some bookkeeping so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that CUTOFF-TEST(*state*, *depth*) returns *true* for all *depth* greater than some fixed depth *d*. (It must also return *true* for all terminal states, just as TERMINAL-TEST did.) The depth *d* is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. (See Chapter 3.) When time runs out, the program returns the move selected by the deepest completed search. As a bonus, iterative deepening also helps with move ordering.

These simple approaches can lead to errors due to the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 5.8(b), where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply.

QUIESCENCE

Obviously, a more sophisticated cutoff test is needed. The evaluation function should be applied only to positions that are **quiescent**—that is, unlikely to exhibit wild swings in value in the near future. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a **quiescence search**; sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

QUIESCENCE  
SEARCH

HORIZON EFFECT

The **horizon effect** is more difficult to eliminate. It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics. Consider the chess game in Figure 5.9. It is clear that there is no way for the black bishop to escape. For example, the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply. But Black does have a sequence of moves that pushes the capture of the bishop “over the horizon.” Suppose Black searches to depth 8 ply. Most moves by Black will lead to the eventual capture of the bishop, and thus will be marked as “bad” moves. But Black will consider checking the white king with the pawn at e4. This will lead to the king capturing the pawn. Now Black will consider checking again, with the pawn at f5, leading to another pawn capture. That takes up 4 ply, and from there the remaining 4 ply is not enough to capture the bishop. Black thinks that the line of play has saved the bishop at the price of two pawns, when actually all it has done is push the inevitable capture of the bishop beyond the horizon that Black can see.

SINGULAR  
EXTENSION

One strategy to mitigate the horizon effect is the **singular extension**, a move that is “clearly better” than all other moves in a given position. Once discovered anywhere in the tree in the course of a search, this singular move is remembered. When the search reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered. This makes the tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

### 5.4.3 Forward pruning

FORWARD PRUNING

So far, we have talked about cutting off search at a certain level and about doing alpha-beta pruning that provably has no effect on the result (at least with respect to the heuristic evaluation values). It is also possible to do **forward pruning**, meaning that some moves at a given node are pruned immediately without further consideration. Clearly, most humans playing chess consider only a few moves from each position (at least consciously). One approach to forward pruning is **beam search**: on each ply, consider only a “beam” of the  $n$  best moves (according to the evaluation function) rather than considering all possible moves.

BEAM SEARCH

would not want alpha–beta to waste time determining a precise value for the lone good move. Better to just make the move quickly and save the time for later. This leads to the idea of the *utility of a node expansion*. A good search algorithm should select node expansions of high utility—that is, ones that are likely to lead to the discovery of a significantly better move. If there are no node expansions whose utility is higher than their cost (in terms of time), then the algorithm should stop searching and make a move. Notice that this works not only for clear-favorite situations but also for the case of *symmetrical* moves, for which no amount of search will show that one move is better than another.

METAREASONING

This kind of reasoning about what computations to do is called **metareasoning** (reasoning about reasoning). It applies not just to game playing but to any kind of reasoning at all. All computations are done in the service of trying to reach better decisions, all have costs, and all have some likelihood of resulting in a certain improvement in decision quality. Alpha–beta incorporates the simplest kind of metareasoning, namely, a theorem to the effect that certain branches of the tree can be ignored without loss. It is possible to do much better. In Chapter 16, we see how these ideas can be made precise and implementable.

Finally, let us reexamine the nature of search itself. Algorithms for heuristic search and for game playing generate sequences of concrete states, starting from the initial state and then applying an evaluation function. Clearly, this is not how humans play games. In chess, one often has a particular goal in mind—for example, trapping the opponent’s queen—and can use this goal to *selectively* generate plausible plans for achieving it. This kind of goal-directed reasoning or planning sometimes eliminates combinatorial search altogether. David Wilkins’ (1980) PARADISE is the only program to have used goal-directed reasoning successfully in chess: it was capable of solving some chess problems requiring an 18-move combination. As yet there is no good understanding of how to *combine* the two kinds of algorithms into a robust and efficient system, although Bridge Baron might be a step in the right direction. A fully integrated system would be a significant achievement not just for game-playing research but also for AI research in general, because it would be a good basis for a general intelligent agent.

## 5.9 SUMMARY

We have looked at a variety of games to understand what optimal play means and to understand how to play well in practice. The most important ideas are as follows:

- A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, the **result** of each action, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states.
- In two-player zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves by a depth-first enumeration of the game tree.
- The **alpha–beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we



need to cut the search off at some point and apply a heuristic **evaluation function** that estimates the utility of a state.

- Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search.
- Games of chance can be handled by an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all its children, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as Kriegspiel and bridge, requires reasoning about the current and future **belief states** of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.
- Programs have bested even champion human players at games such as chess, checkers, and Othello. Humans retain the edge in several games of imperfect information, such as poker, bridge, and Kriegspiel, and in games with very large branching factors and little good heuristic knowledge, such as Go.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

The early history of mechanical game playing was marred by numerous frauds. The most notorious of these was Baron Wolfgang von Kempelen's (1734–1804) "The Turk," a supposed chess-playing automaton that defeated Napoleon before being exposed as a magician's trick cabinet housing a human chess expert (see Levitt, 2000). It played from 1769 to 1854. In 1846, Charles Babbage (who had been fascinated by the Turk) appears to have contributed the first serious discussion of the feasibility of computer chess and checkers (Morrison and Morrison, 1961). He did not understand the exponential complexity of search trees, claiming "the combinations involved in the Analytical Engine enormously surpassed any required, even by the game of chess." Babbage also designed, but did not build, a special-purpose machine for playing tic-tac-toe. The first true game-playing machine was built around 1890 by the Spanish engineer Leonardo Torres y Quevedo. It specialized in the "KRK" (king and rook vs. king) chess endgame, guaranteeing a win with king and rook from any position.

The minimax algorithm is traced to a 1912 paper by Ernst Zermelo, the developer of modern set theory. The paper unfortunately contained several errors and did not describe minimax correctly. On the other hand, it did lay out the ideas of retrograde analysis and proposed (but did not prove) what became known as Zermelo's theorem: that chess is determined—White can force a win or Black can or it is a draw; we just don't know which. Zermelo says that should we eventually know, "Chess would of course lose the character of a game at all." A solid foundation for game theory was developed in the seminal work *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944), which included an analysis showing that some games *require* strategies that are randomized (or otherwise unpredictable). See Chapter 17 for more information.

# 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

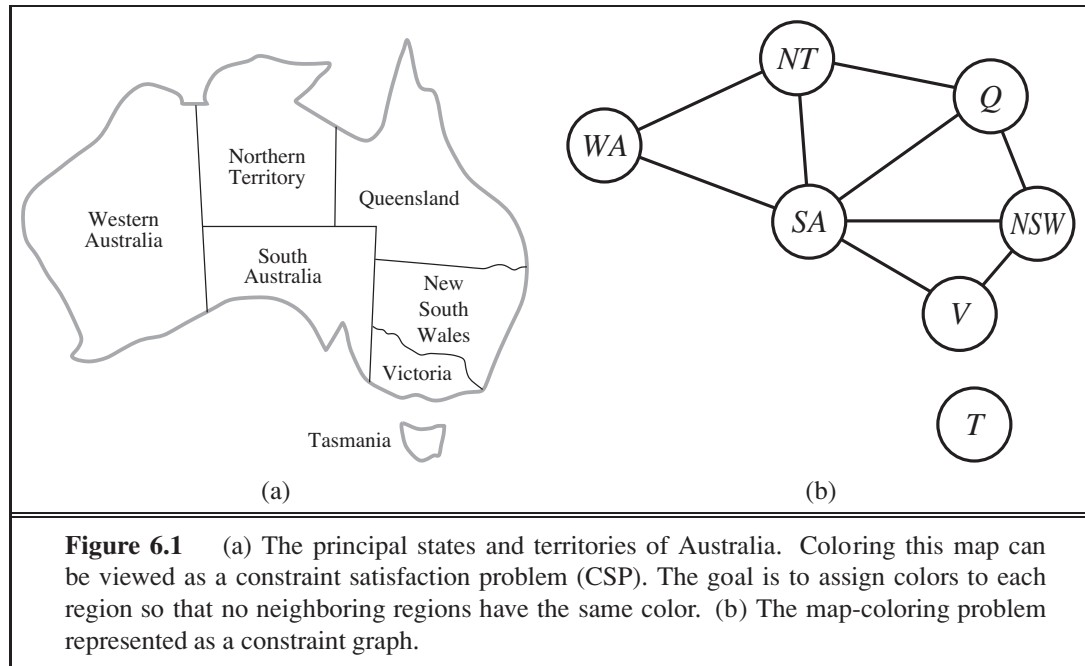
$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

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:

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

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).

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

GLOBAL  
CONSTRAINT

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  $\text{Alldiff}(F, T, U, W, R, O)$ . The addition constraints on the four columns of the puzzle can be written as the following  $n$ -ary constraints:

CRYPTARITHMETIC

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

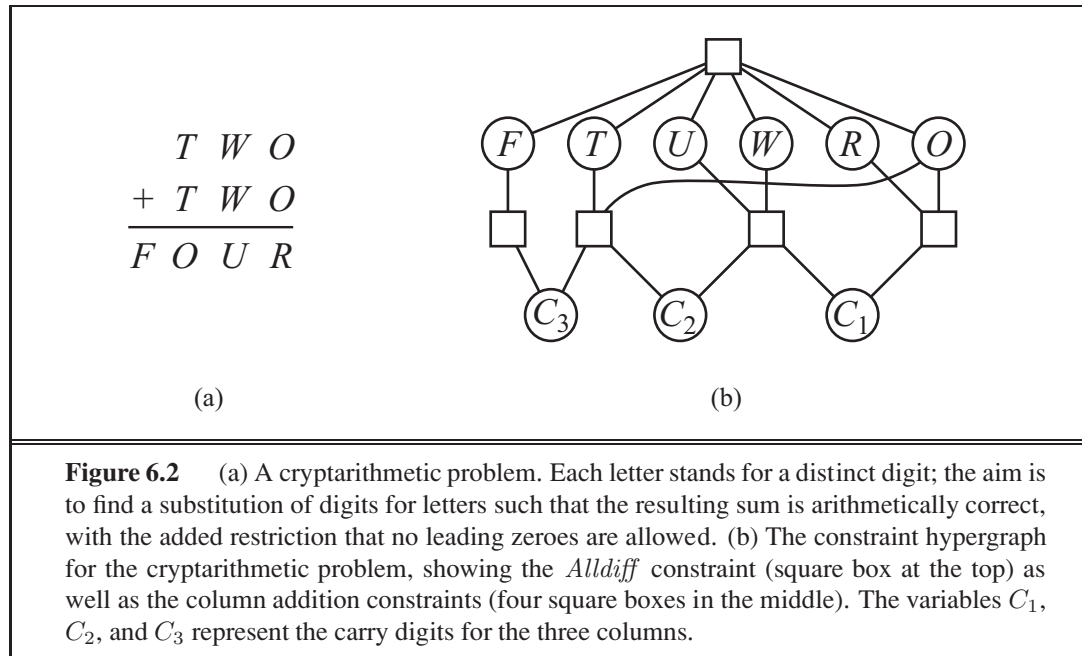
where  $C_{10}$ ,  $C_{100}$ , and  $C_{1000}$  are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes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

CONSTRAINT  
HYPERGRAPH

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

DUAL GRAPH





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.

PREFERENCE  
CONSTRAINTSCONSTRAINT  
OPTIMIZATION  
PROBLEM



## 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
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if REVISE(csp,  $X_i, X_j$ ) then
      if size of  $D_i = 0$  then return false
      for each  $X_k$  in  $X_i.\text{NEIGHBORS} - \{X_j\}$  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.<sup>1</sup>

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.

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.

<sup>1</sup> 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

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

RESOURCE  
CONSTRAINT

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.

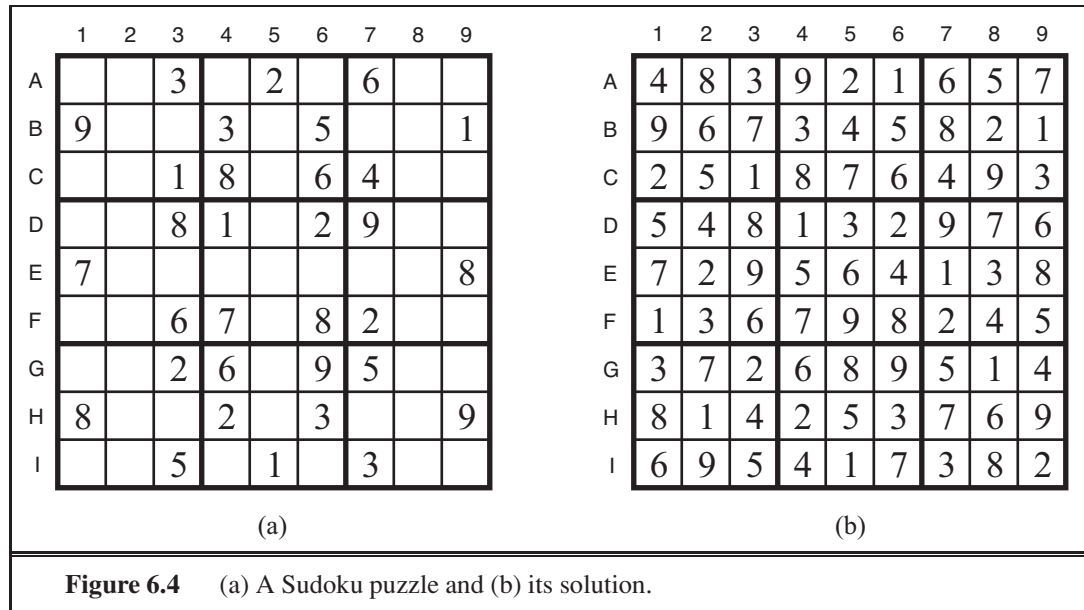
### 6.2.6 Sudoku example

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  $3 \times 3$  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.

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

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.

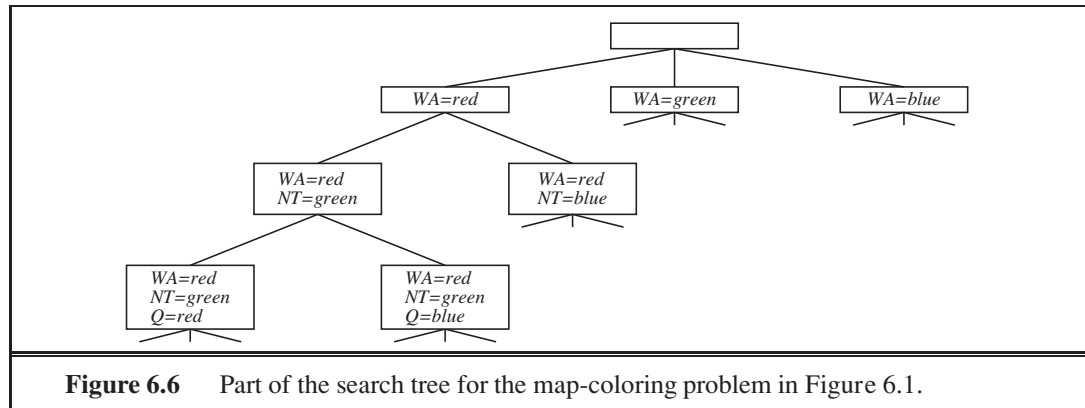
BACKTRACKING  
SEARCH

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)?



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

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp) .$

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.

MINIMUM-  
REMAINING-VALUES

DEGREE HEURISTIC

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-

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

FORWARD  
CHECKING

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.

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 $WA=red$	(R)	G B	R G B	R G B	R G B	G B	R G B
After $Q=green$	(R)	B	(G)	R B	R G B	B	R G B
After $V=blue$	(R)	B	(G)	R	(B)		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.

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.

CONFLICT SET  
BACKJUMPING

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

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

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.<sup>2</sup>

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**

<sup>2</sup> 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



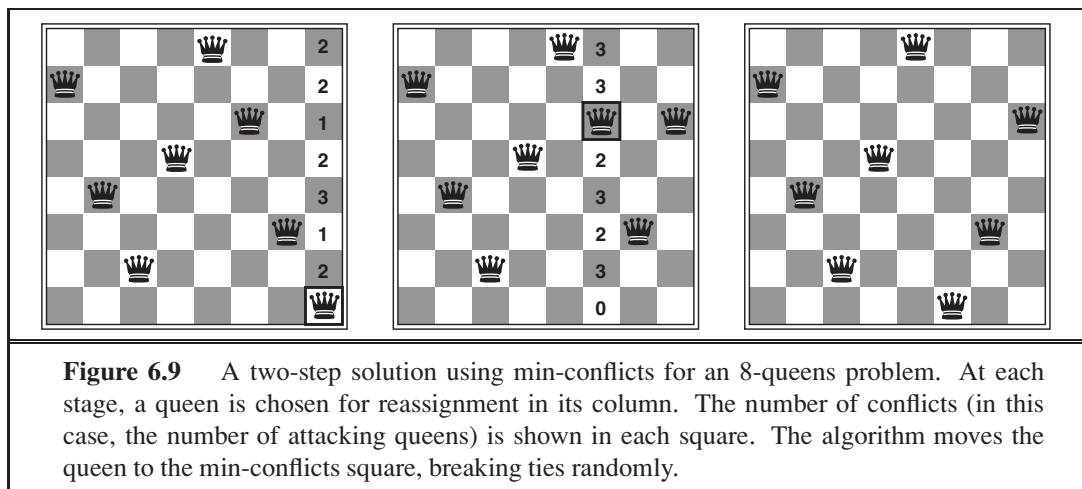
```

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.<sup>3</sup> 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

<sup>3</sup> 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.

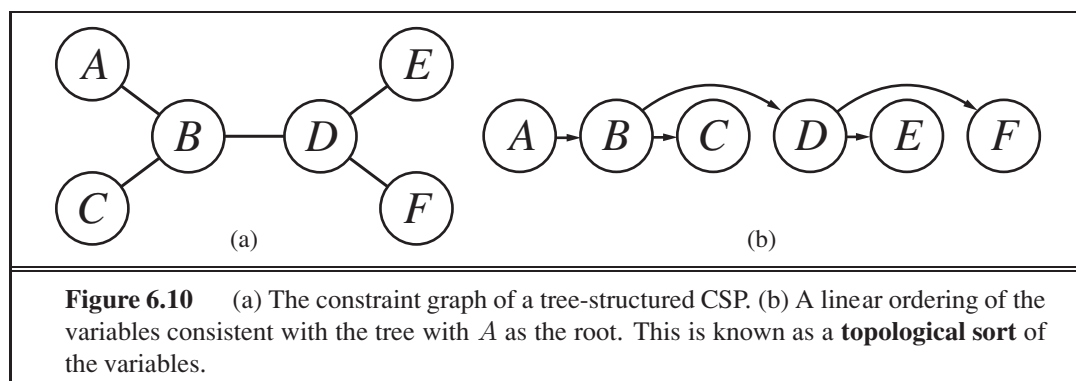
where  $d$  is the size of the domain. Hence, the total work is  $O(d^c n/c)$ , which is *linear* in  $n$ ; without the decomposition, the total work is  $O(d^n)$ , which is exponential in  $n$ . Let's make this more concrete: dividing a Boolean CSP with 80 variables into four subproblems reduces the 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*.<sup>4</sup> 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

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

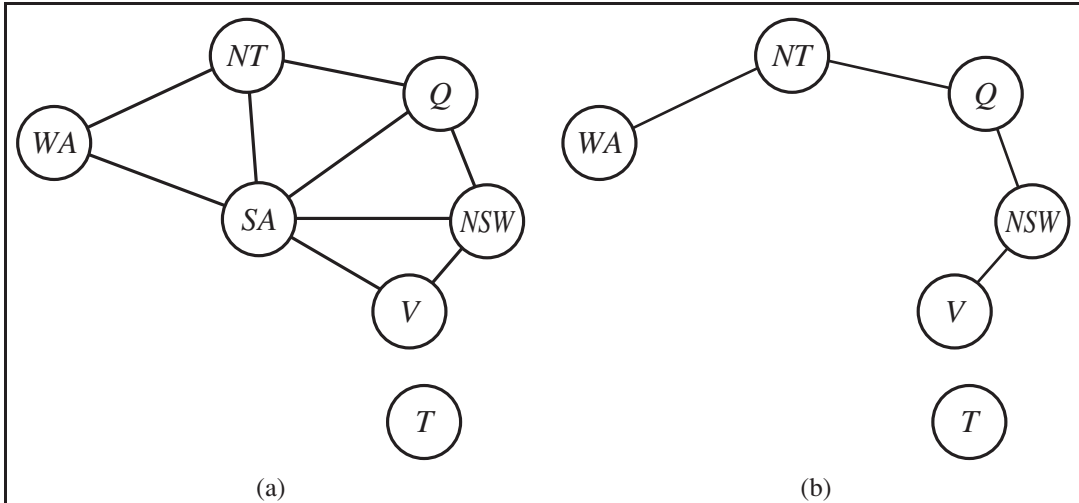
```

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:

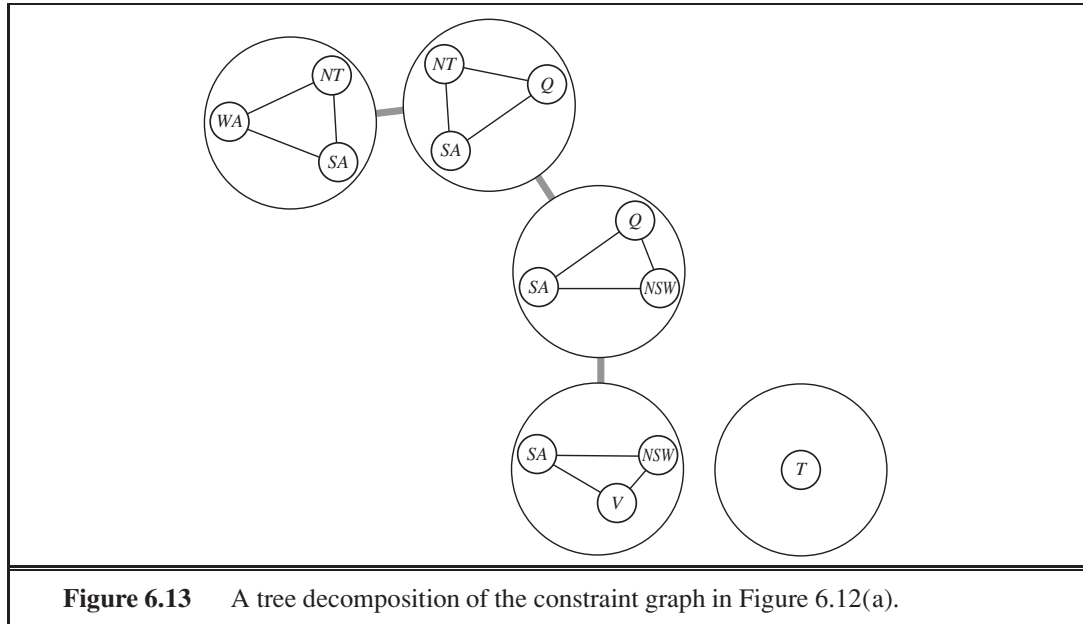
- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, 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  $\{WA = red, SA = blue, NT = green\}$ . 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  $\{WA = red, SA = blue, NT = green\}$  for the first subproblem, the only consistent solution for the next subproblem is  $\{SA = blue, NT = green, Q = red\}$ .

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.



VALUE SYMMETRY  
SYMMETRY-  
BREAKING  
CONSTRAINT

## 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 Hoesve (2001). Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998). A survey of global constraints is provided by van Hoesve 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:<sup>5</sup> 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?"

---

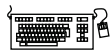
<sup>5</sup> Ginsberg *et al.* (1990) discuss several methods for constructing crossword puzzles. Littman *et al.* (1999) tackle the harder problem of solving them.

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.

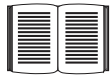
**6.12** What is the worst-case complexity of running AC-3 on a tree-structured CSP?

**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** 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.

**SENTENCE**

**KNOWLEDGE REPRESENTATION LANGUAGE AXIOM**

**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.

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**.

**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.

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.

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

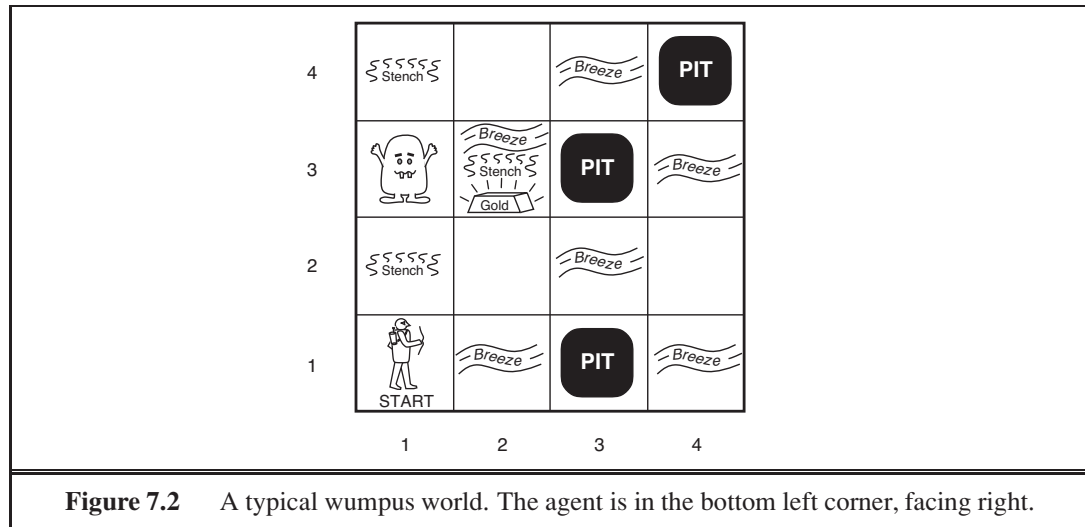
bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates 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  $4 \times 4$  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  $90^\circ$ , or *TurnRight* by  $90^\circ$ . 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



known; or we could say that the transition model itself is unknown because the agent doesn't know which *Forward* actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model.

For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

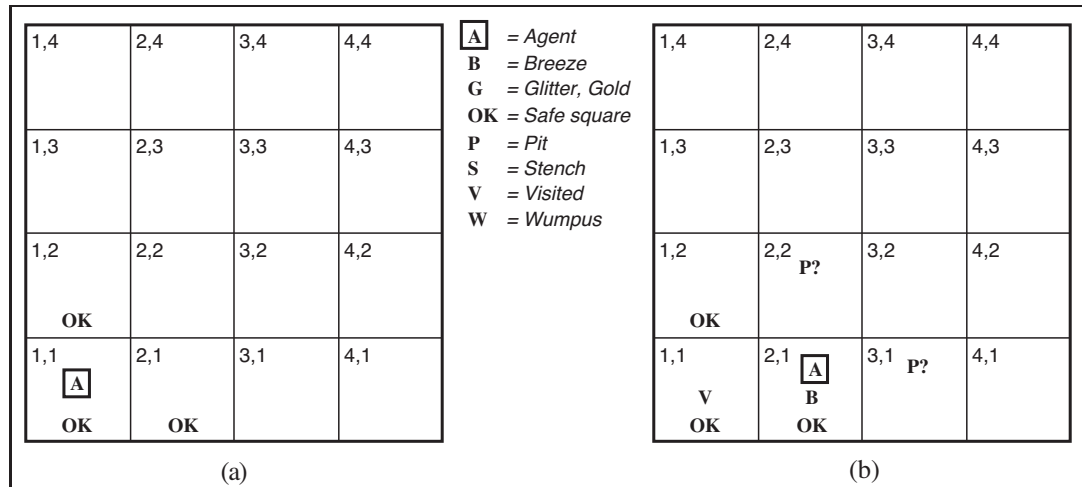
Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 7.3 and 7.4).

The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].

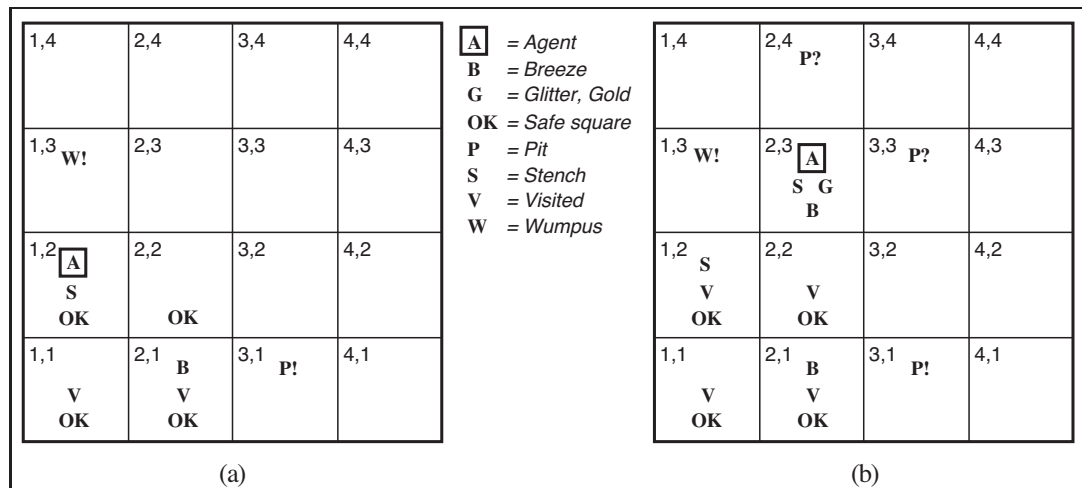
The first percept is  $[None, None, None, None, None]$ , from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3(a) shows the agent's state of knowledge at this point.

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the



**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 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.”<sup>1</sup>

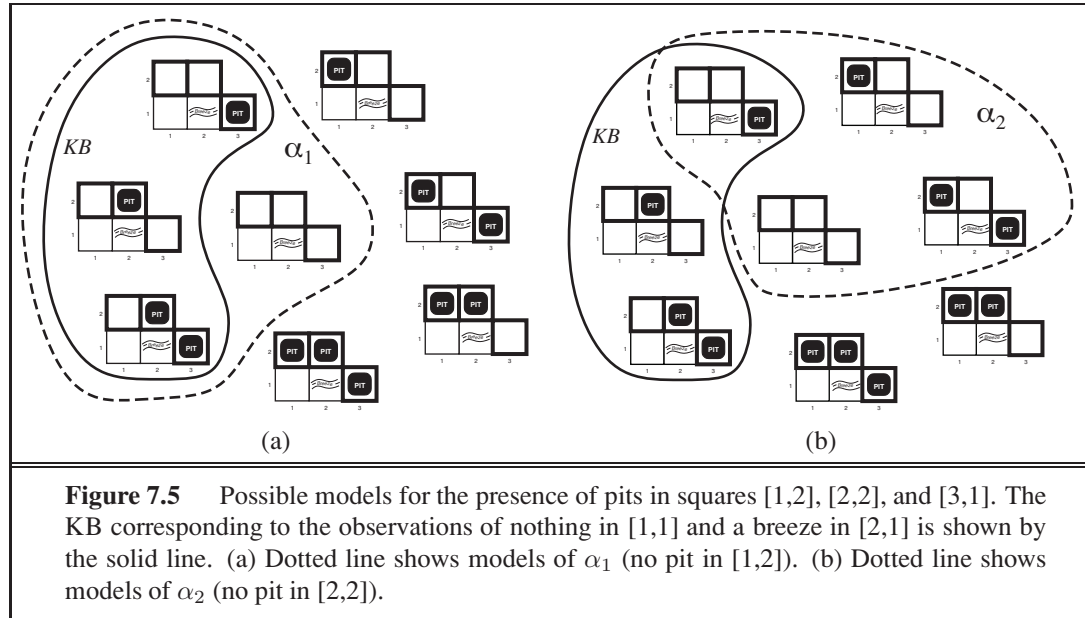
MODEL When we need to be precise, we use the term **model** in place of “possible world.” Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, **models are mathematical abstractions, each of which simply fixes the truth or falsehood of 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  $\alpha$  is true in model  $m$ , we say that  $m$  **satisfies**  $\alpha$  or sometimes  $m$  **is a model of**  $\alpha$ . We use the notation  $M(\alpha)$  to mean the set of all models of  $\alpha$ .

SATISFACTION 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

$$\alpha \models \beta$$

<sup>1</sup> **Fuzzy logic**, discussed in Chapter 14, allows for degrees of truth.





to mean that the sentence  $\alpha$  entails the sentence  $\beta$ . The formal definition of entailment is this:  $\alpha \models \beta$  if and only if, in every model in which  $\alpha$  is true,  $\beta$  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  $\alpha$  is a *stronger* assertion than  $\beta$ : 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.<sup>2</sup>

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

<sup>2</sup> 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:

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

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

We have surrounded the models of  $\alpha_1$  and  $\alpha_2$  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,  $\alpha_1$  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,  $\alpha_2$  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].)<sup>3</sup>

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  $\alpha$  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  $\alpha$  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  $\alpha$  from  $KB$ , we write

$KB \vdash_i \alpha$ ,

which is pronounced “ $\alpha$  is derived from  $KB$  by  $i$ ” or “ $i$  derives  $\alpha$  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,<sup>4</sup> 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.<sup>5</sup> Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

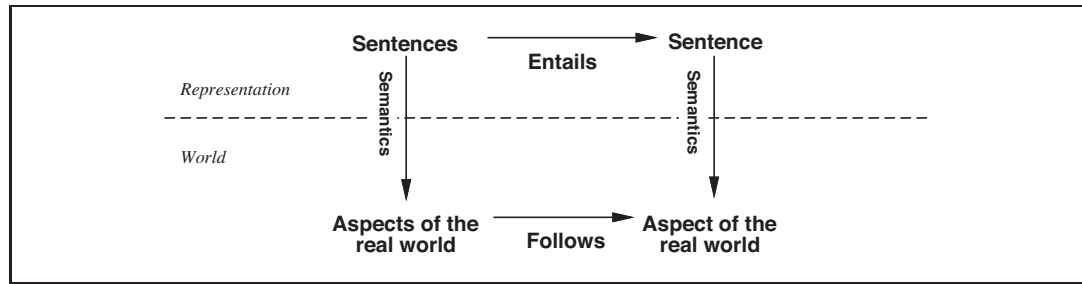


We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, **if  $KB$  is true in the real world, then any sentence  $\alpha$  derived from  $KB$  by a sound inference procedure is also true in the real world**. So, while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

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

<sup>4</sup> 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  $x + y = 4$ .

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



**Figure 7.6** Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

to the real-world relationship whereby some aspect of the real world is the case<sup>6</sup> 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.

<sup>6</sup> 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  
LOGICAL  
CONNECTIVES

**Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**. There are **five connectives** in common use:

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

$\wedge$  (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.”)

CONJUNCTION

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  $\equiv$ .

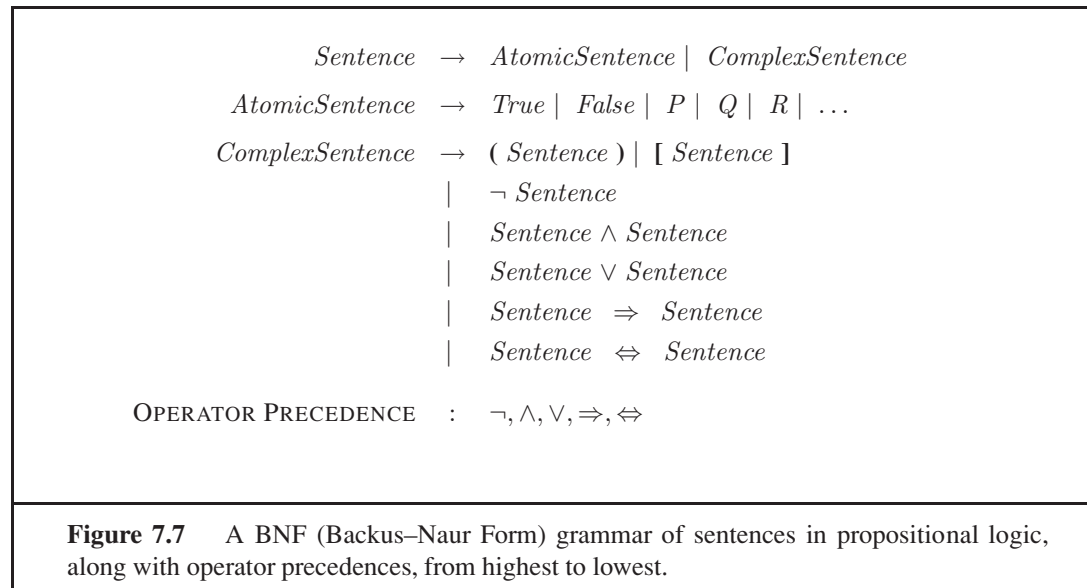


Figure 7.7 gives a formal grammar of propositional logic; see page 1060 if you are not familiar with the BNF notation. The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. 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,

$P$	$Q$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

**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.<sup>7</sup> 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:

<sup>7</sup> 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  $\alpha$ . 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  $\alpha$  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  $\alpha$  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  $\alpha$  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  $\alpha$ ,  $\beta$ , and  $\gamma$  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  $\alpha$  and  $\beta$  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  $\alpha$  and  $\beta$  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  $\alpha$  and  $\beta$ ,  $\alpha \models \beta$  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:  $\alpha$  is valid iff  $\neg\alpha$  is unsatisfiable; contrapositively,  $\alpha$  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  $\beta$  from  $\alpha$  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  $\beta$  to be false and shows that this leads to a contradiction with known axioms  $\alpha$ . 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  $\alpha$  are given, then the sentence  $\beta$  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  $\alpha$  and  $\beta$ , 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  $\alpha$  from  $\beta$ .

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  $R_5$ ; 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.<sup>8</sup> For any sentences  $\alpha$  and  $\beta$ ,

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

<sup>8</sup> **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  $\beta$  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  $\alpha$  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,

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

where each  $\ell$  is a literal and  $\ell_i$  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  $\ell_i$  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.<sup>9</sup> 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  $\ell_i$  that is complementary to literal  $m_j$  in the other clause. If  $\ell_i$  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  $\ell_i$  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  $\ell_i$  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  $\alpha$  and  $\beta$  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  $\neg\alpha \vee \beta$ :

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

<sup>9</sup> 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  $\alpha$ ; or,
- two clauses resolve to yield the *empty* clause, in which case  $KB$  entails  $\alpha$ .

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  $\alpha$  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.

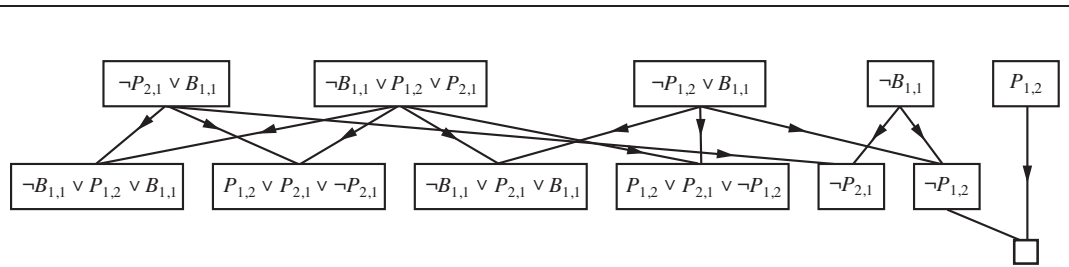
```

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  $(false \vee false \vee \dots \vee false \vee P_i)$  or like  $(false \vee false \vee \dots \vee false \vee \neg P_i)$ . 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  $Q \Rightarrow P$ .

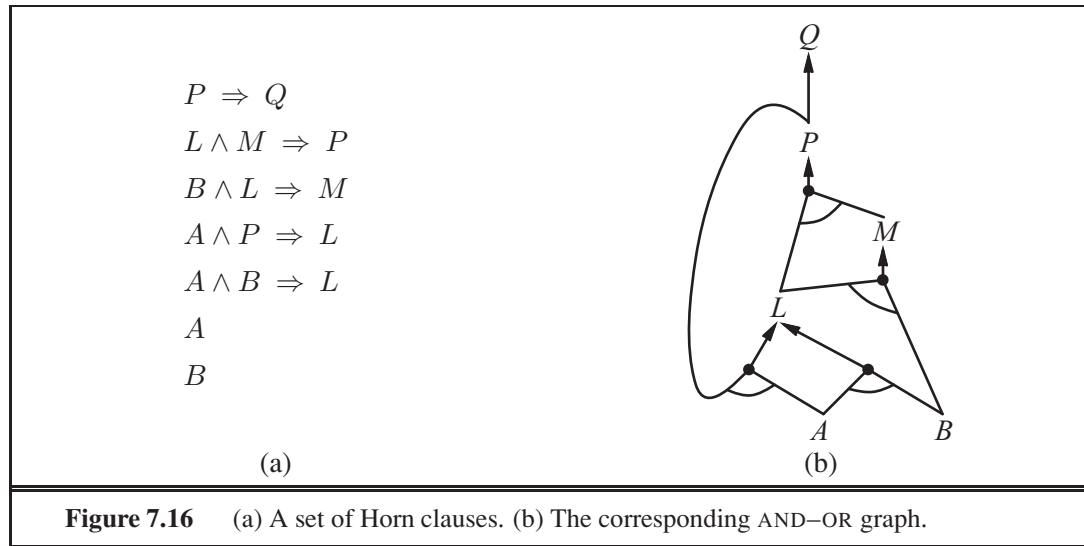
FIXED POINT



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.

DATA-DRIVEN

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



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.

## UNIT PROPAGATION

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.)

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 ← 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 ← 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:

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

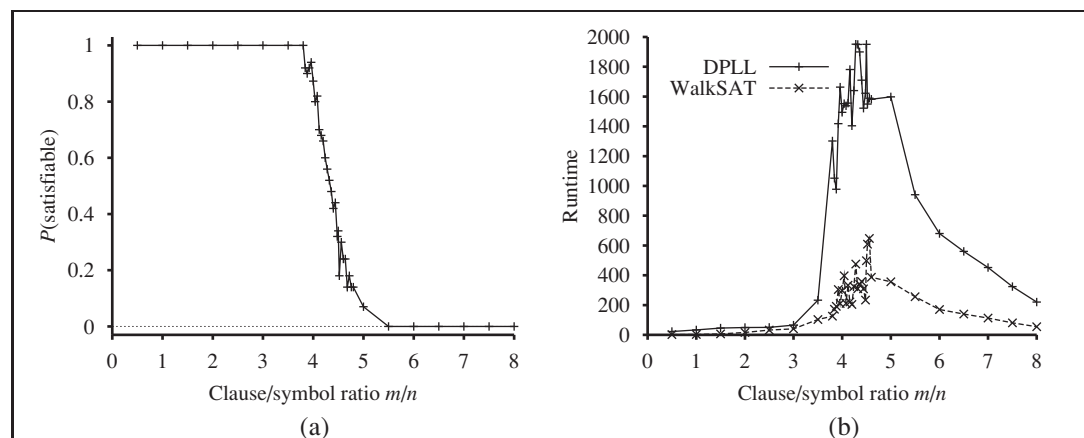
Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are

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.<sup>10</sup> 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*.

FRAME PROBLEM

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**.<sup>11</sup> One possible solution to the frame problem would

<sup>10</sup> Section 7.4.3 conveniently glossed over this requirement.

<sup>11</sup> 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

INFERENTIAL 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<sup>t</sup>* 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.

CACHING

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.<sup>12</sup> 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

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

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

<sup>12</sup> 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  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \text{OK}_{x,y}^t) = \text{true}\}$ 
  if ASK(KB, Glittert) = true then
    plan  $\leftarrow [\text{Grab}] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [\text{Climb}]$ 
  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  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \neg \text{OK}_{x,y}^t) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{not\_unsafe}, \text{safe})$ 
  if plan is empty then
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [\text{Climb}]$ 
  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

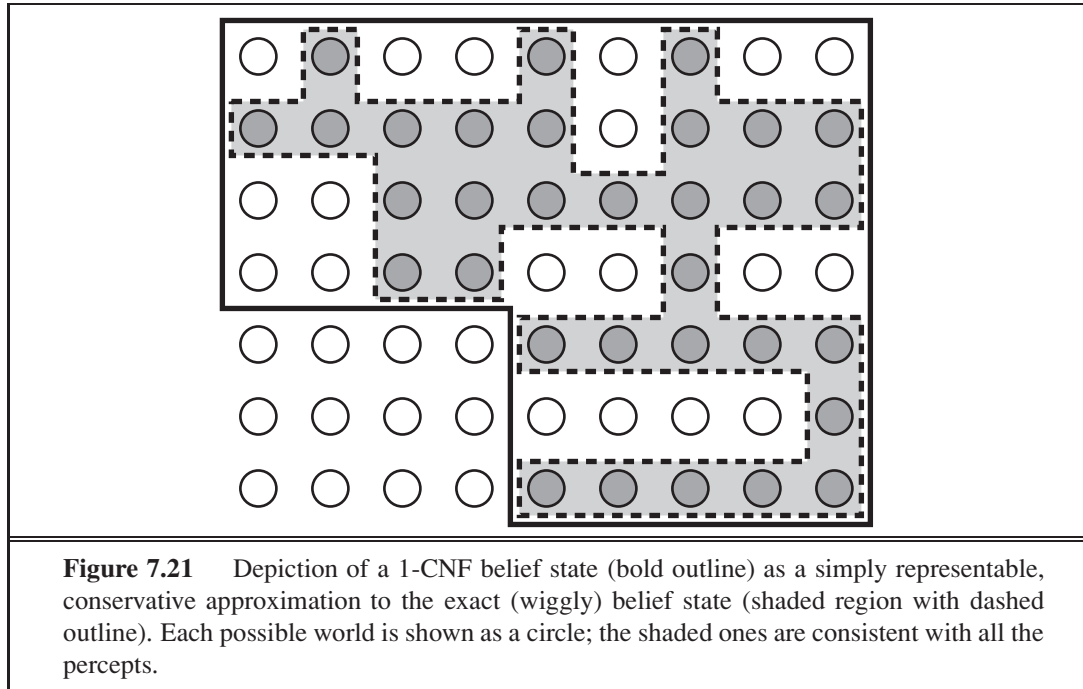
  problem  $\leftarrow \text{ROUTE-PROBLEM}(\text{current}, \text{goals}, \text{allowed})$ 
  return A*-GRAPH-SEARCH(problem)

```

**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.<sup>13</sup> 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  $\text{Forward}^0$  and  $\text{Shoot}^0$  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

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

“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  $100 \times 100$  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  $\alpha$  entails another sentence  $\beta$  if  $\beta$  is true in all worlds where  $\alpha$  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.
-

# CHAP 8 - FOL

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).<sup>2</sup> **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,

<sup>2</sup> 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).<sup>3</sup> 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

DOMAIN

DOMAIN ELEMENTS

<sup>3</sup> 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.

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

$$\{ \langle \text{Richard the Lionheart}, \text{King John} \rangle, \langle \text{King John}, \text{Richard the Lionheart} \rangle \}. \quad (8.1)$$

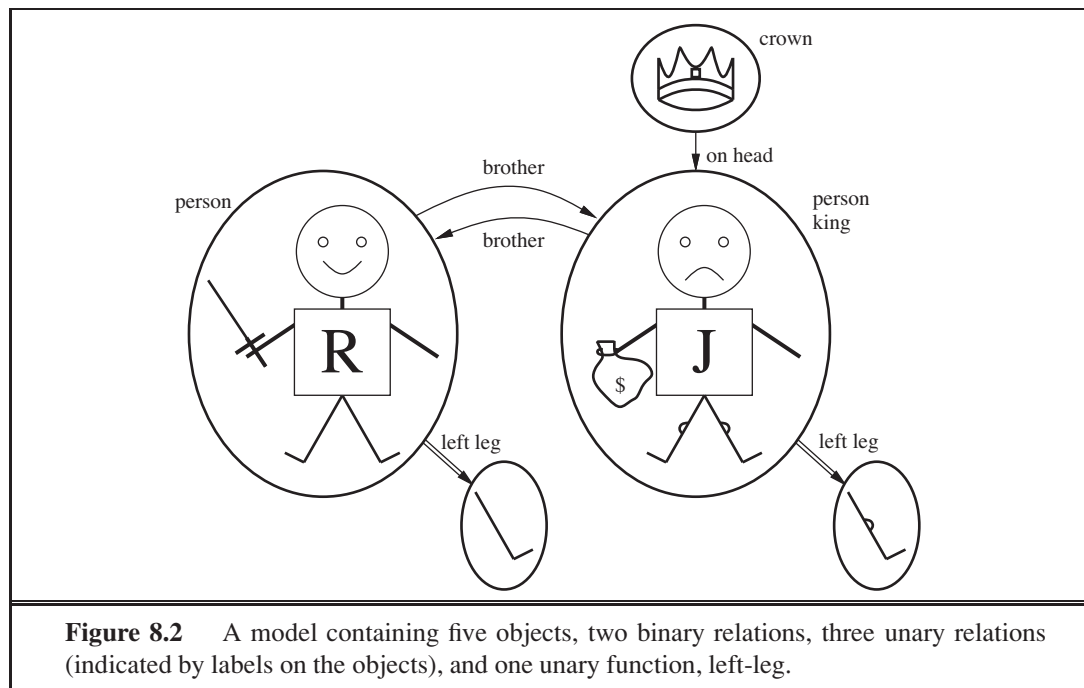
(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John's head, so the “on head” relation contains just one tuple,  $\langle \text{the crown}, \text{King John} \rangle$ . The “brother” and “on head” relations are **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:

$$\begin{aligned} \langle \text{Richard the Lionheart} \rangle &\rightarrow \text{Richard's left leg} \\ \langle \text{King John} \rangle &\rightarrow \text{John's left leg} \end{aligned} \quad (8.2)$$

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.

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.

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:

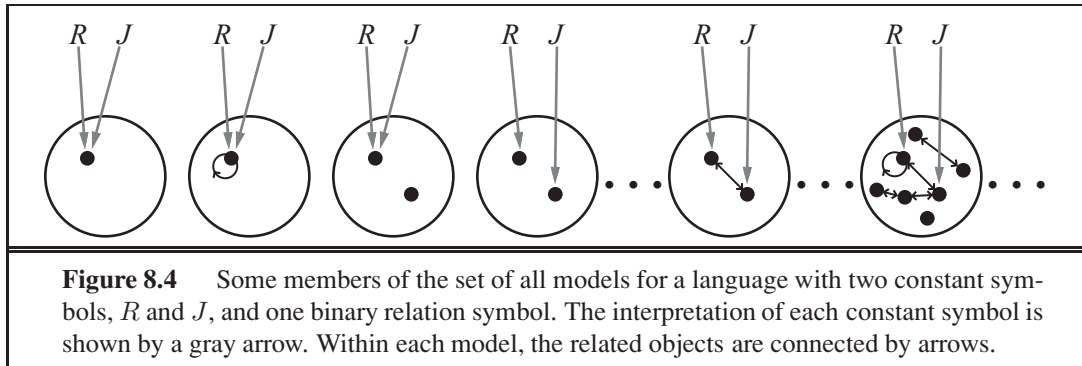
- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); *OnHead* refers to the “on head” relation that holds between the crown and King John; *Person*, *King*, and *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.<sup>4</sup> 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.

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

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

**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.



**Figure 8.4** Some members of the set of all models for a language with two constant symbols,  $R$  and  $J$ , and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

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.<sup>5</sup>

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**

<sup>5</sup>  $\lambda$ -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  $\lambda$ -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  $\lambda$  in this way does not increase the formal expressive power of first-order logic, because any sentence that includes a  $\lambda$ -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.<sup>6</sup> 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,  $\text{LeftLeg}(x)$ . 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

<sup>6</sup> 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.”<sup>7</sup>

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

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

<sup>7</sup> 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  $\text{Henry}$  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  $\text{Father}$  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.<sup>8</sup> 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:

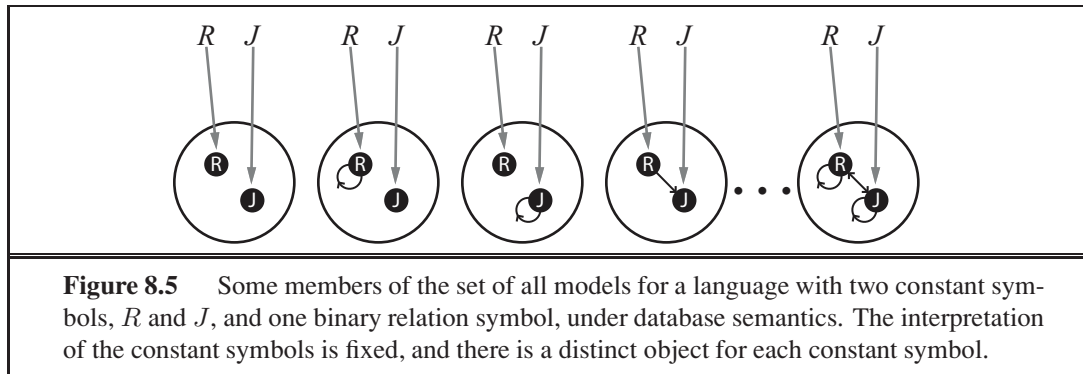
$$\begin{aligned} &\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) \wedge \text{John} \neq \text{Geoffrey} \\ &\wedge \forall x \text{ Brother}(x, \text{Richard}) \Rightarrow (x = \text{John} \vee x = \text{Geoffrey}) . \end{aligned}$$

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

<sup>8</sup> 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, **ASKVARS**, which we call with

$$\text{ASKVARS}(KB, \text{Person}(x))$$

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**. **ASKVARS** 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

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.

THEOREM

**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.<sup>9</sup> 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:

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

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

<sup>9</sup> 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

$$\text{Percept}([Stench, Breeze, Glitter, None, None], 5) .$$

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:

$$\begin{aligned} \forall t, s, g, m, c \quad \text{Percept}([s, Breeze, g, m, c], t) &\Rightarrow \text{Breeze}(t) , \\ \forall t, s, b, m, c \quad \text{Percept}([s, b, Glitter, m, c], t) &\Rightarrow \text{Glitter}(t) , \end{aligned}$$

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

$$\forall t \quad \text{Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t) .$$

# 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.<sup>1</sup> To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let  $\text{SUBST}(\theta, \alpha)$  denote the result of applying the substitution  $\theta$  to the sentence  $\alpha$ . Then the rule is written

$$\frac{\forall v \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

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  $\alpha$ , variable  $v$ , and constant symbol  $k$  that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \alpha}{\text{SUBST}(\{v/k\}, \alpha)} .$$

For example, from the sentence

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

we can infer the sentence

$$\text{Crown}(C_1) \wedge \text{OnHead}(C_1, John)$$

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  $\pi$ . 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  $\exists x \text{Kill}(x, \text{Victim})$  once we have added the sentence  $\text{Kill}(\text{Murderer}, \text{Victim})$ . 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

<sup>1</sup> 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  $\theta$  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  $\theta$ . 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**:<sup>2</sup> For atomic sentences  $p_i$ ,  $p_i'$ , and  $q$ , where there is a substitution  $\theta$

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  $\theta$  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  $\theta$ ,

$$p \models \text{SUBST}(\theta, p)$$

holds by Universal Instantiation. It holds in particular for a  $\theta$  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,  $\theta$  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

UNIFICATION

UNIFIER

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

<sup>2</sup> 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  $\alpha$  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  $\theta$ 

```

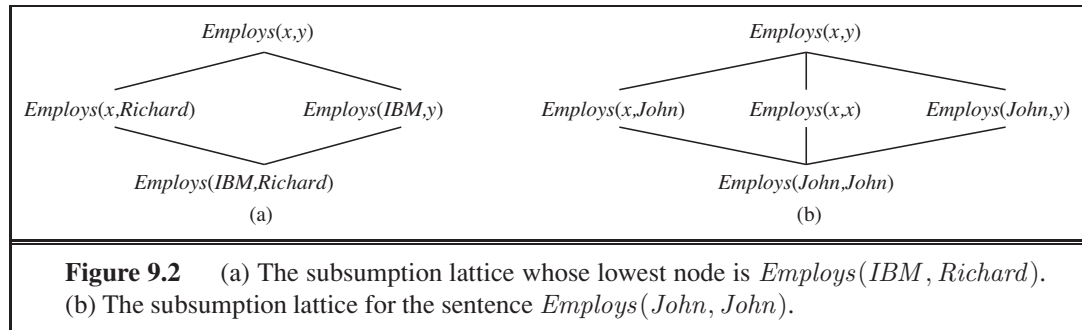
**Figure 9.1** The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution  $\theta$  that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established 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  $\exists x Owns(Nono, x) \wedge Missile(x)$  is transformed into two definite clauses by Existential Instantiation, introducing a new constant  $M_1$ :

$$Owns(Nono, M_1) \quad (9.4)$$

$$Missile(M_1) \quad (9.5)$$

“All of its missiles were sold to it by Colonel West”:

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$



and we must know that an enemy of America counts as “hostile”:

$$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$$

“West, who is American ...”:

$$American(West) . \quad (9.9)$$

“The country Nono, an enemy of America ...”:

$$Enemy(Nono, America) . \quad (9.10)$$

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,  $Likes(x, IceCream)$  and  $Likes(y, IceCream)$  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  $Sells(West, M_1, Nono)$  is added.  
Rule (9.7) is satisfied with  $\{x/M_1\}$ , and  $Weapon(M_1)$  is added.  
Rule (9.8) is satisfied with  $\{x/Nono\}$ , and  $Hostile(Nono)$  is added.
- On the second iteration, rule (9.3) is satisfied with  $\{x/West, y/M_1, z/Nono\}$ , and  $Criminal(West)$  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



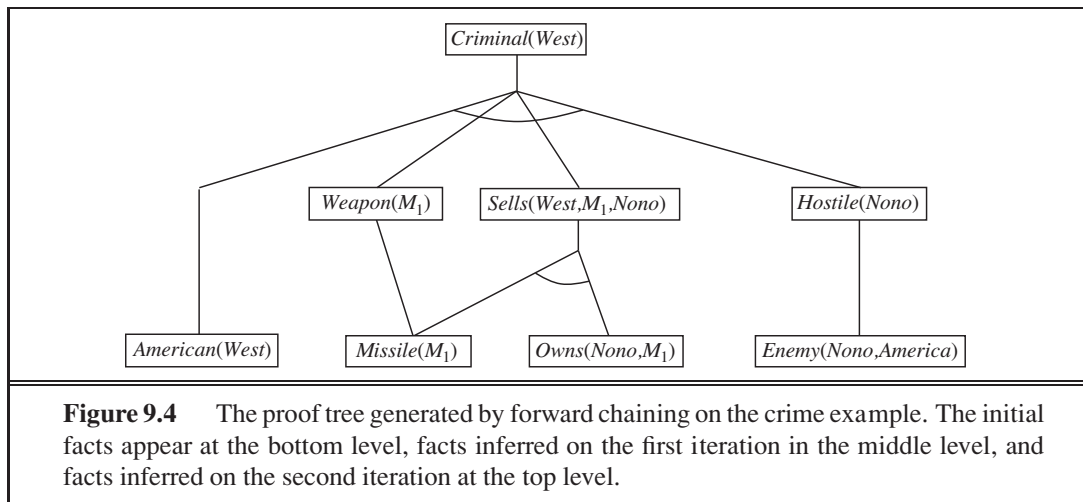
```

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
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(\text{rule})$ 
      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

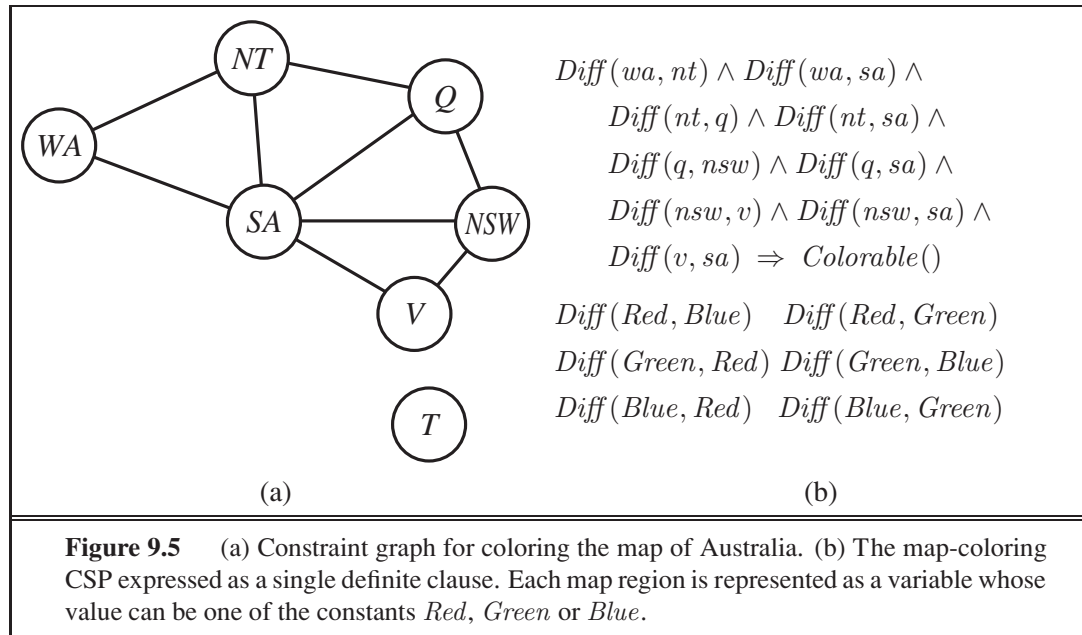
$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) .$$

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** algorithm<sup>3</sup> 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

RETE

<sup>3</sup> 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.<sup>4</sup> 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  $x$ :

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x).$$

<sup>4</sup> 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. FOL-BC-ASK(*KB*, *goal*) will be proved if the knowledge base contains a clause of the form *lhs*  $\Rightarrow$  *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  $\{x/John\}$  as well as with  $\{x/Richard\}$ . 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  $\theta'$ 

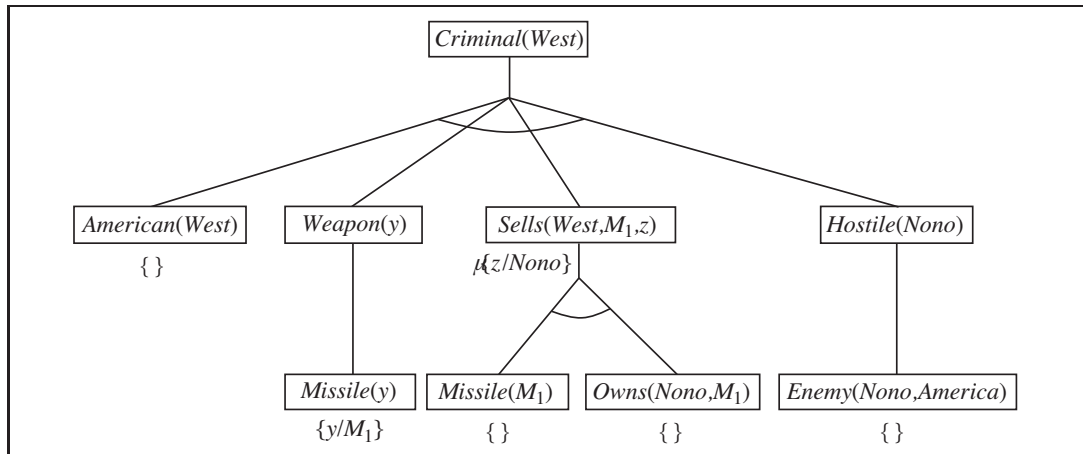


---


generator FOL-BC-AND( $KB, goals, \theta$ ) yields a substitution
  if  $\theta = failure$  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,

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

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.<sup>5</sup>

- 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.

CHOICE POINT

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.

TRAIL

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

<sup>5</sup> 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.

OPEN-CODE

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:

CONTINUATION

- 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.

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.

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).
```

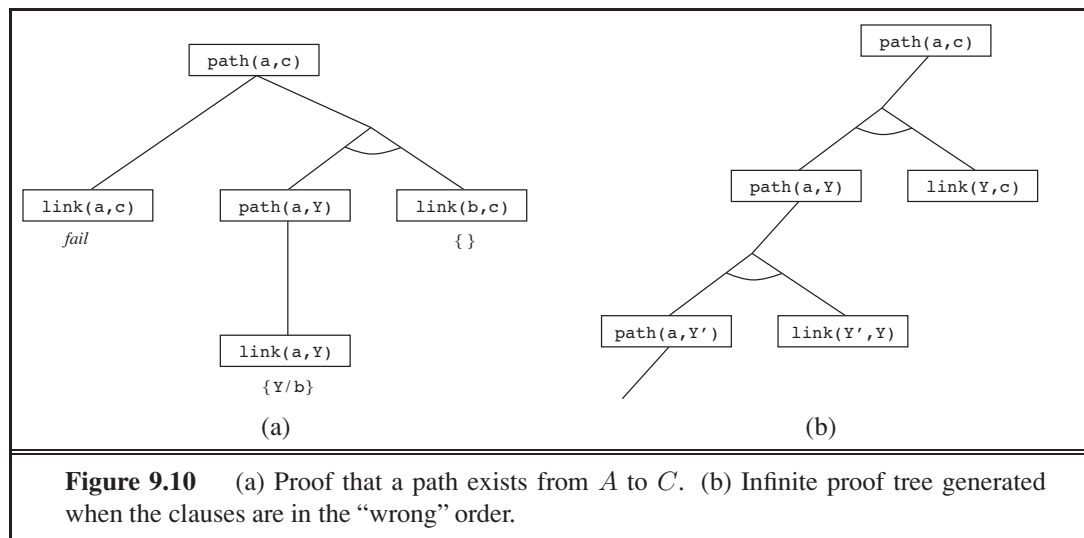
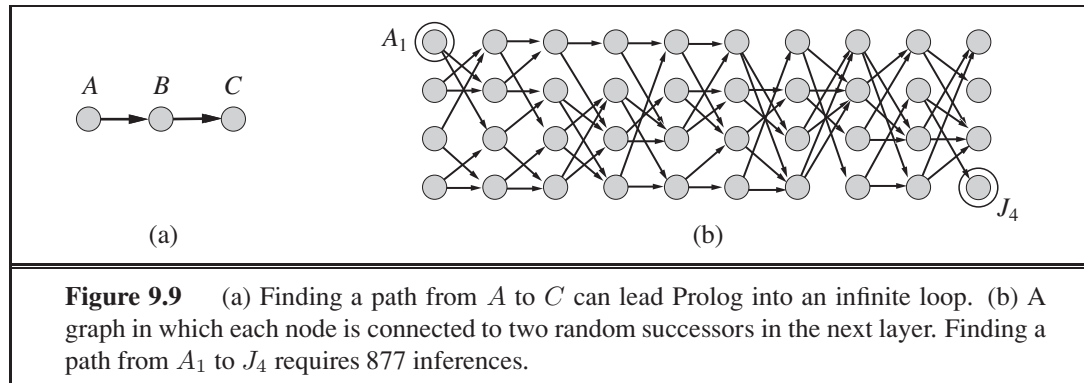
A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 9.9(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 9.10(a). On the other hand, if we put the two clauses in the order

```
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.

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  $Z \geq 1$ . 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.<sup>6</sup> 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.

<sup>6</sup> 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.



---

## 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

# 12 KNOWLEDGE REPRESENTATION

*In which we show how to use first-order logic to represent the most important aspects of the real world, such as action, space, time, thoughts, and shopping.*

The previous chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent’s knowledge base—how to represent facts about the world.

Section 12.1 introduces the idea of a general ontology, which organizes everything in the world into a hierarchy of categories. Section 12.2 covers the basic categories of objects, substances, and measures; Section 12.3 covers events, and Section 12.4 discusses knowledge about beliefs. We then return to consider the technology for reasoning with this content: Section 12.5 discusses reasoning systems designed for efficient inference with categories, and Section 12.6 discusses reasoning with default information. Section 12.7 brings all the knowledge together in the context of an Internet shopping environment.

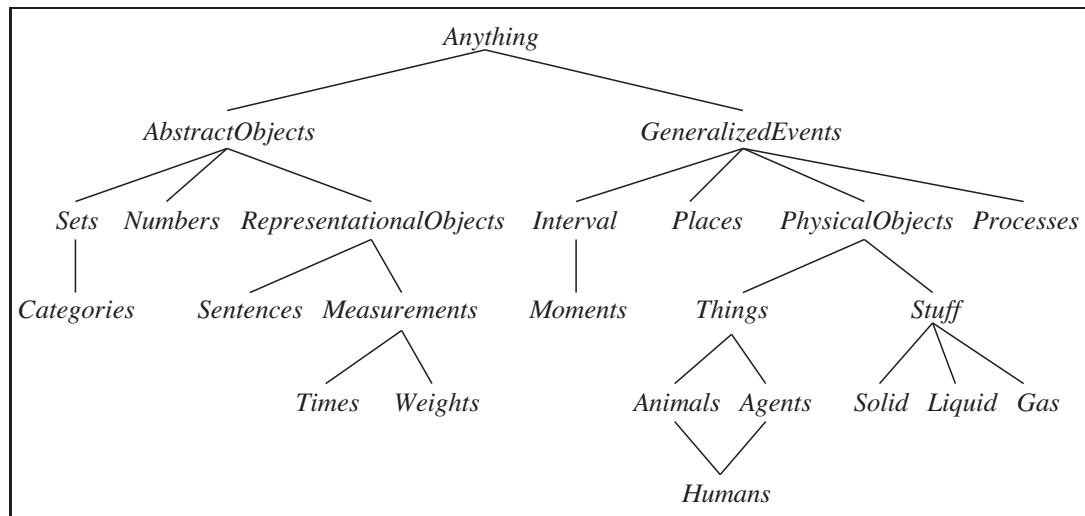
## 12.1 ONTOLOGICAL ENGINEERING

---

In “toy” domains, the choice of representation is not that important; many choices will work. Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Events, Time, Physical Objects, and Beliefs*—that occur in many different domains. *Representing these abstract concepts* is sometimes called **ontological engineering**.

ONTOLOGICAL  
ENGINEERING

The prospect of representing *everything* in the world is daunting. Of course, we won’t actually write a complete description of everything—that would be far too much for even a 1000-page textbook—but we will leave placeholders where new knowledge for any domain can fit in. For example, we will define what it means to be a physical object, and the details of different types of objects—robots, televisions, books, or whatever—can be filled in later. This is analogous to the way that designers of an object-oriented programming framework (such as the Java Swing graphical framework) define general concepts like *Window*, expecting users to



**Figure 12.1** The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example. We will see in Section 12.3.3 why physical objects come under generalized events.

#### UPPER ONTOLOGY

use these to define more specific concepts like *SpreadsheetWindow*. The general framework of concepts is called an **upper ontology** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure 12.1.

Before considering the ontology further, we should state one important caveat. We have elected to use first-order logic to discuss the content and organization of knowledge, although certain aspects of the real world are hard to capture in FOL. The principal difficulty is that most generalizations have exceptions or hold only to a degree. For example, although “tomatoes are red” is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the rules in this chapter. The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we delay the discussion of exceptions until Section 12.5 of this chapter, and the more general topic of reasoning with uncertainty until Chapter 13.

Of what use is an upper ontology? Consider the ontology for circuits in Section 8.4.2. It makes many simplifying assumptions: time is omitted completely; signals are fixed and do not propagate; the structure of the circuit remains constant. A more general ontology would consider signals at particular times, and would include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers. We could also introduce more interesting classes of gates, for example, by describing the technology (TTL, CMOS, and so on) as well as the input–output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to represent where the wires are on the board.

If we look at the wumpus world, similar considerations apply. Although we do represent time, it has a simple structure: Nothing happens except when the agent acts, and all changes are instantaneous. A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. We also used a *Pit* predicate to say which squares have pits. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits, each having different properties. Similarly, we might want to allow for other animals besides wumpuses. It might not be possible to pin down the exact species from the available percepts, so we would need to build up a biological taxonomy to help the agent predict the behavior of cave-dwellers from scanty clues.

For any special-purpose ontology, it is possible to make changes like these to move toward greater generality. An obvious question then arises: do all these ontologies converge on a general-purpose ontology? After centuries of philosophical and computational investigation, the answer is “Maybe.” In this section, we present one general-purpose ontology that synthesizes ideas from those centuries. Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that no representational issue can be finessed or brushed under the carpet.
- In any sufficiently demanding domain, different areas of knowledge must be *unified*, because reasoning and problem solving could involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters.

We should say up front that the enterprise of general ontological engineering has so far had only limited success. None of the top AI applications (as listed in Chapter 1) make use of a shared ontology—they all use special-purpose knowledge engineering. Social/political considerations can make it difficult for competing parties to agree on an ontology. As Tom Gruber (2004) says, “Every ontology is a treaty—a social agreement—among people with some common motive in sharing.” When competing concerns outweigh the motivation for sharing, there can be no common ontology. Those ontologies that do exist have been created along four routes:

1. By a team of trained ontologist/logicians, who architect the ontology and write axioms. The CYC system was mostly built this way (Lenat and Guha, 1990).
2. By importing categories, attributes, and values from an existing database or databases. DBPEDIA was built by importing structured facts from Wikipedia (Bizer *et al.*, 2007).
3. By parsing text documents and extracting information from them. TEXTRUNNER was built by reading a large corpus of Web pages (Banko and Etzioni, 2008).
4. By enticing unskilled amateurs to enter commonsense knowledge. The OPENMIND system was built by volunteers who proposed facts in English (Singh *et al.*, 2002; Chklovski and Gil, 2005).

## 12.2 CATEGORIES AND OBJECTS

CATEGORY



The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories*. For example, a shopper would normally have the goal of buying a basketball, rather than a *particular* basketball such as  $BB_9$ . Categories also serve to make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, black seeds, and presence in the fruit aisle, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

REIFICATION

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate  $Basketball(b)$ , or we can **reify**<sup>1</sup> the category as an object,  $Basketballs$ . We could then say  $Member(b, Basketballs)$ , which we will abbreviate as  $b \in Basketballs$ , to say that  $b$  is a member of the category of basketballs. We say  $Subset(Basketballs, Balls)$ , abbreviated as  $Basketballs \subset Balls$ , to say that  $Basketballs$  is a **subcategory** of  $Balls$ . We will use subcategory, subclass, and subset interchangeably.

SUBCATEGORY

INHERITANCE

Categories serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category  $Food$  are edible, and if we assert that  $Fruit$  is a subclass of  $Food$  and  $Apples$  is a subclass of  $Fruit$ , then we can infer that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the  $Food$  category.

TAXONOMY

Subclass relations organize categories into a **taxonomy**, or **taxonomic hierarchy**. Taxonomies have been used explicitly for centuries in technical fields. The largest such taxonomy organizes about 10 million living and extinct species, many of them beetles,<sup>2</sup> into a single hierarchy; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some types of facts, with examples of each:

- An object is a member of a category.  
 $BB_9 \in Basketballs$
- A category is a subclass of another category.  
 $Basketballs \subset Balls$
- All members of a category have some properties.  
 $(x \in Basketballs) \Rightarrow Spherical(x)$

<sup>1</sup> Turning a proposition into an object is called **reification**, from the Latin word *res*, or thing. John McCarthy proposed the term “thingification,” but it never caught on.

<sup>2</sup> The famous biologist J. B. S. Haldane deduced “An inordinate fondness for beetles” on the part of the Creator.

- Members of a category can be recognized by some properties.  
 $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties.  
 $Dogs \in DomesticatedSpecies$

Notice that because *Dogs* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories. Of course there are exceptions to many of the above rules (punctured basketballs are not spherical); we deal with these exceptions later.

Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that *Males* and *Females* are subclasses of *Animals*, then we have not said that a male cannot be a female. We say that **two or more categories are disjoint if they have no members in common**. And even if we know that males and females are disjoint, we will not know that an animal that is not a male must be a female, unless we say that males and females constitute an **exhaustive decomposition** of the animals. A **disjoint exhaustive decomposition is known as a partition**. The following examples illustrate these three concepts:

DISJOINT

EXHAUSTIVE  
DECOMPOSITION  
PARTITION

$Disjoint(\{Animals, Vegetables\})$   
 $ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\},$   
 $NorthAmericans)$   
 $Partition(\{Males, Females\}, Animals) .$

(Note that the *ExhaustiveDecomposition* of *NorthAmericans* is not a *Partition*, because some people have dual citizenship.) The three predicates are defined as follows:

$Disjoint(s) \Leftrightarrow (\forall c_1, c_2 \ c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersection(c_1, c_2) = \{ \})$   
 $ExhaustiveDecomposition(s, c) \Leftrightarrow (\forall i \ i \in c \Leftrightarrow \exists c_2 \ c_2 \in s \wedge i \in c_2)$   
 $Partition(s, c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s, c) .$

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

$x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males .$

As we discuss in the sidebar on natural kinds on page 443, strict logical definitions for categories are neither always possible nor always necessary.

### 12.2.1 Physical composition

The **idea that one object can be part of another** is a familiar one. One's nose is part of one's head, Romania is part of Europe, and this chapter is part of this book. We use the general **PartOf** relation to say that **one thing is part of another**. Objects can be grouped into *PartOf* hierarchies, reminiscent of the *Subset* hierarchy:

$PartOf(Bucharest, Romania)$   
 $PartOf(Romania, EasternEurope)$   
 $PartOf(EasternEurope, Europe)$   
 $PartOf(Europe, Earth) .$



The *PartOf* relation is transitive and reflexive; that is,

$$\begin{aligned} \text{PartOf}(x, y) \wedge \text{PartOf}(y, z) &\Rightarrow \text{PartOf}(x, z) . \\ \text{PartOf}(x, x) & . \end{aligned}$$

Therefore, we can conclude  $\text{PartOf}(\text{Bucharest}, \text{Earth})$ .

COMPOSITE OBJECT

Categories of **composite objects** are often characterized by structural relations among parts. For example, a biped has two legs attached to a body:

$$\begin{aligned} \text{Biped}(a) \Rightarrow \exists l_1, l_2, b \quad &\text{Leg}(l_1) \wedge \text{Leg}(l_2) \wedge \text{Body}(b) \wedge \\ &\text{PartOf}(l_1, a) \wedge \text{PartOf}(l_2, a) \wedge \text{PartOf}(b, a) \wedge \\ &\text{Attached}(l_1, b) \wedge \text{Attached}(l_2, b) \wedge \\ &l_1 \neq l_2 \wedge [\forall l_3 \quad \text{Leg}(l_3) \wedge \text{PartOf}(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)] . \end{aligned}$$

The notation for “exactly two” is a little awkward; we are forced to say that there are two legs, that they are not the same, and that if anyone proposes a third leg, it must be the same as one of the other two. In Section 12.5.2, we describe a formalism called description logic makes it easier to represent constraints like “exactly two.”

We can define a *PartPartition* relation analogous to the *Partition* relation for categories. (See Exercise 12.8.) An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts. Notice that this is not the case with categories, which have no mass, even though their elements might.

It is also useful to define composite objects with definite parts but no particular structure. For example, we might want to say “The apples in this bag weigh two pounds.” The temptation would be to ascribe this weight to the *set* of apples in the bag, but this would be a mistake because the set is an abstract mathematical concept that has elements but does not have weight. Instead, we need a new concept, which we will call a **bunch**. For example, if the apples are  $\text{Apple}_1$ ,  $\text{Apple}_2$ , and  $\text{Apple}_3$ , then

BUNCH

$$\text{BunchOf}(\{\text{Apple}_1, \text{Apple}_2, \text{Apple}_3\})$$

denotes the composite object with the three apples as parts (not elements). We can then use the bunch as a normal, albeit unstructured, object. Notice that  $\text{BunchOf}(\{x\}) = x$ . Furthermore,  $\text{BunchOf}(\text{Apples})$  is the composite object consisting of all apples—not to be confused with *Apples*, the category or set of all apples.

We can define *BunchOf* in terms of the *PartOf* relation. Obviously, each element of  $s$  is part of  $\text{BunchOf}(s)$ :

$$\forall x \quad x \in s \Rightarrow \text{PartOf}(x, \text{BunchOf}(s)) .$$

Furthermore,  $\text{BunchOf}(s)$  is the smallest object satisfying this condition. In other words,  $\text{BunchOf}(s)$  must be part of any object that has all the elements of  $s$  as parts:

$$\forall y \quad [\forall x \quad x \in s \Rightarrow \text{PartOf}(x, y)] \Rightarrow \text{PartOf}(\text{BunchOf}(s), y) .$$


LOGICAL MINIMIZATION

These axioms are an example of a general technique called **logical minimization**, which means defining an object as the smallest one satisfying certain conditions.



### 12.2.2 Measurements

MEASURE

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**. Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract “measure objects,” such as the *length* that is the length of this line segment: . We can call this length 1.5 inches or 3.81 centimeters. Thus,

UNITS FUNCTION

the same length has different names in our language. We represent the length with a **units function** that takes a number as argument. (An alternative scheme is explored in Exercise 12.9.) If the line segment is called  $L_1$ , we can write

$$Length(L_1) = Inches(1.5) = Centimeters(3.81) .$$

Conversion between units is done by equating multiples of one unit to another:

$$Centimeters(2.54 \times d) = Inches(d) .$$

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

$$Diameter(Basketball_{12}) = Inches(9.5) .$$

$$ListPrice(Basketball_{12}) = \$(19) .$$

$$d \in Days \Rightarrow Duration(d) = Hours(24) .$$

Note that  $\$(1)$  is *not* a dollar bill! One can have two dollar bills, but there is only one object named  $\$(1)$ . Note also that, while  $Inches(0)$  and  $Centimeters(0)$  refer to the same zero length, they are not identical to other zero measures, such as  $Seconds(0)$ .

Simple, quantitative measures are easy to represent. Other measures present more of a problem, because they have no agreed scale of values. Exercises have difficulty, desserts have deliciousness, and poems have beauty, yet numbers cannot be assigned to these qualities. One might, in a moment of pure accountancy, dismiss such properties as useless for the purpose of logical reasoning; or, still worse, attempt to impose a numerical scale on beauty. This would be a grave mistake, because it is unnecessary. The most important aspect of measures is not the particular numerical values, but the fact that measures can be ordered.

Although measures are not numbers, we can still compare them, using an ordering symbol such as  $>$ . For example, we might well believe that Norvig’s exercises are tougher than Russell’s, and that one scores less on tougher exercises:

$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Wrote(Norvig, e_1) \wedge Wrote(Russell, e_2) \Rightarrow \\ Difficulty(e_1) > Difficulty(e_2) .$$

$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Difficulty(e_1) > Difficulty(e_2) \Rightarrow \\ ExpectedScore(e_1) < ExpectedScore(e_2) .$$

This is enough to allow one to decide which exercises to do, even though no numerical values for difficulty were ever used. (One does, however, have to discover who wrote which exercises.) These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without plunging into detailed equations and numerical simulations. Qualitative physics is discussed in the historical notes section.

### 12.2.3 Objects: Things and stuff

INDIVIDUATION

STUFF

The real world can be seen as consisting of primitive objects (e.g., atomic particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually. There is, however, a significant portion of reality that seems to defy any obvious **individuation**—division into distinct objects. We give this portion the generic name **stuff**. For example, suppose I have some butter and an aardvark in front of me. I can say there is one aardvark, but there is no obvious number of “butter-objects,” because any part of a butter-object is also a butter-object, at least until we get to very small parts indeed. This is the major distinction between *stuff* and *things*. If we cut an aardvark in half, we do not get two aardvarks (unfortunately).

COUNT NOUNS

MASS NOUN

The English language distinguishes clearly between *stuff* and *things*. We say “an aardvark,” but, except in pretentious California restaurants, one cannot say “a butter.” Linguists distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy. Several competing ontologies claim to handle this distinction. Here we describe just one; the others are covered in the historical notes section.

To represent *stuff* properly, we begin with the obvious. We need to have as objects in our ontology at least the gross “lumps” of *stuff* we interact with. For example, we might recognize a lump of butter as the one left on the table the night before; we might pick it up, weigh it, sell it, or whatever. In these senses, it is an object just like the aardvark. Let us call it *Butter*<sub>3</sub>. We also define the category *Butter*. Informally, its elements will be all those things of which one might say “It’s butter,” including *Butter*<sub>3</sub>. With some caveats about very small parts that we omit for now, any part of a butter-object is also a butter-object:

$$b \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter .$$

We can now say that butter melts at around 30 degrees centigrade:

$$b \in Butter \Rightarrow MeltingPoint(b, Centigrade(30)) .$$

We could go on to say that butter is yellow, is less dense than water, is soft at room temperature, has a high fat content, and so on. On the other hand, **butter has no particular size, shape, or weight**. We can define more specialized categories of butter such as *UnsaltedButter*, which is also a kind of *stuff*. Note that the category *PoundOfButter*, which includes as members all butter-objects weighing one pound, is not a kind of *stuff*. If we cut a pound of butter in half, we do not, alas, get two pounds of butter.

INTRINSIC

EXTRINSIC

What is actually going on is this: some properties are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut an instance of *stuff* in half, the **two pieces retain the intrinsic properties**—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, their **extrinsic properties**—weight, length, shape, and so on—are not retained under subdivision. A category of objects that includes in its definition **only intrinsic properties** is then a substance, or mass noun; a class that includes **any extrinsic properties** in its definition is a count noun. The category *Stuff* is the most general substance category, specifying no intrinsic properties. The category *Thing* is the most general discrete object category, specifying no extrinsic properties.

### 12.3 EVENTS

#### EVENT CALCULUS

In Section 10.4.2, we showed how situation calculus represents actions and their effects. Situation calculus is limited in its applicability: it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time. Consider a continuous action, such as filling a bathtub. Situation calculus can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens *during* the action. It also can't describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an alternative formalism known as **event calculus**, which is based on points of time rather than on situations.<sup>3</sup>

Event calculus reifies fluents and events. The fluent  $At(Shankar, Berkeley)$  is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true. To assert that a fluent is actually true at some point in time we use the predicate  $T$ , as in  $T(At(Shankar, Berkeley), t)$ .

Events are described as instances of event categories.<sup>4</sup> The event  $E_1$  of Shankar flying from San Francisco to Washington, D.C. is described as

$$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, DC).$$

If this is too verbose, we can define an alternative three-argument version of the category of flying events and say

$$E_1 \in Flyings(Shankar, SF, DC).$$

We then use  $Happens(E_1, i)$  to say that the event  $E_1$  took place over the time interval  $i$ , and we say the same thing in functional form with  $Extent(E_1) = i$ . We represent time intervals by a (start, end) pair of times; that is,  $i = (t_1, t_2)$  is the time interval that starts at  $t_1$  and ends at  $t_2$ . The complete set of predicates for one version of the event calculus is

$T(f, t)$	Fluent $f$ is true at time $t$
$Happens(e, i)$	Event $e$ happens over the time interval $i$
$Initiates(e, f, t)$	Event $e$ causes fluent $f$ to start to hold at time $t$
$Terminates(e, f, t)$	Event $e$ causes fluent $f$ to cease to hold at time $t$
$Clipped(f, i)$	Fluent $f$ ceases to be true at some point during time interval $i$
$Restored(f, i)$	Fluent $f$ becomes true sometime during time interval $i$

We assume a distinguished event, *Start*, that describes the initial state by saying which fluents are initiated or terminated at the start time. We define  $T$  by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and

<sup>3</sup> The terms “event” and “action” may be used interchangeably. Informally, “action” connotes an agent while “event” connotes the possibility of agentless actions.

<sup>4</sup> Some versions of event calculus do not distinguish event categories from instances of the categories.

not made true (restored) by another event. Formally, the axioms are:

$$\begin{aligned} & \text{Happens}(e, (t_1, t_2)) \wedge \text{Initiates}(e, f, t_1) \wedge \neg \text{Clipped}(f, (t_1, t)) \wedge t_1 < t \Rightarrow \\ & \quad T(f, t) \\ & \text{Happens}(e, (t_1, t_2)) \wedge \text{Terminates}(e, f, t_1) \wedge \neg \text{Restored}(f, (t_1, t)) \wedge t_1 < t \Rightarrow \\ & \quad \neg T(f, t) \end{aligned}$$

where *Clipped* and *Restored* are defined by

$$\begin{aligned} \text{Clipped}(f, (t_1, t_2)) & \Leftrightarrow \\ & \exists e, t, t_3 \text{ Happens}(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge \text{Terminates}(e, f, t) \\ \text{Restored}(f, (t_1, t_2)) & \Leftrightarrow \\ & \exists e, t, t_3 \text{ Happens}(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge \text{Initiates}(e, f, t) \end{aligned}$$

It is convenient to extend *T* to work over intervals as well as time points; a fluent holds over an interval if it holds on every point within the interval:

$$T(f, (t_1, t_2)) \Leftrightarrow [\forall t (t_1 \leq t < t_2) \Rightarrow T(f, t)]$$

Fluents and actions are defined with domain-specific axioms that are similar to successor-state axioms. For example, we can say that the only way a wumpus-world agent gets an arrow is at the start, and the only way to use up an arrow is to shoot it:

$$\begin{aligned} \text{Initiates}(e, \text{HaveArrow}(a), t) & \Leftrightarrow e = \text{Start} \\ \text{Terminates}(e, \text{HaveArrow}(a), t) & \Leftrightarrow e \in \text{Shootings}(a) \end{aligned}$$

By reifying events we make it possible to add any amount of arbitrary information about them. For example, we can say that Shankar's flight was bumpy with *Bumpy*( $E_1$ ). In an ontology where events are  $n$ -ary predicates, there would be no way to add extra information like this; moving to an  $n + 1$ -ary predicate isn't a scalable solution.

We can extend event calculus to make it possible to represent simultaneous events (such as two people being necessary to ride a seesaw), exogenous events (such as the wind blowing and changing the location of an object), continuous events (such as the level of water in the bathtub continuously rising) and other complications.

### 12.3.1 Processes

DISCRETE EVENTS

The events we have seen so far are what we call **discrete events**—they have a definite structure. Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be something different—it would not be a trip from San Francisco to Washington, but instead a trip from San Francisco to somewhere over Kansas. On the other hand, the category of events denoted by *Flyings* has a different quality. If we take a small interval of Shankar's flight, say, the third 20-minute segment (while he waits anxiously for a bag of peanuts), that event is still a member of *Flyings*. In fact, this is true for any subinterval.

PROCESS

LIQUID EVENT

Categories of events with this property are called **process categories** or **liquid event categories**. Any process  $e$  that happens over an interval also happens over any subinterval:

$$(e \in \text{Processes}) \wedge \text{Happens}(e, (t_1, t_4)) \wedge (t_1 < t_2 < t_3 < t_4) \Rightarrow \text{Happens}(e, (t_2, t_3)).$$

The distinction between liquid and nonliquid events is exactly analogous to the difference between substances, or *stuff*, and individual objects, or *things*. In fact, some have called liquid events **temporal substances**, whereas substances like butter are **spatial substances**.

TEMPORAL  
SUBSTANCE

SPATIAL SUBSTANCE

### 12.3.2 Time intervals

Event calculus opens us up to the possibility of talking about time, and time intervals. We will consider two kinds of time intervals: **moments and extended intervals**. The distinction is that only **moments have zero duration**:

$$\begin{aligned} & \text{Partition}(\{\text{Moments}, \text{ExtendedIntervals}\}, \text{Intervals}) \\ & i \in \text{Moments} \Leftrightarrow \text{Duration}(i) = \text{Seconds}(0) . \end{aligned}$$

Next we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we measure it in seconds and say that the moment at midnight (GMT) on January 1, 1900, has time 0. The functions **Begin** and **End** pick out the earliest and latest moments in an interval, and the function **Time** delivers the point on the time scale for a moment. The function **Duration** gives the difference between the end time and the start time.

$$\begin{aligned} & \text{Interval}(i) \Rightarrow \text{Duration}(i) = (\text{Time}(\text{End}(i)) - \text{Time}(\text{Begin}(i))) . \\ & \text{Time}(\text{Begin}(\text{AD1900})) = \text{Seconds}(0) . \\ & \text{Time}(\text{Begin}(\text{AD2001})) = \text{Seconds}(3187324800) . \\ & \text{Time}(\text{End}(\text{AD2001})) = \text{Seconds}(3218860800) . \\ & \text{Duration}(\text{AD2001}) = \text{Seconds}(31536000) . \end{aligned}$$

To make these numbers easier to read, we also introduce a function *Date*, which takes six arguments (hours, minutes, seconds, day, month, and year) and returns a time point:

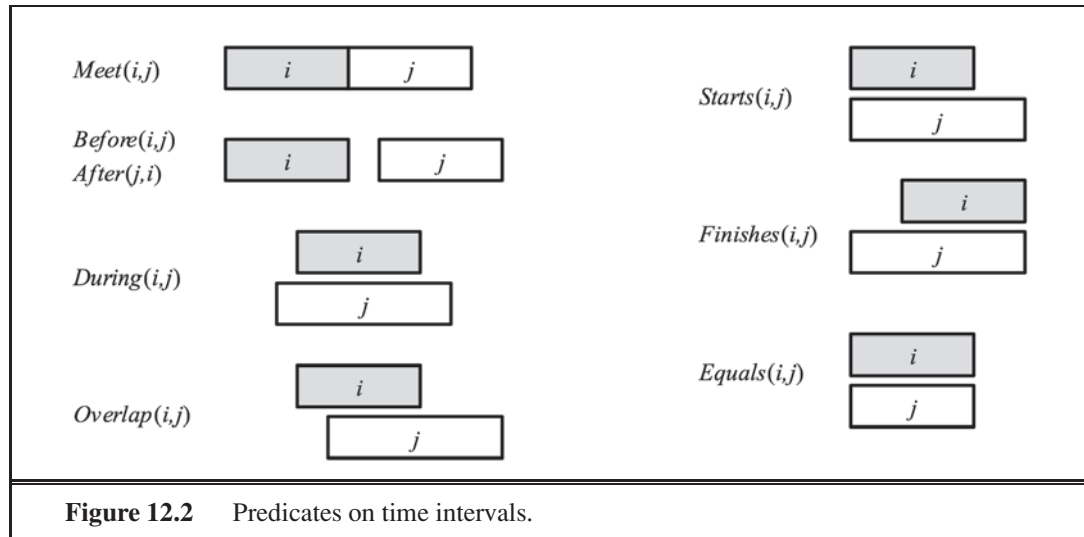
$$\begin{aligned} & \text{Time}(\text{Begin}(\text{AD2001})) = \text{Date}(0, 0, 0, 1, \text{Jan}, 2001) \\ & \text{Date}(0, 20, 21, 24, 1, 1995) = \text{Seconds}(3000000000) . \end{aligned}$$

Two intervals **Meet** if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

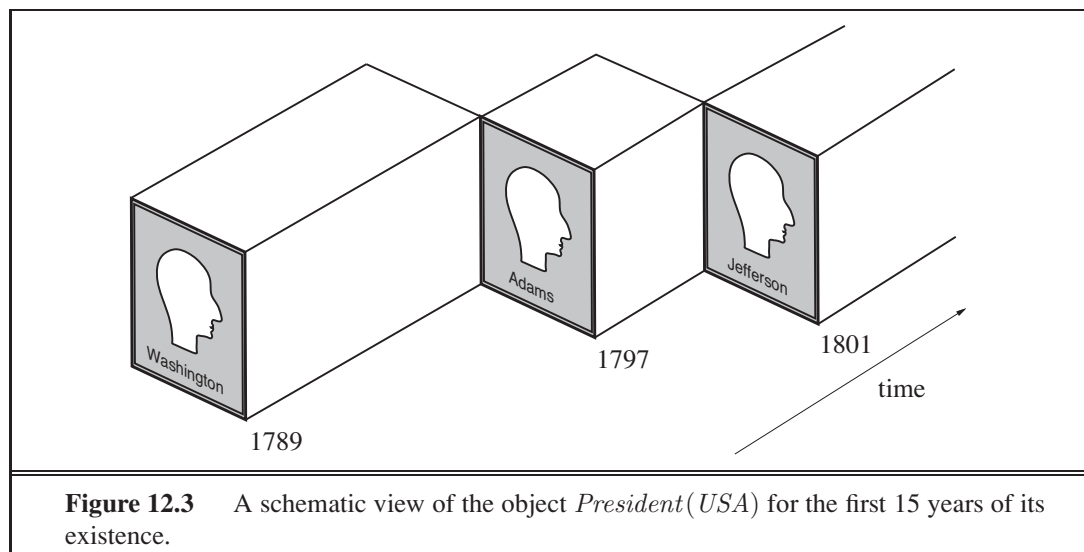
$$\begin{aligned} \text{Meet}(i, j) & \Leftrightarrow \text{End}(i) = \text{Begin}(j) \\ \text{Before}(i, j) & \Leftrightarrow \text{End}(i) < \text{Begin}(j) \\ \text{After}(j, i) & \Leftrightarrow \text{Before}(i, j) \\ \text{During}(i, j) & \Leftrightarrow \text{Begin}(j) < \text{Begin}(i) < \text{End}(i) < \text{End}(j) \\ \text{Overlap}(i, j) & \Leftrightarrow \text{Begin}(i) < \text{Begin}(j) < \text{End}(i) < \text{End}(j) \\ \text{Begins}(i, j) & \Leftrightarrow \text{Begin}(i) = \text{Begin}(j) \\ \text{Finishes}(i, j) & \Leftrightarrow \text{End}(i) = \text{End}(j) \\ \text{Equals}(i, j) & \Leftrightarrow \text{Begin}(i) = \text{Begin}(j) \wedge \text{End}(i) = \text{End}(j) \end{aligned}$$

These all have their intuitive meaning, with the exception of *Overlap*: we tend to think of overlap as symmetric (if *i* overlaps *j* then *j* overlaps *i*), but in this definition, **Overlap(*i*, *j*)** only holds if *i* begins before *j*. To say that the reign of Elizabeth II immediately followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$\begin{aligned} & \text{Meets}(\text{ReignOf}(\text{George VI}), \text{ReignOf}(\text{Elizabeth II})) . \\ & \text{Overlap}(\text{Fifties}, \text{ReignOf}(\text{Elvis})) . \\ & \text{Begin}(\text{Fifties}) = \text{Begin}(\text{AD1950}) . \\ & \text{End}(\text{Fifties}) = \text{End}(\text{AD1959}) . \end{aligned}$$



**Figure 12.2** Predicates on time intervals.



**Figure 12.3** A schematic view of the object *President(USA)* for the first 15 years of its existence.

### 12.3.3 Fluents and objects

Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time. For example, *USA* can be thought of as an event that began in, say, 1776 as a union of 13 states and is still in progress today as a union of 50. We can describe the changing properties of *USA* using state fluents, such as *Population(USA)*. A property of the USA that changes every four or eight years, barring mishaps, is its president. One might propose that *President(USA)* is a logical term that denotes a different object at different times. Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term *President(USA, t)* can denote different objects, depending on the value of  $t$ , but our ontology keeps time indices separate from fluents.) The



only possibility is that *President(USA)* denotes a single object that consists of different people at different times. It is the object that is George Washington from 1789 to 1797, John Adams from 1797 to 1801, and so on, as in Figure 12.3. To say that George Washington was president throughout 1790, we can write

$$T(\text{Equals}(\text{President}(\text{USA}), \text{GeorgeWashington}), \text{AD1790}) .$$

We use the function symbol *Equals* rather than the standard logical predicate  $=$ , because we cannot have a predicate as an argument to *T*, and because the interpretation is *not* that *GeorgeWashington* and *President(USA)* are logically identical in 1790; logical identity is not something that can change over time. The identity is between the subevents of each object that are defined by the period 1790.

## 12.4 MENTAL EVENTS AND MENTAL OBJECTS

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge *about* beliefs or *about* deduction. Knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, suppose Alice asks "what is the square root of 1764" and Bob replies "I don't know." If Alice insists "think harder," Bob should realize that with some more thought, this question can in fact be answered. On the other hand, if the question were "Is your mother sitting down right now?" then Bob should realize that thinking harder is unlikely to help. Knowledge about the knowledge of other agents is also important; Bob should realize that his mother knows whether she is sitting or not, and that asking her would be a way to find out.

What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

PROPOSITIONAL  
ATTITUDE

We begin with the **propositional attitudes** that an agent can have toward mental objects: attitudes such as *Believes*, *Knows*, *Wants*, *Intends*, and *Informs*. The difficulty is that these attitudes do not behave like "normal" predicates. For example, suppose we try to assert that Lois knows that Superman can fly:

$$\text{Knows}(\text{Lois}, \text{CanFly}(\text{Superman})) .$$

One minor issue with this is that we normally think of *CanFly(Superman)* as a sentence, but here it appears as a term. That issue can be patched up just by reifying *CanFly(Superman)*; making it a fluent. A more serious problem is that, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly:

$$\begin{aligned} &(\text{Superman} = \text{Clark}) \wedge \text{Knows}(\text{Lois}, \text{CanFly}(\text{Superman})) \\ &\models \text{Knows}(\text{Lois}, \text{CanFly}(\text{Clark})) . \end{aligned}$$

This is a consequence of the fact that equality reasoning is built into logic. Normally that is a good thing; if our agent knows that  $2 + 2 = 4$  and  $4 < 5$ , then we want our agent to know



REFERENTIAL  
TRANSPARENCY

that  $2 + 2 < 5$ . This property is called **referential transparency**—it doesn't matter what term a logic uses to refer to an object, what matters is the object that the term names. But for propositional attitudes like *believes* and *knows*, we would like to have **referential opacity**—the terms used *do* matter, because not all agents know which terms are co-referential.

MODAL LOGIC

**Modal logic** is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express “ $P$  is true.” Modal logic includes special modal operators that take sentences (rather than terms) as arguments. For example, “ $A$  knows  $P$ ” is represented with the notation  $\mathbf{K}_A P$ , where  $\mathbf{K}$  is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators.

POSSIBLE WORLD  
ACCESSIBILITY  
RELATIONS

The semantics of modal logic is more complicated. In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of **possible worlds** rather than just one true world. The worlds are connected in a graph by **accessibility relations**, one relation for each modal operator. We say that world  $w_1$  is accessible from world  $w_0$  with respect to the modal operator  $\mathbf{K}_A$  if everything in  $w_1$  is consistent with what  $A$  knows in  $w_0$ , and we write this as  $Acc(\mathbf{K}_A, w_0, w_1)$ . In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds. As an example, in the real world, Bucharest is the capital of Romania, but for an agent that did not know that, other possible worlds are accessible, including ones where the capital of Romania is Sibiu or Sofia. Presumably a world where  $2 + 2 = 5$  would not be accessible to any agent.

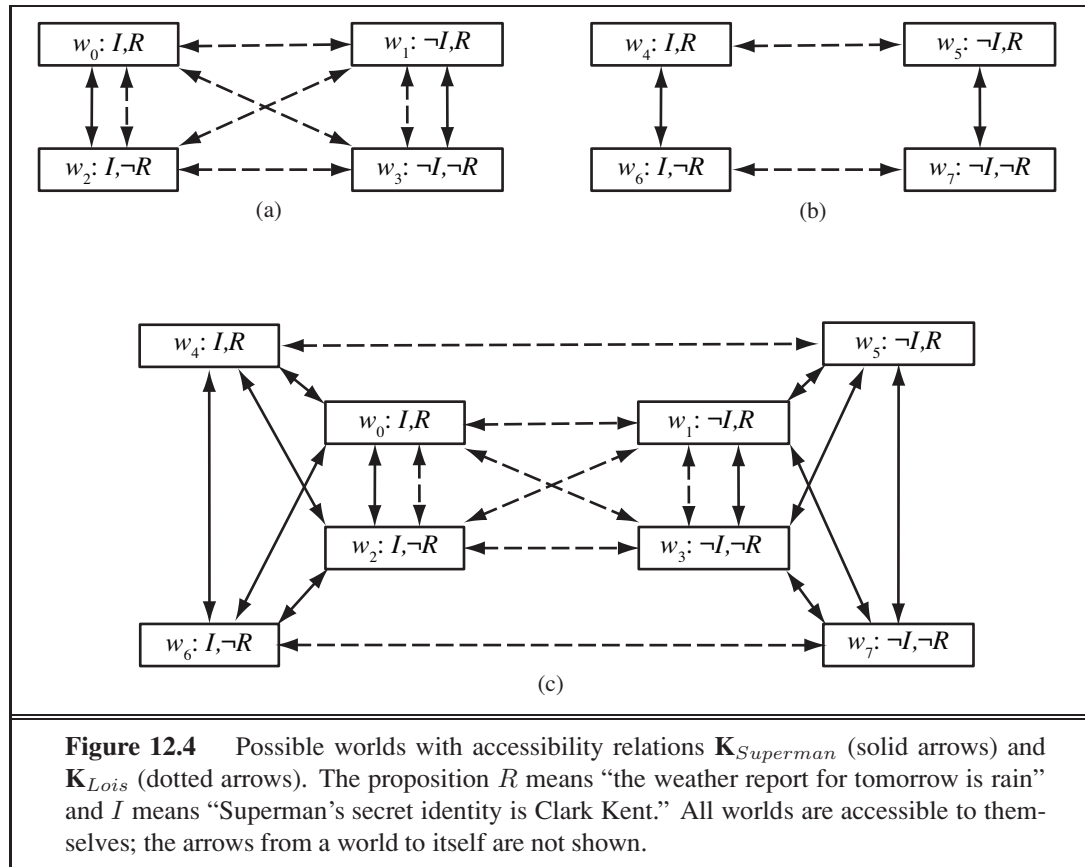
In general, a knowledge atom  $\mathbf{K}_A P$  is true in world  $w$  if and only if  $P$  is true in every world accessible from  $w$ . The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

$$\mathbf{K}_{Lois}[\mathbf{K}_{Clark} Identity(Superman, Clark) \vee \mathbf{K}_{Clark} \neg Identity(Superman, Clark)]$$

Figure 12.4 shows some possible worlds for this domain, with accessibility relations for Lois and Superman.

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in  $w_0$  the worlds  $w_0$  and  $w_2$  are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows  $I$ , or he knows  $\neg I$ . Lois does not know which is the case, but either way she knows Superman knows.

In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in  $w_4$  she knows rain is predicted and in  $w_6$  she knows rain is not predicted.



Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows  $R$  or she knows  $\neg R$ .

In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don’t need to add any arrows for her. In  $w_0$  Superman still knows  $I$  but not  $R$ , and now he does not know whether Lois knows  $R$ . From what Superman knows, he might be in  $w_0$  or  $w_2$ , in which case Lois does not know whether  $R$  is true, or he could be in  $w_4$ , in which case she knows  $R$ , or  $w_6$ , in which case she knows  $\neg R$ .

There are an infinite number of possible worlds, so the trick is to introduce just the ones you need to represent what you are trying to model. A new possible world is needed to talk about different possible facts (e.g., rain is predicted or not), or to talk about different states of knowledge (e.g., does Lois know that rain is predicted). That means two possible worlds, such as  $w_4$  and  $w_0$  in Figure 12.4, might have the same base facts about the world, but differ in their accessibility relations, and therefore in facts about knowledge.

Modal logic solves some tricky issues with the interplay of quantifiers and knowledge. The English sentence “Bond knows that someone is a spy” is ambiguous. The first reading is

that there is a particular someone who Bond knows is a spy; we can write this as

$$\exists x \mathbf{K}_{Bond} Spy(x),$$

which in modal logic means that there is an  $x$  that, in all accessible worlds, Bond knows to be a spy. The second reading is that Bond just knows that there is at least one spy:

$$\mathbf{K}_{Bond} \exists x Spy(x).$$

The modal logic interpretation is that in each accessible world there is an  $x$  that is a spy, but it need not be the same  $x$  in each world.

Now that we have a modal operator for knowledge, we can write axioms for it. First, we can say that agents are able to draw deductions; if an agent knows  $P$  and knows that  $P$  implies  $Q$ , then the agent knows  $Q$ :

$$(\mathbf{K}_a P \wedge \mathbf{K}_a (P \Rightarrow Q)) \Rightarrow \mathbf{K}_a Q.$$

From this (and a few other rules about logical identities) we can establish that  $\mathbf{K}_A(P \vee \neg P)$  is a tautology; every agent knows every proposition  $P$  is either true or false. On the other hand,  $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$  is not a tautology; in general, there will be lots of propositions that an agent does not know to be true and does not know to be false.

It is said (going back to Plato) that knowledge is justified true belief. That is, if it is true, if you believe it, and if you have an unassailably good reason, then you know it. That means that if you know something, it must be true, and we have the axiom:

$$\mathbf{K}_a P \Rightarrow P.$$

Furthermore, logical agents should be able to introspect on their own knowledge. If they know something, then they know that they know it:

$$\mathbf{K}_a P \Rightarrow \mathbf{K}_a (\mathbf{K}_a P).$$

LOGICAL  
OMNISCIENCE

We can define similar axioms for belief (often denoted by  $\mathbf{B}$ ) and other modalities. However, one problem with the modal logic approach is that it assumes **logical omniscience** on the part of agents. That is, if an agent knows a set of axioms, then it knows all consequences of those axioms. This is on shaky ground even for the somewhat abstract notion of knowledge, but it seems even worse for belief, because belief has more connotation of referring to things that are physically represented in the agent, not just potentially derivable. There have been attempts to define a form of limited rationality for agents; to say that agents believe those assertions that can be derived with the application of no more than  $k$  reasoning steps, or no more than  $s$  seconds of computation. These attempts have been generally unsatisfactory.

## 12.5 REASONING SYSTEMS FOR CATEGORIES

Categories are the primary building blocks of large-scale knowledge representation schemes. This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: **semantic networks** provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties

of an object on the basis of its category membership; and **description logics** provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

### 12.5.1 Semantic networks

EXISTENTIAL  
GRAPHS

In 1909, Charles S. Peirce proposed a graphical notation of nodes and edges called **existential graphs** that he called “the logic of the future.” Thus began a long-running debate between advocates of “logic” and advocates of “semantic networks.” Unfortunately, the debate obscured the fact that semantics networks—at least those with well-defined semantics—are a form of logic. The notation that semantic networks provide for certain kinds of sentences is often more convenient, but if we strip away the “human interface” issues, the underlying concepts—objects, relations, quantification, and so on—are the same.

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, Figure 12.5 has a *MemberOf* link between *Mary* and *FemalePersons*, corresponding to the logical assertion  $Mary \in FemalePersons$ ; similarly, the *SisterOf* link between *Mary* and *John* corresponds to the assertion  $SisterOf(Mary, John)$ . We can connect categories using *SubsetOf* links, and so on. It is such fun drawing bubbles and arrows that one can get carried away. For example, we know that persons have female persons as mothers, so can we draw a *HasMother* link from *Persons* to *FemalePersons*? The answer is no, because *HasMother* is a relation between a person and his or her mother, and categories do not have mothers.<sup>5</sup>

For this reason, we have used a special notation—the double-boxed link—in Figure 12.5. This link asserts that

$$\forall x \ x \in Persons \Rightarrow [\forall y \ HasMother(x, y) \Rightarrow y \in FemalePersons].$$

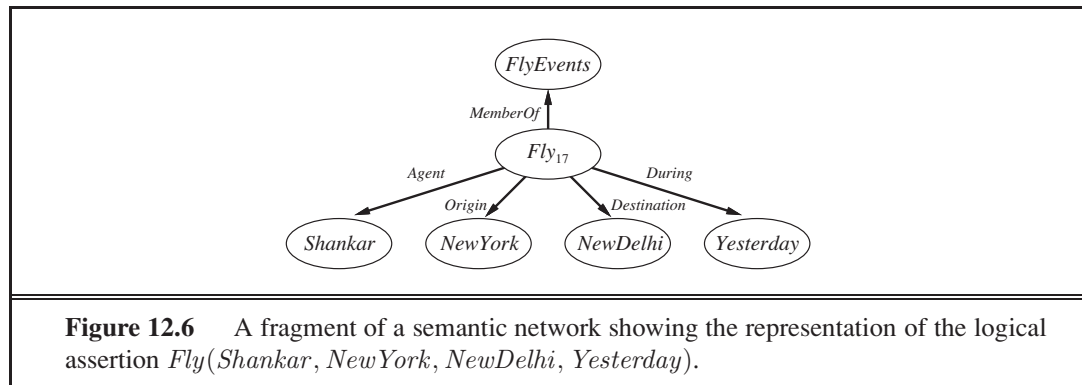
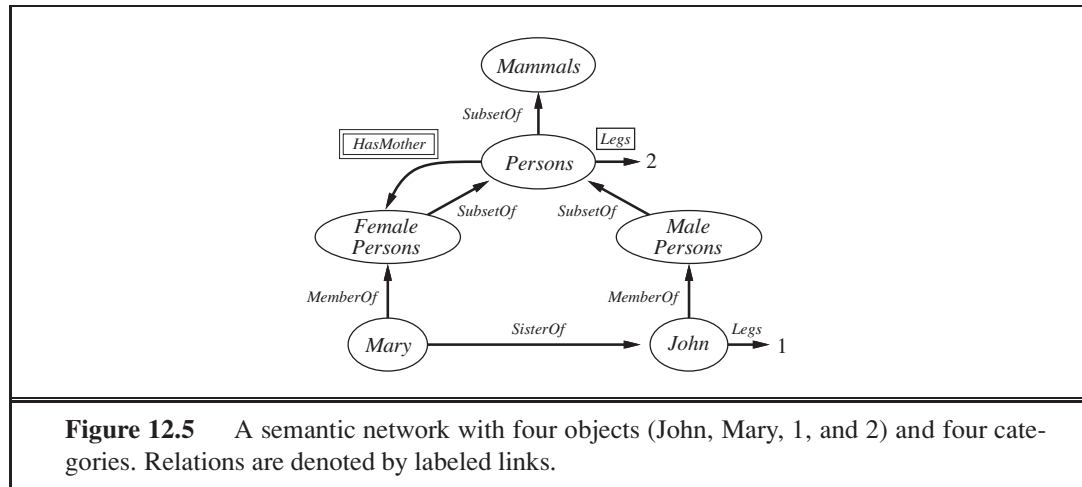
We might also want to assert that persons have two legs—that is,

$$\forall x \ x \in Persons \Rightarrow Legs(x, 2).$$

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 12.5 is used to assert properties of every member of a category.

The semantic network notation makes it convenient to perform **inheritance reasoning** of the kind introduced in Section 12.2. For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm follows the *MemberOf* link from *Mary* to the category she belongs to, and then follows *SubsetOf* links up the hierarchy until it finds a category for which there is a boxed *Legs* link—in this case, the *Persons* category. The simplicity and efficiency of this inference

<sup>5</sup> Several early systems failed to distinguish between properties of members of a category and properties of the category as a whole. This can lead directly to inconsistencies, as pointed out by Drew McDermott (1976) in his article “Artificial Intelligence Meets Natural Stupidity.” Another common problem was the use of *IsA* links for both subset and membership relations, in correspondence with English usage: “a cat is a mammal” and “Fifi is a cat.” See Exercise 12.22 for more on these issues.



mechanism, compared with logical theorem proving, has been one of the main attractions of semantic networks.

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called **multiple inheritance**. In such cases, the inheritance algorithm might find two or more conflicting values answering the query. For this reason, multiple inheritance is **banned in some object-oriented programming (OOP) languages**, such as Java, that use inheritance in a class hierarchy. It is usually allowed in semantic networks, but we defer discussion of that until Section 12.6.

The reader might have noticed an obvious drawback of semantic network notation, compared to first-order logic: the fact that **links between bubbles represent only binary relations**. For example, the sentence  $Fly(Shankar, NewYork, NewDelhi, Yesterday)$  cannot be asserted directly in a semantic network. Nonetheless, we *can* obtain the effect of  **$n$ -ary assertions by reifying the proposition itself as an event belonging to an appropriate event category**. Figure 12.6 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts.

Reification of propositions makes it possible to represent every ground, function-free atomic sentence of first-order logic in the semantic network notation. Certain kinds of univer-

sally quantified sentences can be asserted using inverse links and the singly boxed and doubly boxed arrows applied to categories, but that still leaves us a long way short of full first-order logic. Negation, disjunction, nested function symbols, and existential quantification are all missing. Now it is *possible* to extend the notation to make it equivalent to first-order logic—as in Peirce’s existential graphs—but doing so negates one of the main advantages of semantic networks, which is the simplicity and transparency of the inference processes. Designers can build a large network and still have a good idea about what queries will be efficient, because (a) it is easy to visualize the steps that the inference procedure will go through and (b) in some cases the query language is so simple that difficult queries cannot be posed. In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about (or sometimes an assertion of) a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.

One of the most important aspects of semantic networks is their **ability to represent default values for categories**. Examining Figure 12.5 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information. **The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself** (John in this case) **and stops as soon as it finds a value**. We say that the **default is overridden by the more specific value**. Notice that we could also override the default number of legs by creating a category of *OneLeggedPersons*, a subset of *Persons* of which *John* is a member.

We can retain a strictly logical semantics for the network if we say that the *Legs* assertion for *Persons* includes an exception for John:

$$\forall x \ x \in Persons \wedge x \neq John \Rightarrow Legs(x, 2) .$$

For a *fixed* network, this is semantically adequate but will be much less concise than the network notation itself if there are lots of exceptions. For a network that will be updated with more assertions, however, such an approach fails—we really want to say that any persons as yet unknown with one leg are exceptions too. Section 12.6 goes into more depth on this issue and on default reasoning in general.

### 12.5.2 Description logics

The syntax of first-order logic is designed to make it easy to say things about objects. **Description logics** are notations that are designed to make it easier to describe definitions and properties of categories. Description logic systems evolved from semantic networks in response to pressure to formalize what the networks mean while retaining the **emphasis on taxonomic structure** as an organizing principle.

The principal inference tasks for description logics are **subsumption** (checking if one category is a subset of another by comparing their definitions) and **classification** (checking whether an object belongs to a category).. Some systems also include **consistency** of a category definition—whether the membership criteria are logically satisfiable.



$  \begin{aligned}  \text{Concept} &\rightarrow \mathbf{Thing} \mid \text{ConceptName} \\  &\mid \mathbf{And}(\text{Concept}, \dots) \\  &\mid \mathbf{All}(\text{RoleName}, \text{Concept}) \\  &\mid \mathbf{AtLeast}(\text{Integer}, \text{RoleName}) \\  &\mid \mathbf{AtMost}(\text{Integer}, \text{RoleName}) \\  &\mid \mathbf{Fills}(\text{RoleName}, \text{IndividualName}, \dots) \\  &\mid \mathbf{SameAs}(\text{Path}, \text{Path}) \\  &\mid \mathbf{OneOf}(\text{IndividualName}, \dots) \\  \text{Path} &\rightarrow [\text{RoleName}, \dots]  \end{aligned}  $
--

**Figure 12.7** The syntax of descriptions in a subset of the CLASSIC language.

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 12.7.<sup>6</sup> For example, to say that bachelors are unmarried adult males we would write

$$\text{Bachelor} = \text{And}(\text{Unmarried}, \text{Adult}, \text{Male}) .$$

The equivalent in first-order logic would be

$$\text{Bachelor}(x) \Leftrightarrow \text{Unmarried}(x) \wedge \text{Adult}(x) \wedge \text{Male}(x) .$$

Notice that the description logic has an algebra of operations on predicates, which of course we can't do in first-order logic. Any description in CLASSIC can be translated into an equivalent first-order sentence, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or math departments, we would use

$$\begin{aligned}
 &\text{And}(\text{Man}, \text{AtLeast}(3, \text{Son}), \text{AtMost}(2, \text{Daughter}), \\
 &\quad \text{All}(\text{Son}, \text{And}(\text{Unemployed}, \text{Married}, \text{All}(\text{Spouse}, \text{Doctor}))), \\
 &\quad \text{All}(\text{Daughter}, \text{And}(\text{Professor}, \text{Fills}(\text{Department}, \text{Physics}, \text{Math})))) .
 \end{aligned}$$

We leave it as an exercise to translate this into first-order logic.

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.<sup>7</sup>

<sup>6</sup> Notice that the language does *not* allow one to simply state that one concept, or category, is a subset of another. This is a deliberate policy: subsumption between categories must be derivable from some aspects of the descriptions of the categories. If not, then something is missing from the descriptions.

<sup>7</sup> CLASSIC provides efficient subsumption testing in practice, but the worst-case run time is exponential.



This sounds wonderful in principle, until one realizes that it can only have one of two consequences: either hard problems cannot be stated at all, or they require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave. For example, *description logics* usually lack *negation* and *disjunction*. Each forces first-order logical systems to go through a potentially exponential case analysis in order to ensure completeness. *CLASSIC* allows only a limited form of disjunction in the *Fills* and *OneOf* constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

## 12.6 REASONING WITH DEFAULT INFORMATION

In the preceding section, we saw a simple example of an assertion with default status: people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. We saw that the inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way. In this section, we study defaults more generally, with a view toward understanding the *semantics* of defaults rather than just providing a procedural mechanism.

### 12.6.1 Circumscription and default logic

We have seen two examples of reasoning processes that violate the **monotonicity** property of logic that was proved in Chapter 7.<sup>8</sup> In this chapter we saw that a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In Section 9.4.5, we saw that under the closed-world assumption, if a proposition  $\alpha$  is not mentioned in  $KB$  then  $KB \models \neg\alpha$ , but  $KB \wedge \alpha \models \alpha$ .

Simple introspection suggests that these failures of monotonicity are widespread in commonsense reasoning. It seems that humans often “jump to conclusions.” For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. Now, probability theory can certainly provide a conclusion that the fourth wheel exists with high probability, yet, for most people, the possibility of the car’s not having four wheels *does not arise unless some new evidence presents itself*. Thus, it seems that the four-wheel conclusion is reached by default, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. **Nonmonotonic logics** have been devised with modified notions of truth and entailment in order to capture such behavior. We will look at two such logics that have been studied extensively: circumscription and default logic.

NONMONOTONICITY  
NONMONOTONIC  
LOGIC

<sup>8</sup> Recall that monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if  $KB \models \alpha$  then  $KB \wedge \beta \models \alpha$ .

CIRCUMSCRIPTION

**Circumscription** can be seen as a more powerful and precise version of the closed-world assumption. The idea is to specify particular predicates that are assumed to be “as false as possible”—that is, false for every object except those for which they are known to be true. For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say  $Abnormal_1(x)$ , and write

$$Bird(x) \wedge \neg Abnormal_1(x) \Rightarrow Flies(x) .$$

If we say that  $Abnormal_1$  is to be **circumscribed**, a circumscriptive reasoner is entitled to assume  $\neg Abnormal_1(x)$  unless  $Abnormal_1(x)$  is known to be true. This allows the conclusion  $Flies(Tweety)$  to be drawn from the premise  $Bird(Tweety)$ , but the conclusion no longer holds if  $Abnormal_1(Tweety)$  is asserted.

MODEL  
PREFERENCE

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB, as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.<sup>9</sup> Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the “Nixon diamond.” It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

$$\begin{aligned} & Republican(Nixon) \wedge Quaker(Nixon) . \\ & Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) . \\ & Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x) . \end{aligned}$$

If we circumscribe  $Abnormal_2$  and  $Abnormal_3$ , there are two preferred models: one in which  $Abnormal_2(Nixon)$  and  $Pacifist(Nixon)$  hold and one in which  $Abnormal_3(Nixon)$  and  $\neg Pacifist(Nixon)$  hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon was a pacifist. If we wish, in addition, to assert that religious beliefs take precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where  $Abnormal_3$  is minimized.

PRIORITIZED  
CIRCUMSCRIPTION

DEFAULT LOGIC

DEFAULT RULES

**Default logic** is a formalism in which **default rules** can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

$$Bird(x) : Flies(x) / Flies(x) .$$

This rule means that if  $Bird(x)$  is true, and if  $Flies(x)$  is consistent with the knowledge base, then  $Flies(x)$  may be concluded by default. In general, a default rule has the form

$$P : J_1, \dots, J_n / C$$

where  $P$  is called the **prerequisite**,  $C$  is the **conclusion**, and  $J_i$  are the **justifications**—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that

<sup>9</sup> For the closed-world assumption, one model is preferred to another if it has fewer true atoms—that is, preferred models are **minimal** models. There is a natural connection between the closed-world assumption and definite-clause KBs, because the fixed point reached by forward chaining on definite-clause KBs is the unique minimal model. See page 258 for more on this point.

appears in  $J_i$  or  $C$  must also appear in  $P$ . The Nixon-diamond example can be represented in default logic with one fact and two default rules:

$$\begin{aligned} & \text{Republican}(\text{Nixon}) \wedge \text{Quaker}(\text{Nixon}) . \\ & \text{Republican}(x) : \neg \text{Pacifist}(x) / \neg \text{Pacifist}(x) . \\ & \text{Quaker}(x) : \text{Pacifist}(x) / \text{Pacifist}(x) . \end{aligned}$$

EXTENSION

To interpret what the default rules mean, we define the notion of an **extension** of a default theory to be a maximal set of consequences of the theory. That is, an extension  $S$  consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from  $S$  and the justifications of every default conclusion in  $S$  are consistent with  $S$ . As in the case of the preferred models in circumscription, we have two possible extensions for the Nixon diamond: one wherein he is a pacifist and one wherein he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. There are still unresolved questions, however. For example, if “Cars have four wheels” is false, what does it mean to have it in one’s knowledge base? What is a good set of default rules to have? If we cannot decide, for each rule separately, whether it belongs in our knowledge base, then we have a serious problem of nonmodularity. Finally, how can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions, and the *costs* of making a wrong decision. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as “threshold probability” statements. For example, the default rule “My brakes are always OK” really means “The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them.” When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence of faulty brakes. These considerations have led some researchers to consider how to embed default reasoning within probability theory or utility theory.

### 12.6.2 Truth maintenance systems

We have seen that many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these **inferred facts will turn out to be wrong and will have to be retracted in the face of new information**. This process is called **belief revision**.<sup>10</sup> Suppose that a knowledge base  $KB$  contains a sentence  $P$ —perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute  $\text{TELL}(KB, \neg P)$ . To avoid creating a contradiction, we must first execute  $\text{RETRACT}(KB, P)$ . This sounds easy enough.

BELIEF REVISION

<sup>10</sup> Belief revision is often contrasted with **belief update**, which occurs when a knowledge base is revised to reflect a change in the world rather than new information about a fixed world. Belief update combines belief revision with reasoning about time and change; it is also related to the process of **filtering** described in Chapter 15.

TRUTH  
MAINTENANCE  
SYSTEM

Problems arise, however, if any *additional* sentences were inferred from  $P$  and asserted in the KB. For example, the implication  $P \Rightarrow Q$  might have been used to add  $Q$ . The obvious “solution”—retracting all sentences inferred from  $P$ —fails because such sentences may have other justifications besides  $P$ . For example, if  $R$  and  $R \Rightarrow Q$  are also in the KB, then  $Q$  does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

One simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from  $P_1$  to  $P_n$ . When the call  $\text{RETRACT}(KB, P_i)$  is made, the system reverts to the state just before  $P_i$  was added, thereby removing both  $P_i$  and any inferences that were derived from  $P_i$ . The sentences  $P_{i+1}$  through  $P_n$  can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but retracting  $P_i$  requires retracting and reasserting  $n - i$  sentences as well as undoing and redoing all the inferences drawn from those sentences. For systems to which many facts are being added—such as large commercial databases—this is impractical.

JTMS  
JUSTIFICATION

A more efficient approach is the **justification-based truth maintenance system**, or **JTMS**. In a JTMS, each sentence in the knowledge base is annotated with a **justification** consisting of the set of sentences from which it was inferred. For example, if the knowledge base already contains  $P \Rightarrow Q$ , then  $\text{TELL}(P)$  will cause  $Q$  to be added with the justification  $\{P, P \Rightarrow Q\}$ . In general, a sentence can have any number of justifications. Justifications make retraction efficient. Given the call  $\text{RETRACT}(P)$ , the JTMS will delete exactly those sentences for which  $P$  is a member of every justification. So, if a sentence  $Q$  had the single justification  $\{P, P \Rightarrow Q\}$ , it would be removed; if it had the additional justification  $\{P, P \vee R \Rightarrow Q\}$ , it would still be removed; but if it also had the justification  $\{R, P \vee R \Rightarrow Q\}$ , then it would be spared. In this way, the time required for retraction of  $P$  depends only on the number of sentences derived from  $P$  rather than on the number of other sentences added since  $P$  entered the knowledge base.

The JTMS assumes that sentences that are considered once will probably be considered again, so rather than deleting a sentence from the knowledge base entirely when it loses all justifications, we merely mark the sentence as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then we mark the sentence as being back *in*. In this way, the JTMS retains all the inference chains that it uses and need not rederive sentences when a justification becomes valid again.

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations. Suppose, for example, that the Romanian Olympic Committee is choosing sites for the swimming, athletics, and equestrian events at the 2048 Games to be held in Romania. For example, let the first hypothesis be  $\text{Site}(\text{Swimming}, \text{Pitesti})$ ,  $\text{Site}(\text{Athletics}, \text{Bucharest})$ , and  $\text{Site}(\text{Equestrian}, \text{Arad})$ . A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider  $\text{Site}(\text{Athletics}, \text{Sibiu})$  instead, the TMS avoids the need to start again from scratch. Instead, we simply retract  $\text{Site}(\text{Athletics}, \text{Bucharest})$  and assert  $\text{Site}(\text{Athletics}, \text{Sibiu})$  and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

ATMS

An assumption-based truth maintenance system, or **ATMS**, makes this type of context-switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents *all* the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being *in* or *out*, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases in which all the assumptions in one of the assumption sets hold.

EXPLANATION

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence  $P$  is a set of sentences  $E$  such that  $E$  entails  $P$ . If the sentences in  $E$  are already known to be true, then  $E$  simply provides a sufficient basis for proving that  $P$  must be the case. But explanations can also include **assumptions**—sentences that are not known to be true, but would suffice to prove  $P$  if they were true. For example, one might not have enough information to prove that one's car won't start, but a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed nonbehavior. In most cases, we will prefer an explanation  $E$  that is minimal, meaning that there is no proper subset of  $E$  that is also an explanation. An ATMS can generate explanations for the "car won't start" problem by making assumptions (such as "gas in car" or "battery dead") in any order we like, even if some assumptions are contradictory. Then we look at the label for the sentence "car won't start" to read off the sets of assumptions that would justify the sentence.

ASSUMPTION

The exact algorithms used to implement truth maintenance systems are a little complicated, and we do not cover them here. The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. Therefore, you should not expect truth maintenance to be a panacea. When used carefully, however, a TMS can provide a substantial increase in the ability of a logical system to handle complex environments and hypotheses.

## 12.8 SUMMARY

---

By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed and a feeling for the interesting philosophical issues that arise. The major points are as follows:

- Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
- A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
- Building a large, general-purpose ontology is a significant challenge that has yet to be fully realized, although current frameworks seem to be quite robust.
- We presented an **upper ontology** based on categories and the event calculus. We covered categories, subcategories, parts, structured objects, measurements, substances, events, time and space, change, and beliefs.

- Natural kinds cannot be defined completely in logic, but properties of natural kinds can be represented.
- Actions, events, and time can be represented either in situation calculus or in more expressive representations such as event calculus. Such representations enable an agent to construct plans by logical inference.
- We presented a detailed analysis of the Internet shopping domain, exercising the general ontology and showing how the domain knowledge can be used by a shopping agent.
- Special-purpose representation systems, such as **semantic networks** and **description logics**, have been devised to help in organizing a hierarchy of categories. **Inheritance** is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.
- The **closed-world assumption**, as implemented in logic programs, provides a simple way to avoid having to specify lots of negative information. It is best interpreted as a **default** that can be overridden by additional information.
- **Nonmonotonic logics**, such as **circumscription** and **default logic**, are intended to capture default reasoning in general.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.