

10 CLASSICAL PLANNING

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action.

We have defined AI as the study of rational action, which means that **planning**—devising a plan of action to achieve one’s goals—is a critical part of AI. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the hybrid logical agent of Chapter 7. In this chapter we introduce a representation for planning problems that scales up to problems that could not be handled by those earlier approaches.

Section 10.1 develops an expressive yet carefully constrained language for representing planning problems. Section 10.2 shows how forward and backward search algorithms can take advantage of this representation, primarily through accurate heuristics that can be derived automatically from the structure of the representation. (This is analogous to the way in which effective domain-independent heuristics were constructed for constraint satisfaction problems in Chapter 6.) Section 10.3 shows how a data structure called the planning graph can make the search for a plan more efficient. We then describe a few of the other approaches to planning, and conclude by comparing the various approaches.

This chapter covers fully observable, deterministic, static environments with single agents. Chapters 11 and 17 cover partially observable, stochastic, dynamic environments with multiple agents.

10.1 DEFINITION OF CLASSICAL PLANNING

The problem-solving agent of Chapter 3 can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well. The hybrid propositional logical agent of Chapter 7 can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

PDDL

In response to this, planning researchers have settled on a **factored representation**—one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language, that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL; we select a simple version and alter its syntax to be consistent with the rest of the book.¹ We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

SET SEMANTICS

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and a state in a package delivery problem might be $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$. **Database semantics** is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that $Truck_1$ and $Truck_2$ are distinct. The following fluents are *not* allowed in a state: $At(x, y)$ (because it is non-ground), $\neg Poor$ (because it is a negation), and $At(Father(Fred), Sydney)$ (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations. The **set semantics** is sometimes easier to deal with.

ACTION SCHEMA

Actions are described by a set of action schemas that implicitly define the $ACTIONS(s)$ and $RESULT(s, a)$ functions needed to do a problem-solving search. We saw in Chapter 7 that any system for action description needs to solve the frame problem—to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector Δ . A concise description of the action should mention only Δ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single **action schema**. The schema is a **lifted** representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

$$\begin{aligned} &Action(Fly(p, from, to), \\ &\quad PRECOND: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\ &\quad EFFECT: \neg At(p, from) \wedge At(p, to)) \end{aligned}$$

PRECONDITION

EFFECT

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground

¹ PDDL was derived from the original STRIPS planning language (Fikes and Nilsson, 1971), which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals.

action that results from substituting values for all the variables:

$$\begin{aligned} & \text{Action}(\text{Fly}(P_1, \text{SFO}, \text{JFK}), \\ & \quad \text{PRECOND: } \text{At}(P_1, \text{SFO}) \wedge \text{Plane}(P_1) \wedge \text{Airport}(\text{SFO}) \wedge \text{Airport}(\text{JFK}) \\ & \quad \text{EFFECT: } \neg \text{At}(P_1, \text{SFO}) \wedge \text{At}(P_1, \text{JFK})) \end{aligned}$$

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action a can be executed in state s if s entails the precondition of a . Entailment can also be expressed with the set semantics: $s \models q$ iff every positive literal in q is in s and every negated literal in q is not. In formal notation we say

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

where any variables in a are universally quantified. For example,

$$\begin{aligned} \forall p, \text{from}, \text{to} \quad & (\text{Fly}(p, \text{from}, \text{to}) \in \text{ACTIONS}(s)) \Leftrightarrow \\ & s \models (\text{At}(p, \text{from}) \wedge \text{Plane}(p) \wedge \text{Airport}(\text{from}) \wedge \text{Airport}(\text{to})) \end{aligned}$$

APPLICABLE

We say that action a is **applicable** in state s if the preconditions are satisfied by s . When an action schema a contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the *Fly* action can be instantiated as *Fly*($P_1, \text{SFO}, \text{JFK}$) or as *Fly*($P_2, \text{JFK}, \text{SFO}$), both of which are applicable in the initial state. If an action a has v variables, then, in a domain with k unique names of objects, it takes $O(v^k)$ time in the worst case to find the applicable ground actions.

PROPOSITIONALIZE

Sometimes we want to **propositionalize** a PDDL problem—replace each action schema with a set of ground actions and then use a propositional solver such as SATPLAN to find a solution. However, this is impractical when v and k are large.

DELETE LIST

ADD LIST

The **result** of executing action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s , removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or $\text{DEL}(a)$), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or $\text{ADD}(a)$):

$$\text{RESULT}(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a). \quad (10.1)$$

For example, with the action *Fly*($P_1, \text{SFO}, \text{JFK}$), we would remove $\text{At}(P_1, \text{SFO})$ and add $\text{At}(P_1, \text{JFK})$. It is a requirement of action schemas that any variable in the effect must also appear in the precondition. That way, when the precondition is matched against the state s , all the variables will be bound, and $\text{RESULT}(s, a)$ will therefore have only ground atoms. In other words, ground states are closed under the **RESULT** operation.

Also note that the fluents do not explicitly refer to time, as they did in Chapter 7. There we needed superscripts for time, and successor-state axioms of the form

$$F^{t+1} \Leftrightarrow \text{ActionCauses}F^t \vee (F^t \wedge \neg \text{ActionCausesNot}F^t).$$

In PDDL the times and states are implicit in the action schemas: the precondition always refers to time t and the effect to time $t + 1$.

A set of action schemas serves as a definition of a planning *domain*. A specific *problem* within the domain is defined with the addition of an initial state and a goal. The **initial**

```

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
    PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    EFFECT: ¬ At(p, from) ∧ At(p, to))

```

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

INITIAL STATE

GOAL

state is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.) The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as *At*(*p*, *SFO*) ∧ *Plane*(*p*). Any variables are treated as existentially quantified, so this goal is to have *any* plane at SFO. The problem is solved when we can find a sequence of actions that end in a state *s* that entails the goal. For example, the state *Rich* ∧ *Famous* ∧ *Miserable* entails the goal *Rich* ∧ *Famous*, and the state *Plane*(*Plane*₁) ∧ *At*(*Plane*₁, *SFO*) entails the goal *At*(*p*, *SFO*) ∧ *Plane*(*p*).

Now we have defined planning as a search problem: we have an initial state, an ACTIONS function, a RESULT function, and a goal test. We'll look at some example problems before investigating efficient search algorithms.

10.1.1 Example: Air cargo transport

Figure 10.1 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: *Load*, *Unload*, and *Fly*. The actions affect two predicates: *In*(*c*, *p*) means that cargo *c* is inside plane *p*, and *At*(*x*, *a*) means that object *x* (either plane or cargo) is at airport *a*. Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *At* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
 Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```

Finally, there is the problem of spurious actions such as $Fly(P_1, JFK, JFK)$, which should be a no-op, but which has contradictory effects (according to the definition, the effect would include $At(P_1, JFK) \wedge \neg At(P_1, JFK)$). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the *from* and *to* airports must be different; see another example of this in Figure 10.3.

10.1.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$.

```

Init(Tire(Flat)  $\wedge$  Tire(Spare)  $\wedge$  At(Flat, Axle)  $\wedge$  At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(obj, loc),
  PRECOND: At(obj, loc)
  EFFECT:  $\neg At(obj, loc) \wedge At(obj, Ground)$ )
Action(PutOn(t, Axle),
  PRECOND: Tire(t)  $\wedge At(t, Ground) \wedge \neg At(Flat, Axle)$ 
  EFFECT:  $\neg At(t, Ground) \wedge At(t, Axle)$ )
Action(LeaveOvernight,
  PRECOND:
  EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$ 
          $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

```

Figure 10.2 The simple spare tire problem.

10.1.3 Example: The blocks world

BLOCKS WORLD

One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table.² The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top

² The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

```

Init( $On(A, Table) \wedge On(B, Table) \wedge On(C, A)$ 
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)$ )
Goal( $On(A, B) \wedge On(B, C)$ )
Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$ 
 $(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$ ,
  EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ )
Action( $MoveToTable(b, x)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x)$ ,
  EFFECT:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$ )

```

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

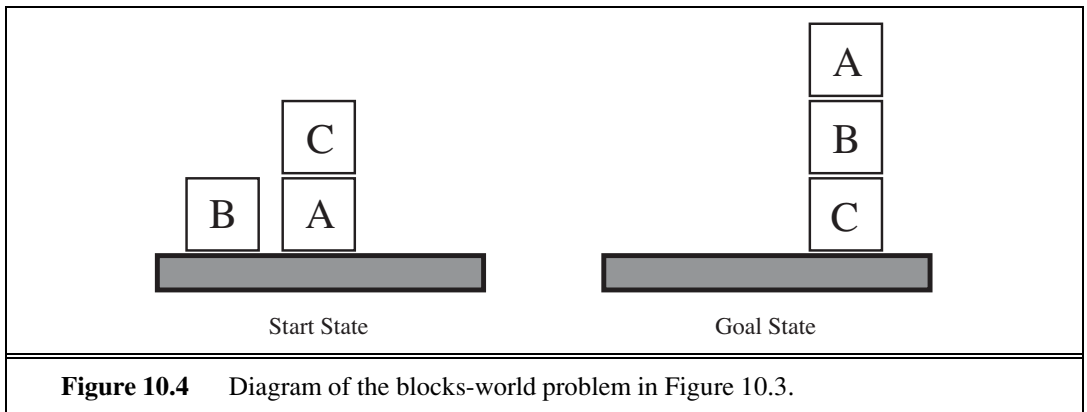


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

of what other blocks. For example, a goal might be to get block A on B and block B on C (see Figure 10.4).

We use $On(b, x)$ to indicate that block b is on x , where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b, x, y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg \exists x On(x, b)$ or, alternatively, $\forall x \neg On(x, b)$. Basic PDDL does not allow quantifiers, so instead we introduce a predicate $Clear(x)$ that is true when nothing is on x . (The complete problem description is in Figure 10.3.)

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the $Move$ schema is

```

Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y)$ ,
  EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ ) .

```

Unfortunately, this does not maintain $Clear$ properly when x or y is the table. When x is the *Table*, this action has the effect $Clear(Table)$, but the table should not become clear; and when $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear

for us to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

$$\begin{aligned} & \text{Action}(\text{MoveToTable}(b, x), \\ & \quad \text{PRECOND: } On(b, x) \wedge \text{Clear}(b), \\ & \quad \text{EFFECT: } On(b, \text{Table}) \wedge \text{Clear}(x) \wedge \neg On(b, x)) . \end{aligned}$$

Second, we take the interpretation of $\text{Clear}(x)$ to be “there is a clear space on x to hold a block.” Under this interpretation, $\text{Clear}(\text{Table})$ will always be true. The only problem is that nothing prevents the planner from using $\text{Move}(b, x, \text{Table})$ instead of $\text{MoveToTable}(b, x)$. We could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate Block and add $\text{Block}(b) \wedge \text{Block}(y)$ to the precondition of Move .

10.1.4 The complexity of classical planning

In this subsection we consider the theoretical complexity of planning and distinguish two decision problems. **PlanSAT** is the question of whether there exists any plan that solves a planning problem. **Bounded PlanSAT** asks whether there is a solution of length k or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols. For proofs of the assertions in this section, see Ghallab *et al.* (2004).

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult. For example, if we disallow negative effects, both problems are still NP-hard. However, if we also disallow negative preconditions, PlanSAT reduces to the class P.

These worst-case results may seem discouraging. We can take solace in the fact that agents are usually not asked to find plans for arbitrary worst-case problem instances, but rather are asked for plans in specific domains (such as blocks-world problems with n blocks), which can be much easier than the theoretical worst case. For many domains (including the blocks world and the air cargo world), Bounded PlanSAT is NP-complete while PlanSAT is in P; in other words, optimal planning is usually hard, but sub-optimal planning is sometimes easy. To do well on easier-than-worst-case problems, we will need good search heuristics. That’s the true advantage of the classical planning formalism: it has facilitated the development of very accurate domain-independent heuristics, whereas systems based on successor-state axioms in first-order logic have had less success in coming up with good heuristics.

PlanSAT

Bounded PlanSAT

10.2 ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.5 compares forward and backward searches.

10.2.1 Forward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms from Chapter 3 or a local search algorithm from Chapter 4 (provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until around 1998 it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why.

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an

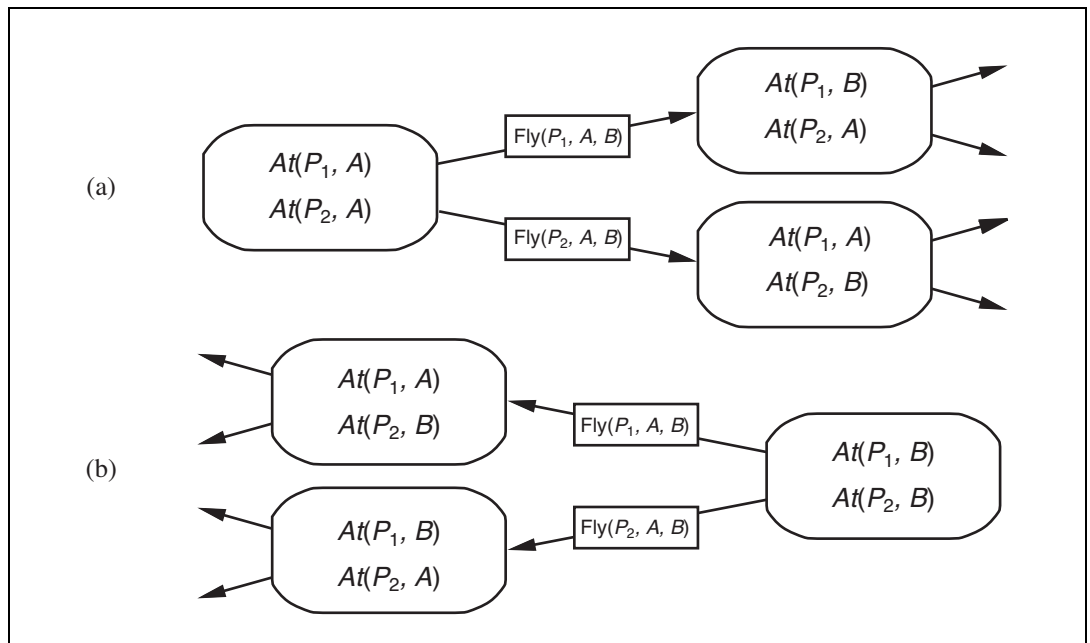


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

action schema $Buy(isbn)$ with effect $Own(isbn)$. ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B . There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A , fly the plane to B , and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 10.2.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

10.2.2 Backward (regression) relevant-states search

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). As in belief-state search (Section 4.4), there is a *set* of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal $\neg Poor \wedge Famous$ describes those states in which *Poor* is false, *Famous* is true, and any other fluent can have any value. If there are n ground fluents in a domain, then there are 2^n ground states (each fluent can be true or false), but 3^n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n -queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a , the regression from g over a gives us a state description g' defined by

$$g' = (g - \text{ADD}(a)) \cup \text{Precond}(a) .$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. Note that $\text{DEL}(a)$ does not appear in the formula; that's because while we know the fluents in $\text{DEL}(a)$ are no longer true after the action, we don't know whether or not they were true before, so there's nothing to be said about them.

To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $At(C_2, SFO)$. That suggests the action $Unload(C_2, p', SFO)$:

$Action(Unload(C_2, p', SFO),$
 PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$
 EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p')$.

(Note that we have **standardized** variable names (changing p to p' in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan. The same approach was used in Chapter 9 for first-order logical inference.) This represents unloading the package from an *unspecified* plane at SFO; any plane will do, but we need not say which one now. We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any* of the planes by implicitly quantifying over p' . The regressed state description is

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) .$$

The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the *last* step in a plan leading up to the current goal state.

RELEVANCE

For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal. Now, if the goal is $A \wedge B \wedge C$ and an action has the effect $A \wedge B \wedge \neg C$ then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there. But it is not relevant in the technical sense defined here, because this action could not be the *final* step of a solution—we would always need at least one more step to achieve C .

Given the goal $At(C_2, SFO)$, several instantiations of $Unload$ are relevant: we could chose any specific plane to unload from, or we could leave the plane unspecified by using the action $Unload(C_2, p', SFO)$. We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

As another example, consider the goal $Own(0136042597)$, given an initial state with 10 billion ISBNs, and the single action schema

$$A = Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i)) .$$

As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions. But with backward search, we would unify the goal $Own(0136042597)$ with the (standardized) effect $Own(i')$, yielding the substitution $\theta = \{i'/0136042597\}$. Then we would regress over the action $Subst(\theta, A')$ to yield the predecessor state description $ISBN(0136042597)$. This is part of, and thus entailed by, the initial state, so we are done.

We can make this more formal. Assume a goal description g which contains a goal literal g_i and an action schema A that is standardized to produce A' . If A' has an effect literal e'_j where $\text{Unify}(g_i, e'_j) = \theta$ and where we define $a' = \text{SUBST}(\theta, A')$ and if there is no effect in a' that is the negation of a literal in g , then a' is a relevant action towards g .

Backward search keeps the branching factor lower than forward search, for most problem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

10.2.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function $h(s)$ estimates the distance from a state s to the goal and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate—then we can use A^* search to find optimal solutions. An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

Think of a search problem as a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two ways we can relax this problem to make it easier: by adding more edges to the graph, making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of $\log n$ of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding-block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning

IGNORE
PRECONDITIONS
HEURISTIC

SET-COVER
PROBLEM

problem involving tiles with a single schema *Slide*:

$Action(Slide(t, s_1, s_2),$

PRECOND: $On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)$

EFFECT: $On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2)$)

As we saw in Section 3.6, if we remove the preconditions $Blank(s_2) \wedge Adjacent(s_1, s_2)$ then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove $Blank(s_2)$ then we get the Manhattan-distance heuristic. It is easy to see how these heuristics could be derived automatically from the action schema description. The ease of manipulating the schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

IGNORE DELETE
LISTS

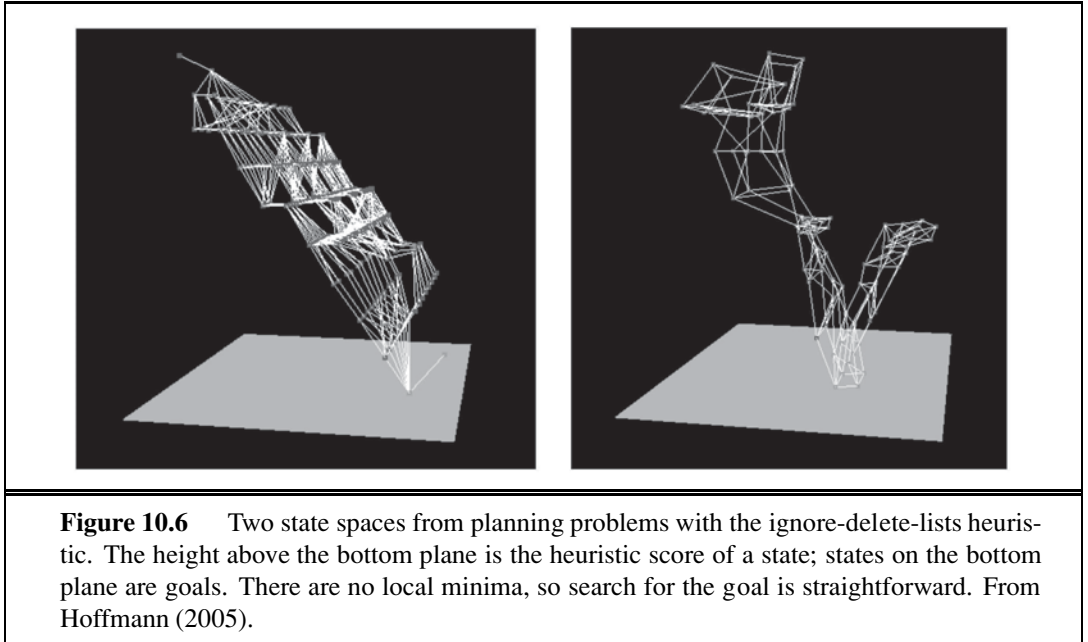
Another possibility is the **ignore delete lists** heuristic. Assume for a moment that all goals and preconditions contain only positive literals³ We want to create a relaxed version of the original problem that will be easier to solve, and where the length of the solution will serve as a good heuristic. We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill-climbing. Figure 10.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

STATE ABSTRACTION

The relaxed problems leave us with a simplified—but still expensive—planning problem just to calculate the value of the heuristic function. Many planning problems have 10^{100} states or more, and relaxing the *actions* does nothing to reduce the number of states. Therefore, we now look at relaxations that decrease the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

The easiest form of state abstraction is to ignore some fluents. For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. So there are $50^{10} \times 200^{50+10} \approx 10^{155}$ states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $5^{10} \times 5^{5+10} \approx 10^{17}$ states. A solution in this abstract state space will be shorter than a solution in the original space (and thus will be an admissible heuristic), and the abstract solution is easy to extend to a solution to the original problem (by adding additional *Load* and *Unload* actions).

³ Many problems are written with this convention. For problems that aren't, replace every negative literal $\neg P$ in a goal or precondition with a new positive literal, P' .



DECOMPOSITION
SUBGOAL
INDEPENDENCE

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently*. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

Suppose the goal is a set of fluents G , which we divide into disjoint subsets G_1, \dots, G_n . We then find plans P_1, \dots, P_n that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of G ? We can think of each $\text{Cost}(P_i)$ as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So $\max_i \text{COST}(P_i)$ is admissible, and sometimes it is exactly correct: it could be that P_1 serendipitously achieves all the G_i . But in most cases, in practice the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when we can determine that G_i and G_j are **independent**. If the effects of P_i leave all the preconditions and goals of P_j unchanged, then the estimate $\text{COST}(P_i) + \text{COST}(P_j)$ is admissible, and more accurate than the max estimate. We show in Section 10.3.1 that planning graphs can help provide better heuristic estimates.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The tech-

niques of **pattern databases** from Section 3.6.3 can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

An example of a system that makes use of effective heuristics is FF, or FASTFORWARD (Hoffmann, 2005), a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph (see Section 10.3). FF then uses hill-climbing search (modified to keep track of the plan) with the heuristic to find a solution. When it hits a plateau or local maximum—when no action leads to a state with better heuristic score—then FF uses iterative deepening search until it finds a state that is better, or it gives up and restarts hill-climbing.

10.3 PLANNING GRAPHS

PLANNING GRAPH

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a **planning graph** can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question “can we reach state G from state S_0 ” immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial-size approximation to this tree that can be constructed quickly. The planning graph can’t answer definitively whether G is reachable from S_0 , but it can *estimate* how many steps it takes to reach G . The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

LEVEL

A planning graph is a directed graph organized into **levels**: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0 ; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0 ; then alternating levels S_i followed by A_i ; until we reach a termination condition (to be discussed later).

Roughly speaking, S_i contains all the literals that *could* hold at time i , depending on the actions executed at preceding time steps. If it is possible that either P or $\neg P$ could hold, then both will be represented in S_i . Also roughly speaking, A_i contains all the actions that *could* have their preconditions satisfied at time i . We say “roughly speaking” because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level S_j when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of ac-

depending on the choice of actions in A_0 , either, but not both, could be the result. In other words, S_1 represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**. The graph in Figure 10.8 levels off at S_2 .

LEVELED OFF

What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i , along with constraints saying that two actions cannot both be executed at the same level. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1} , along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links.

We now define mutex links for both actions and literals. A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:

- *Inconsistent effects*: one action negates an effect of the other. For example, $Eat(Cake)$ and the persistence of $Have(Cake)$ have inconsistent effects because they disagree on the effect $Have(Cake)$.
- *Interference*: one of the effects of one action is the negation of a precondition of the other. For example $Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition.
- *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $Bake(Cake)$ and $Eat(Cake)$ are mutex because they compete on the value of the $Have(Cake)$ precondition.

A mutex relation holds between two *literals* at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called *inconsistent support*. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistence action, is mutex with the only way of achieving $Eaten(Cake)$, namely $Eat(Cake)$. In S_2 the two literals are not mutex, because there are new ways of achieving them, such as $Bake(Cake)$ and the persistence of $Eaten(Cake)$, that are not mutex.

A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including the no-ops), $(a + l)^2$ mutex links, and $2(al + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

10.3.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s . We call this the

LEVEL COST

level cost of g_i . In Figure 10.8, *Have*(*Cake*) has level cost 0 and *Eaten*(*Cake*) has level cost 1. It is easy to show (Exercise 10.10) that these estimates are admissible for the individual goals. The estimate might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a **serial planning graph** for computing heuristics. A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

SERIAL PLANNING GRAPH

MAX-LEVEL

To estimate the cost of a *conjunction* of goals, there are three simple approaches. The **max-level** heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily accurate.

LEVEL SUM

The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 10.2. For our problem, the level-sum heuristic estimate for the conjunctive goal *Have*(*Cake*) \wedge *Eaten*(*Cake*) will be $0 + 1 = 1$, whereas the correct answer is 2, achieved by the plan [*Eat*(*Cake*), *Bake*(*Cake*)]. That doesn't seem so bad. A more serious error is that if *Bake*(*Cake*) were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

SET-LEVEL

Finally, the **set-level** heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without *Bake*(*Cake*). It is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently solvable. To understand the nature of the relaxed problem, we need to understand exactly what it means for a literal g to appear at level S_i in the planning graph. Ideally, we would like it to be a guarantee that there exists a plan with i action levels that achieves g , and also that if g does not appear, there is no such plan. Unfortunately, making that guarantee is as difficult as solving the original planning problem. So the planning graph makes the second half of the guarantee (if g does not appear, there is no plan), but if g does appear, then all the planning graph promises is that there is a plan that *possibly* achieves g and has no “obvious” flaws. An obvious flaw is defined as a flaw that can be detected by considering two actions or two literals at a time—in other words, by looking at the mutex relations. There could be more subtle flaws involving three, four, or more actions, but experience has shown that it is not worth the computational effort to keep track of these possible flaws. This is similar to a lesson learned from constraint satisfaction problems—that it is often worthwhile to compute 2-consistency before searching for a solution, but less often worthwhile to compute 3-consistency or higher. (See page 211.)

One example of an unsolvable problem that cannot be recognized as such by a planning graph is the blocks-world problem where the goal is to get block A on B , B on C , and C on A . This is an impossible goal; a tower with the bottom on top of the top. But a planning graph

cannot detect the impossibility, because any two of the three subgoals are achievable. There are no mutexes between any pair of literals, only between the three as a whole. To detect that this problem is impossible, we would have to search over the planning graph.

10.3.2 The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm (Figure 10.9) repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

function GRAPHPLAN(problem) returns solution or failure
  graph  $\leftarrow$  INITIAL-PLANNING-GRAPH(problem)
  goals  $\leftarrow$  CONJUNCTS(problem.GOAL)
  nogoods  $\leftarrow$  an empty hash table
  for tl = 0 to  $\infty$  do
    if goals all non-mutex in  $S_t$  of graph then
      solution  $\leftarrow$  EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution  $\neq$  failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph  $\leftarrow$  EXPAND-GRAPH(graph, problem)

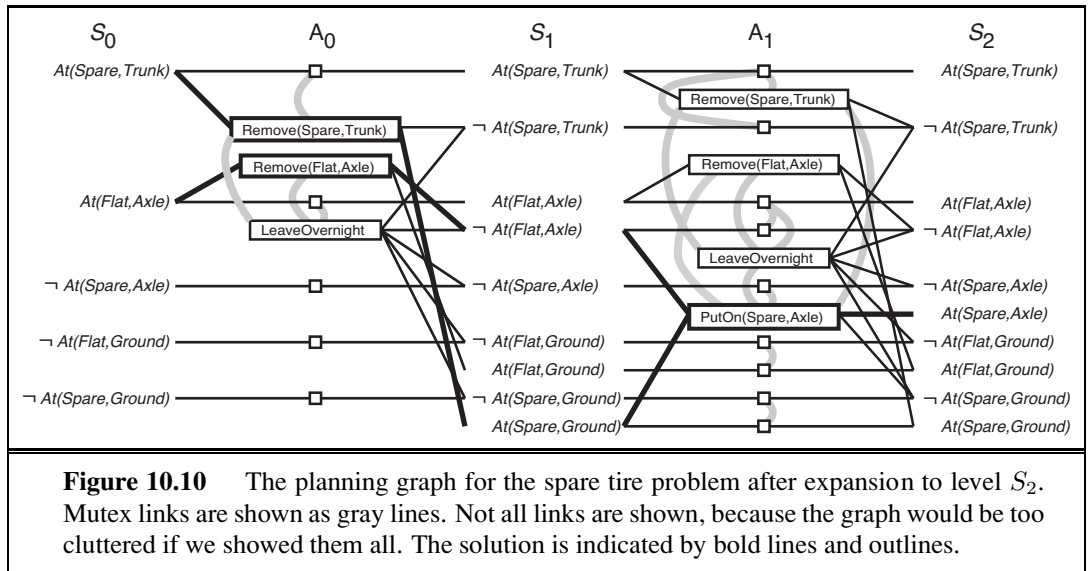
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph representing the initial state. The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as $Tire(Spare)$) and the irrelevant negative literals. The goal $At(Spare, Axle)$ is not present in S_0 , so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at level S_0 (i.e., all the actions except $PutOn(Spare, Axle)$), along with persistence actions for all the literals in S_0 . The effects of the actions are added at level S_1 . EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

$At(Spare, Axle)$ is still not present in S_1 , so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown in Figure 10.10. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:

- *Inconsistent effects*: $Remove(Spare, Trunk)$ is mutex with $LeaveOvernight$ because one has the effect $At(Spare, Ground)$ and the other has its negation.



- *Interference*: $Remove(Flat, Axle)$ is mutex with $LeaveOvernight$ because one has the precondition $At(Flat, Axle)$ and the other has its negation as an effect.
- *Competing needs*: $PutOn(Spare, Axle)$ is mutex with $Remove(Flat, Axle)$ because one has $At(Flat, Axle)$ as a precondition and the other has its negation.
- *Inconsistent support*: $At(Spare, Axle)$ is mutex with $At(Flat, Axle)$ in S_2 because the only way of achieving $At(Spare, Axle)$ is by $PutOn(Spare, Axle)$, and that is mutex with the persistence action that is the only way of achieving $At(Flat, Axle)$. Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

This time, when we go back to the start of the loop, all the literals from the goal are present in S_2 , and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n , along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By “conflict free,” we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.

- The goal is to reach a state at level S_0 such that all the goals are satisfied.
- The cost of each action is 1.

For this particular problem, we start at S_2 with the goal $At(Spare, Axle)$. The only choice we have for achieving the goal set is $PutOn(Spare, Axle)$. That brings us to a search state at S_1 with goals $At(Spare, Ground)$ and $\neg At(Flat, Axle)$. The former can be achieved only by $Remove(Spare, Trunk)$, and the latter by either $Remove(Flat, Axle)$ or $LeaveOvernight$. But $LeaveOvernight$ is mutex with $Remove(Spare, Trunk)$, so the only solution is to choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$. That brings us to a search state at S_0 with the goals $At(Spare, Trunk)$ and $At(Flat, Axle)$. Both of these are present in the state, so we have a solution: the actions $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0 , followed by $PutOn(Spare, Axle)$ in A_1 .

In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the $(level, goals)$ pair as a **no-good**, just as we did in constraint learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

10.3.3 Termination of GRAPHPLAN

So far, we have skated over the question of termination. Here we show that GRAPHPLAN will in fact terminate and return failure when there is no solution.

The first thing to understand is why we can't stop expanding the graph as soon as it has leveled off. Consider an air cargo domain with one plane and n pieces of cargo at airport A , all of which have airport B as their destination. In this version of the problem, only one piece of cargo can fit in the plane at a time. The graph will level off at level 4, reflecting the fact that for any single piece of cargo, we can load it, fly it, and unload it at the destination in three steps. But that does not mean that a solution can be extracted from the graph at level 4; in fact a solution will require $4n - 1$ steps: for each piece of cargo we load, fly, and unload, and for all but the last piece we need to fly back to airport A to get the next piece.

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution.

Now all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is that certain properties of planning graphs are monotonically increasing or decreasing. “X increases monotonically” means that the set of Xs at level $i + 1$ is a superset (not necessarily proper) of the set at level i . The properties are as follows:

- *Literals increase monotonically*: Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- *Actions increase monotonically*: Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- *Mutexes decrease monotonically*: If two actions are mutex at a given level A_i , then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level S_i nor actions that cannot be executed at level A_i . We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything.

The proof can be handled by cases: if actions A and B are mutex at level A_i , it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i , they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i : that level must contain a precondition of A that is mutex with a precondition of B . Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so, by induction, the mutexes must be decreasing.

- *No-goods decrease monotonically*: If a set of goals is not achievable at a given level, then they are not achievable in any *previous* level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.

Because the actions and literals increase monotonically and because there are only a finite number of actions and literals, there must come a level that has the same number of actions and literals as the previous level. Because mutexes and no-goods decrease, and because there can never be fewer than zero mutexes or no-goods, there must come a level that has the same number of mutexes and no-goods as the previous level. Once a graph has reached this state, then if one of the goals is missing or is mutex with another goal, then we can stop the GRAPHPLAN algorithm and return failure. That concludes a sketch of the proof; for more details see Ghallab *et al.* (2004).

Year	Track	Winning Systems (approaches)
2008	Optimal	GAMER (model checking, bidirectional search)
2008	Satisficing	LAMA (fast downward search with FF heuristic)
2006	Optimal	SATPLAN, MAXPLAN (Boolean satisfiability)
2006	Satisficing	SGPLAN (forward search; partitions into independent subproblems)
2004	Optimal	SATPLAN (Boolean satisfiability)
2004	Satisficing	FAST DIAGONALLY DOWNWARD (forward search with causal graph)
2002	Automated	LPG (local search, planning graphs converted to CSPs)
2002	Hand-coded	TLPLAN (temporal action logic with control rules for forward search)
2000	Automated	FF (forward search)
2000	Hand-coded	TALPLANNER (temporal action logic with control rules for forward search)
1998	Automated	IPP (planning graphs); HSP (forward search)

Figure 10.11 Some of the top-performing systems in the International Planning Competition. Each year there are various tracks: “Optimal” means the planners must produce the shortest possible plan, while “Satisficing” means nonoptimal solutions are accepted. “Hand-coded” means domain-specific heuristics are allowed; “Automated” means they are not.

10.4 OTHER CLASSICAL PLANNING APPROACHES

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics (Section 10.2)
- Search using a planning graph (Section 10.3)

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

10.4.1 Classical planning as Boolean satisfiability

In Section 7.7.4 we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
- Define the initial state: assert F^0 for every fluent F in the problem’s initial state, and $\neg F$ for every fluent not mentioned in the initial state.
- Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A

on another block, $On(A, x) \wedge Block(x)$ in a world with objects A, B and C , would be replaced by the goal

$$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C)).$$

- Add successor-state axioms: For each fluent F , add an axiom of the form

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

where $ActionCausesF$ is a disjunction of all the ground actions that have F in their add list, and $ActionCausesNotF$ is a disjunction of all the ground actions that have F in their delete list.

- Add precondition axioms: For each ground action A , add the axiom $A^t \Rightarrow PRE(A)^t$, that is, if an action is taken at time t , then the preconditions must have been true.
- Add action exclusion axioms: say that every action is distinct from every other action.

The resulting translation is in the form that we can hand to SATPLAN to find a solution.

10.4.2 Planning as first-order logical deduction: Situation calculus

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from A to B regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.

The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, $South^2$ means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching *situations*, using a representation called **situation calculus** that works like this:

- The initial state is called a **situation**. If s is a situation and a is an action, then $RESULT(s, a)$ is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same: $(RESULT(s, a) = RESULT(s', a')) \Leftrightarrow (s = s' \wedge a = a')$. Some examples of actions and situations are shown in Figure 10.12.
- A function or relation that can vary from one situation to the next is a **fluent**. By convention, the situation s is always the last argument to the fluent, for example $At(x, l, s)$ is a relational fluent that is true when object x is at location l in situation s , and $Location$ is a functional fluent such that $Location(x, s) = l$ holds in the same situations as $At(x, l, s)$.
- Each action's preconditions are described with a **possibility axiom** that says when the action can be taken. It has the form $\Phi(s) \Rightarrow Poss(a, s)$ where $\Phi(s)$ is some formula

SITUATION
CALCULUS

SITUATION

POSSIBILITY AXIOM

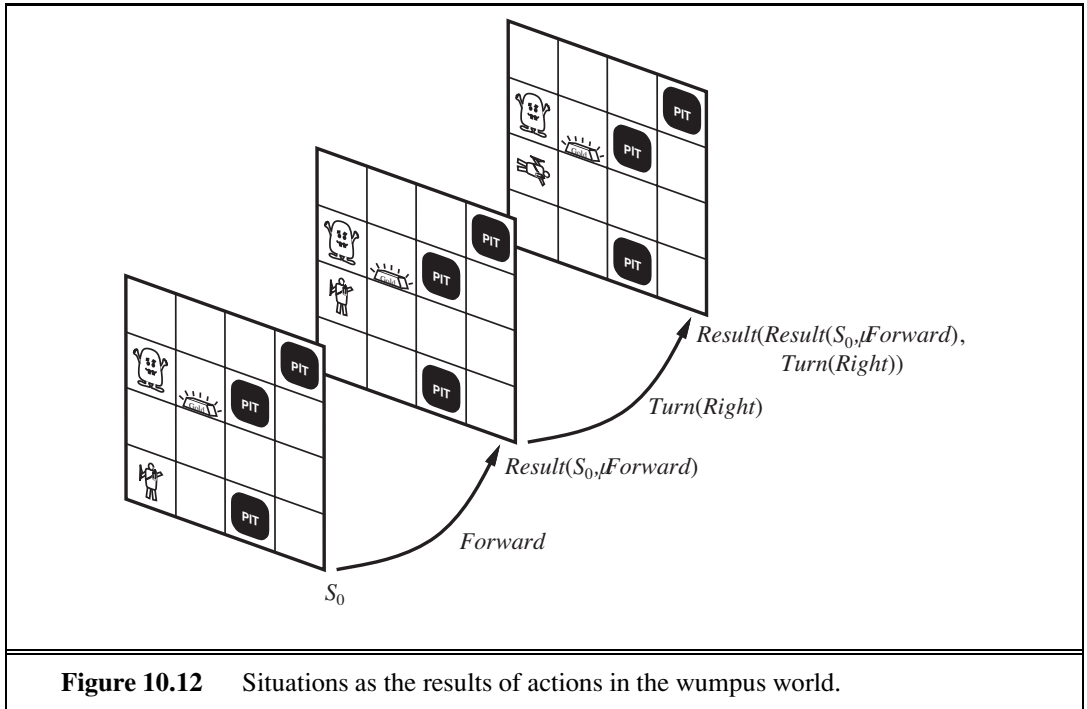


Figure 10.12 Situations as the results of actions in the wumpus world.

involving s that describes the preconditions. An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:

$$Alive(Agent, s) \wedge Have(Arrow, s) \Rightarrow Poss(Shoot, s)$$

- Each fluent is described with a **successor-state axiom** that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form

$$\begin{aligned} \text{Action is possible} &\Rightarrow \\ &(\text{Fluent is true in result state} \Leftrightarrow \text{Action's effect made it true} \\ &\quad \vee \text{It was true before and action left it alone}) . \end{aligned}$$

For example, the axiom for the relational fluent *Holding* says that the agent is holding some gold g after executing a possible action if and only if the action was a *Grab* of g or if the agent was already holding g and the action was not releasing it:

$$\begin{aligned} Poss(a, s) &\Rightarrow \\ &(Holding(Agent, g, Result(a, s)) \Leftrightarrow \\ &\quad a = Grab(g) \vee (Holding(Agent, g, s) \wedge a \neq Release(g))) . \end{aligned}$$

- We need **unique action axioms** so that the agent can deduce that, for example, $a \neq Release(g)$. For each distinct pair of action names A_i and A_j we have an axiom that says the actions are different:

$$A_i(x, \dots) \neq A_j(y, \dots)$$

and for each action name A_i we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

$$A_i(x_1, \dots, x_n) = A_i(y_1, \dots, y_n) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n.$$

- A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Work in situation calculus has done a lot to define the formal semantics of planning and to open up new areas of investigation. But so far there have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus.

10.4.3 Planning as constraint satisfaction

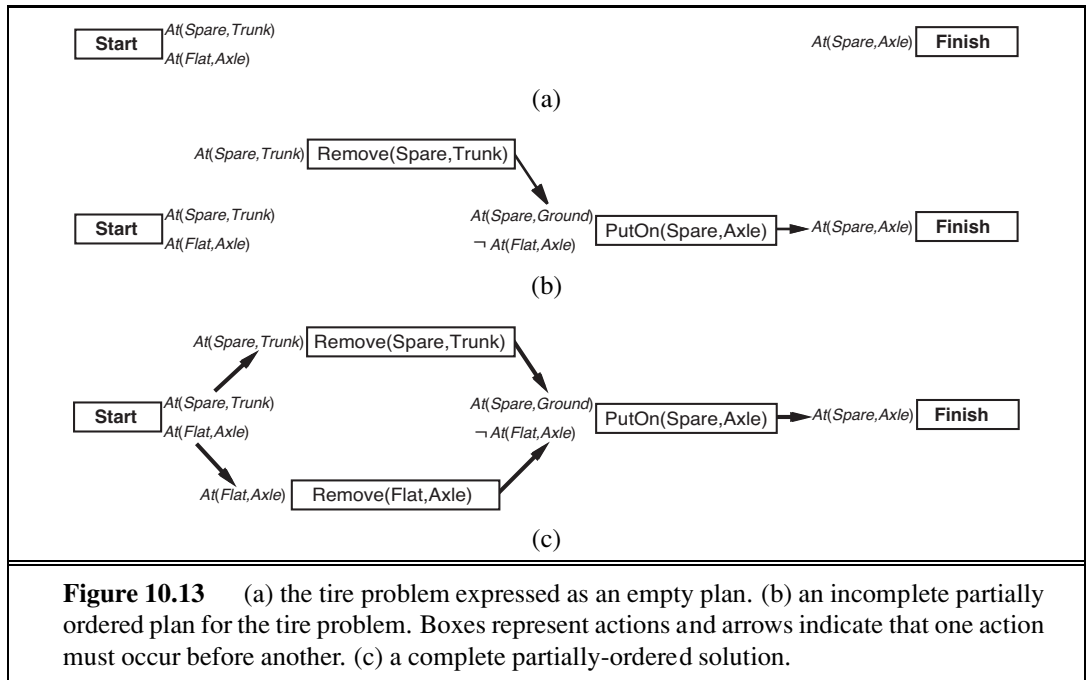
We have seen that constraint satisfaction has a lot in common with Boolean satisfiability, and we have seen that CSP techniques are effective for scheduling problems, so it is not surprising that it is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem (Section 10.4.1), with one important simplification: at each time step we need only a single variable, $Action^t$, whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms. It is also possible to encode a planning graph into a CSP. This is the approach taken by GP-CSP (Do and Kambhampati, 2003).

10.4.4 Planning as refinement of partially ordered plans

All the approaches we have seen so far construct *totally ordered* plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently.

An alternative is to represent plans as *partially ordered* structures: a plan is a set of actions and a set of constraints of the form $Before(a_i, a_j)$ saying that one action occurs before another. In the bottom of Figure 10.13, we see a partially ordered plan that is a solution to the spare tire problem. Actions are boxes and ordering constraints are arrows. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order as long as they are both completed before the $PutOn(Spare, Axle)$ action.

Partially ordered plans are created by a *search through the space of plans* rather than through the state space. We start with the empty plan consisting of just the initial state and the goal, with no actions in between, as in the top of Figure 10.13. The search procedure then looks for a **flaw** in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else). A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves $At(Spare, Axle)$. One way to correct the flaw is to insert into the plan



the action $PutOn(Spare, Axle)$. Of course that introduces some new flaws: the preconditions of the new action are not achieved. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13. At every step, we make the **least commitment** possible to fix the flaw. For example, in adding the action $Remove(Spare, Trunk)$ we need to commit to having it occur before $PutOn(Spare, Axle)$, but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

LEAST COMMITMENT

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial-order planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems.

However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

10.5 ANALYSIS OF PLANNING APPROACHES

Planning combines the two major areas of AI we have covered so far: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are n propositions in a domain, then there are 2^n states. As we have seen, planning is PSPACE-hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B , which in turn is on C , which in turn is on the *Table*, as in Figure 10.4 on page 371), then the subgoals are serializable bottom to top: if we first achieve C on *Table*, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner that commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is *designed* by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

SERIALIZABLE
SUBGOAL

10.6 SUMMARY

In this chapter, we defined the problem of planning in deterministic, fully observable, static environments. We described the PDDL representation for planning problems and several algorithmic approaches for solving them. The points to remember:

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.
- A **planning graph** can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the literals or actions that could occur at that time step and encodes mutual exclusion (mutex) relations among literals or actions that cannot co-occur. Planning graphs yield useful heuristics for state-space and partial-order planners and can be used directly in the GRAPHPLAN algorithm.
- Other approaches include first-order deduction over situation calculus axioms; encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.
- Each of the major approaches to planning has its adherents, and there is as yet no consensus on which is best. Competition and cross-fertilization among the approaches have resulted in significant gains in efficiency for planning systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI. Its overall control structure was modeled on that of GPS, the General Problem Solver (Newell and Simon, 1961), a state-space search system that used means–ends analysis. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and its relationship to more recent planning efforts.

The representation language used by STRIPS has been far more influential than its algorithmic approach; what we call the “classical” language is close to what STRIPS used.

The Action Description Language, or ADL (Pednault, 1986), relaxed some of the STRIPS restrictions and made it possible to encode more realistic problems. Nebel (2000) explores schemes for compiling ADL into STRIPS. The Problem Domain Description Language, or PDDL (Ghallab *et al.*, 1998), was introduced as a computer-parsable, standardized syntax for representing planning problems and has been used as the standard language for the International Planning Competition since 1998. There have been several extensions; the most recent version, PDDL 3.0, includes plan constraints and preferences (Gerevini and Long, 2005).

LINEAR PLANNING

Planners in the early 1970s generally considered totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then stringing the subplans together in some order. This approach, called **linear planning** by Sacerdoti (1975), was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly (see Exercise 10.7), found by Allen Brown during experimentation with the HACKER system (Sussman, 1975). A complete planner must allow for **interleaving** of actions from different subplans within a single sequence. The notion of serializable subgoals (Korf, 1987) corresponds exactly to the set of problems for which noninterleaved planners are complete.

INTERLEAVING

One solution to the interleaving problem was goal-regression planning, a technique in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals. This was introduced by Waldinger (1975) and also used by Warren's (1974) WARPLAN. WARPLAN is also notable in that it was the first planner to be written in a logic programming language (Prolog) and is one of the best examples of the remarkable economy that can sometimes be gained with logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time.

The ideas underlying partial-order planning include the detection of conflicts (Tate, 1975a) and the protection of achieved conditions from interference (Sussman, 1975). The construction of partially ordered plans (then called **task networks**) was pioneered by the NOAH planner (Sacerdoti, 1975, 1977) and by Tate's (1975b, 1977) NONLIN system.

Partial-order planning dominated the next 20 years of research, yet the first clear formal exposition was TWEAK (Chapman, 1987), a planner that was simple enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various planning problems. Chapman's work led to a straightforward description of a complete partial-order planner (McAllester and Rosenblitt, 1991), then to the widely distributed implementations SNLP (Soderland and Weld, 1991) and UCPOP (Penberthy and Weld, 1992). Partial-order planning fell out of favor in the late 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a reconsideration is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN in parallelizable domains and is competitive with the fastest state-space planners.

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest the ignore-delete-list heuristic. The name UNPOP was a reaction to the overwhelming concentration on partial-order planning at the time; McDermott suspected that other approaches were not getting the attention they deserved. Bonet and Geffner's Heuristic Search Planner (HSP) and its later derivatives (Bonet and Geffner, 1999; Haslum *et al.*, 2005; Haslum, 2006) were the first to make

state-space search practical for large planning problems. HSP searches in the forward direction while HSPR (Bonet and Geffner, 1999) searches backward. The most successful state-space searcher to date is FF (Hoffmann, 2001; Hoffmann and Nebel, 2001; Hoffmann, 2005), winner of the AIPS 2000 planning competition. FASTDOWNWARD (Helmert, 2006) is a forward state-space search planner that preprocesses the action schemas into an alternative representation which makes some of the constraints more explicit. FASTDOWNWARD (Helmert and Richter, 2004; Helmert, 2006) won the 2004 planning competition, and LAMA (Richter and Westphal, 2008), a planner based on FASTDOWNWARD with improved heuristics, won the 2008 competition.

Bylander (1994) and Ghallab *et al.* (2004) discuss the computational complexity of several variants of the planning problem. Helmert (2003) proves complexity bounds for many of the standard benchmark problems, and Hoffmann (2005) analyzes the search space of the ignore-delete-list heuristic. Heuristics for the set-covering problem are discussed by Caprara *et al.* (1995) for scheduling operations of the Italian railway. Edelkamp (2009) and Haslum *et al.* (2007) describe how to construct pattern databases for planning heuristics. As we mentioned in Chapter 3, Felner *et al.* (2004) show encouraging results using pattern databases for sliding blocks puzzles, which can be thought of as a planning domain, but Hoffmann *et al.* (2006) show some limitations of abstraction for classical planning problems.

Avrim Blum and Merrick Furst (1995, 1997) revitalized the field of planning with their GRAPHPLAN system, which was orders of magnitude faster than the partial-order planners of the time. Other graph-planning systems, such as IPP (Koehler *et al.*, 1997), STAN (Fox and Long, 1998), and SGP (Weld *et al.*, 1998), soon followed. A data structure closely resembling the planning graph had been developed slightly earlier by Ghallab and Laruelle (1994), whose IXTET partial-order planner used it to derive accurate heuristics to guide searches. Nguyen *et al.* (2001) thoroughly analyze heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes and articles by Subbarao Kambhampati (Bryce and Kambhampati, 2007). As mentioned in the chapter, a planning graph can be used in many different ways to guide the search for a solution. The winner of the 2002 AIPS planning competition, LPG (Gerevini and Serina, 2002, 2003), searched planning graphs using a local search technique inspired by WALKSAT.

The situation calculus approach to planning was introduced by John McCarthy (1963). The version we show here was proposed by Ray Reiter (1991, 2001).

Kautz *et al.* (1996) investigated various ways to propositionalize action schemas, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernst *et al.* (1997), who also developed an automatic “compiler” for generating propositional representations from PDDL problems. The BLACKBOX planner, which combines ideas from GRAPHPLAN and SATPLAN, was developed by Kautz and Selman (1998). CPLAN, a planner based on constraint satisfaction, was described by van Beek and Chen (1999).

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, compact data structures for Boolean expressions widely studied in the hardware verification community (Clarke and Grumberg, 1987; McMillan, 1993). There are techniques for proving properties of binary decision diagrams, including the property of being a solution

to a planning problem. Cimatti *et al.* (1998) present a planner based on this approach. Other representations have also been used; for example, Vossen *et al.* (2001) survey the use of integer programming for planning.

The jury is still out, but there are now some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches such as GRAPHPLAN and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. GRAPHPLAN and SATPLAN have trouble in domains with many objects because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

Readings in Planning (Allen *et al.*, 1990) is a comprehensive anthology of early work in the field. Weld (1994, 1999) provides two excellent surveys of planning algorithms of the 1990s. It is interesting to see the change in the five years between the two surveys: the first concentrates on partial-order planning, and the second introduces GRAPHPLAN and SATPLAN. *Automated Planning* (Ghallab *et al.*, 2004) is an excellent textbook on all aspects of planning. LaValle's text *Planning Algorithms* (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences. There are also specialized conferences such as the International Conference on AI Planning Systems, the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

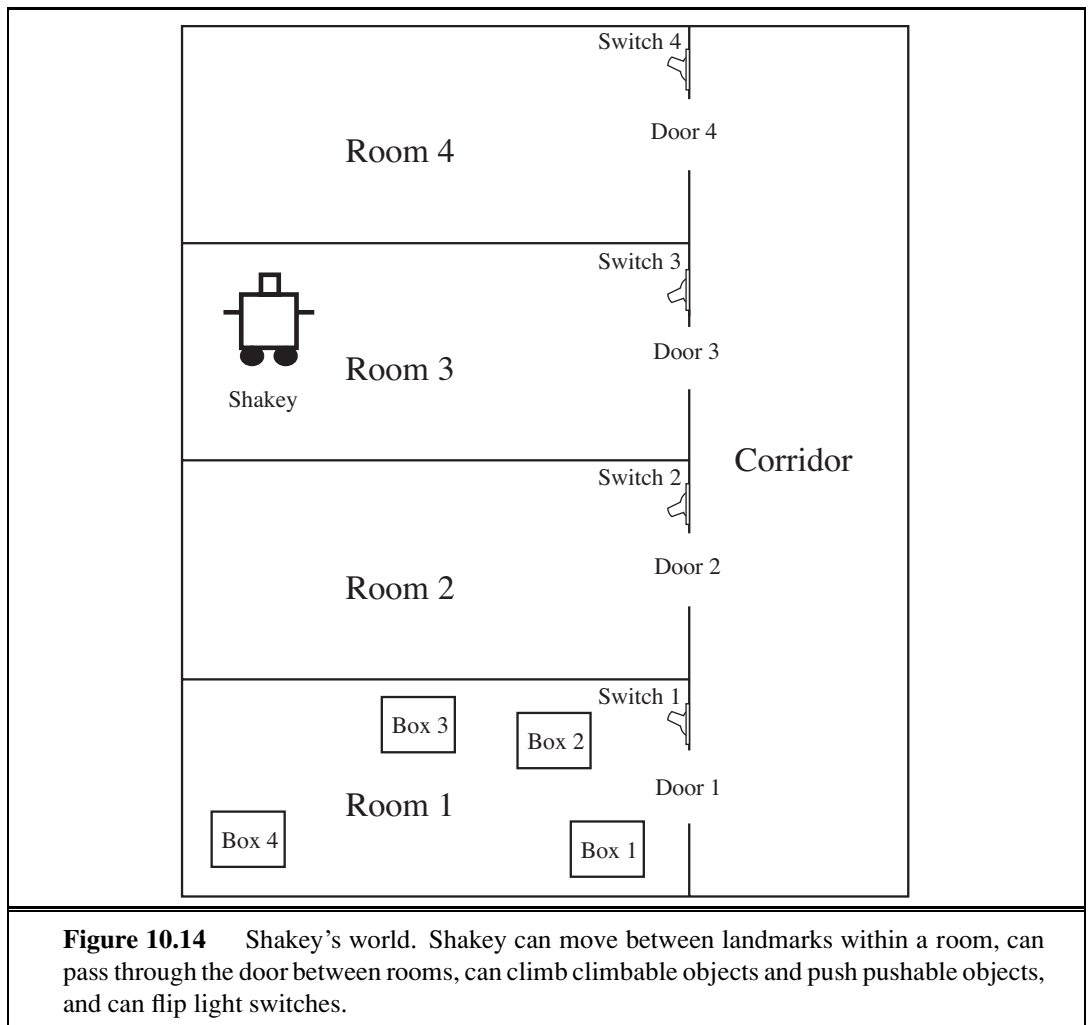
10.1 Describe the differences and similarities between problem solving and planning.

10.2 Given the action schemas and initial state from Figure 10.1, what are all the applicable concrete instances of $Fly(p, from, to)$ in the state described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) ?$$

10.3 The monkey-and-bananas problem is faced by a monkey in a laboratory with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it. Initially, the monkey is at A , the bananas at B , and the box at C . The monkey and box have height *Low*, but if the monkey climbs onto the box he will have height *High*, the same as the bananas. The actions available to the monkey include *Go* from one place to another, *Push* an object from one place to another, *ClimbUp* onto or *ClimbDown* from an object, and *Grasp* or *Ungrasp* an object. The result of a *Grasp* is that the monkey holds the object if the monkey and object are in the same place at the same height.

a. Write down the initial state description.



- b. Write the six action schemas.
- c. Suppose the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Write this as a general goal (i.e., not assuming that the box is necessarily at C) in the language of situation calculus. Can this goal be solved by a classical planning system?
- d. Your schema for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* schema is applied. Fix your action schema to account for heavy objects.

10.4 The original STRIPS planner was designed to control Shakey the robot. Figure 10.14 shows a version of Shakey's world consisting of four rooms lined up along a corridor, where each room has a door and a light switch. The actions in Shakey's world include moving from place to place, pushing movable objects (such as boxes), climbing onto and down from rigid

objects (such as boxes), and turning light switches on and off. The robot itself could not climb on a box or toggle a switch, but the planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y, r)$, which requires that Shakey be *At* x and that x and y are locations *In* the same room r . By convention a door between two rooms is in both of them.
- Push a box b from location x to location y within the same room: $Push(b, x, y, r)$. You will need the predicate *Box* and constants for the boxes.
- Climb onto a box from position x : $ClimbUp(x, b)$; climb down from a box to position x : $ClimbDown(b, x)$. We will need the predicate *On* and the constant *Floor*.
- Turn a light switch on or off: $TurnOn(s, b)$; $TurnOff(s, b)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.

Write PDDL sentences for Shakey's six actions and the initial state from Figure 10.14. Construct a plan for Shakey to get Box_2 into $Room_2$.

10.5 A finite Turing machine has a finite one-dimensional tape of cells, each cell containing one of a finite number of symbols. One cell has a read and write head above it. There is a finite set of states the machine can be in, one of which is the accept state. At each time step, depending on the symbol on the cell under the head and the machine's current state, there are a set of actions we can choose from. Each action involves writing a symbol to the cell under the head, transitioning the machine to a state, and optionally moving the head left or right. The mapping that determines which actions are allowed is the Turing machine's program. Your goal is to control the machine into the accept state.

Represent the Turing machine acceptance problem as a planning problem. If you can do this, it demonstrates that determining whether a planning problem has a solution is at least as hard as the Turing acceptance problem, which is PSPACE-hard.

10.6 Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem.

10.7 Figure 10.4 (page 371) shows a blocks-world problem that is known as the **Sussman anomaly**. The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Write a definition of the problem and solve it, either by hand or with a planning program. A noninterleaved planner is a planner that, when given two subgoals G_1 and G_2 , produces either a plan for G_1 concatenated with a plan for G_2 , or vice versa. Explain why a noninterleaved planner cannot solve this problem.

SUSSMAN ANOMALY

10.8 Prove that backward search with PDDL problems is complete.

10.9 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 10.1.

10.10 Prove the following assertions about planning graphs:

- a. A literal that does not appear in the final level of the graph cannot be achieved.

- b. The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

10.11 The set-level heuristic (see page 382) uses a planning graph to estimate the cost of achieving a conjunctive goal from the current state. What relaxed problem is the set-level heuristic the solution to?

10.12 Examine the definition of **bidirectional search** in Chapter 3.

- a. Would bidirectional state-space search be a good idea for planning?
- b. What about bidirectional search in the space of partial-order plans?
- c. Devise a version of partial-order planning in which an action can be added to a plan if its preconditions can be achieved by the effects of actions already in the plan. Explain how to deal with conflicts and ordering constraints. Is the algorithm essentially identical to forward state-space search?

10.13 We contrasted forward and backward state-space searchers with partial-order planners, saying that the latter is a plan-space searcher. Explain how forward and backward state-space search can also be considered plan-space searchers, and say what the plan refinement operators are.

10.14 Up to now we have assumed that the plans we create always make sure that an action's preconditions are satisfied. Let us now investigate what propositional successor-state axioms such as $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t)$ have to say about actions whose preconditions are not satisfied.

- a. Show that the axioms predict that nothing will happen when an action is executed in a state where its preconditions are not satisfied.
- b. Consider a plan p that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

$$initial\ state \wedge successor\text{-}state\ axioms \wedge p \models goal?$$

- c. With first-order successor-state axioms in situation calculus, is it possible to prove that a plan containing illegal actions will achieve the goal?

10.15 Consider how to translate a set of action schemas into the successor-state axioms of situation calculus.

- a. Consider the schema for $Fly(p, from, to)$. Write a logical definition for the predicate $Poss(Fly(p, from, to), s)$, which is true if the preconditions for $Fly(p, from, to)$ are satisfied in situation s .
- b. Next, assuming that $Fly(p, from, to)$ is the only action schema available to the agent, write down a successor-state axiom for $At(p, x, s)$ that captures the same information as the action schema.

- c. Now suppose there is an additional method of travel: $Teleport(p, from, to)$. It has the additional precondition $\neg Warped(p)$ and the additional effect $Warped(p)$. Explain how the situation calculus knowledge base must be modified.
- d. Finally, develop a general and precisely specified procedure for carrying out the translation from a set of action schemas to a set of successor-state axioms.

10.16 In the SATPLAN algorithm in Figure 7.22 (page 272), each call to the satisfiability algorithm asserts a goal g^T , where T ranges from 0 to T_{\max} . Suppose instead that the satisfiability algorithm is called only once, with the goal $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$.

- a. Will this always return a plan if one exists with length less than or equal to T_{\max} ?
- b. Does this approach introduce any new spurious “solutions”?
- c. Discuss how one might modify a satisfiability algorithm such as WALKSAT so that it finds short solutions (if they exist) when given a disjunctive goal of this form.

11 PLANNING AND ACTING IN THE REAL WORLD

In which we see how more expressive representations and more interactive agent architectures lead to planners that are useful in the real world.

The previous chapter introduced the most basic concepts, representations, and algorithms for planning. Planners that are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This chapter shows how. Section 11.1 extends the classical language for planning to talk about actions with durations and resource constraints. Section 11.2 describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Section 11.3 presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems. Section 11.4 shows how to plan when the environment contains other agents.

11.1 TIME, SCHEDULES, AND RESOURCES

The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is “plan first, schedule later”: that is, we divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

JOB

DURATION

CONSUMABLE

REUSABLE

MAKESPAN

$$\begin{aligned} &Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\}, \\ &\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\}) \\ &Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500)) \\ &Action(AddEngine1, DURATION:30, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddEngine2, DURATION:60, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddWheels1, DURATION:30, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(AddWheels2, DURATION:15, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(Inspect_i, DURATION:10, \\ &\quad USE:Inspectors(1)) \end{aligned}$$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B .

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

11.1.1 Representing temporal and resource constraints

A typical **job-shop scheduling problem**, as first introduced in Section 6.1.2, consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. Each constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be *produced* by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form $[AddEngine, AddWheels, Inspect]$. Then the

Resources statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

AGGREGATION

The representation of resources as numerical quantities, such as *Inspectors*(2), rather than as named entities, such as *Inspector*(I_1) and *Inspector*(I_2), is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent *Inspect* actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

11.1.2 Solving scheduling problems

CRITICAL PATH
METHOD

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*. (For example, there are two paths in the partial-order plan in Figure 11.2.)

CRITICAL PATH

The **critical path** is that path whose total duration is longest; the path is “critical” because it determines the duration of the entire plan—shortening other paths doesn’t shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, *ES*, and a latest possible start time, *LS*. The quantity $LS - ES$ is known as the **slack** of an action. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the *ES* and *LS* times for all the actions constitute a **schedule** for the problem.

SLACK

SCHEDULE

The following formulas serve as a definition for *ES* and *LS* and also as the outline of a dynamic-programming algorithm to compute them. *A* and *B* are actions, and $A \prec B$ means that *A* comes before *B*:

$$\begin{aligned} ES(Start) &= 0 \\ ES(B) &= \max_{A \prec B} ES(A) + Duration(A) \\ LS(Finish) &= ES(Finish) \\ LS(A) &= \min_{B \succ A} LS(B) - Duration(A) . \end{aligned}$$

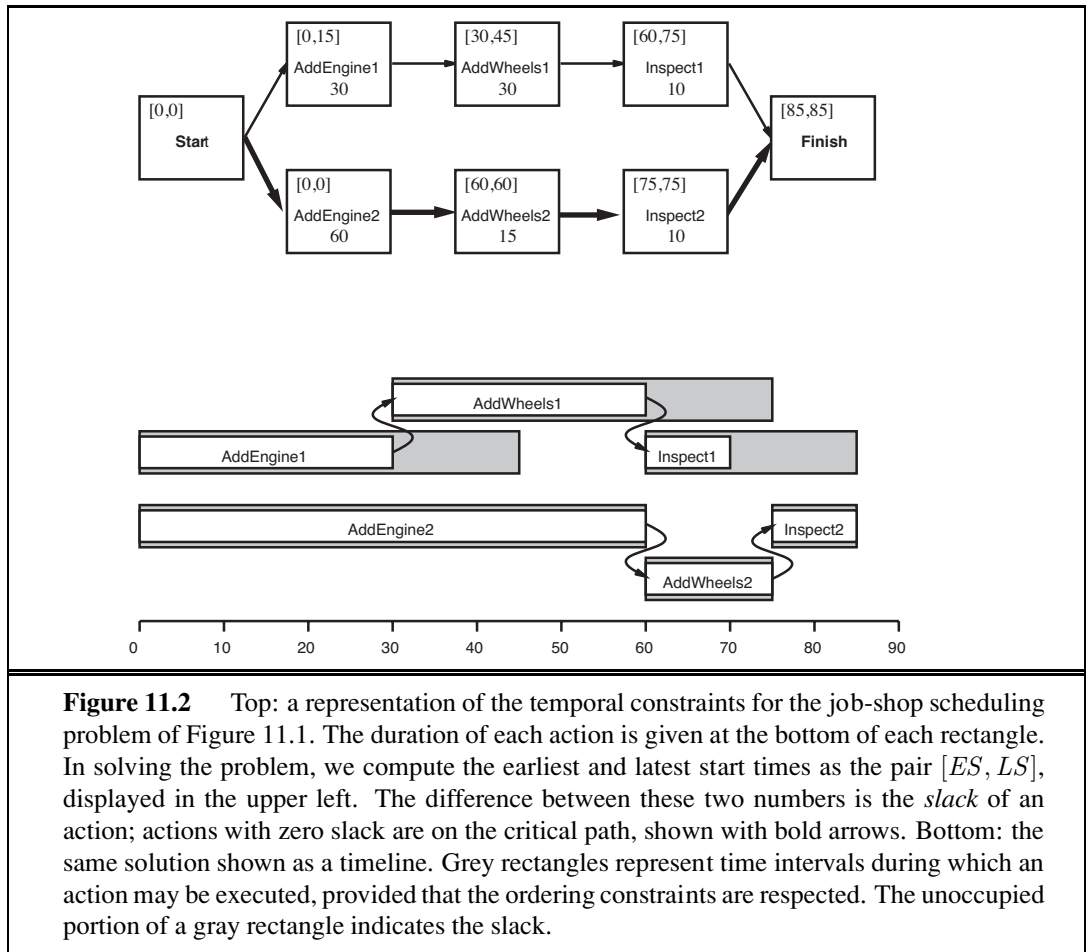


Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

The idea is that we start by assigning $ES(Start)$ to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the *Finish* action.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the *AddEngine* actions, which begin at the same time in Figure 11.2,

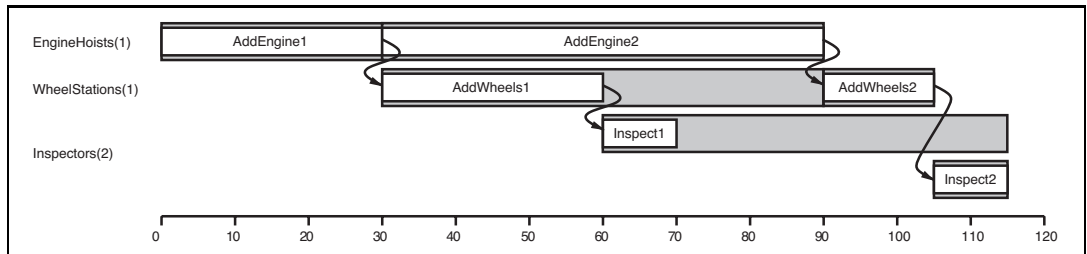


Figure 11.3 A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we’ve shown the shortest-duration solution, which takes 115 minutes.

require the same *EngineHoist* and so cannot overlap. The “cannot overlap” constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.*, 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Chapters 3 and 4. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the *ES* and *LS* times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130-minute solution, not the 115-minute solution of Figure 11.3.

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.

11.2 HIERARCHICAL PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

For plans executed by the human brain, atomic actions are muscle activations. In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all. Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude. Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be “Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.” Given such a plan, the action “Go to San Francisco airport” can be viewed as a planning task in itself, with a solution such as “Drive to the long-term parking lot; park; take the shuttle to the terminal.” Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. We defer discussion of this topic until Section 11.3. Here, we concentrate on the aspect of **hierarchical decomposition**, an idea that pervades almost all attempts to manage complexity. For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices. The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a *large* number of individual actions; for large-scale problems, this is completely impractical.

11.2.1 High-level actions

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. As in classical planning (Chapter 10), we assume full observability and determinism and the availability of a set of actions, now called **primitive actions**, with standard precondition–effect schemas. The key additional concept is the **high-level action** or HLA—for example, the action “Go to San Francisco airport” in the

HIERARCHICAL
DECOMPOSITION

HIERARCHICAL TASK
NETWORK

PRIMITIVE ACTION

HIGH-LEVEL ACTION

```

Refinement(Go(Home, SFO),
  STEPS: [Drive(Home, SFOLongTermParking),
         Shuttle(SFOLongTermParking, SFO)] )
Refinement(Go(Home, SFO),
  STEPS: [Taxi(Home, SFO)] )

Refinement(Navigate([a, b], [x, y]),
  PRECOND: a = x ∧ b = y
  STEPS: [] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a − 1, b])
  STEPS: [Left, Navigate([a − 1, b], [x, y])] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a + 1, b])
  STEPS: [Right, Navigate([a + 1, b], [x, y])] )
...

```

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

REFINEMENT

example given earlier. Each HLA has one or more possible **refinements**, into a sequence¹ of actions, each of which may be an HLA or a primitive action (which has no refinements by definition). For example, the action “Go to San Francisco airport,” represented formally as *Go*(*Home*, *SFO*), might have two possible refinements, as shown in Figure 11.4. The same figure shows a **recursive** refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

These examples show that high-level actions and their refinements embody knowledge about *how to do things*. For instance, the refinements for *Go*(*Home*, *SFO*) say that to get to the airport you can drive or take a taxi; buying milk, sitting down, and moving the knight to e4 are not to be considered.

IMPLEMENTATION

An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. For example, in the vacuum world, the sequences [*Right*, *Right*, *Down*] and [*Down*, *Right*, *Right*] both implement the HLA *Navigate*([1, 3], [3, 2]). An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal. We can say, then, that *a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state*. The “at least one” in this definition is crucial—not *all* implementations need to achieve the goal, because the agent gets



¹ HTN planners often allow refinement into partially ordered plans, and they allow the refinements of two different HLAs in a plan to *share* actions. We omit these important complications in the interest of understanding the basic concepts of hierarchical planning.

to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning. There, we required that a plan work for *all* outcomes because the agent doesn't get to choose the outcome; nature does.

The simplest case is an HLA that has exactly one implementation. In that case, we can compute the preconditions and effects of the HLA from those of the implementation (see Exercise 11.3) and then treat the HLA exactly as if it were a primitive action itself. It can be shown that the right collection of HLAs can result in the time complexity of blind search dropping from exponential in the solution depth to linear in the solution depth, although devising such a collection of HLAs may be a nontrivial task in itself. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works, as in Section 11.2.2; the other is to reason directly about the HLAs—despite the multiplicity of implementations—as explained in Section 11.2.3. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

11.2.2 Searching for primitive solutions

HTN planning is often formulated with a single “top level” action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action a_i , provide one refinement of *Act* with steps $[a_i, Act]$. That creates a recursive definition of *Act* that lets us add actions. But we need some way to stop the recursion; we do that by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal. One possible implementation based on breadth-first tree search is shown in Figure 11.5. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

In essence, this form of hierarchical search explores the space of sequences that conform to the knowledge contained in the HLA library about how things are to be done. A great deal of knowledge can be encoded, not just in the action sequences specified in each refinement but also in the preconditions for the refinements. For some domains, HTN planners have been able to generate huge plans with very little search. For example, O-PLAN (Bell and Tate, 1985), which combines HTN planning with scheduling, has been used to develop production plans for Hitachi. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner generates a 30-day schedule with three 8-hour shifts a day, involving tens of millions of steps. Another important aspect of HTN plans is that they are, by definition, hierarchically structured; usually this makes them easy for humans to understand.

```

function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier  $\leftarrow$  a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest plan in frontier */
    hla  $\leftarrow$  the first HLA in plan, or null if none
    prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
    outcome  $\leftarrow$  RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)

```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

The computational benefits of hierarchical search can be seen by examining an idealized case. Suppose that a planning problem has a solution with d primitive actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^d)$, as explained in Chapter 3. For an HTN planner, let us suppose a very regular refinement structure: each nonprimitive action has r possible refinements, each into k actions at the next lower level. We want to know how many different refinement trees there are with this structure. Now, if there are d actions at the primitive level, then the number of levels below the root is $\log_k d$, so the number of internal refinement nodes is $1 + k + k^2 + \dots + k^{\log_k d - 1} = (d - 1)/(k - 1)$. Each internal node has r possible refinements, so $r^{(d-1)/(k-1)}$ possible regular decomposition trees could be constructed. Examining this formula, we see that keeping r small and k large can result in huge savings: essentially we are taking the k th root of the nonhierarchical cost, if b and r are comparable. Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence (that nonetheless allows us to solve any problem). This is not always possible: long action sequences that are usable across a wide range of problems are extremely precious.

The key to HTN planning, then, is the construction of a plan library containing known methods for implementing complex, high-level actions. One method of constructing the library is to *learn* the methods from problem-solving experience. After the excruciating experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high-level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance (e.g., the name of

the builder or the address of the plot of land) and keeping just the key elements of the plan. Methods for achieving this kind of generalization are described in Chapter 19. It seems to us inconceivable that humans could be as competent as they are without some such mechanism.

11.2.3 Searching for abstract solutions

The hierarchical search algorithm in the preceding section refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan

$[Drive(Home, SFO_{LongTermParking}), Shuttle(SFO_{LongTermParking}, SFO)]$

gets one to the airport without having to determine a precise route, choice of parking spot, and so on. The solution seems obvious: write precondition–effect descriptions of the HLAs, just as we write down what the primitive actions do. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal. This is the holy grail, so to speak, of hierarchical planning because if we derive a high-level plan that provably achieves the goal, working in a small search space of high-level actions, then we can commit to that plan and work on the problem of refining each step of the plan. This gives us the exponential reduction we seek. For this to work, it has to be the case that every high-level plan that “claims” to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

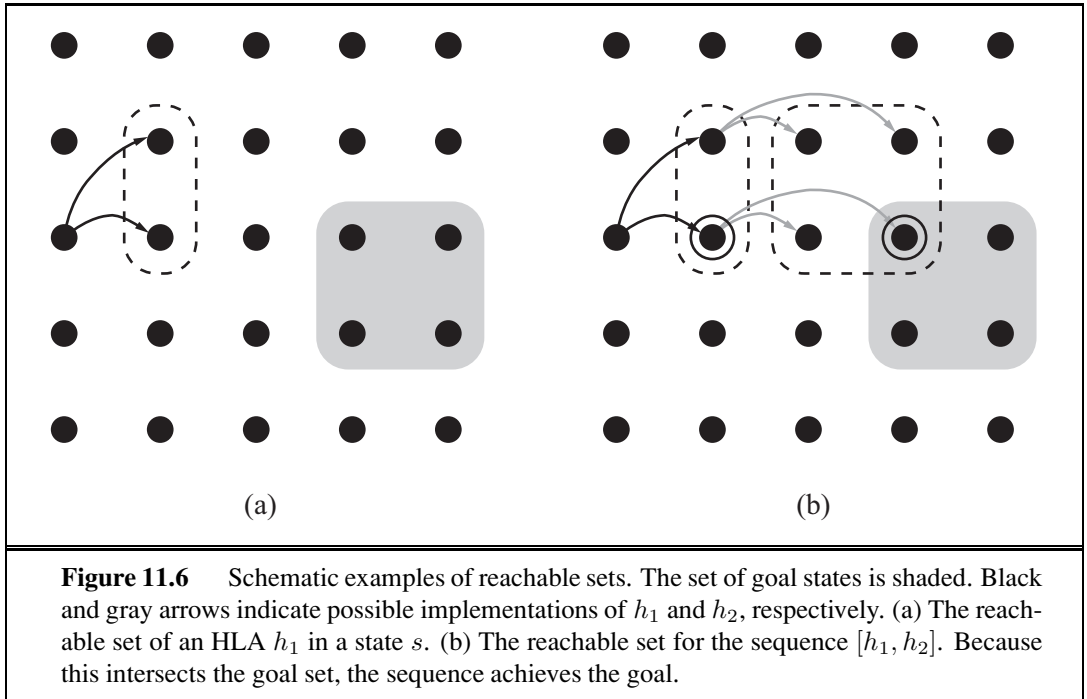
DOWNWARD
REFINEMENT
PROPERTY

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do. We have already seen how to write true descriptions for HLAs that have exactly one implementation (Exercise 11.3); a problem arises when the HLA has *multiple* implementations. How can we describe the effects of an action that can be implemented in many different ways?

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative. Consider again the HLA $Go(Home, SFO)$, which has two refinements, and suppose, for the sake of argument, a simple world in which one can always drive to the airport and park, but taking a taxi requires *Cash* as a precondition. In that case, $Go(Home, SFO)$ doesn’t always get you to the airport. In particular, it fails if *Cash* is false, and so we cannot assert $At(Agent, SFO)$ as an effect of the HLA. This makes no sense, however; if the agent didn’t have *Cash*, it would drive itself. Requiring that an effect hold for *every* implementation is equivalent to assuming that *someone else*—an adversary—will choose the implementation. It treats the HLA’s multiple outcomes exactly as if the HLA were a **nondeterministic** action, as in Section 4.3. For our case, the agent itself will choose the implementation.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nonde-**

DEMONIC
NONDETERMINISM



ANGELIC
NONDETERMINISM
ANGELIC SEMANTICS
REACHABLE SET

terminism, where the agent itself makes the choices. We borrow this term to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state s , the reachable set for an HLA h , written as $\text{REACH}(s, h)$, is the set of states reachable by any of the HLA's implementations. The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more “powerful” than the same HLA with fewer refinements. We can also define the reachable set of a sequences of HLAs. For example, the reachable set of a sequence $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1 :

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2).$$

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. (Compare this to the much stronger condition for demonic semantics, where every member of the reachable set has to be a goal state.) Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. Figure 11.6 illustrates these ideas.

The notion of reachable sets yields a straightforward algorithm: search among high-level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further. We will come back to the algorithmic issues later; first, we consider the question of how the effects of an HLA—the reachable set for each possible initial state—are represented. As with the classical action schemas of Chapter 10, we represent the *changes*

made to each fluent. Think of a fluent as a state variable. A primitive action can *add* or *delete* a variable or leave it *unchanged*. (With conditional effects (see Section 11.3.1) there is a fourth possibility: flipping a variable to its opposite.)

An HLA under angelic semantics can do more: it can *control* the value of a variable, setting it to true or false depending on which implementation is chosen. In fact, an HLA can have nine different effects on a variable: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the variable starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for each case can be combined arbitrarily, making nine. Notationally, this is a bit challenging. We'll use the \sim symbol to mean “possibly, if the agent so chooses.” Thus, an effect $\tilde{+}A$ means “possibly add A ,” that is, either leave A unchanged or make it true. Similarly, $\tilde{-}A$ means “possibly delete A ” and $\tilde{\pm}A$ means “possibly add or delete A .” For example, the HLA $Go(Home, SFO)$, with the two refinements shown in Figure 11.4, possibly deletes $Cash$ (if the agent decides to take a taxi), so it should have the effect $\tilde{-}Cash$. Thus, we see that the descriptions of HLAs are *derivable*, in principle, from the descriptions of their refinements—in fact, this is required if we want true HLA descriptions, such that the downward refinement property holds. Now, suppose we have the following schemas for the HLAs h_1 and h_2 :

$$\begin{aligned} Action(h_1, \text{PRECOND: } \neg A, \text{EFFECT: } A \wedge \tilde{-}B) , \\ Action(h_2, \text{PRECOND: } \neg B, \text{EFFECT: } \tilde{+}A \wedge \tilde{\pm}C) . \end{aligned}$$

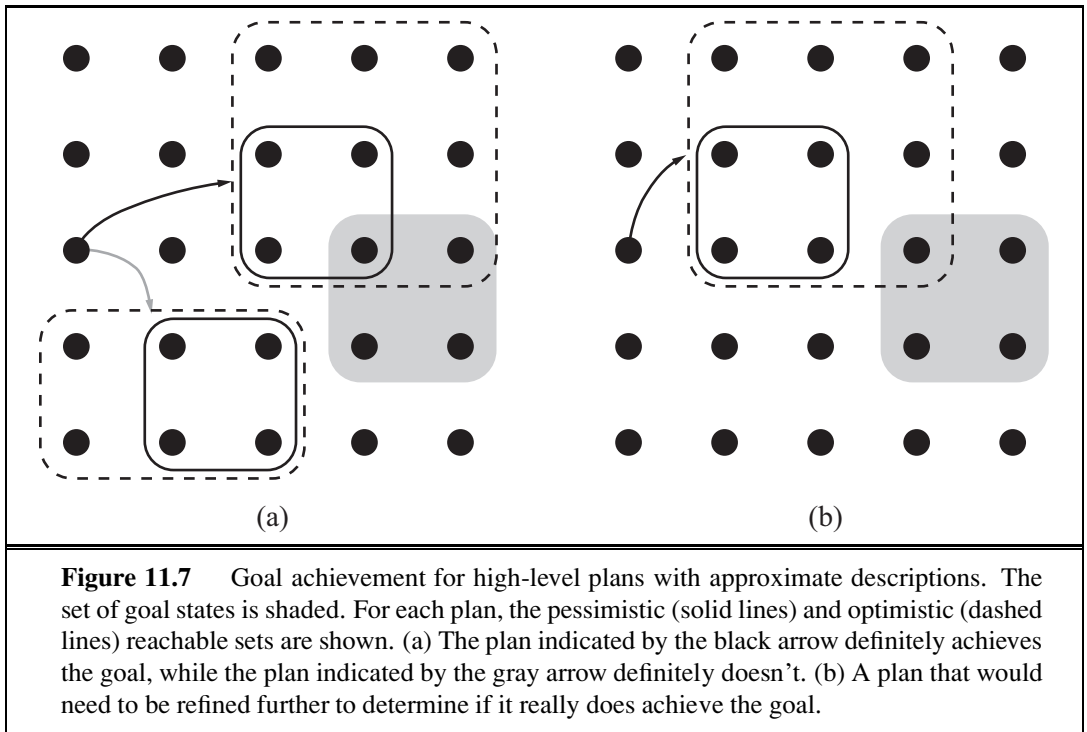
That is, h_1 adds A and possibly deletes B , while h_2 possibly adds A and has full control over C . Now, if only B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal: we choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

The preceding discussion assumes that the effects of an HLA—the reachable set for any given initial state—can be described exactly by describing the effect on each variable. It would be nice if this were always true, but in many cases we can only approximate the effects because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets—rather like the wiggly-belief-state problem illustrated in Figure 7.21 on page 271. For example, we said that $Go(Home, SFO)$ possibly deletes $Cash$; it also possibly adds $At(Car, SFO\text{LongTermParking})$; but it cannot do both—in fact, it must do exactly one. As with belief states, we may need to write *approximate* descriptions. We will use two kinds of approximation: an **optimistic description** $REACH^+(s, h)$ of an HLA h may overstate the reachable set, while a **pessimistic description** $REACH^-(s, h)$ may understate the reachable set. Thus, we have

$$REACH^-(s, h) \subseteq REACH(s, h) \subseteq REACH^+(s, h) .$$

For example, an optimistic description of $Go(Home, SFO)$ says that it possibly deletes $Cash$ and possibly adds $At(Car, SFO\text{LongTermParking})$. Another good example arises in the 8-puzzle, half of whose states are unreachable from any given state (see Exercise 3.4 on page 113): the optimistic description of Act might well include the whole state space, since the exact reachable set is quite wiggly.

With approximate descriptions, the test for whether a plan achieves the goal needs to be modified slightly. If the optimistic reachable set for the plan doesn't intersect the goal,



then the plan doesn't work; if the pessimistic reachable set intersects the goal, then the plan does work (Figure 11.7(a)). With exact descriptions, a plan either works or it doesn't, but with approximate descriptions, there is a middle ground: if the optimistic set intersects the goal but the pessimistic set doesn't, then we cannot tell if the plan works (Figure 11.7(b)). When this circumstance arises, the uncertainty can be resolved by refining the plan. This is a very common situation in human reasoning. For example, in planning the aforementioned two-week Hawaii vacation, one might propose to spend two days on each of seven islands. Prudence would indicate that this ambitious plan needs to be refined by adding details of inter-island transportation.

An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure 11.8. For simplicity, we have kept to the same overall scheme used previously in Figure 11.5, that is, a breadth-first search in the space of refinements. As just explained, the algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. (The details of how to compute the reachable sets of a plan, given approximate descriptions of each step, are covered in Exercise 11.5.) When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. (See Section 10.2.2 for a discussion of how regression works.) Figure 11.6(b) illustrates the basic idea: the right-hand circled state is the guaranteed-reachable goal state, and the left-hand circled state is the intermediate goal obtained by regressing the


```

function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns solution or fail
  frontier  $\leftarrow$  a FIFO queue with initialPlan as the only element
  loop do
    if EMPTY?(frontier) then return fail
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    if REACH+(problem.INITIAL-STATE, plan) intersects problem.GOAL then
      if plan is primitive then return plan /* REACH+ is exact for primitive plans */
      guaranteed  $\leftarrow$  REACH-(problem.INITIAL-STATE, plan)  $\cap$  problem.GOAL
      if guaranteed  $\neq \{\}$  and MAKING-PROGRESS(plan, initialPlan) then
        finalState  $\leftarrow$  any element of guaranteed
        return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
      hla  $\leftarrow$  some HLA in plan
      prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
      for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)



---


function DECOMPOSE(hierarchy, s0, plan, sf) returns a solution
  solution  $\leftarrow$  an empty plan
  while plan is not empty do
    action  $\leftarrow$  REMOVE-LAST(plan)
    si  $\leftarrow$  a state in REACH-(s0, plan) such that sf  $\in$  REACH-(si, action)
    problem  $\leftarrow$  a problem with INITIAL-STATE = si and GOAL = sf
    solution  $\leftarrow$  APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf  $\leftarrow$  si
  return solution

```

Figure 11.8 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the *initialPlan*.

goal through the final action.

The ability to commit to or reject high-level plans can give ANGELIC-SEARCH a significant computational advantage over HIERARCHICAL-SEARCH, which in turn may have a large advantage over plain old BREADTH-FIRST-SEARCH. Consider, for example, cleaning up a large vacuum world consisting of rectangular rooms connected by narrow corridors. It makes sense to have an HLA for *Navigate* (as shown in Figure 11.4) and one for *CleanWholeRoom*. (Cleaning the room could be implemented with the repeated application of another HLA to clean each row.) Since there are five actions in this domain, the cost for BREADTH-FIRST-SEARCH grows as 5^d , where d is the length of the shortest solution (roughly twice the total number of squares); the algorithm cannot manage even two 2×2 rooms. HIERARCHICAL-SEARCH is more efficient, but still suffers from exponential growth because it tries all ways of cleaning that are consistent with the hierarchy. ANGELIC-SEARCH scales approximately linearly in the number of squares—it commits to a good high-level se-

quence and prunes away the other options. Notice that cleaning a set of rooms by cleaning each room in turn is hardly rocket science: it is easy for humans precisely because of the hierarchical structure of the task. When we consider how difficult humans find it to solve small puzzles such as the 8-puzzle, it seems likely that the human capacity for solving complex problems derives to a great extent from their skill in abstracting and decomposing the problem to eliminate combinatorics.

The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set. Instead of a state being reachable or not, it has a cost for the most efficient way to get there. (The cost is ∞ for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without considering their implementations. The same approach can be used to obtain effective **hierarchical lookahead** algorithms for online search, in the style of LRTA* (page 152). In some ways, such algorithms mirror aspects of human deliberation in tasks such as planning a vacation to Hawaii—consideration of alternatives is done initially at an abstract level over long time scales; some parts of the plan are left quite abstract until execution time, such as how to spend two lazy days on Molokai, while others parts are planned in detail, such as the flights to be taken and lodging to be reserved—without these refinements, there is no guarantee that the plan would be feasible.

HIERARCHICAL
LOOKAHEAD

11.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. Chapter 4 extended search in similar ways, and the methods here are also similar: **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments.

While the basic concepts are the same as in Chapter 4, there are also significant differences. These arise because planners deal with factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent **belief states**—the sets of possible physical states the agent might be in—for unobservable and partially observable environments. We can also take advantage of many of the domain-independent methods given in Chapter 10 for calculating search heuristics.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

$$\begin{aligned} &Init(Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2) \wedge InView(Table)) \\ &Goal(Color(Chair, c) \wedge Color(Table, c)) \end{aligned}$$

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can. The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable

list. That is, $Paint(x, can)$ does not mention the variable c , representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action $Paint(x, can, c)$. But in the partially observable case, we might or might not know what color is in the can. (The variable c is universally quantified, just like all the other variables in an action schema.)

$$\begin{aligned} &Action(RemoveLid(can), \\ &\quad PRECOND: Can(can) \\ &\quad EFFECT: Open(can)) \\ &Action(Paint(x, can), \\ &\quad PRECOND: Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can) \\ &\quad EFFECT: Color(x, c)) \end{aligned}$$

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, $PERCEPT(s)$. For planning, we augment PDDL with a new type of schema, the **percept schema**:

PERCEPT SCHEMA

$$\begin{aligned} &Percept(Color(x, c), \\ &\quad PRECOND: Object(x) \wedge InView(x)) \\ &Percept(Color(can, c), \\ &\quad PRECOND: Can(can) \wedge InView(can) \wedge Open(can)) \end{aligned}$$

The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object x , the agent will learn the truth value of $Color(x, c)$ for all c). The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$\begin{aligned} &Action(LookAt(x), \\ &\quad PRECOND: InView(y) \wedge (x \neq y) \\ &\quad EFFECT: InView(x) \wedge \neg InView(y)) \end{aligned}$$

For a fully observable environment, we would have a *Percept* axiom with no preconditions for each fluent. A sensorless agent, on the other hand, has no *Percept* axioms at all. Note that even a sensorless agent can solve the painting problem. One solution is to open any can of paint and apply it to both chair and table, thus **coercing** them to be the same color (even though the agent doesn't know what the color is).

A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done. If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal

with problems when they arise by replanning. It could also deal with incorrectness of its action schemas. Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes. On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents. In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening. Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning. We next look at each of the three approaches in more detail.

11.3.1 Sensorless planning

Section 4.4.1 (page 138) introduced the basic idea of searching in belief-state space to find a solution for sensorless problems. Conversion of a sensorless planning problem to a belief-state planning problem works much the same way as it did in Section 4.4.1; the main differences are that the underlying physical transition model is represented by a collection of action schemas and the belief state can be represented by a logical formula instead of an explicitly enumerated set of states. For simplicity, we assume that the underlying planning problem is deterministic.

The initial belief state for the sensorless painting problem can ignore *InView* fluents because the agent has no sensors. Furthermore, we take as given the unchanging facts $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$ because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: $\forall x \exists c Color(x, c)$. After Skolemizing, (see Section 9.5), we obtain the initial belief state:

$$b_0 = Color(x, C(x)) .$$

In classical planning, where the **closed-world assumption** is made, we would assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula. Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] .$$

We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

First, note that in a given belief state b , the agent can consider any action whose preconditions are satisfied by b . (The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.) According

to Equation (4.4) (page 139), the general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

where RESULT_P defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b' , we must consider what happens to each literal ℓ in each physical state s in b when action a is applied. For literals whose truth value is already known in b , the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b' .) What about a literal whose truth value is unknown in b ? There are three cases:

1. If the action adds ℓ , then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ , then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ , then ℓ will retain its initial value (which is unknown) and will not appear in b' .

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (10.1) on page 368:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a) .$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\text{RESULT}(b, a)$ is computed by starting with b , setting any atom that appears in $\text{DEL}(a)$ to false, and setting any atom that appears in $\text{ADD}(a)$ to true. For example, if we apply $\text{RemoveLid}(\text{Can}_1)$ to the initial belief state b_0 , we get

$$b_1 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) .$$

When we apply the action $\text{Paint}(\text{Chair}, \text{Can}_1)$, the precondition $\text{Color}(\text{Can}_1, c)$ is satisfied by the known literal $\text{Color}(x, C(x))$ with binding $\{x/\text{Can}_1, c/C(\text{Can}_1)\}$ and the new belief state is

$$b_2 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) .$$

Finally, we apply the action $\text{Paint}(\text{Table}, \text{Can}_1)$ to obtain

$$b_3 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) \\ \wedge \text{Color}(\text{Table}, C(\text{Can}_1)) .$$

The final belief state satisfies the goal, $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{Chair}, c)$, with the variable c bound to $C(\text{Can}_1)$.



The preceding analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas*. That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with n fluents, any belief state can be represented by a conjunction of size $O(n)$. This is a very comforting result, considering that there are 2^n states in the world. It says we can compactly represent all the subsets of those 2^n states that we will ever need. Moreover, the process of checking for belief

states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

The fly in the ointment of this pleasant picture is that it only works for action schemas that have the *same effects* for all states in which their preconditions are satisfied. It is this property that enables the preservation of the 1-CNF belief-state representation. As soon as the effect can depend on the state, dependencies are introduced between fluents and the 1-CNF property is lost. Consider, for example, the simple vacuum world defined in Section 3.2.1. Let the fluents be *AtL* and *AtR* for the location of the robot and *CleanL* and *CleanR* for the state of the squares. According to the definition of the problem, the *Suck* action has no precondition—it can always be done. The difficulty is that its effect depends on the robot’s location: when the robot is *AtL*, the result is *CleanL*, but when it is *AtR*, the result is *CleanR*. For such actions, our action schemas will need something new: a **conditional effect**. These have the syntax “**when** *condition*: *effect*,” where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state. For the vacuum world, we have

Action(*Suck*,
EFFECT:**when** *AtL*: *CleanL* \wedge **when** *AtR*: *CleanR*) .

When applied to the initial belief state *True*, the resulting belief state is $(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$, which is no longer in 1-CNF. (This transition can be seen in Figure 4.14 on page 141.) In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states of exponential size in the worst case.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action. For example, we could split *Suck* into two actions with unconditional effects as follows:

Action(*SuckL*,
PRECOND:*AtL*; EFFECT:*CleanL*)
Action(*SuckR*,
PRECOND:*AtR*; EFFECT:*CleanR*) .

Now we have only unconditional schemas, so the belief states all remain in 1-CNF; unfortunately, we cannot determine the applicability of *SuckL* and *SuckR* in the initial belief state.

It seems inevitable, then, that nontrivial problems will involve wiggly belief states, just like those encountered when we considered the problem of state estimation for the wumpus world (see Figure 7.21 on page 271). The solution suggested then was to use a **conservative approximation** to the exact belief state; for example, the belief state can remain in 1-CNF if it contains all literals whose truth values can be determined and treats all other literals as unknown. While this approach is *sound*, in that it never generates an incorrect plan, it is *incomplete* because it may be unable to find solutions to problems that necessarily involve interactions among literals. To give a trivial example, if the goal is for the robot to be on

a clean square, then $[Suck]$ is a solution but a sensorless agent that insists on 1-CNF belief states will not find it.

Perhaps a better solution is to look for action sequences that keep the belief state as simple as possible. For example, in the sensorless vacuum world, the action sequence $[Right, Suck, Left, Suck]$ generates the following sequence of belief states:

$$\begin{aligned} b_0 &= True \\ b_1 &= AtR \\ b_2 &= AtR \wedge CleanR \\ b_3 &= AtL \wedge CleanR \\ b_4 &= AtL \wedge CleanR \wedge CleanL \end{aligned}$$

That is, the agent *can* solve the problem while retaining a 1-CNF belief state, even though some sequences (e.g., those beginning with *Suck*) go outside 1-CNF. The general lesson is not lost on humans: we are always performing little actions (checking the time, patting our pockets to make sure we have the car keys, reading street signs as we navigate through a city) to eliminate uncertainty and keep our belief state manageable.

There is another, quite different approach to the problem of unmanageably wiggly belief states: don't bother computing them at all. Suppose the initial belief state is b_0 and we would like to know the belief state resulting from the action sequence $[a_1, \dots, a_m]$. Instead of computing it explicitly, just represent it as " b_0 then $[a_1, \dots, a_m]$." This is a lazy but unambiguous representation of the belief state, and it's quite concise— $O(n + m)$ where n is the size of the initial belief state (assumed to be in 1-CNF) and m is the maximum length of an action sequence. As a belief-state representation, it suffers from one drawback, however: determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

The computation can be implemented as an entailment test: if A_m represents the collection of successor-state axioms required to define occurrences of the actions a_1, \dots, a_m —as explained for SATPLAN in Section 10.4.1—and G_m asserts that the goal is true after m steps, then the plan achieves the goal if $b_0 \wedge A_m \models G_m$, that is, if $b_0 \wedge A_m \wedge \neg G_m$ is unsatisfiable. Given a modern SAT solver, it may be possible to do this much more quickly than computing the full belief state. For example, if none of the actions in the sequence has a particular goal fluent in its add list, the solver will detect this immediately. It also helps if partial results about the belief state—for example, fluents known to be true or false—are cached to simplify subsequent computations.

The final piece of the sensorless planning puzzle is a heuristic function to guide the search. The meaning of the heuristic function is the same as for classical planning: an estimate (perhaps admissible) of the cost of achieving the goal from the given belief state. With belief states, we have one additional fact: solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2) .$$

Hence, any admissible heuristic computed for a subset is admissible for the belief state itself. The most obvious candidates are the singleton subsets, that is, individual physical states. We

can take any random collection of states s_1, \dots, s_N that are in the belief state b , apply any admissible heuristic h from Chapter 10, and return

$$H(b) = \max\{h(s_1), \dots, h(s_N)\}$$

as the heuristic estimate for solving b . We could also use a planning graph directly on b itself: if it is a conjunction of literals (1-CNF), simply set those literals to be the initial state layer of the graph. If b is not in 1-CNF, it may be possible to find sets of literals that together entail b . For example, if b is in disjunctive normal form (DNF), each term of the DNF formula is a conjunction of literals that entails b and can form the initial layer of a planning graph. As before, we can take the maximum of the heuristics obtained from each set of literals. We can also use inadmissible heuristics such as the ignore-delete-lists heuristic (page 377), which seems to work quite well in practice.

11.3.2 Contingent planning

We saw in Chapter 4 that contingent planning—the generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, non-determinism, or both. For the partially observable painting problem with the percept axioms given earlier, one possible contingent solution is as follows:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1), RemoveLid(Can2), LookAt(Can2),
    if Color(Table, c) ∧ Color(can, c) then Paint(Chair, can)
    else if Color(Chair, c) ∧ Color(can, c) then Paint(Table, can)
    else [Paint(Chair, Can1), Paint(Table, Can1)]]]
```

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation. (It is up to the contingent-planning algorithm to make sure that the agent will never end up in a belief state where the condition formula's truth value is unknown.) Note that with first-order conditions, the formula may be satisfied in more than one way; for example, the condition $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{can}, c)$ might be satisfied by $\{\text{can}/\text{Can}_1\}$ and by $\{\text{can}/\text{Can}_2\}$ if both cans are the same color as the table. In that case, the agent can choose any satisfying substitution to apply to the rest of the plan.

As shown in Section 4.4.2, calculating the new belief state after an action and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:

$$\hat{b} = (b - \text{DEL}(a)) \cup \text{ADD}(a)$$

where, as before, we have assumed a belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals p_1, \dots, p_k are received. One might think that we simply need to add these into the belief state; in fact, we can also infer

that the preconditions for sensing are satisfied. Now, if a percept p has exactly one percept axiom, $Percept(p, PRECOND:c)$, where c is a conjunction of literals, then those literals can be thrown into the belief state along with p . On the other hand, if p has more than one percept axiom whose preconditions might hold according to the predicted belief state \hat{b} , then we have to add in the *disjunction* of the preconditions. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, we can generate contingent plans with an extension of the AND–OR forward search over belief states used in Section 4.4. Actions with nondeterministic effects—which are defined simply by using a disjunction in the EFFECT of the action schema—can be accommodated with minor changes to the belief-state update calculation and no change to the search algorithm.² For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

11.3.3 Online replanning

Imagine watching a spot-welding robot in a car plant. The robot’s fast, accurate motions are repeated over and over again as each car passes down the line. Although technically impressive, the robot probably does not seem at all *intelligent* because the motion is a fixed, preprogrammed sequence; the robot obviously doesn’t “know what it’s doing” in any meaningful sense. Now suppose that a poorly attached door falls off the car just as the robot is about to apply a spot-weld. The robot quickly replaces its welding actuator with a gripper, picks up the door, checks it for scratches, reattaches it to the car, sends an email to the floor supervisor, switches back to the welding actuator, and resumes its work. All of a sudden, the robot’s behavior seems *purposive* rather than rote; we assume it results not from a vast, precomputed contingent plan but from an online replanning process—which means that the robot *does* need to know what it’s trying to do.

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency, such as whether the sky might fall on its head.³ Some branches of a partially constructed contingent plan can simply say *Replan*; if such a branch is reached during execution, the agent reverts to planning mode. As we mentioned earlier, the decision as to how much of the problem to solve in advance and how much to leave to replanning is one that involves tradeoffs among possible events with different costs and probabilities of occurring. Nobody wants to have their car break down in the middle of the Sahara desert and only then think about having enough water.

EXECUTION
MONITORING

² If cyclic solutions are required for a nondeterministic problem, AND–OR search must be generalized to a loopy version such as LAO* (Hansen and Zilberstein, 2001).

³ In 1954, a Mrs. Hodges of Alabama was hit by meteorite that crashed through her roof. In 1992, a piece of the Mbale meteorite hit a small boy on the head; fortunately, its descent was slowed by banana leaves (Jenniskens *et al.*, 1994). And in 2009, a German boy claimed to have been hit in the hand by a pea-sized meteorite. No serious injuries resulted from any of these incidents, suggesting that the need for preplanning against such contingencies is sometimes overstated.

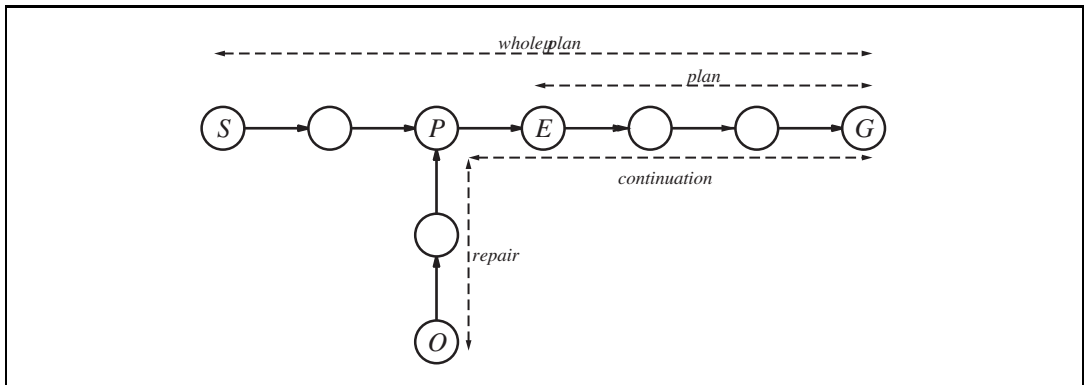


Figure 11.9 Before execution, the planner comes up with a plan, here called *whole plan*, to get from *S* to *G*. The agent executes steps of the plan until it expects to be in state *E*, but observes it is actually in *O*. The agent then replans for the minimal repair plus continuation to reach *G*.

MISSING
PRECONDITION

MISSING EFFECT
MISSING STATE
VARIABLE

EXOGENOUS EVENT

Replanning may also be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver; the model may have a **missing effect**—for example, painting an object may get paint on the floor as well; or the model may have a **missing state variable**—for example, the model given earlier has no notion of the amount of paint in a can, of how its actions affect this amount, or of the need for the amount to be nonzero. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be extremely fragile if it relies on absolute correctness of its model.

The online agent has a choice of how carefully to monitor the environment. We distinguish three levels:

ACTION MONITORING

- **Action monitoring:** before executing an action, the agent verifies that all the preconditions still hold.

PLAN MONITORING

- **Plan monitoring:** before executing an action, the agent verifies that the remaining plan will still succeed.

GOAL MONITORING

- **Goal monitoring:** before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

In Figure 11.9 we see a schematic of action monitoring. The agent keeps track of both its original plan, *wholeplan*, and the part of the plan that has not been executed yet, which is denoted by *plan*. After executing the first few steps of the plan, the agent expects to be in state *E*. But the agent observes it is actually in state *O*. It then needs to repair the plan by finding some point *P* on the original plan that it can get back to. (It may be that *P* is the goal state, *G*.) The agent tries to minimize the total cost of the plan: the repair part (from *O* to *P*) plus the continuation (from *P* to *G*).

Now let's return to the example problem of achieving a chair and table of matching color. Suppose the agent comes up with this plan:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1),
    if Color(Table, c) ∧ Color(Can1, c) then Paint(Chair, Can1)
    else REPLAN]] .
```

Now the agent is ready to execute the plan. Suppose the agent observes that the table and can of paint are white and the chair is black. It then executes *Paint(Chair, Can₁)*. At this point a classical planner would declare victory; the plan has been executed. But an online execution monitoring agent needs to check the preconditions of the remaining empty plan—that the table and chair are the same color. Suppose the agent perceives that they do not have the same color—in fact, the chair is now a mottled gray because the black paint is showing through. The agent then needs to figure out a position in *whole plan* to aim for and a repair action sequence to get there. The agent notices that the current state is identical to the precondition before the *Paint(Chair, Can₁)* action, so the agent chooses the empty sequence for *repair* and makes its *plan* be the same [*Paint*] sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the *Paint* action is retried. This behavior will loop until the chair is perceived to be completely painted. But notice that the loop is created by a process of plan–execute–replan, rather than by an explicit loop in a plan. Note also that the original plan need not cover every contingency. If the agent reaches the step marked REPLAN, it can then generate a new plan (perhaps involving *Can₂*).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red. Plan monitoring achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs.⁴ Plan monitoring also allows for **serendipity**—accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly

⁴ Plan monitoring means that finally, after 424 pages, we have an agent that is smarter than a dung beetle (see page 39). A plan-monitoring agent would notice that the dung ball was missing from its grasp and would replan to get another ball and plug its hole.

more complex to enable plan monitoring. Partial-order and planning-graph planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

Now that we have described a method for monitoring and replanning, we need to ask, “Does it work?” This is a surprisingly tricky question. If we mean, “Can we guarantee that the agent will always achieve the goal?” then the answer is no, because the agent could inadvertently arrive at a dead end from which there is no repair. For example, the vacuum agent might have a faulty model of itself and not know that its batteries can run out. Once they do, it cannot repair any plans. If we rule out dead ends—assume that there exists a plan to reach the goal from *any* state in the environment—and assume that the environment is really nondeterministic, in the sense that such a plan always has *some* chance of success on any given execution attempt, then the agent will eventually reach the goal.

Trouble occurs when an action is actually not nondeterministic, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this.⁵ One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair. This kind of learning is described in Chapters 18 and 19.

11.4 MULTIAGENT PLANNING

So far, we have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a **multiagent planning problem** in which it tries to achieve its own goals with the help or hindrance of others.

Between the purely single-agent and truly multiagent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do **multieffector planning** to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory—multieffector planning becomes **multibody planning**. A multibody problem is still a “standard” single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body. When communication constraints make this impossible, we have

MULTIAGENT
PLANNING PROBLEM

MULTIEFFECTOR
PLANNING

MULTIBODY
PLANNING

⁵ Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp (page 39).

DECENTRALIZED
PLANNING

what is sometimes called a **decentralized planning** problem; this is perhaps a misnomer, because the planning phase is centralized but the execution phase is at least partially decoupled. In this case, the subplan constructed for each body may need to include explicit communicative actions with other bodies. For example, multiple reconnaissance robots covering a wide area may often be out of radio contact with each other and should share their findings during times when communication is feasible.

COORDINATION

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multiagent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multiagent doubles team, on the other hand, each agent decides what to do; without some method for **coordination**, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

The clearest case of a multiagent problem, of course, is when the agents have different goals. In tennis, the goals of two opposing teams are in direct conflict, leading to the zero-sum situation of Chapter 5. Spectators could be viewed as agents if their support or disdain is a significant factor and can be influenced by the players' conduct; otherwise, they can be treated as an aspect of nature—just like the weather—that is assumed to be indifferent to the players' intentions.⁶

INCENTIVE

Finally, some systems are a mixture of centralized and multiagent planning. For example, a delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.

The issues involved in multiagent planning can be divided roughly into two sets. The first, covered in Section 11.4.1, involves issues of representing and planning for multiple simultaneous actions; these issues occur in all settings from multieffector to multiagent planning. The second, covered in Section 11.4.2, involves issues of cooperation, coordination, and competition arising in true multiagent settings.

11.4.1 Planning with multiple simultaneous actions

MULTIACTOR
ACTOR

For the time being, we will treat the multieffector, multibody, and multiagent settings in the same way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting. A correct plan is one that, if executed by the actors, achieves the goal. (In the true multiagent setting, of course, the agents may not agree to execute any particular plan, but at least they

⁶ We apologize to residents of the United Kingdom, where the mere act of contemplating a game of tennis guarantees rain.

```

Actors( $A, B$ )
Init( $At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$ 
     $Approaching(Ball, RightBaseline) \wedge Partner(A, B) \wedge Partner(B, A)$ 
     $Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet)))$ 
     $Action(Hit(actor, Ball),$ 
         $PRECOND: Approaching(Ball, loc) \wedge At(actor, loc)$ 
         $EFFECT: Returned(Ball))$ 
     $Action(Go(actor, to),$ 
         $PRECOND: At(actor, loc) \wedge to \neq loc,$ 
         $EFFECT: At(actor, to) \wedge \neg At(actor, loc))$ 

```

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: *LeftBaseline*, *RightBaseline*, *LeftNet*, and *RightNet*. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

will know what plans *would* work if they *did* agree to execute them.) For simplicity, we assume perfect **synchronization**: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

We begin with the transition model; for the deterministic case, this is the function $RESULT(s, a)$. In the single-agent setting, there might be b different choices for the action; b can be quite large, especially for first-order representations with many objects to act on, but action schemas provide a concise representation nonetheless. In the multiactor setting with n actors, the single action a is replaced by a **joint action** $\langle a_1, \dots, a_n \rangle$, where a_i is the action taken by the i th actor. Immediately, we see two problems: first, we have to describe the transition model for b^n different joint actions; second, we have a joint planning problem with a branching factor of b^n .

Having put the actors together into a multiactor system with a huge branching factor, the principal focus of research on multiactor planning has been to *decouple* the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are **loosely coupled**, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI. We have seen it explicitly in the context of CSPs, where “tree like” constraint graphs yielded efficient solution methods (see page 225), as well as in the context of disjoint pattern databases (page 106) and additive heuristics for planning (page 378).

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let’s see how this works for the doubles tennis problem. Let’s suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

A first pass at a multiactor definition might look like Figure 11.10. With this definition, it is easy to see that the following **joint plan** works:

JOINT PLAN

PLAN 1:

$A : [Go(A, RightBaseline), Hit(A, Ball)]$
 $B : [NoOp(B), NoOp(B)] .$

Problems arise, however, when a plan has both agents hitting the ball at the same time. In the real world, this won't work, but the action schema for *Hit* says that the ball will be returned successfully. Technically, the difficulty is that preconditions constrain the *state* in which an action can be executed successfully, but do not constrain other actions that might mess it up. We solve this by augmenting action schemas with one new feature: a **concurrent action list** stating which actions must or must not be executed concurrently. For example, the *Hit* action could be described as follows:

CONCURRENT ACTION LIST

$Action(Hit(a, Ball),$
 $CONCURRENT: b \neq a \Rightarrow \neg Hit(b, Ball)$
 $PRECOND: Approaching(Ball, loc) \wedge At(a, loc)$
 $EFFECT: Returned(Ball)) .$

In other words, the *Hit* action has its stated effect only if no other *Hit* action by another agent occurs at the same time. (In the SATPLAN approach, this would be handled by a partial **action exclusion axiom**.) For some actions, the desired effect is achieved *only* when another action occurs concurrently. For example, two agents are needed to carry a cooler full of beverages to the tennis court:

$Action(Carry(a, cooler, here, there),$
 $CONCURRENT: b \neq a \wedge Carry(b, cooler, here, there)$
 $PRECOND: At(a, here) \wedge At(cooler, here) \wedge Cooler(cooler)$
 $EFFECT: At(a, there) \wedge At(cooler, there) \wedge \neg At(a, here) \wedge \neg At(cooler, here)) .$

With these kinds of action schemas, any of the planning algorithms described in Chapter 10 can be adapted with only minor modifications to generate multiactor plans. To the extent that the coupling among subplans is loose—meaning that concurrency constraints come into play only rarely during plan search—one would expect the various heuristics derived for single-agent planning to also be effective in the multiactor context. We could extend this approach with the refinements of the last two chapters—HTNs, partial observability, conditionals, execution monitoring, and replanning—but that is beyond the scope of this book.

11.4.2 Planning with multiple agents: Cooperation and coordination

Now let us consider the true multiagent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the “*the*” in “*the joint solution*” is misleading. For our doubles team, more than one joint solution exists:

PLAN 2:

$A : [Go(A, LeftNet), NoOp(A)]$
 $B : [Go(B, RightBaseline), Hit(B, Ball)] .$

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if *A* chooses plan 2 and *B* chooses plan 1, then nobody will return the ball. Conversely, if *A* chooses 1 and *B* chooses 2, then they will both try to hit the ball. The agents may realize this, but how can they coordinate to make sure they agree on the plan?

CONVENTION

One option is to adopt a **convention** before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention “stick to your side of the court” would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention “stay on the right side of the road” in most countries; the alternative, “stay on the left side,” works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called **social laws**.

SOCIAL LAWS

In the absence of a convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a tennis player could shout “Mine!” or “Yours!” to indicate a preferred joint plan. We cover mechanisms for communication in more depth in Chapter 22, where we observe that communication does not necessarily involve a verbal exchange. For example, one player can communicate a preferred joint plan to the other simply by executing the first part of it. If agent *A* heads for the net, then agent *B* is obliged to go back to the baseline to hit the ball, because plan 2 is the only joint plan that begins with *A*’s heading for the net. This approach to coordination, sometimes called **plan recognition**, works when a single action (or short sequence of actions) is enough to determine a joint plan unambiguously. Note that communication can work as well with competitive agents as with cooperative ones.

PLAN RECOGNITION

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen’s job is to reproduce, not to do centralized planning—and with very limited computation, communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role according to the local conditions it observes. For example, foragers travel away from the nest, search for a seed, and when they find one, bring it back immediately. Thus, the rate at which foragers return to the nest is an approximation of the availability of food today. When the rate is high, other ants abandon their current role and take on the role of scavenger. The ants appear to have a convention on the importance of roles—foraging is the most important—and ants will easily switch into the more important roles, but not into the less important. There is some learning mechanism: a colony learns to make more successful and prudent actions over the course of its decades-long life, even though individual ants live only about a year.

One final example of cooperative multiagent behavior appears in the flocking behavior of birds. We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a **boid**) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components:

BOID

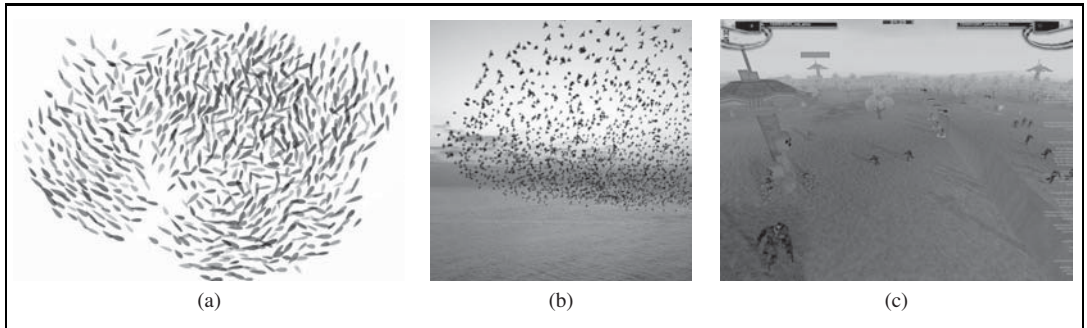


Figure 11.11 (a) A simulated flock of birds, using Reynold's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors

EMERGENT
BEHAVIOR

If all the boids execute this policy, the flock exhibits the **emergent behavior** of flying as a pseudorigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents.

The most difficult multiagent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control. We see this in games such as robotic soccer or the NERO game shown in Figure 11.11(c), in which two teams of software agents compete to capture the control towers. As yet, methods for efficient planning in these kinds of environments—for example, taking advantage of loose coupling—are in their infancy.

11.5 SUMMARY

This chapter has addressed some of the complications of planning and acting in the real world. The main points:

- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these resources as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Actions can generate and consume resources, and it is usually cheap and effective to check partial plans for satisfaction of resource constraints before attempting further refinements.
- Time is one of the most important resources. It can be handled by specialized scheduling algorithms, or scheduling can be integrated with planning.

- **Hierarchical task network** (HTN) planning allows the agent to take advice from the domain designer in the form of **high-level actions** (HLAs) that can be implemented in various ways by lower-level action sequences. The effects of HLAs can be defined with **angelic semantics**, allowing provably correct high-level plans to be derived without consideration of lower-level implementations. HTN methods can create the very large plans required by many real-world applications.
- Standard planning algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption.
- **Contingent plans** allow the agent to sense the world during execution to decide what branch of the plan to follow. In some cases, **sensorless** or **conformant planning** can be used to construct a plan that works without the need for perception. Both conformant and contingent plans can be constructed by search in the space of **belief states**. Efficient representation or computation of belief states is a key problem.
- An **online planning agent** uses execution monitoring and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.
- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate or compete. Joint plans can be constructed, but must be augmented with some form of coordination if two agents are to agree on which joint plan to execute.
- This chapter extends classic planning to cover nondeterministic environments (where outcomes of actions are uncertain), but it is not the last word on planning. Chapter 17 describes techniques for stochastic environments (in which outcomes of actions have probabilities associated with them): Markov decision processes, partially observable Markov decision processes, and game theory. In Chapter 21 we show that reinforcement learning allows an agent to learn how to behave from past successes and failures.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with time constraints was first dealt with by DEVISER (Vere, 1983). The representation of time in plans was addressed by Allen (1984) and by Dean *et al.* (1990) in the FORBIN system. NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1988, 1990) could reason about the allocation of limited resources to various plan steps. O-PLAN (Bell and Tate, 1985), an HTN planner, had a uniform, general representation for constraints on time and resources. In addition to the Hitachi application mentioned in the text, O-PLAN has been applied to software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars.

The two planners SAPA (Do and Kambhampati, 2001) and T4 (Haslum and Geffner, 2001) both used forward state-space search with sophisticated heuristics to handle actions with durations and resources. An alternative is to use very expressive action languages, but guide them by human-written domain-specific heuristics, as is done by ASPEN (Fukunaga *et al.*, 1997), HSTS (Jonsson *et al.*, 2000), and IxTeT (Ghallab and Laruelle, 1994).

A number of hybrid planning-and-scheduling systems have been deployed: ISIS (Fox *et al.*, 1982; Fox, 1990) has been used for job shop scheduling at Westinghouse, GARI (Desclotte and Latombe, 1985) planned the machining and construction of mechanical parts, FORBIN was used for factory control, and NONLIN+ was used for naval logistics planning. We chose to present planning and scheduling as two separate problems; (Cushing *et al.*, 2007) show that this can lead to incompleteness on certain problems. There is a long history of scheduling in aerospace. T-SCHED (Drabble, 1990) was used to schedule mission-command sequences for the UOSAT-II satellite. OPTIMUM-AIV (Aarup *et al.*, 1994) and PLAN-ERS1 (Fuchs *et al.*, 1990), both based on O-PLAN, were used for spacecraft assembly and observation planning, respectively, at the European Space Agency. SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble Space Telescope, while the Space Shuttle Ground Processing Scheduling System (Deale *et al.*, 1994) does job-shop scheduling of up to 16,000 worker-shifts. Remote Agent (Muscettola *et al.*, 1998) became the first autonomous planner-scheduler to control a spacecraft when it flew onboard the Deep Space One probe in 1999. Space applications have driven the development of algorithms for resource allocations; see Laborie (2003) and Muscettola (2002). The literature on scheduling is presented in a classic survey article (Lawler *et al.*, 1993), a recent book (Pinedo, 2008), and an edited handbook (Blazewicz *et al.*, 2007).

MACROPS

ABSTRACTION
HIERARCHY

The facility in the STRIPS program for learning **macrops**—“macro-operators” consisting of a sequence of primitive steps—could be considered the first mechanism for hierarchical planning (Fikes *et al.*, 1972). Hierarchy was also used in the LAWALY system (Siklossy and Dreussi, 1973). The ABSTRIPS system (Sacerdoti, 1974) introduced the idea of an **abstraction hierarchy**, whereby planning at higher levels was permitted to ignore lower-level preconditions of actions in order to derive the general structure of a working plan. Austin Tate’s Ph.D. thesis (1975b) and work by Earl Sacerdoti (1977) developed the basic ideas of HTN planning in its modern form. Many practical planners, including O-PLAN and SIPE, are HTN planners. Yang (1990) discusses properties of actions that make HTN planning efficient. Erol, Hendler, and Nau (1994, 1996) present a complete hierarchical decomposition planner as well as a range of complexity results for pure HTN planners. Our presentation of HLAs and angelic semantics is due to Marthi *et al.* (2007, 2008). Kambhampati *et al.* (1998) have proposed an approach in which decompositions are just another form of plan refinement, similar to the refinements for non-hierarchical partial-order planning.

CASE-BASED
PLANNING

Beginning with the work on macro-operators in STRIPS, one of the goals of hierarchical planning has been the reuse of previous planning experience in the form of generalized plans. The technique of **explanation-based learning**, described in depth in Chapter 19, has been applied in several systems as a means of generalizing previously computed plans, including SOAR (Laird *et al.*, 1986) and PRODIGY (Carbonell *et al.*, 1989). An alternative approach is to store previously computed plans in their original form and then reuse them to solve new, similar problems by analogy to the original problem. This is the approach taken by the field called **case-based planning** (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) argues that case-based planning should be analyzed as a form of refinement planning and provides a formal foundation for case-based partial-order planning.

Early planners lacked conditionals and loops, but some could use coercion to form conformant plans. Sacerdoti's NOAH solved the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state, using coercion. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and described a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions, *regardless* of the initial position.

Goldman and Boddy (1996) introduced the term **conformant planning**, noting that sensorless plans are often effective even if the agent has sensors. The first moderately efficient conformant planner was Smith and Weld's (1998) Conformant Graphplan or CGP. Ferraris and Giunchiglia (2000) and Rintanen (1999) independently developed SATPLAN-based conformant planners. Bonet and Geffner (2000) describe a conformant planner based on heuristic search in the space of belief states, drawing on ideas first developed in the 1960s for partially observable Markov decision processes, or POMDPs (see Chapter 17).

Currently, there are three main approaches to conformant planning. The first two use heuristic search in belief-state space: HSCP (Bertoli *et al.*, 2001a) uses binary decision diagrams (BDDs) to represent belief states, whereas Hoffmann and Brafman (2006) adopt the lazy approach of computing precondition and goal tests on demand using a SAT solver. The third approach, championed primarily by Jussi Rintanen (2007), formulates the entire sensorless planning problem as a quantified Boolean formula (QBF) and solves it using a general-purpose QBF solver. Current conformant planners are five orders of magnitude faster than CGP. The winner of the 2006 conformant-planning track at the International Planning Competition was T_0 (Palacios and Geffner, 2007), which uses heuristic search in belief-state space while keeping the belief-state representation simple by defining derived literals that cover conditional effects. Bryce and Kambhampati (2007) discuss how a planning graph can be generalized to generate good heuristics for conformant and contingent planning.

There has been some confusion in the literature between the terms "conditional" and "contingent" planning. Following Majercik and Littman (2003), we use "conditional" to mean a plan (or action) that has different effects depending on the actual state of the world, and "contingent" to mean a plan in which the agent can choose different actions depending on the results of sensing. The problem of contingent planning received more attention after the publication of Drew McDermott's (1978a) influential article, *Planning and Acting*.

The contingent-planning approach described in the chapter is based on Hoffmann and Brafman (2005), and was influenced by the efficient search algorithms for cyclic AND-OR graphs developed by Jimenez and Torras (2000) and Hansen and Zilberstein (2001). Bertoli *et al.* (2001b) describe MBP (Model-Based Planner), which uses binary decision diagrams to do conformant and contingent planning.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for uncertain domains. Fast-forward heuristic search through state space led to forward search in belief space (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 2003) and to planning with quantified Boolean logic (Rintanen, 2007); partial order planning led to UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992); GRAPHPLAN led to Sensory Graphplan or SGP (Weld *et al.*, 1998).

The first online planner with execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. The NASL planner (McDermott, 1978a) treated a planning problem simply as a specification for carrying out a complex action, so that execution and planning were completely unified. SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988, 1990) was the first planner to deal systematically with the problem of replanning. It has been used in demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier, job-shop scheduling for an Australian beer factory, and planning the construction of multistory buildings (Kartam and Levitt, 1990).

REACTIVE PLANNING

In the mid-1980s, pessimism about the slow run times of planning systems led to the proposal of reflex agents called **reactive planning** systems (Brooks, 1986; Agre and Chapman, 1987). PENG (Agre and Chapman, 1987) could play a (fully observable) video game by using Boolean circuits combined with a “visual” representation of current goals and the agent’s internal state. “Universal plans” (Schoppers, 1987, 1989) were developed as a lookup-table method for reactive planning, but turned out to be a rediscovery of the idea of **policies** that had long been used in Markov decision processes (see Chapter 17). A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Koenig (2001) surveys online planning techniques, under the name *Agent-Centered Search*.

POLICY

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) formalizes multiagent planning in first-order logic, while Pednault (1986) gives a STRIPS-style description. The notion of joint intention, which is essential if agents are to execute a joint plan, comes from work on communicative acts (Cohen and Levesque, 1990; Cohen *et al.*, 1990). Boutilier and Brafman (2001) show how to adapt partial-order planning to a multiactor setting. Brafman and Domshlak (2008) devise a multiactor planning algorithm whose complexity grows only linearly with the number of actors, provided that the degree of coupling (measured partly by the **tree width** of the graph of interactions among agents) is bounded. Petrik and Zilberstein (2009) show that an approach based on bilinear programming outperforms the cover-set approach we outlined in the chapter.

We have barely skimmed the surface of work on negotiation in multiagent planning. Durfee and Lesser (1989) discuss how tasks can be shared out among agents by negotiation. Kraus *et al.* (1991) describe a system for playing Diplomacy, a board game requiring negotiation, coalition formation, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. In a later article, Stone (2003) analyzes two competitive multiagent environments—RoboCup, a robotic soccer competition, and TAC, the auction-based Trading Agents Competition—and finds that the computational intractability of our current theoretically well-founded approaches has led to many multiagent systems being designed by *ad hoc* methods.

In his highly influential *Society of Mind* theory, Marvin Minsky (1986, 2007) proposes that human minds are constructed from an ensemble of agents. Livnat and Pippenger (2006) prove that, for the problem of optimal path-finding, and given a limitation on the total amount of computing resources, the best architecture for an agent is an ensemble of subagents, each of which tries to optimize its own objective, and all of which are in conflict with one another.

The boid model on page 429 is due to Reynolds (1987), who won an Academy Award for its application to swarms of penguins in *Batman Returns*. The NERO game and the methods for learning strategies are described by Bryant and Miikkulainen (2007).

Recent book on multiagent systems include those by Weiss (2000a), Young (2004), Vlassis (2008), and Shoham and Leyton-Brown (2009). There is an annual conference on autonomous agents and multiagent systems (AAMAS).

EXERCISES

11.1 The goals we have considered so far all ask the planner to make the world satisfy the goal at just one time step. Not all goals can be expressed this way: you do not achieve the goal of suspending a chandelier above the ground by throwing it in the air. More seriously, you wouldn't want your spacecraft life-support system to supply oxygen one day but not the next. A *maintenance goal* is achieved when the agent's plan causes a condition to hold continuously from a given state onward. Describe how to extend the formalism of this chapter to support maintenance goals.

11.2 You have a number of trucks with which to deliver a set of packages. Each package starts at some location on a grid map, and has a destination somewhere else. Each truck is directly controlled by moving forward and turning. Construct a hierarchy of high-level actions for this problem. What knowledge about the solution does your hierarchy encode?

11.3 Suppose that a high-level action has exactly one implementation as a sequence of primitive actions. Give an algorithm for computing its preconditions and effects, given the complete refinement hierarchy and schemas for the primitive actions.

11.4 Suppose that the optimistic reachable set of a high-level plan is a superset of the goal set; can anything be concluded about whether the plan achieves the goal? What if the pessimistic reachable set doesn't intersect the goal set? Explain.

11.5 Write an algorithm that takes an initial state (specified by a set of propositional literals) and a sequence of HLAs (each defined by preconditions and angelic specifications of optimistic and pessimistic reachable sets) and computes optimistic and pessimistic descriptions of the reachable set of the sequence.

11.6 In Figure 11.2 we showed how to describe actions in a scheduling problem by using separate fields for DURATION, USE, and CONSUME. Now suppose we wanted to combine scheduling with nondeterministic planning, which requires nondeterministic and conditional effects. Consider each of the three fields and explain if they should remain separate fields, or if they should become effects of the action. Give an example for each of the three.

11.7 Some of the operations in standard programming languages can be modeled as actions that change the state of the world. For example, the assignment operation changes the contents of a memory location, and the print operation changes the state of the output stream. A program consisting of these operations can also be considered as a plan, whose goal is given

by the specification of the program. Therefore, planning algorithms can be used to construct programs that achieve a given specification.

- a. Write an action schema for the assignment operator (assigning the value of one variable to another). Remember that the original value will be overwritten!
- b. Show how object creation can be used by a planner to produce a plan for exchanging the values of two variables by using a temporary variable.

11.8 Suppose the *Flip* action always changes the truth value of variable *L*. Show how to define its effects by using an action schema with conditional effects. Show that, despite the use of conditional effects, a 1-CNF belief state representation remains in 1-CNF after a *Flip*.

11.9 In the blocks world we were forced to introduce two action schemas, *Move* and *MoveToTable*, in order to maintain the *Clear* predicate properly. Show how conditional effects can be used to represent both of these cases with a single action.

11.10 Conditional effects were illustrated for the *Suck* action in the vacuum world—which square becomes clean depends on which square the robot is in. Can you think of a new set of propositional variables to define states of the vacuum world, such that *Suck* has an *unconditional* description? Write out the descriptions of *Suck*, *Left*, and *Right*, using your propositions, and demonstrate that they suffice to describe all possible states of the world.

11.11 Find a suitably dirty carpet, free of obstacles, and vacuum it. Draw the path taken by the vacuum cleaner as accurately as you can. Explain it, with reference to the forms of planning discussed in this chapter.

11.12 To the medication problem in the previous exercise, add a *Test* action that has the conditional effect *CultureGrowth* when *Disease* is true and in any case has the perceptual effect *Known(CultureGrowth)*. Diagram a conditional plan that solves the problem and minimizes the use of the *Medicate* action.

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.

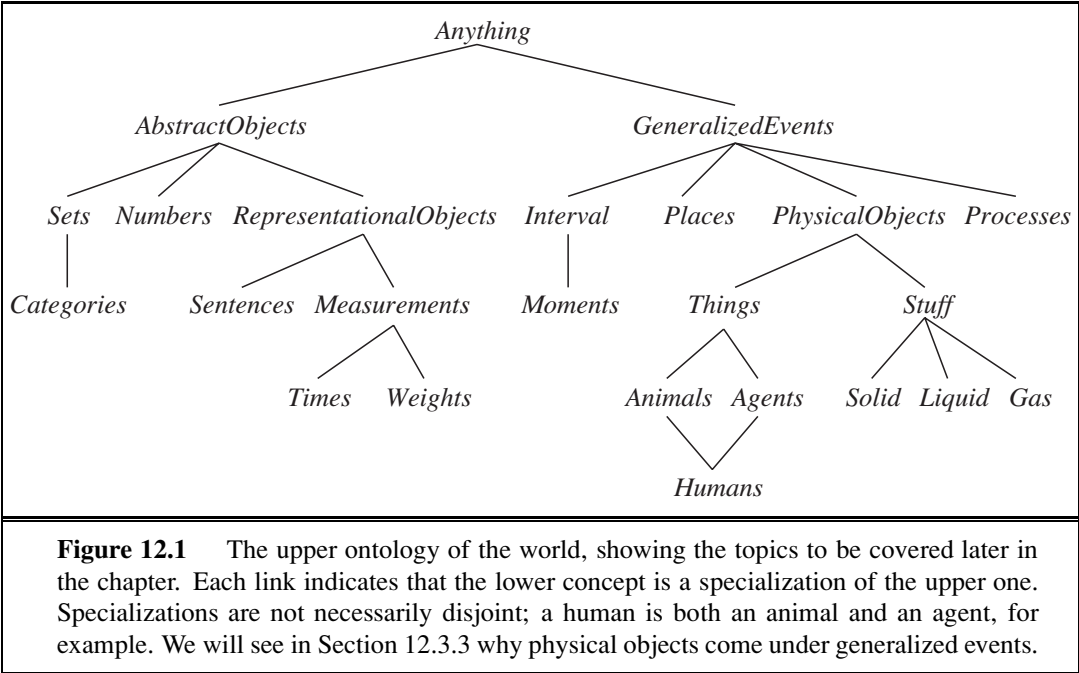
The previous chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent's knowledge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

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



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**¹ 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,² 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.
 $(x \in Basketballs) \Rightarrow Spherical(x)$

¹ 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.

² 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 *Apple*₁, *Apple*₂, and *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.

NATURAL KINDS

Some categories have strict definitions: an object is a triangle if and only if it is a polygon with three sides. On the other hand, most categories in the real world have no clear-cut definition; these are called **natural kind** categories. For example, tomatoes tend to be a dull scarlet; roughly spherical; with an indentation at the top where the stem was; about two to four inches in diameter; with a thin but tough skin; and with flesh, seeds, and juice inside. There is, however, variation: some tomatoes are yellow or orange, unripe tomatoes are green, some are smaller or larger than average, and cherry tomatoes are uniformly small. Rather than having a complete definition of tomatoes, we have a set of features that serves to identify objects that are clearly typical tomatoes, but might not be able to decide for other objects. (Could there be a tomato that is fuzzy like a peach?)

This poses a problem for a logical agent. The agent cannot be sure that an object it has perceived is a tomato, and even if it were sure, it could not be certain which of the properties of typical tomatoes this one has. This problem is an inevitable consequence of operating in partially observable environments.

One useful approach is to separate what is true of all instances of a category from what is true only of typical instances. So in addition to the category *Tomatoes*, we will also have the category *Typical(Tomatoes)*. Here, the *Typical* function maps a category to the subclass that contains only typical instances:

$$\text{Typical}(c) \subseteq c.$$

Most knowledge about natural kinds will actually be about their typical instances:


$$x \in \text{Typical}(\text{Tomatoes}) \Rightarrow \text{Red}(x) \wedge \text{Round}(x).$$

Thus, we can write down useful facts about categories without exact definitions. The difficulty of providing exact definitions for most natural categories was explained in depth by Wittgenstein (1953). He used the example of *games* to show that members of a category shared “family resemblances” rather than necessary and sufficient characteristics: what strict definition encompasses chess, tag, solitaire, and dodgeball?

The utility of the notion of strict definition was also challenged by Quine (1953). He pointed out that even the definition of “bachelor” as an unmarried adult male is suspect; one might, for example, question a statement such as “the Pope is a bachelor.” While not strictly *false*, this usage is certainly *infelicitous* because it induces unintended inferences on the part of the listener. The tension could perhaps be resolved by distinguishing between logical definitions suitable for internal knowledge representation and the more nuanced criteria for felicitous linguistic usage. The latter may be achieved by “filtering” the assertions derived from the former. It is also possible that failures of linguistic usage serve as feedback for modifying internal definitions, so that filtering becomes unnecessary.

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, 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

UNITS FUNCTION

$$\text{Length}(L_1) = \text{Inches}(1.5) = \text{Centimeters}(3.81) .$$

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

$$\text{Centimeters}(2.54 \times d) = \text{Inches}(d) .$$

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:

$$\text{Diameter}(\text{Basketball}_{12}) = \text{Inches}(9.5) .$$

$$\text{ListPrice}(\text{Basketball}_{12}) = \$ (19) .$$

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

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 $\text{Inches}(0)$ and $\text{Centimeters}(0)$ refer to the same zero length, they are not identical to other zero measures, such as $\text{Seconds}(0)$.

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:

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

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

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*₃. We also define the category *Butter*. Informally, its elements will be all those things of which one might say “It’s butter,” including *Butter*₃. 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.³

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.⁴ 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

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

⁴ 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:

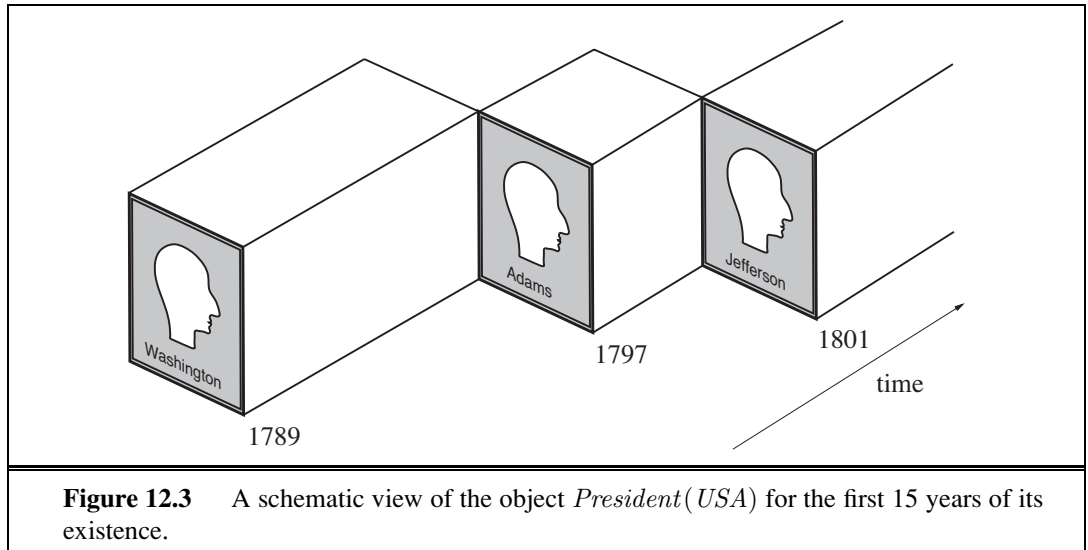
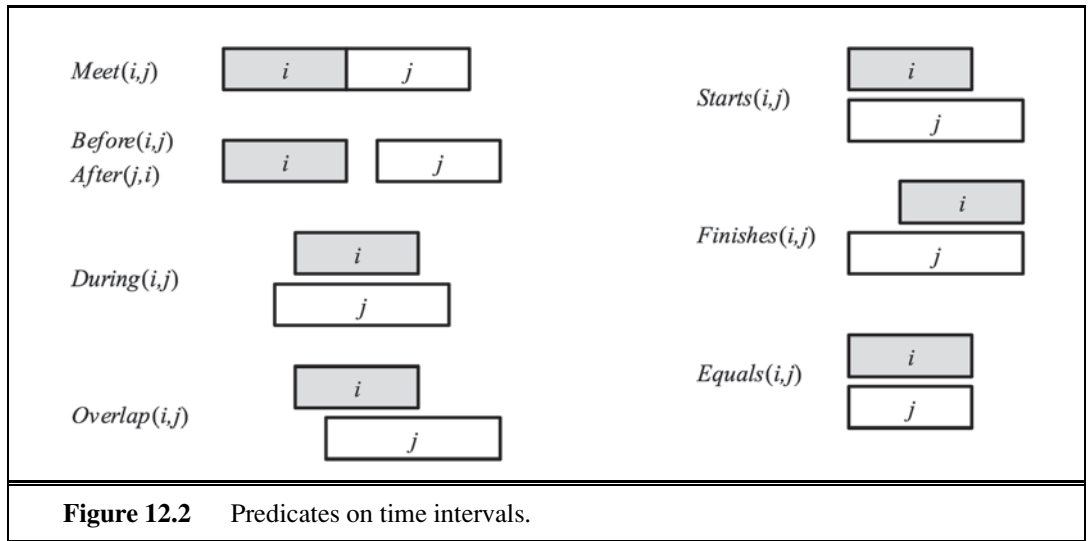
$$\begin{aligned} \text{Time}(\text{Begin}(\text{AD2001})) & = \text{Date}(0, 0, 0, 1, \text{Jan}, 2001) \\ \text{Date}(0, 20, 21, 24, 1, 1995) & = \text{Seconds}(3000000000) . \end{aligned}$$

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:

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



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

$$T(Equals(President(USA), GeorgeWashington), AD1790) .$$

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.

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:

$$Knows(Lois, CanFly(Superman)) .$$

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:

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

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 $\text{Acc}(\mathbf{K}_A, w_0, w_1)$. 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.

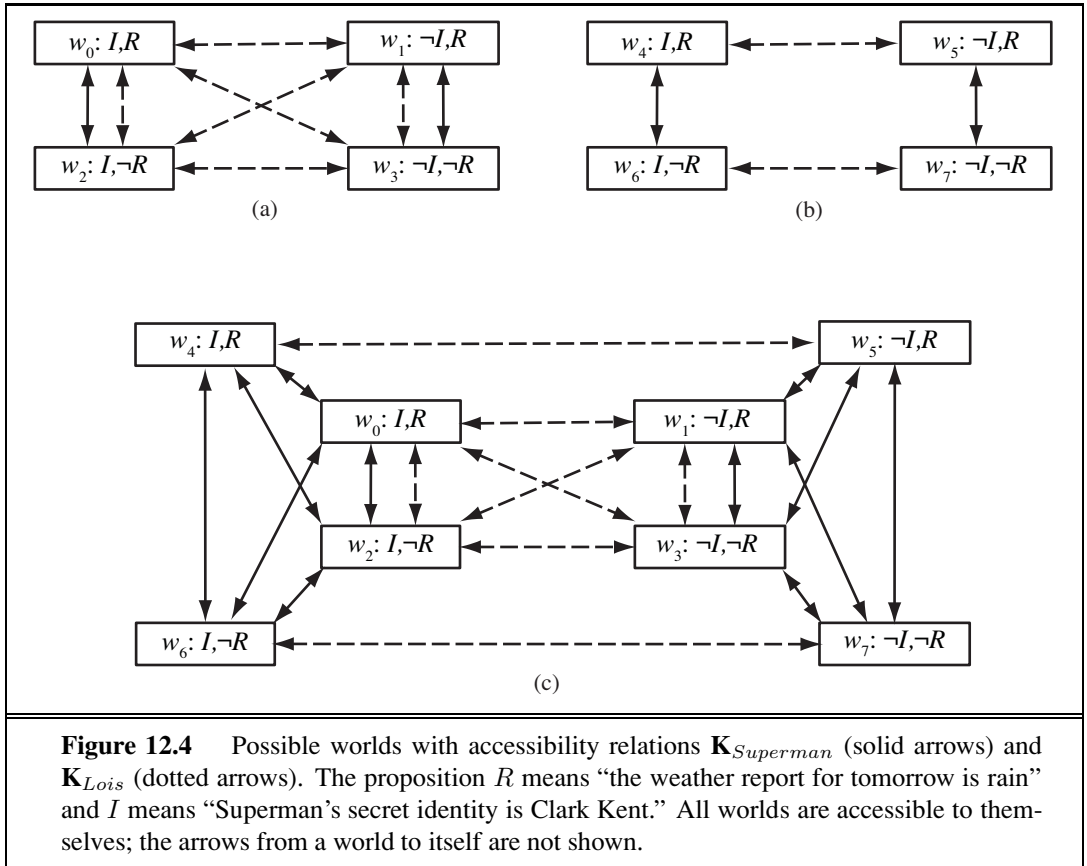
In general, a knowledge atom $\mathbf{K}_A P$ is true in world w if and only if P is true in every world accessible from w . The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason 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:

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

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 knows R . From what Superman knows, he might 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:

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

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.

LOGICAL
OMNISCIENCE

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.⁵

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

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

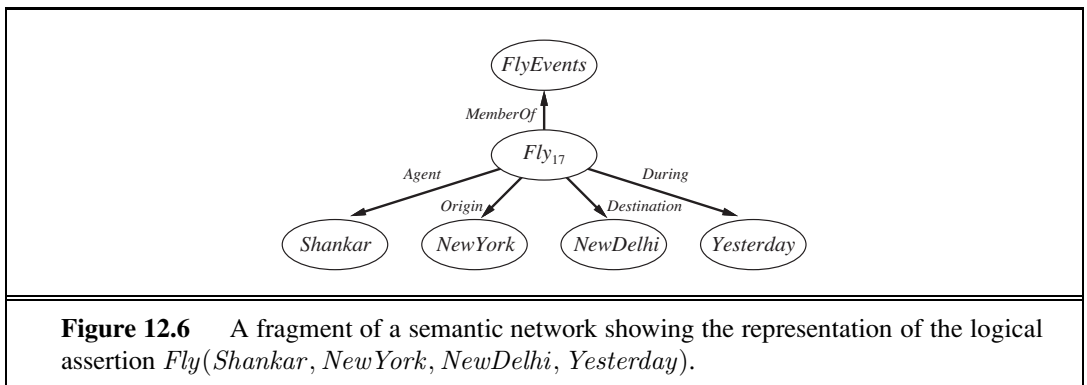
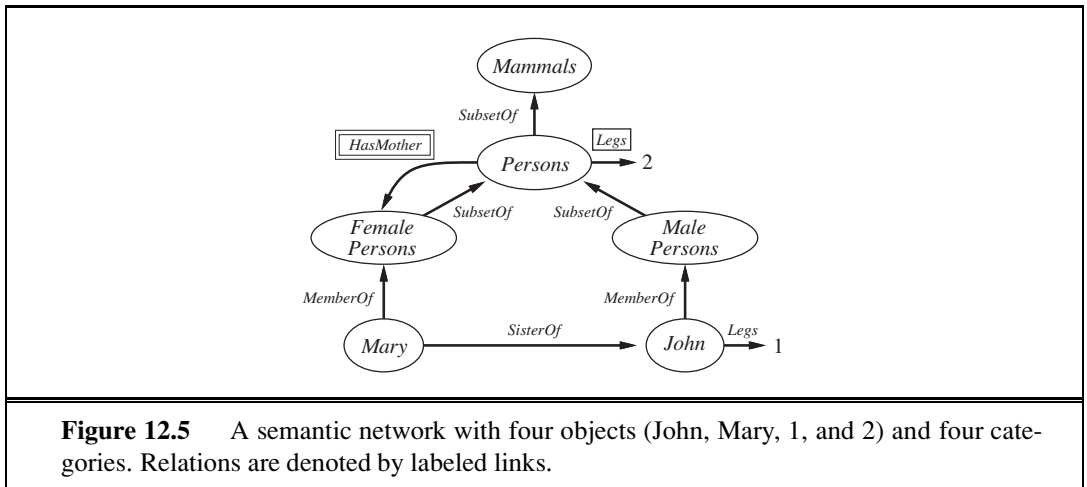
We might also want to assert that persons have two legs—that is,

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

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

⁵ 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.

DEFAULT VALUE

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.

OVERRIDING

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

DESCRIPTION LOGIC

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.

SUBSUMPTION

CLASSIFICATION

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} \textit{Concept} &\rightarrow \mathbf{Thing} \mid \textit{ConceptName} \\ &\mid \mathbf{And}(\textit{Concept}, \dots) \\ &\mid \mathbf{All}(\textit{RoleName}, \textit{Concept}) \\ &\mid \mathbf{AtLeast}(\textit{Integer}, \textit{RoleName}) \\ &\mid \mathbf{AtMost}(\textit{Integer}, \textit{RoleName}) \\ &\mid \mathbf{Fills}(\textit{RoleName}, \textit{IndividualName}, \dots) \\ &\mid \mathbf{SameAs}(\textit{Path}, \textit{Path}) \\ &\mid \mathbf{OneOf}(\textit{IndividualName}, \dots) \\ \textit{Path} &\rightarrow [\textit{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 12.7.⁶ For example, to say that bachelors are unmarried adult males we would write

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

The equivalent in first-order logic would be

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

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

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

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.⁷

⁶ 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.

⁷ 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 7.⁸ 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 α 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

⁸ 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.⁹ 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

⁹ 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**.¹⁰ 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

¹⁰ 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 $\text{RETRACT}(\text{KB}, P_i)$ 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.

12.7 THE INTERNET SHOPPING WORLD

In this final section we put together all we have learned to encode knowledge for a shopping research agent that helps a buyer find product offers on the Internet. The shopping agent is given a product description by the buyer and has the task of producing a list of Web pages that offer such a product for sale, and ranking which offers are best. In some cases the buyer's product description will be precise, as in *Canon Rebel XTi digital camera*, and the task is then to find the store(s) with the best offer. In other cases the description will be only partially specified, as in *digital camera for under \$300*, and the agent will have to compare different products.

The shopping agent's environment is the entire World Wide Web in its full complexity—not a toy simulated environment. The agent's percepts are Web pages, but whereas a human

Example Online Store

Select from our fine line of products:

- Computers
- Cameras
- Books
- Videos
- Music

```
<h1>Example Online Store</h1>
<i>Select</i> from our fine line of products:
<ul>
<li> <a href="http://example.com/compu">Computers</a>
<li> <a href="http://example.com/camer">Cameras</a>
<li> <a href="http://example.com/books">Books</a>
<li> <a href="http://example.com/video">Videos</a>
<li> <a href="http://example.com/music">Music</a>
</ul>
```

Figure 12.8 A Web page from a generic online store in the form perceived by the human user of a browser (top), and the corresponding HTML string as perceived by the browser or the shopping agent (bottom). In HTML, characters between `<` and `>` are markup directives that specify how the page is displayed. For example, the string `<i>Select</i>` means to switch to italic font, display the word *Select*, and then end the use of italic font. A page identifier such as `http://example.com/books` is called a **uniform resource locator (URL)**. The markup `Books` means to create a hypertext link to *url* with the **anchor text** *Books*.

Web user would see pages displayed as an array of pixels on a screen, the shopping agent will perceive a page as a character string consisting of ordinary words interspersed with formatting commands in the HTML markup language. Figure 12.8 shows a Web page and a corresponding HTML character string. The perception problem for the shopping agent involves extracting useful information from percepts of this kind.

Clearly, perception on Web pages is easier than, say, perception while driving a taxi in Cairo. Nonetheless, there are complications to the Internet perception task. The Web page in Figure 12.8 is simple compared to real shopping sites, which may include CSS, cookies, Java, Javascript, Flash, robot exclusion protocols, malformed HTML, sound files, movies, and text that appears only as part of a JPEG image. An agent that can deal with *all* of the Internet is almost as complex as a robot that can move in the real world. We concentrate on a simple agent that ignores most of these complications.

The agent's first task is to collect product offers that are relevant to a query. If the query is "laptops," then a Web page with a review of the latest high-end laptop would be relevant, but if it doesn't provide a way to buy, it isn't an offer. For now, we can say a page is an offer if it contains the words "buy" or "price" or "add to cart" within an HTML link or form on the

page. For example, if the page contains a string of the form “<a...add to cart...” then it is an offer. This could be represented in first-order logic, but it is more straightforward to encode it into program code. We show how to do more sophisticated information extraction in Section 22.4.

12.7.1 Following links

The strategy is to start at the home page of an online store and consider all pages that can be reached by following relevant links.¹¹ The agent will have knowledge of a number of stores, for example:

$$\begin{aligned} Amazon &\in OnlineStores \wedge Homepage(Amazon, "amazon.com") . \\ Ebay &\in OnlineStores \wedge Homepage(Ebay, "ebay.com") . \\ ExampleStore &\in OnlineStores \wedge Homepage(ExampleStore, "example.com") . \end{aligned}$$

These stores classify their goods into product categories, and provide links to the major categories from their home page. Minor categories can be reached through a chain of relevant links, and eventually we will reach offers. In other words, a page is relevant to the query if it can be reached by a chain of zero or more relevant category links from a store’s home page, and then from one more link to the product offer. We can define relevance:

$$\begin{aligned} Relevant(page, query) &\Leftrightarrow \\ &\exists store, home \ store \in OnlineStores \wedge Homepage(store, home) \\ &\wedge \exists url, url_2 \ RelevantChain(home, url_2, query) \wedge Link(url_2, url) \\ &\wedge page = Contents(url) . \end{aligned}$$

Here the predicate $Link(from, to)$ means that there is a hyperlink from the *from* URL to the *to* URL. To define what counts as a *RelevantChain*, we need to follow not just any old hyperlinks, but only those links whose associated anchor text indicates that the link is relevant to the product query. For this, we use $LinkText(from, to, text)$ to mean that there is a link between *from* and *to* with *text* as the anchor text. A chain of links between two URLs, *start* and *end*, is relevant to a description *d* if the anchor text of each link is a relevant category name for *d*. The existence of the chain itself is determined by a recursive definition, with the empty chain ($start = end$) as the base case:

$$\begin{aligned} RelevantChain(start, end, query) &\Leftrightarrow (start = end) \\ &\vee (\exists u, text \ LinkText(start, u, text) \wedge RelevantCategoryName(query, text) \\ &\wedge RelevantChain(u, end, query)) . \end{aligned}$$

Now we must define what it means for *text* to be a *RelevantCategoryName* for *query*. First, we need to relate strings to the categories they name. This is done using the predicate $Name(s, c)$, which says that string *s* is a name for category *c*—for example, we might assert that $Name("laptops", LaptopComputers)$. Some more examples of the *Name* predicate appear in Figure 12.9(b). Next, we define relevance. Suppose that *query* is “laptops.” Then $RelevantCategoryName(query, text)$ is true when one of the following holds:

- The *text* and *query* name the same category—e.g., “notebooks” and “laptops.”

¹¹ An alternative to the link-following strategy is to use an Internet search engine; the technology behind Internet search, information retrieval, will be covered in Section 22.3.

$Books \subset Products$	$Name("books", Books)$
$MusicRecordings \subset Products$	$Name("music", MusicRecordings)$
$MusicCDs \subset MusicRecordings$	$Name("CDs", MusicCDs)$
$Electronics \subset Products$	$Name("electronics", Electronics)$
$DigitalCameras \subset Electronics$	$Name("digital cameras", DigitalCameras)$
$StereoEquipment \subset Electronics$	$Name("stereos", StereoEquipment)$
$Computers \subset Electronics$	$Name("computers", Computers)$
$DesktopComputers \subset Computers$	$Name("desktops", DesktopComputers)$
$LaptopComputers \subset Computers$	$Name("laptops", LaptopComputers)$
\dots	$Name("notebooks", LaptopComputers)$
(a)	(b)

Figure 12.9 (a) Taxonomy of product categories. (b) Names for those categories.

- The *text* names a supercategory such as “computers.”
- The *text* names a subcategory such as “ultralight notebooks.”

The logical definition of *RelevantCategoryName* is as follows:

$$\begin{aligned}
 &RelevantCategoryName(query, text) \Leftrightarrow \\
 &\exists c_1, c_2 \quad Name(query, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1) .
 \end{aligned} \tag{12.1}$$

Otherwise, the anchor text is irrelevant because it names a category outside this line, such as “clothes” or “lawn & garden.”

To follow relevant links, then, it is essential to have a rich hierarchy of product categories. The top part of this hierarchy might look like Figure 12.9(a). It will not be feasible to list *all* possible shopping categories, because a buyer could always come up with some new desire and manufacturers will always come out with new products to satisfy them (electric kneecap warmers?). Nonetheless, an ontology of about a thousand categories will serve as a very useful tool for most buyers.

In addition to the product hierarchy itself, we also need to have a rich vocabulary of names for categories. Life would be much easier if there were a one-to-one correspondence between categories and the character strings that name them. We have already seen the problem of **synonymy**—two names for the same category, such as “laptop computers” and “laptops.” There is also the problem of **ambiguity**—one name for two or more different categories. For example, if we add the sentence

$$Name("CDs", CertificatesOfDeposit)$$

to the knowledge base in Figure 12.9(b), then “CDs” will name two different categories.

Synonymy and ambiguity can cause a significant increase in the number of paths that the agent has to follow, and can sometimes make it difficult to determine whether a given page is indeed relevant. A much more serious problem is the very broad range of descriptions that a user can type and category names that a store can use. For example, the link might say “laptop” when the knowledge base has only “laptops” or the user might ask for “a computer

I can fit on the tray table of an economy-class airline seat.” It is impossible to enumerate in advance all the ways a category can be named, so the agent will have to be able to do additional reasoning in some cases to determine if the *Name* relation holds. In the worst case, this requires full natural language understanding, a topic that we will defer to Chapter 22. In practice, a few simple rules—such as allowing “laptop” to match a category named “laptops”—go a long way. Exercise 12.10 asks you to develop a set of such rules after doing some research into online stores.

Given the logical definitions from the preceding paragraphs and suitable knowledge bases of product categories and naming conventions, are we ready to apply an inference algorithm to obtain a set of relevant offers for our query? Not quite! The missing element is the *Contents(url)* function, which refers to the HTML page at a given URL. The agent doesn’t have the page contents of every URL in its knowledge base; nor does it have explicit rules for deducing what those contents might be. Instead, we can arrange for the right HTTP procedure to be executed whenever a subgoal involves the *Contents* function. In this way, it appears to the inference engine as if the entire Web is inside the knowledge base. This is an example of a general technique called **procedural attachment**, whereby particular predicates and functions can be handled by special-purpose methods.

PROCEDURAL
ATTACHMENT

12.7.2 Comparing offers

Let us assume that the reasoning processes of the preceding section have produced a set of offer pages for our “laptops” query. To compare those offers, the agent must extract the relevant information—price, speed, disk size, weight, and so on—from the offer pages. This can be a difficult task with real Web pages, for all the reasons mentioned previously. A common way of dealing with this problem is to use programs called **wrappers** to extract information from a page. The technology of information extraction is discussed in Section 22.4. For now we assume that wrappers exist, and when given a page and a knowledge base, they add assertions to the knowledge base. Typically, a hierarchy of wrappers would be applied to a page: a very general one to extract dates and prices, a more specific one to extract attributes for computer-related products, and if necessary a site-specific one that knows the format of a particular store. Given a page on the example.com site with the text

WRAPPER

IBM ThinkBook 970. Our price: \$399.00

followed by various technical specifications, we would like a wrapper to extract information such as the following:

$$\begin{aligned} \exists c, offer \quad & c \in LaptopComputers \wedge offer \in ProductOffers \wedge \\ & Manufacturer(c, IBM) \wedge Model(c, ThinkBook970) \wedge \\ & ScreenSize(c, Inches(14)) \wedge ScreenType(c, ColorLCD) \wedge \\ & MemorySize(c, Gigabytes(2)) \wedge CPUSpeed(c, GHz(1.2)) \wedge \\ & OfferedProduct(offer, c) \wedge Store(offer, GenStore) \wedge \\ & URL(offer, \text{“example.com/computers/34356.html”}) \wedge \\ & Price(offer, \$(399)) \wedge Date(offer, Today) . \end{aligned}$$

This example illustrates several issues that arise when we take seriously the task of knowledge engineering for commercial transactions. For example, notice that the price is an attribute of

the *offer*, not the product itself. This is important because the offer at a given store may change from day to day even for the same individual laptop; for some categories—such as houses and paintings—the same individual object may even be offered simultaneously by different intermediaries at different prices. There are still more complications that we have not handled, such as the possibility that the price depends on the method of payment and on the buyer’s qualifications for certain discounts. The final task is to compare the offers that have been extracted. For example, consider these three offers:

A : 1.4 GHz CPU, 2GB RAM, 250 GB disk, \$299 .

B : 1.2 GHz CPU, 4GB RAM, 350 GB disk, \$500 .

C : 1.2 GHz CPU, 2GB RAM, 250 GB disk, \$399 .

C is **dominated** by *A*; that is, *A* is cheaper and faster, and they are otherwise the same. In general, *X* dominates *Y* if *X* has a better value on at least one attribute, and is not worse on any attribute. But neither *A* nor *B* dominates the other. To decide which is better we need to know how the buyer weighs CPU speed and price against memory and disk space. The general topic of preferences among multiple attributes is addressed in Section 16.4; for now, our shopping agent will simply return a list of all undominated offers that meet the buyer’s description. In this example, both *A* and *B* are undominated. Notice that this outcome relies on the assumption that everyone prefers cheaper prices, faster processors, and more storage. Some attributes, such as screen size on a notebook, depend on the user’s particular preference (portability versus visibility); for these, the shopping agent will just have to ask the user.

The shopping agent we have described here is a simple one; many refinements are possible. Still, it has enough capability that with the right domain-specific knowledge it can actually be of use to a shopper. Because of its declarative construction, it extends easily to more complex applications. The main point of this section is to show that some knowledge representation—in particular, the product hierarchy—is necessary for such an agent, and that once we have some knowledge in this form, the rest follows naturally.

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.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Briggs (1985) claims that formal knowledge representation research began with classical Indian theorizing about the grammar of Shastric Sanskrit, which dates back to the first millennium B.C. In the West, the use of definitions of terms in ancient Greek mathematics can be regarded as the earliest instance: Aristotle's *Metaphysics* (literally, what comes after the book on physics) is a near-synonym for *Ontology*. Indeed, the development of technical terminology in any field can be regarded as a form of knowledge representation.

Early discussions of representation in AI tended to focus on “*problem* representation” rather than “*knowledge* representation.” (See, for example, Amarel's (1968) discussion of the Missionaries and Cannibals problem.) In the 1970s, AI emphasized the development of “expert systems” (also called “knowledge-based systems”) that could, if given the appropriate domain knowledge, match or exceed the performance of human experts on narrowly defined tasks. For example, the first expert system, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpreted the output of a mass spectrometer (a type of instrument used to analyze the structure of organic chemical compounds) as accurately as expert chemists. Although the success of DENDRAL was instrumental in convincing the AI research community of the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry. Over time, researchers became interested in standardized knowledge representation formalisms and ontologies that could streamline the process of creating new expert systems. In so doing, they ventured into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one's theories to “work” has led to more rapid and deeper progress than was the case

when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle (384–322 B.C.) strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called *Categories* in which he attempted to construct what we would now call an upper ontology. He also introduced the notions of **genus** and **species** for lower-level classification. Our present system of biological classification, including the use of “binomial nomenclature” (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linné (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

Interest in larger-scale ontologies is increasing, as documented by the *Handbook on Ontologies* (Staab, 2004). The OPENCYC project (Lenat and Guha, 1990; Matuszek *et al.*, 2006) has released a 150,000-concept ontology, with an upper ontology similar to the one in Figure 12.1 as well as specific concepts like “OLED Display” and “iPhone,” which is a type of “cellular phone,” which in turn is a type of “consumer electronics,” “phone,” “wireless communication device,” and other concepts. The DBPEDIA project extracts structured data from Wikipedia; specifically from Infoboxes: the boxes of attribute/value pairs that accompany many Wikipedia articles (Wu and Weld, 2008; Bizer *et al.*, 2007). As of mid-2009, DBPEDIA contains 2.6 million concepts, with about 100 facts per concept. The IEEE working group P1600.1 created the Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001; Pease and Niles, 2002), which contains about 1000 terms in the upper ontology and links to over 20,000 domain-specific terms. Stoffel *et al.* (1997) describe algorithms for efficiently managing a very large ontology. A survey of techniques for extracting knowledge from Web pages is given by Etzioni *et al.* (2008).

On the Web, representation languages are emerging. RDF (Brickley and Guha, 2004) allows for assertions to be made in the form of relational triples, and provides some means for evolving the meaning of names over time. OWL (Smith *et al.*, 2004) is a description logic that supports inferences over these triples. So far, usage seems to be inversely proportional to representational complexity: the traditional HTML and CSS formats account for over 99% of Web content, followed by the simplest representation schemes, such as microformats (Khare, 2006) and RDFa (Adida and Birbeck, 2008), which use HTML and XHTML markup to add attributes to literal text. Usage of sophisticated RDF and OWL ontologies is not yet widespread, and the full vision of the Semantic Web (Berners-Lee *et al.*, 2001) has not yet been realized. The conferences on *Formal Ontology in Information Systems* (FOIS) contain many interesting papers on both general and domain-specific ontologies.

The taxonomy used in this chapter was developed by the authors and is based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990, 2005). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes’s (1978, 1985b) “Naive Physics Manifesto.”

Successful deep ontologies within a specific field include the Gene Ontology project (Consortium, 2008) and CML, the Chemical Markup Language (Murray-Rust *et al.*, 2003).

Doubts about the feasibility of a single ontology for *all* knowledge are expressed by Doctorow (2001), Gruber (2004), Halevy *et al.* (2009), and Smith (2004), who states, “the initial project of building one single ontology ... has ... largely been abandoned.”

The event calculus was introduced by Kowalski and Sergot (1986) to handle continuous time, and there have been several variations (Sadri and Kowalski, 1995; Shanahan, 1997) and overviews (Shanahan, 1999; Mueller, 2006). van Lambalgen and Hamm (2005) show how the logic of events maps onto the language we use to talk about events. An alternative to the event and situation calculi is the fluent calculus (Thielscher, 1999). James Allen introduced time intervals for the same reason (Allen, 1984), arguing that intervals were much more natural than situations for reasoning about extended and concurrent events. Peter Ladkin (1986a, 1986b) introduced “concave” time intervals (intervals with gaps; essentially, unions of ordinary “convex” time intervals) and applied the techniques of mathematical abstract algebra to time representation. Allen (1991) systematically investigates the wide variety of techniques available for time representation; van Beek and Manchak (1996) analyze algorithms for temporal reasoning. There are significant commonalities between the event-based ontology given in this chapter and an analysis of events due to the philosopher Donald Davidson (1980). The **histories** in Pat Hayes’s (1985a) ontology of liquids and the **chronicles** in McDermott’s (1985) theory of plans were also important influences on the field and this chapter.

The question of the ontological status of substances has a long history. Plato proposed that substances were abstract entities entirely distinct from physical objects; he would say *MadeOf(Butter₃, Butter)* rather than *Butter₃ ∈ Butter*. This leads to a substance hierarchy in which, for example, *UnsaltedButter* is a more specific substance than *Butter*. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack. The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed originally by the Polish logician Leśniewski (1916). His **mereology** (the name is derived from the Greek word for “part”) used the part–whole relation as a substitute for mathematical set theory, with the aim of eliminating abstract entities such as sets. A more readable exposition of these ideas is given by Leonard and Goodman (1940), and Goodman’s *The Structure of Appearance* (1977) applies the ideas to various problems in knowledge representation. While some aspects of the mereological approach are awkward—for example, the need for a separate inheritance mechanism based on part–whole relations—the approach gained the support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation. Casati and Varzi (1999) cover parts, wholes, and the spatial locations.

Mental objects have been the subject of intensive study in philosophy and AI. There are three main approaches. The one taken in this chapter, based on modal logic and possible worlds, is the classical approach from philosophy (Hintikka, 1962; Kripke, 1963; Hughes and Cresswell, 1996). The book *Reasoning about Knowledge* (Fagin *et al.*, 1995) provides a thorough introduction. The second approach is a first-order theory in which mental objects are fluents. Davis (2005) and Davis and Morgenstern (2005) describe this approach. It relies on the possible-worlds formalism, and builds on work by Robert Moore (1980, 1985). The third approach is a **syntactic theory**, in which mental objects are represented by character

strings. A string is just a complex term denoting a list of symbols, so *CanFly(Clark)* can be represented by the list of symbols $[C, a, n, F, l, y, (, C, l, a, r, k,)]$. The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Ernie Davis (1990) provides an excellent comparison of the syntactic and modal theories of knowledge.

The Greek philosopher Porphyry (c. 234–305 A.D.), commenting on Aristotle's *Categories*, drew what might qualify as the first semantic network. Charles S. Peirce (1909) developed existential graphs as the first semantic network formalism using modern logic. Ross Quillian (1961), driven by an interest in human memory and language processing, initiated work on semantic networks within AI. An influential paper by Marvin Minsky (1975) presented a version of semantic networks called **frames**; a frame was a representation of an object or category, with attributes and relations to other objects or categories. The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "IS-A links" Woods's (1975) famous article "What's In a Link?" drew the attention of AI researchers to the need for precise semantics in knowledge representation formalisms. Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper, claiming that "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation without Denotation!" argued that the model-theoretic approach to semantics used in first-order logic should be applied to all knowledge representation formalisms. This remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

The development of description logics is the most recent stage in a long line of research aimed at finding useful subsets of first-order logic for which inference is computationally tractable. Hector Levesque and Ron Brachman (1987) showed that certain logical constructs—notably, certain uses of disjunction and negation—were primarily responsible for the intractability of logical inference. Building on the KL-ONE system (Schmolze and Lipkis, 1983), several researchers developed systems that incorporate theoretical complexity analysis, most notably KRYPTON (Brachman *et al.*, 1983) and Classic (Borgida *et al.*, 1989). The result has been a marked increase in the speed of inference and a much better understanding of the interaction between complexity and expressiveness in reasoning systems. Calvanese *et al.* (1999) summarize the state of the art, and Baader *et al.* (2007) present a comprehensive handbook of description logic. Against this trend, Doyle and Patil (1991) have argued that restricting the expressiveness of a language either makes it impossible to solve certain problems or encourages the user to circumvent the language restrictions through nonlogical means.

The three main formalisms for dealing with nonmonotonic inference—circumscription (McCarthy, 1980), default logic (Reiter, 1980), and modal nonmonotonic logic (McDermott and Doyle, 1980)—were all introduced in one special issue of the AI Journal. Delgrande and Schaub (2003) discuss the merits of the variants, given 25 years of hindsight. Answer set programming can be seen as an extension of negation as failure or as a refinement of circum-

scription; the underlying theory of stable model semantics was introduced by Gelfond and Lifschitz (1988), and the leading answer set programming systems are DLV (Eiter *et al.*, 1998) and SMODELS (Niemelä *et al.*, 2000). The disk drive example comes from the SMODELS user manual (Syrjänen, 2000). Lifschitz (2001) discusses the use of answer set programming for planning. Brewka *et al.* (1997) give a good overview of the various approaches to nonmonotonic logic. Clark (1978) covers the negation-as-failure approach to logic programming and Clark completion. Van Emden and Kowalski (1976) show that every Prolog program without negation has a unique minimal model. Recent years have seen renewed interest in applications of nonmonotonic logics to large-scale knowledge representation systems. The BENINQ systems for handling insurance-benefit inquiries was perhaps the first commercially successful application of a nonmonotonic inheritance system (Morgenstern, 1998). Lifschitz (2001) discusses the application of answer set programming to planning. A variety of nonmonotonic reasoning systems based on logic programming are documented in the proceedings of the conferences on *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. Forbus and de Kleer (1993) explain in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient incremental TMS called an ITMS makes it feasible to plan the operations of a NASA spacecraft in real time.

This chapter could not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

QUALITATIVE PHYSICS

Qualitative physics: Qualitative physics is a subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD, a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modeling the physics of the blocks world to calculate the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics-like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD's physical modeling. Hayes (1985a) uses "histories"—four-dimensional slices of space-time similar to Davidson's events—to construct a fairly complex naive physics of liquids. Hayes was the first to prove that a bath with the plug in will eventually overflow if the tap keeps running and that a person who falls into a lake will get wet all over. Davis (2008) gives an update to the ontology of liquids that describes the pouring of liquids into containers.

De Kleer and Brown (1985), Ken Forbus (1985), and Benjamin Kuipers (1985) independently and almost simultaneously developed systems that can reason about a physical system based on qualitative abstractions of the underlying equations. Qualitative physics soon developed to the point where it became possible to analyze an impressive variety of complex physical systems (Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windshield wipers, and six-legged walkers (Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and

de Kleer, 1990) an encyclopedia article by Kuipers (2001), and a handbook article by Davis (2007) introduce to the field.

SPATIAL REASONING

Spatial reasoning: The reasoning necessary to navigate in the wumpus world and shopping world is trivial in comparison to the rich spatial structure of the real world. The earliest serious attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986, 1990). The region connection calculus of Cohn *et al.* (1997) supports a form of qualitative spatial reasoning and has led to new kinds of geographical information systems; see also (Davis, 2006). As with qualitative physics, an agent can go a long way, so to speak, without resorting to a full metric representation. When such a representation is necessary, techniques developed in robotics (Chapter 25) can be used.

PSYCHOLOGICAL REASONING

Psychological reasoning: Psychological reasoning involves the development of a working *psychology* for artificial agents to use in reasoning about themselves and other agents. This is often based on so-called folk psychology, the theory that humans in general are believed to use in reasoning about themselves and other humans. When AI researchers provide their artificial agents with psychological theories for reasoning about other agents, the theories are frequently based on the researchers' description of the logical agents' own design. Psychological reasoning is currently most useful within the context of natural language understanding, where divining the speaker's intentions is of paramount importance.

Minker (2001) collects papers by leading researchers in knowledge representation, summarizing 40 years of work in the field. The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. Davis (1990), Stefik (1995), and Sowa (1999) provide textbook introductions to knowledge representation, van Harmelen *et al.* (2007) contributes a handbook, and a special issue of *AI Journal* covers recent progress (Davis and Morgenstern, 2004). The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

EXERCISES

12.1 Define an ontology in first-order logic for tic-tac-toe. The ontology should contain situations, actions, squares, players, marks (X, O, or blank), and the notion of winning, losing, or drawing a game. Also define the notion of a forced win (or draw): a position from which a player can force a win (or draw) with the right sequence of actions. Write axioms for the domain. (Note: The axioms that enumerate the different squares and that characterize the winning positions are rather long. You need not write these out in full, but indicate clearly what they look like.)

12.2 Figure 12.1 shows the top levels of a hierarchy for everything. Extend it to include as many real categories as possible. A good way to do this is to cover all the things in your everyday life. This includes objects and events. Start with waking up, and proceed in an orderly fashion noting everything that you see, touch, do, and think about. For example, a random sampling produces music, news, milk, walking, driving, gas, Soda Hall, carpet, talking, Professor Fateman, chicken curry, tongue, \$7, sun, the daily newspaper, and so on.

You should produce both a single hierarchy chart (on a large sheet of paper) and a listing of objects and categories with the relations satisfied by members of each category. Every object should be in a category, and every category should be in the hierarchy.

12.3 Develop a representational system for reasoning about windows in a window-based computer interface. In particular, your representation should be able to describe:

- The state of a window: minimized, displayed, or nonexistent.
- Which window (if any) is the active window.
- The position of every window at a given time.
- The order (front to back) of overlapping windows.
- The actions of creating, destroying, resizing, and moving windows; changing the state of a window; and bringing a window to the front. Treat these actions as atomic; that is, do not deal with the issue of relating them to mouse actions. Give axioms describing the effects of actions on fluents. You may use either event or situation calculus.

Assume an ontology containing *situations*, *actions*, *integers* (for x and y coordinates) and *windows*. Define a language over this ontology; that is, a list of constants, function symbols, and predicates with an English description of each. If you need to add more categories to the ontology (e.g., pixels), you may do so, but be sure to specify these in your write-up. You may (and should) use symbols defined in the text, but be sure to list these explicitly.

12.4 State the following in the language you developed for the previous exercise:

- a. In situation S_0 , window W_1 is behind W_2 but sticks out on the left and right. Do *not* state exact coordinates for these; describe the *general* situation.
- b. If a window is displayed, then its top edge is higher than its bottom edge.
- c. After you create a window w , it is displayed.
- d. A window can be minimized if it is displayed.

12.5 (Adapted from an example by Doug Lenat.) Your mission is to capture, in logical form, enough knowledge to answer a series of questions about the following simple scenario:

Yesterday John went to the North Berkeley Safeway supermarket and bought two pounds of tomatoes and a pound of ground beef.

Start by trying to represent the content of the sentence as a series of assertions. You should write sentences that have straightforward logical structure (e.g., statements that objects have certain properties, that objects are related in certain ways, that all objects satisfying one property satisfy another). The following might help you get started:

- Which classes, objects, and relations would you need? What are their parents, siblings and so on? (You will need events and temporal ordering, among other things.)
- Where would they fit in a more general hierarchy?
- What are the constraints and interrelationships among them?
- How detailed must you be about each of the various concepts?

To answer the questions below, your knowledge base must include background knowledge. You'll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what the purchases will be used for, and so on. Try to make your representation as general as possible. To give a trivial example: don't say "People buy food from Safeway," because that won't help you with those who shop at another supermarket. Also, don't turn the questions into answers; for example, question (c) asks "Did John buy any meat?"—not "Did John buy a pound of ground beef?"

Sketch the chains of reasoning that would answer the questions. If possible, use a logical reasoning system to demonstrate the sufficiency of your knowledge base. Many of the things you write might be only approximately correct in reality, but don't worry too much; the idea is to extract the common sense that lets you answer these questions at all. A truly complete answer to this question is *extremely* difficult, probably beyond the state of the art of current knowledge representation. But you should be able to put together a consistent set of axioms for the limited questions posed here.

- a. Is John a child or an adult? [Adult]
- b. Does John now have at least two tomatoes? [Yes]
- c. Did John buy any meat? [Yes]
- d. If Mary was buying tomatoes at the same time as John, did he see her? [Yes]
- e. Are the tomatoes made in the supermarket? [No]
- f. What is John going to do with the tomatoes? [Eat them]
- g. Does Safeway sell deodorant? [Yes]
- h. Did John bring some money or a credit card to the supermarket? [Yes]
- i. Does John have less money after going to the supermarket? [Yes]

12.6 Make the necessary additions or changes to your knowledge base from the previous exercise so that the questions that follow can be answered. Include in your report a discussion of your changes, explaining why they were needed, whether they were minor or major, and what kinds of questions would necessitate further changes.

- a. Are there other people in Safeway while John is there? [Yes—staff!]
- b. Is John a vegetarian? [No]
- c. Who owns the deodorant in Safeway? [Safeway Corporation]
- d. Did John have an ounce of ground beef? [Yes]
- e. Does the Shell station next door have any gas? [Yes]
- f. Do the tomatoes fit in John's car trunk? [Yes]

12.7 Represent the following seven sentences using and extending the representations developed in the chapter:

- a. Water is a liquid between 0 and 100 degrees.
- b. Water boils at 100 degrees.
- c. The water in John's water bottle is frozen.
- d. Perrier is a kind of water.
- e. John has Perrier in his water bottle.
- f. All liquids have a freezing point.
- g. A liter of water weighs more than a liter of alcohol.

12.8 Write definitions for the following:

- a. *ExhaustivePartDecomposition*
- b. *PartPartition*
- c. *PartwiseDisjoint*

These should be analogous to the definitions for *ExhaustiveDecomposition*, *Partition*, and *Disjoint*. Is it the case that $PartPartition(s, BunchOf(s))$? If so, prove it; if not, give a counterexample and define sufficient conditions under which it does hold.

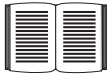
12.9 An alternative scheme for representing measures involves applying the units function to an abstract length object. In such a scheme, one would write $Inches(Length(L_1)) = 1.5$. How does this scheme compare with the one in the chapter? Issues include conversion axioms, names for abstract quantities (such as “50 dollars”), and comparisons of abstract measures in different units (50 inches is more than 50 centimeters).

12.10 Add sentences to extend the definition of the predicate $Name(s, c)$ so that a string such as “laptop computer” matches the appropriate category names from a variety of stores. Try to make your definition general. Test it by looking at ten online stores, and at the category names they give for three different categories. For example, for the category of laptops, we found the names “Notebooks,” “Laptops,” “Notebook Computers,” “Notebook,” “Laptops and Notebooks,” and “Notebook PCs.” Some of these can be covered by explicit *Name* facts, while others could be covered by sentences for handling plurals, conjunctions, etc.

12.11 Write event calculus axioms to describe the actions in the wumpus world.

12.12 State the interval-algebra relation that holds between every pair of the following real-world events:

- LK*: The life of President Kennedy.
- IK*: The infancy of President Kennedy.
- PK*: The presidency of President Kennedy.
- LJ*: The life of President Johnson.
- PJ*: The presidency of President Johnson.
- LO*: The life of President Obama.

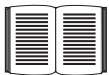


12.13 Investigate ways to extend the event calculus to handle *simultaneous* events. Is it possible to avoid a combinatorial explosion of axioms?

12.14 Construct a representation for exchange rates between currencies that allows for daily fluctuations.

12.15 Define the predicate *Fixed*, where $Fixed(Location(x))$ means that the location of object x is fixed over time.

12.16 Describe the event of trading something for something else. Describe buying as a kind of trading in which one of the objects traded is a sum of money.



12.17 The two preceding exercises assume a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. The picture is complicated still further by borrowing, leasing, renting, and bailment. Investigate the various commonsense and legal concepts of ownership, and propose a scheme by which they can be represented formally.

12.18 (Adapted from Fagin *et al.* (1995).) Consider a game played with a deck of just 8 cards, 4 aces and 4 kings. The three players, Alice, Bob, and Carlos, are dealt two cards each. Without looking at them, they place the cards on their foreheads so that the other players can see them. Then the players take turns either announcing that they know what cards are on their own forehead, thereby winning the game, or saying "I don't know." Everyone knows the players are truthful and are perfect at reasoning about beliefs.

- a. Game 1. Alice and Bob have both said "I don't know." Carlos sees that Alice has two aces (A-A) and Bob has two kings (K-K). What should Carlos say? (*Hint*: consider all three possible cases for Carlos: A-A, K-K, A-K.)
- b. Describe each step of Game 1 using the notation of modal logic.
- c. Game 2. Carlos, Alice, and Bob all said "I don't know" on their first turn. Alice holds K-K and Bob holds A-K. What should Carlos say on his second turn?
- d. Game 3. Alice, Carlos, and Bob all say "I don't know" on their first turn, as does Alice on her second turn. Alice and Bob both hold A-K. What should Carlos say?
- e. Prove that there will always be a winner to this game.

12.19 The assumption of *logical omniscience*, discussed on page 453, is of course not true of any actual reasoners. Rather, it is an *idealization* of the reasoning process that may be more or less acceptable depending on the applications. Discuss the reasonableness of the assumption for each of the following applications of reasoning about knowledge:

- a. Partial knowledge adversary games, such as card games. Here one player wants to reason about what his opponent knows about the state of the game.
- b. Chess with a clock. Here the player may wish to reason about the limits of his opponent's or his own ability to find the best move in the time available. For instance, if player A has much more time left than player B, then A will sometimes make a move that greatly complicates the situation, in the hopes of gaining an advantage because he has more time to work out the proper strategy.

- c. A shopping agent in an environment in which there are costs of gathering information.
- d. Reasoning about public key cryptography, which rests on the intractability of certain computational problems.

12.20 Translate the following description logic expression (from page 457) into first-order logic, and comment on the result:

*And(Man, AtLeast(3, Son), AtMost(2, Daughter),
 All(Son, And(Unemployed, Married, All(Spouse, Doctor))),
 All(Daughter, And(Professor, Fills(Department, Physics, Math)))) .*

12.21 Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. This exercise investigates the efficiency of using such sentences for inheritance.

- a. Consider the information in a used-car catalog such as Kelly's Blue Book—for example, that 1973 Dodge vans are (or perhaps were once) worth \$575. Suppose all this information (for 11,000 models) is encoded as logical sentences, as suggested in the chapter. Write down three such sentences, including that for 1973 Dodge vans. How would you use the sentences to find the value of a *particular* car, given a backward-chaining theorem prover such as Prolog?
- b. Compare the time efficiency of the backward-chaining method for solving this problem with the inheritance method used in semantic nets.
- c. Explain how forward chaining allows a logic-based system to solve the same problem efficiently, assuming that the KB contains only the 11,000 sentences about prices.
- d. Describe a situation in which neither forward nor backward chaining on the sentences will allow the price query for an individual car to be handled efficiently.
- e. Can you suggest a solution enabling this type of query to be solved efficiently in all cases in logic systems? (*Hint:* Remember that two cars of the same year and model have the same price.)

12.22 One might suppose that the syntactic distinction between unboxed links and singly boxed links in semantic networks is unnecessary, because singly boxed links are always attached to categories; an inheritance algorithm could simply assume that an unboxed link attached to a category is intended to apply to all members of that category. Show that this argument is fallacious, giving examples of errors that would arise.

12.23 One part of the shopping process that was not covered in this chapter is checking for compatibility between items. For example, if a digital camera is ordered, what accessory batteries, memory cards, and cases are compatible with the camera? Write a knowledge base that can determine the compatibility of a set of items and suggest replacements or additional items if the shopper makes a choice that is not compatible. The knowledge base should work with at least one line of products and extend easily to other lines.

12.24 A complete solution to the problem of inexact matches to the buyer's description in shopping is very difficult and requires a full array of natural language processing and

information retrieval techniques. (See Chapters 22 and 23.) One small step is to allow the user to specify minimum and maximum values for various attributes. The buyer must use the following grammar for product descriptions:

$$\begin{aligned} \textit{Description} &\rightarrow \textit{Category} [\textit{Connector} \textit{ Modifier}]^* \\ \textit{Connector} &\rightarrow \text{“with”} \mid \text{“and”} \mid \text{“,”} \\ \textit{Modifier} &\rightarrow \textit{Attribute} \mid \textit{Attribute Op Value} \\ \textit{Op} &\rightarrow \text{“=”} \mid \text{“>”} \mid \text{“<”} \end{aligned}$$

Here, *Category* names a product category, *Attribute* is some feature such as “CPU” or “price,” and *Value* is the target value for the attribute. So the query “computer with at least a 2.5 GHz CPU for under \$500” must be re-expressed as “computer with CPU > 2.5 GHz and price < \$500.” Implement a shopping agent that accepts descriptions in this language.

12.25 Our description of Internet shopping omitted the all-important step of actually *buying* the product. Provide a formal logical description of buying, using event calculus. That is, define the sequence of events that occurs when a buyer submits a credit-card purchase and then eventually gets billed and receives the product.