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

The task of coming up with a sequence of actions that will achieve a goal is called **planning**. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the logical planning agent of Chapter 10. This chapter is concerned primarily with scaling up to complex planning problems that defeat the approaches we have seen so far.

Section 11.1 develops an expressive yet carefully constrained language for representing planning problems, including actions and states. The language is closely related to the propositional and first-order representations of actions in Chapters 7 and 10. Section 11.2 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 heuristics were constructed for constraint satisfaction problems in Chapter 5.) Sections 11.3 through 11.5 describe planning algorithms that go beyond forward and backward search, taking advantage of the representation of the problem. In particular, we explore approaches that are not constrained to consider only totally ordered sequences of actions.

For this chapter, we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments. In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs, outlined in Chapters 12 and 17.

### 11.1 The Planning Problem

Let us consider what can happen when an ordinary problem-solving agent using standard search algorithms — depth-first A*, and so on — comes up against large, real-world problems. That will help us design better planning agents.
The most obvious difficulty is that the problem-solving agent can be overwhelmed by irrelevant actions. Consider the task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is one buying action for each 10-digit ISBN number, for a total of 10 billion actions. The search algorithm would have to examine the outcome states of all 10 billion actions to find one that satisfies the goal, which is to own a copy of ISBN 0137903952. A sensible planning agent, on the other hand, should be able to work back from an explicit goal description such as \( \text{Have}(ISBN0137903952) \) and generate the action \( \text{Buy}(ISBN0137903952) \) directly. To do this, the agent simply needs the general knowledge that \( \text{Buy}(x) \) results in \( \text{Have}(x) \). Given this knowledge and the goal, the planner can decide in a single unification step that \( \text{Buy}(ISBN0137903952) \) is the right action.

The next difficulty is finding a good heuristic function. Suppose the agent's goal is to buy four different books online. Then there will be \( 10^4 \) plans of just four steps, so searching without an accurate heuristic is out of the question. It is obvious to a human that a good heuristic estimate for the cost of a state is the number of books that remain to be bought; unfortunately, this insight is not obvious to a problem-solving agent, because it sees the goal test only as a black box that returns true or false for each state. Therefore, the problem-solving agent lacks autonomy; it requires a human to supply a heuristic function for each new problem. On the other hand, if a planning agent has access to an explicit representation of the goal as a conjunction of subgoals, then it can use a single domain-independent heuristic: the number of unsatisfied conjuncts. For the book-buying problem, the goal would be \( \text{Have}(A) \land \text{Have}(B) \land \text{Have}(C) \land \text{Have}(D) \). and a state containing \( \text{Have}(A) \land \text{Have}(C) \) would have cost 2. Thus, the agent automatically gets the right heuristic for this problem, and for many others. We shall see later in the chapter how to construct more sophisticated heuristics that examine the available actions as well as the structure of the goal.

Finally, the problem solver might be inefficient because it cannot take advantage of problem decomposition. Consider the problem of delivering a set of overnight packages to their respective destinations, which are scattered across Australia. It makes sense to find out the nearest airport for each destination and divide the overall problem into several subproblems, one for each airport. Within the set of packages routed through a given airport, whether further decomposition is possible depends on the destination city. We saw in Chapter 5 that the ability to do this kind of decomposition contributes to the efficiency of constraint satisfaction problem solvers. The same holds true for planners: in the worst case, it can take \( O(n!) \) time to find the best plan to deliver \( n \) packages, but only \( O((n/k)^x) \) time if the problem can be decomposed into \( k \) equal parts.

As we noted in Chapter 5, perfectly decomposable problems are delicious but rare.\(^1\) The design of many planning systems—particularly the partial-order planners described in Section 11.3—is based on the assumption that most real-world problems are nearly decomposable. That is, the planner can work on subgoals independently, but might need to do some additional work to combine the resulting subplans. For some problems, this assum-

\(^1\) Notice that even the delivery of a package is not perfectly decomposable. There may be cases in which it is better to assign packages to a more distant airport if that renders a flight to the nearest airport unnecessary. Nevertheless, most delivery companies prefer the computational and organizational simplicity of sticking with decomposed solutions.
tion breaks down because working on one subgoal is likely to undo another subgoal. These interactions among subgoals are what makes puzzles (like the 8-puzzle) puzzling.

**The language of planning problems**

The preceding discussion suggests that the representation of planning problems — states, actions, and goals — should make it possible for planning algorithms to take advantage of the logical structure of the problem. The key is to find a language that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. In this section, we first outline the basic representation language of classical planners, known as the STRIPS language. Later, we point out some of the many possible variations in STRIPS-like languages.

**Representation of states.** Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. We will consider propositional literals; for example, Poor ∧ Unknown might represent the state of a hapless agent. We will also use first-order literals; for example, At(Plane₁, Melbourne) ∧ At(Plane₂, Sydney) might represent a state in the package delivery problem. Literals in first-order state descriptions must be ground and function-free. Literals such as At(x, y) or At(Father(Fred), Sydney) are not allowed. The closed-world assumption is used, meaning that any conditions that are not mentioned in a state are assumed false.

**Representation of goals.** A goal is a partially specified state, represented as a conjunction of positive ground literals, such as Rich ∧ Famous or At(P₂, Tahiti). A propositional state s satisfies a goal g if s contains all the atoms in g (and possibly others). For example, the state Rich ∧ Famous satisfies the goal Rich ∧ Famous.

**Representation of actions.** An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

\[
Action(\text{Fly}(p, \text{from}, \text{to}),
\begin{align*}
\text{PRECOND:} & \quad \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}) \\
\text{EFFECT:} & \quad \text{Not At}(p, \text{from}) \land \text{At}(p, \text{to})
\end{align*}
\]

This is more properly called an action schema, meaning that it represents a number of different actions that can be derived by instantiating the variables p, from, and to to different constants. In general, an action schema consists of three parts:

- The action name and parameter list — for example, Fly(p, from, to) — serves to identify the action.
- The precondition is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
- The effect is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in

---

1 STRIPS stands for Stanford Research Institute Problem Solver.
the state resulting from the action, whereas a negative literal \( \neg P \) is asserted to be false.

Variables in the effect must also appear in the action's parameter list.

To improve readability, some planning systems divide the effect into the **add list** for positive literals and the **delete list** for negative literals.

Having defined the syntax for representations of planning problems, we can now define the semantics. The most straightforward way to do this is to describe how actions affect states. (An alternative method is to specify a direct translation into successor-state axioms, whose semantics comes from first-order logic; see Exercise 11.3.) First, we say that an action is applicable in any state that satisfies the precondition; otherwise, the action has no effect. For a first-order action schema, establishing applicability will involve a substitution \( \theta \) for the variables in the precondition. For example, suppose the current state is described by

\[
\text{At}(P_1, \text{JFK}) \land \text{At}(P_2, \text{SFO}) \land \text{Plane}(P_1) \land \text{Plane}(P_2) \land \text{Airport}(\text{JFK}) \land \text{Airport}(\text{SFO}).
\]

This state satisfies the precondition

\[
\text{At}(p, \text{from}) \land \text{Plane}(p') \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}),
\]

with substitution \( \{p/P_1, \text{from}/\text{JFK}, \text{to}/\text{SFO}\} \) (among others—see Exercise 11.2). Thus, the concrete action \( \text{Fly}(P_1, \text{JFK}, SFO) \) is applicable.

Starting in state \( s \), the result of executing an applicable action \( a \) is a state \( s' \) that is the same as \( s \) except that any positive literal \( P \) in the effect of \( a \) is added to \( s \) and any negative literal \( \neg P \) is removed from \( s' \). Thus, after \( \text{Fly}(P_1, \text{JFK}, SFO) \), the current state becomes

\[
\text{At}(P_1, \text{SFO}) \land \text{At}(P_2, \text{SFO}) \land \text{Plane}(P_1) \land \text{Plane}(P_2) \land \text{Airport}(\text{JFK}) \land \text{Airport}(\text{SFO}).
\]

Note that if a positive effect is already in \( s \) it is not added twice, and if a negative effect is not in \( s \), then that part of the effect is ignored. This definition embodies the so-called STRIPS assumption: that every literal not mentioned in the effect remains unchanged. In this way, STRIPS avoids the representational frame problem described in Chapter 10.

Finally, we can define the solution for a planning problem. In its simplest form, this is just an action sequence that, when executed in the initial state, results in a state that satisfies the goal. Later in the chapter, we will allow solutions to be partially ordered sets of actions, provided that every action sequence that respects the partial order is a solution.

**Expressiveness and extensions**

The various restrictions imposed by the STRIPS representation were chosen in the hope of making planning algorithms simpler and more efficient, without making it too difficult to describe real problems. One of the most important restrictions is that literals be function-free. With this restriction, we can be sure that any action schema for a given problem can be propositionalized—that is, turned into a finite collection of purely propositional action representations with no variables. (See Chapter 9 for more on this topic.) For example, in the air cargo domain for a problem with 10 planes and five airports, we could translate the \( \text{Fly}(p, \text{from}, \text{to}) \) schema into \( 10 \times 5 \times 5 = 250 \) purely propositional actions. The planners
Section 11.1. The Planning Problem

<table>
<thead>
<tr>
<th>STRIPS Language</th>
<th>ADL Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only positive literals in states: Poor &amp; Unknown</td>
<td>Positive and negative literals in states: −Rich &amp; −Famous</td>
</tr>
<tr>
<td>Closed World Assumption: Unmentioned literals are false.</td>
<td>Open World Assumption: Unmentioned literals are unknown.</td>
</tr>
<tr>
<td>Effect ( P \land \neg Q ) means add ( P ) and delete ( Q ).</td>
<td>Effect ( P \land \neg Q ) means add ( P ) and ( \neg Q ) and delete ( \neg P ) and ( Q ).</td>
</tr>
<tr>
<td>Only ground literals in goals: Rich &amp; Famous</td>
<td>Quantified variables in goals: ( \exists x. At(P_1, x) \land At(P_2, x) ) is the goal of having ( P_1 ) and ( P_2 ) in the same place.</td>
</tr>
<tr>
<td>Goals are conjunctions: Rich &amp; Famous</td>
<td>Goals allow conjunction and disjunction: ( \neg Poor \land (Famous \lor Smart) )</td>
</tr>
<tr>
<td>Effects are conjunctions.</td>
<td>Conditional effects allowed: when ( P ): ( E ) means ( E ) is an effect only if ( P ) is satisfied.</td>
</tr>
<tr>
<td>No support for equality.</td>
<td>Equality predicate ( (x = y) ) is built in.</td>
</tr>
<tr>
<td>No support for types.</td>
<td>Variables can have types, as in ((p : Plane)).</td>
</tr>
</tbody>
</table>

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

In Sections 11.4 and 11.5 work directly with propositionalized descriptions. If we allow function symbols, then infinitely many states and actions can be constructed.

In recent years, it has become clear that STRIPS is insufficiently expressive for some real domains. As a result, many language variants have been developed. Figure 11.1 briefly describes one important one, the Action Description Language or ADL, by comparing it with the basic STRIPS language. In ADL, the Fly action could be written as

\[
\text{Action}(\text{Fly}(p : Plane, from : Airport, to : Airport),
\text{PRECOND:} At(p, from) \land (from \neq to)
\text{EFFECT:} \neg At(p, from) \land At(p, to))
\]

The notation \( p : Plane \) in the parameter list is an abbreviation for \( Plane(p) \) in the precondition; this adds no expressive power, but can be easier to read. (It also cuts down on the number of possible propositional actions that can be constructed.) The precondition \( (from \neq to) \) expresses the fact that a flight cannot be made from an airport to itself. This could not be expressed succinctly in STRIPS.

The various planning formalisms used in AI have been systematized within a standard syntax called the Planning Domain Definition Language, or PDDL. This language allows researchers to exchange benchmark problems and compare results. PDDL includes sublanguages for STRIPS, ADL, and the hierarchical task networks we will see in Chapter 12.
The STRIPS and ADL notations are adequate for many real domains. The subsections that follow show some simple examples. There are still some significant restrictions, however. The most obvious is that they cannot represent in a natural way the ramifications of actions. For example, if there are people, packages, or dust motes in the airplane, then they too change location when the plane flies. We can represent these changes as the direct effects of flying, whereas it seems more natural to represent the location of the plane’s contents as a logical consequence of the location of the plane. We will see more examples of such state constraints in Section 11.5. Classical planning systems do not even attempt to address the qualification problem: the problem of unrepresented circumstances that could cause an action to fail. We will see how to address qualifications in Chapter 12.

Example: Air cargo transport

Figure 11.2 shows an air cargo transport problem involving loading and unloading cargo onto and off of planes 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 cargo is not At anywhere when it is In a plane, so At really means "available for use at a given location.” It takes some experience with action definitions to handle such details consistently. The following plan is a solution to the problem:

\[
\{\text{Load}(C_1, P_1, SFO), \text{Fly}(P_1, SFO, JFK), \text{Unload}(C_1, P_1, JFK), \\
\text{Load}(C_2, P_2, JFK), \text{Fly}(P_2, JFK, SFO), \text{Unload}(C_2, P_2, SFO)\}.
\]

Our representation is pure STRIPS. In particular, it allows a plane to fly to and from the same airport. Inequality literals in ADL could prevent this.
Example: The spare tire problem

Consider the problem of changing a flat tire. More precisely, 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 a very 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 in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.

The ADL description of the problem is shown in Figure 11.3. Notice that it is purely propositional. It goes beyond STRIPS in that it uses a negated precondition, $\neg \text{At}(\text{Flat}, \text{Axle})$, for the PutOn($\text{Spare}, \text{Axle}$) action. This could be avoided by using Clear($\text{Axle}$) instead, as we will see in the next example.

```
Init(\text{At(Flat}, \text{Axle}) \land \text{At(Spare, Trunk)})
Goal(\text{At(Spare, Axle)})
Action(\text{Remove(Spare, Trunk)},
   \text{PRECOND: At(Spare, Trunk)}
   \text{EFFECT: } \neg \text{At(Spare, Trunk)} \land \text{At(Spare, Ground)})
Action(\text{Remove(Flat, Axle)},
   \text{PRECOND: At(Flat, Axle)}
   \text{EFFECT: } \neg \text{At(Flat, Axle)} \land \text{At(Flat, Ground)})
Action(\text{PutOn(Spare, Axle)},
   \text{PRECOND: At(Spare, Ground) } \land \neg \text{At(Flat, Axle)}
   \text{EFFECT: } \neg \text{At(Spare, Ground)} \land \text{At(Spare, Axle)})
Action(\text{LeaveOvernight},
   \text{PRECOND: }
   \text{EFFECT: } \neg \text{At(Spare, Ground)} \land \neg \text{At(Spare, Axle)} \land \neg \text{At(Flat, Axle)})
```

Figure 11.3 The simple spare tire problem.

Example: The 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.\textsuperscript{3} 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 of what other blocks. For example, a goal might be to get block A on B and block C on D.

\textsuperscript{3} The blocks world used in planning research is much simpler than SHRDLU’s version, shown on page 20.
We will 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, $\forall x \neg On(x,b)$. These could be stated as preconditions in ADL. We can stay within the STRIPS language, however, by introducing a new predicate, $Clear(x)$, that is true when nothing is on $x$.

The action $Move$ moves a block $b$ from $x$ to $y$ if both $b$ and $y$ are clear. After the move is made, $x$ is clear but $y$ is not. A formal description of $Move$ in STRIPS is

\begin{align*}
Action(Move(b,x,y), \\
\text{PRECOND: } & On(b,x) \land Clear(b) \land Clear(y), \\
\text{EFFECT: } & On(b,y) \land Clear(x) \land \neg On(b,x) \land \neg Clear(y)).
\end{align*}

Unfortunately, this action does not maintain $Clear$ properly when $x$ or $y$ is the table. When $x = Table$, this action has the effect $Clear(\text{Table})$, but the table should not become clear, and when $y = Table$, it has the precondition $Clear(\text{Table})$, but the table does not have to be clear 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{align*}
Action(MoveToTable(b,x), \\
\text{PRECOND: } & On(b,x) \land Clear(b), \\
\text{EFFECT: } & On(b,\text{Table}) \land Clear(x) \land \neg On(b,x)).
\end{align*}

Second, we take the interpretation of $Clear(b)$ to be "there is a clear space on $b$ to hold a block." Under this interpretation, $Clear(\text{Table})$ will always be true. The only problem is that nothing prevents the planner from using $Move(b,x,\text{Table})$ instead of $MoveToTable(b,x)$. 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 $Block(b)$ to the precondition of $Move$.

Finally, there is the problem of spurious actions such as $Move(B,C,C)$, which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add inequality preconditions as shown in Figure 11.4.

### 11.2 Planning with State-Space Search

Now we turn our attention to planning algorithms. The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal, as shown in Figure 11.5. We can also use the explicit action and goal representations to derive effective heuristics automatically.

**Forward state-space search**

Planning with forward state-space search is similar to the problem-solving approach of Chapter 3. It is sometimes called **progression** planning, because it moves in the forward direction.
Section 11.2. Planning with State-Space Search

We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

- The **initial state** of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals; literals not appearing are false.
The actions that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals. (In the first-order case, we must apply the unifier from the preconditions to the effect literals.) Note that a single successor function works for all planning problems—a consequence of using an explicit action representation.

- The goal test checks whether the state satisfies the goal of the planning problem.
- The step cost of each action is typically 1. Although it would be easy to allow different costs for different actions, this is seldom done by STRIPS planners.

Recall that, in the absence of function symbols, the state space of a planning problem is finite. Therefore, any graph search algorithm that is complete—for example, A*—will be a complete planning algorithm.

From the earliest days of planning research (around 1961) until recently (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—just refer back to the start of Section 11.1. First, forward search does not address the irrelevant action problem—all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic. 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. But 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). On average, let’s say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about $1000^{41}$ nodes. It is clear that a very accurate heuristic will be needed to make this kind of search efficient. We will discuss some possible heuristics after looking at backward search.

### Backward state-space search

Backward state-space search was described briefly as part of bidirectional search in Chapter 3. We noted there that backward search can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly. In particular, it is not always obvious how to generate a description of the possible predecessors of the set of goal states. We will see that the STRIPS representation makes this quite easy because sets of states can be described by the literals that must be true in those states.

The main advantage of backward search is that it allows us to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

$$\text{At}(C_1, B) \land \text{At}(C_2, B) \land \ldots \land \text{At}(C_{20}, B).$$

Now consider the conjunct $\text{At}(C_1, B)$. Working backwards, we can seek actions that have this as an effect. There is only one: $\text{Unload}(C_1, p, B)$ where plane $p$ is unspecified.
Notice that there are many irrelevant actions that can also lead to a goal state. For example, we can fly an empty plane from JFK to SFO; this action reaches a goal state from a predecessor state in which the plane is at JFK and all the goal conjuncts are satisfied. A backward search that allows irrelevant actions will still be complete, but it will be much less efficient. If a solution exists, it will be found by a backward search that allows only relevant actions. The restriction to relevant actions means that backward search often has a much lower branching factor than forward search. For example, our air cargo problem has about 1000 actions leading forward from the initial state, but only 20 actions working backward from the goal.

Searching backwards is sometimes called regression planning. The principal question in regression planning is this: what are the states from which applying a given action leads to the goal? Computing the description of these states is called regressing the goal through the action. To see how to do it, consider the air cargo example. We have the goal

\[ \text{At}(C_1, B) \land \text{At}(C_2, B) \land \ldots \land \text{At}(C_{20}, B) \]

and the relevant action \( \text{Unload}(C_1, p, B) \), which achieves the first conjunct. The action will work only if its preconditions are satisfied. Therefore, any predecessor state must include these preconditions: \( \text{In}(C_1, p) \land \text{At}(p, B) \). Moreover, the subgoal \( \text{At}(C_1, B) \) should not be true in the predecessor state.\(^4\) Thus, the predecessor description is

\[ \text{In}(C_1, p) \land \text{At}(p, B) \land \text{At}(C_2, B) \land \ldots \land \text{At}(C_{20}, B). \]

In addition to insisting that actions achieve some desired literal, we must insist that the actions not undo any desired literals. An action that satisfies this restriction is called consistent. For example, the action \( \text{Load}(C_2, p) \) would not be consistent with the current goal, because it would negate the literal \( \text{At}(C_2, B) \).

Given definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search. Given a goal description \( G \), let \( A \) be an action that is relevant and consistent. The corresponding predecessor is as follows:

a. Any positive effects of \( A \) that appear in \( G \) are deleted.

b. Each precondition literal of \( A \) is added, unless it already appears.

Any of the standard search algorithms can be used to carry out the search. Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem. In the first-order case, satisfaction might require a substitution for variables in the predecessor description. For example, the predecessor description in the preceding paragraph is satisfied by the initial state

\[ \text{In}(C_1, P_{12}) \land \text{At}(P_{12}, B) \land \text{At}(C_2, B) \land \ldots \land \text{At}(C_{20}, B) \]

with substitution \( \{ p/P_{12} \} \). The substitution must be applied to the actions leading from the state to the goal, producing the solution \( [\text{Unload}(C_1, P_{12}, B)] \).

\(^4\) If the subgoal were true in the predecessor state, the action would still lead to a goal state. On the other hand, such actions are irrelevant because they do not make the goal true.
Heuristics for state-space search

It turns out that neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 4 that a heuristic function estimates the distance from a state to the goal; in STRIPS planning, the cost of each action is 1, so the distance is the number of actions. The basic idea is to look at the effects of the actions and at the goals that must be achieved and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates most of the time without too much computation. We might also be able to derive an admissible heuristic—one that does not overestimate. This could be used with A* search to find optimal solutions.

There are two approaches that can be tried. The first is to derive a relaxed problem from the given problem specification, as described in Chapter 4. The optimal solution cost for the relaxed problem—which we hope is very easy to solve—gives an admissible heuristic for the original problem. The second approach is to pretend that a pure divide-and-conquer algorithm will work. This is called the subgoal independence assumption: 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.

Let us consider how to derive relaxed planning problems. Since explicit representations of preconditions and effects are available, the process will work by modifying those representations. (Compare this approach with search problems, where the successor function is a black box.) The simplest idea is to relax the problem by removing all preconditions from the actions. Then every action will always be applicable, and any literal can be achieved in one step (if there is an applicable action—if not, the goal is impossible). This almost implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals—almost but not quite, because (1) there may be two actions, each of which deletes the goal literal achieved by the other, and (2) some action may achieve multiple goals. If we combine our relaxed problem with the subgoal independence assumption, both of these issues are assumed away and the resulting heuristic is exactly the number of unsatisfied goals.

In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions that achieve multiple goals. First, we relax the problem further by removing negative effects (see Exercise 11.6). Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal. For example, consider

\[
\text{Goal}(A \land B \land C)
\]
\[
\text{Action}(X, \text{Effect} : A \land P)
\]
\[
\text{Action}(Y, \text{Effect} : B \land C \land Q)
\]
\[
\text{Action}(Z, \text{Effect} : B \land P \land Q)
\].

The minimal set cover of the goal \{A, B, C\} is given by the actions \{X, Y\}, so the set cover heuristic returns a cost of 2. This improves on the subgoal independence assumption, which
Section 11.3. Partial-Order Planning 387

Section 11.3. Partial-Order Planning

Section 11.3. Partial-Order Planning gives a heuristic value of 3. There is one minor irritation: the set cover problem is \( \text{NP-hard} \). A simple greedy set-covering algorithm is guaranteed to return a value that is within a factor of \( \log n \) of the true minimum value, where \( n \) is the number of literals in the goal, and usually works much better than this in practice. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect \( A \land \neg B \) in the original problem, it will have the effect \( A \) in the relaxed problem. This means that we need not worry about negative interactions between subplans, because no action can delete the literals achieved by another action. The solution cost of the resulting relaxed problem gives what is called the empty-delete-list heuristic. The heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm. In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.

The heuristics described here can be used in either the progression or the regression direction. At the time of writing, progression planners using the empty-delete-list heuristic hold the lead. That is likely to change as new heuristics and new search techniques are explored. Since planning is exponentially hard, no algorithm will be efficient for all problems, but many practical problems can be solved with the heuristic methods in this chapter—far more than could be solved just a few years ago.

11.3 PARTIAL-ORDER PLANNING

Forward and backward state-space search are particular forms of totally ordered plan search. They explore only strictly linear sequences of actions directly connected to the start or goal. This means that they cannot take advantage of problem decomposition. Rather than work on each subproblem separately, they must always make decisions about how to sequence actions from all the subproblems. We would prefer an approach that works on several subgoals independently, solves them with several subplans, and then combines the subplans.

Such an approach also has the advantage of flexibility in the order in which it constructs the plan. That is, the planner can work on "obvious" or "important" decisions first, rather than being forced to work on steps in chronological order. For example, a planning agent that is in Berkeley and wishes to be in Monte Carlo might first try to find a flight from San Francisco to Paris; given information about the departure and arrival times, it can then work on ways to get to and from the airports.

The general strategy of delaying a choice during search is called a least commitment strategy. There is no formal definition of least commitment, and clearly some degree of commitment is necessary, lest the search would make no progress. Despite the informality, least commitment is a useful concept for analyzing when decisions should be made in any search problem.

\[5\] Technically \( \text{STIPS-style} \) planning is PSPACE-complete unless actions have only positive preconditions and only one effect literal (Bylander, 1994).
Our first concrete example will be much simpler than planning a vacation. Consider the simple problem of putting on a pair of shoes. We can describe this as a formal planning problem as follows:

\[
\text{Goal} (\text{RightShoeOn} \land \text{LeftShoeOn})
\]

\[
\text{Init}(())
\]

\[
\text{Action} (\text{RightShoe}, \text{PRECOND}: \text{RightSockOn}, \text{EFFECT}: \text{RightShoeOn})
\]

\[
\text{Action} (\text{RightSock}, \text{EFFECT}: \text{RightSockOn})
\]

\[
\text{Action} (\text{LeftShoe}, \text{PRECOND}: \text{LeftSockOn}, \text{EFFECT}: \text{LeftShoeOn})
\]

\[
\text{Action} (\text{LeftSock}, \text{EFFECT}: \text{LeftSockOn})
\]

A planner should be able to come up with the two-action sequence \text{RightSock} followed by \text{RightShoe} to achieve the first conjunct of the goal and the sequence \text{LeftSock} followed by \text{LeftShoe} for the second conjunct. Then the two sequences can be combined to yield the final plan. In doing this, the planner will be manipulating the two subsequences independently, without committing to whether an action in one sequence is before or after an action in the other. Any planning algorithm that can place two actions into a plan without specifying which comes first is called a \text{partial-order planner}. Figure 11.6 shows the partial-order plan that is the solution to the shoes and socks problem. Note that the solution is represented as a graph of actions, not a sequence. Note also the "dummy" actions called Start and Finish, which mark the beginning and end of the plan. Calling them actions simplifies things, because now every step of a plan is an action. The partial-order solution corresponds to six possible total-order plans; each of these is called a \text{linearization} of the partial-order plan.

Partial-order planning can be implemented as a search in the space of partial-order plans. (From now on, we will just call them "plans." ) That is, we start with an empty plan. Then we consider ways of refining the plan until we come up with a complete plan that solves the problem. The actions in this search are not actions in the world, but actions on plans: adding a step to the plan, imposing an ordering that puts one action before another, and so on.

We will define the POP algorithm for partial-order planning. It is traditional to write out the POP algorithm as a stand-alone program, but we will instead formulate partial-order planning as an instance of a search problem. This allows us to focus on the plan refinement steps that can be applied, rather than worrying about how the algorithm explores the space. In fact, a wide variety of uninformed or heuristic search methods can be applied once the search problem is formulated.

Remember that the states of our search problem will be (mostly unfinished) plans. To avoid confusion with the states of the world, we will talk about plans rather than states. Each plan has the following four components, where the first two define the steps of the plan and the last two serve a bookkeeping function to determine how plans can be extended:

- A set of \text{actions} that make up the steps of the plan. These are taken from the set of actions in the planning problem. The "empty" plan contains just the \text{Start} and \text{Finish} actions. Start has no preconditions and has as its effect all the literals in the initial state of the planning problem. Finish has no effects and has as its preconditions the goal literals of the planning problem.
A set of **ordering constraints**. Each ordering constraint is of the form $A \prec B$, which is read as "$A$ before $B$" and means that action $A$ must be executed sometime before action $B$, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle—such as $A \prec B$ and $B \prec A$—represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.

A set of **causal links**. A causal link between two actions $A$ and $B$ in the plan is written as $A \xrightarrow{p} B$ and is read as "$A$ achieves $p$ for $B$." For example, the causal link $\text{RightSock} \xrightarrow{\text{RightSockOn}} \text{RightShoe}$ asserts that $\text{RightSockOn}$ is an effect of the $\text{RightSock}$ action and a precondition of $\text{RightShoe}$. It also asserts that $\text{RightSockOn}$ must remain true from the time of action $\text{RightSock}$ to the time of action $\text{RightShoe}$. In other words, the plan may not be extended by adding a new action $C$ that conflicts with the causal link. An action $C$ conflicts with $A \xrightarrow{p} B$ if $C$ has the effect $\neg p$ and if $C$ could (according to the ordering constraints) come after $A$ and before $B$. Some authors call causal links protection intervals, because the link $A \xrightarrow{p} B$ protects $p$ from being negated over the interval from $A$ to $B$.

A set of **open preconditions**. A precondition is open if it is not achieved by some action in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.
For example, the final plan in Figure 11.6 has the following components (not shown are the ordering constraints that put every other action after Start and before Finish):

Actions: \{ RightShoe, RightShoe, LeftSock, LeftShoe, Start, Finish \}
Orderings: \{ RightSock \prec RightShoe, LeftSock \prec LeftShoe \}
Links: \{ RightSock \xrightarrow{RightShoeOn} RightShoe, LeftSock \xrightarrow{LeftShoeOn} LeftShoe, RightShoe \xrightarrow{RightShoeOn} Finish, LeftShoe \xrightarrow{LeftShoeOn} Finish \}
Open Preconditions: \{ \}

We define a **consistent plan** as a plan in which there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a **solution**. A moment’s thought should convince the reader of the following fact: *every linearization of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state*. This means that we can extend the notion of "executing a plan" from total-order to partial-order plans. A partial-order plan is executed by repeatedly choosing any of the possible next actions. We will see in Chapter 12 that the flexibility available to the agent as it executes the plan can be very useful when the world fails to cooperate. The flexible ordering also makes it easier to combine smaller plans into larger ones, because each of the small plans can reorder its actions to avoid conflict with the other plans.

Now we are ready to formulate the search problem that POP solves. We will begin with a formulation suitable for propositional planning problems, leaving the first-order complications for later. As usual, the definition includes the initial state, actions, and goal test.

- The initial plan contains Start and Finish, the ordering constraint Start \prec Finish, and no causal links and has all the preconditions in Finish as open preconditions.
- The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p. Consistency is enforced as follows:
  1. The causal link A \xrightarrow{p} B and the ordering constraint A \prec B are added to the plan.
     Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add Start \prec A and A \prec Finish.
  2. We resolve conflicts between the new causal link and all existing actions and between the action A (if it is new) and all existing causal links. A conflict between A \xrightarrow{p} B and C is resolved by making C occur at some time outside the protection interval, either by adding B \prec C or C \prec A. We add successor states for either or both if they result in consistent plans.
- The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

Remember that the actions considered by the search algorithms under this formulation are plan refinement steps rather than the real actions from the domain itself. The path cost is therefore irrelevant, strictly speaking, because the only thing that matters is the total cost of the real actions in the plan to which the path leads. Nonetheless, it is possible to specify a path cost function that reflects the real plan costs: we charge 1 for each real action added to
the plan and 0 for all other refinement steps. In this way, \( g(n) \), where \( n \) is a plan, will be equal to the number of real actions in the plan. A heuristic estimate \( h(n) \) can also be used.

At first glance, one might think that the successor function should include successors for every open \( p \), not just for one of them. This would be redundant and inefficient, however, for the same reason that constraint satisfaction algorithms don't include successors for every possible variable: the order in which we consider open preconditions (like the order in which we consider CSP variables) is commutative. (See page 141.) Thus, we can choose an arbitrary ordering and still have a complete algorithm. Choosing the right ordering can lead to a faster search, but all orderings end up with the same set of candidate solutions.

A partial-order planning example

Now let's look at how POP solves the spare tire problem from Section 11.1. The problem description is repeated in Figure 11.7.

```
Init(At(Flat, Axle) ∧ At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(Spare, Trunk),
   PRECOND: At(Spare, Trunk)
   EFFECT: ¬At(Spare, Trunk) ∧ At(Spare, Ground))
Action(Remove(Flat, Axle),
   PRECOND: At(Flat, Axle)
   EFFECT: ¬At(Flat, Axle) ∧ At(Flat, Ground))
Action(PutOn(Spare, Axle),
   PRECOND: At(Spare, Ground) ∧ ¬At(Flat, Axle)
   EFFECT: ¬At(Spare, Ground) ∧ At(Spare, Axle))
Action(LeaveOvernight,
   PRECOND: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare, Trunk)
   EFFECT: ¬At(Flat, Ground) ∧ ¬At(Flat, Axle))
```

Figure 11.7 The simple flat tire problem description.

The search for a solution begins with the initial plan, containing a Start action with the effect \( At(Spare, Trunk) \) \& \( At(Flat, Axle) \) and a Finish action with the sole precondition \( At(Spare, Axle) \). Then we generate successors by picking an open precondition to work on (irrevocably) and choosing among the possible actions to achieve it. For now, we will not worry about a heuristic function to help with these decisions; we will make seemingly arbitrary choices. The sequence of events is as follows:

1. Pick the only open precondition, \( At(Spare, Axle) \) of Finish. Choose the only applicable action, PutOn(Spare, Axle).
2. Pick the \( At(Spare, Ground) \) precondition of PutOn(Spare, Axle). Choose the only applicable action, Remove(Spare, Trunk) to achieve it. The resulting plan is shown in Figure 11.8.
3. Pick the \( \neg \text{At(Flat, Axle)} \) precondition of \( \text{PutOn(Spare, Axle)} \). Just to be contrary, choose the \text{LeaveOvernight} action rather than the \text{Remove(Flat, Axle)} action. Notice that \text{LeaveOvernight} also has the effect \( \neg \text{At(Spare, Ground)} \), which means it conflicts with the causal link

\[
\text{Remove(Spare, Trunk)} \quad \text{At(Spare, Ground)} \quad \text{PutOn(Spare, Axle)}.
\]

To resolve the conflict we add an ordering constraint putting \text{LeaveOvernight} before \text{Remove(Spare, Trunk)}. The resulting plan is shown in Figure 11.9. (Why does this resolve the conflict, and why is there no other way to resolve it?)

4. The only remaining open precondition at this point is the \( \text{At(Spare, Trunk)} \) precondition of the action \text{Remove(Spare, Trunk)}. The only action that can achieve it is the existing \text{Start} action, but the causal link from \text{Start} to \text{Remove(Spare, Trunk)} is in conflict with the \( \neg \text{At(Spare, Trunk)} \) effect of \text{LeaveOvernight}. This time there is no way to resolve the conflict with \text{LeaveOvernight}: we cannot order it before \text{Start} (because nothing can come before \text{Start}), and we cannot order it after \text{Remove(Spare, Trunk)} (because there is already a constraint ordering it before \text{Remove(Spare, Trunk)}). So we are forced to back up, remove the \text{LeaveOvernight} action and the last two causal links, and return to the state in Figure 11.8. In essence, the planner has proved that \text{LeaveOvernight} doesn't work as a way to change a tire.
5. Consider again the \( \neg At(Flat, Axle) \) precondition of \( PutOn(Spare, Axle) \). This time, we choose \( Remove(Flat, Axle) \).

6. Once again, pick the \( At(Spare, Trunk) \) precondition of \( Remove(Spare, Trunk) \) and choose \( Start \) to achieve it. This time there are no conflicts.

7. Pick the \( At(Flat, Axle) \) precondition of \( Remove(Flat, Axle) \), and choose \( Start \) to achieve it. This gives us a complete, consistent plan—in other words a solution—as shown in Figure 11.10.

![Figure 11.10](image)

The final solution to the tire problem. Note that \( Remove(Spare, Trunk) \) and \( Remove(Flat, Axle) \) can be done in either order, as long as they are completed before the \( PutOn(Spare, Axle) \) action.

Although this example is very simple, it illustrates some of the strengths of partial-order planning. First, the causal links lead to early pruning of portions of the search space that, because of irresolvable conflicts, contain no solutions. Second, the solution in Figure 11.10 is a partial-order plan. In this case the advantage is small, because there are only two possible linearizations; nonetheless, an agent might welcome the flexibility—for example, if the tire has to be changed in the middle of heavy traffic.

The example also points to some possible improvement!; that could be made. For example, there is duplication of effort: \( Start \) is linked to \( Remove(Spare, Trunk) \) before the conflict causes a backtrack and is then unlinked by backtracking even though it is not involved in the conflict. It is then relinked as the search continues. This is typical of chronological backtracking and might be mitigated by dependency-directed backtracking.

**Partial-order planning with unbound variables**

In this section, we consider the complications that can arise when POP is used with first-order action representations that include variables. Suppose we have a blocks world problem (Figure 11.4) with the open precondition \( On(A, B) \) and the action

\[
Action(Move(b, x, y), \\
\text{PRECOND: } On(b, x) \land Clear(b) \land Clear(y), \\
\text{EFFECT: } On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y)) .
\]
This action achieves $On(A,B)$ because the effect $On(b,y)$ unifies with $On(A,B)$ with the substitution $\{b/A, y/B\}$. We then apply this substitution to the action, yielding

$\text{Action}(\text{Move}(A, x, B)),$  
$\text{PRECOND}: On(A, x) \land \text{Clear}(A) \land \text{Clear}(B).$  
$\text{EFFECT}: On(A, B) \land \text{Clear}(x) \land \neg On(A, x) \land \neg \text{Clear}(B).$

This leaves the variable $x$ unbound. That is, the action says to move block $A$ from somewhere, without yet saying whence. This is another example of the least commitment principle: we can delay making the choice until some other step in the plan makes it for us. For example, suppose we have $On(A, D)$ in the initial state. Then the Start action can be used to achieve $On(A, x) \land \text{Clear}(x)$, binding $x$ to $D$. This strategy of waiting for more information before choosing $x$ is often more efficient than trying every possible value of $x$ and backtracking for each one that fails.

The presence of variables in preconditions and actions complicates the process of detecting and resolving conflicts. For example, when $\text{Move}(A, x, B)$ is added to the plan, we will need a causal link

$\text{Move}(A, x, B) \xrightarrow{On(A,B)} \text{Finish}.$

If there is another action $M_2$ with effect $\neg On(A, z)$, then $M_2$ conflicts only if $z$ is $B$. To accommodate this possibility, we extend the representation of plans to include a set of inequality constraints of the form $z \neq X$ where $z$ is a variable and $X$ is either another variable or a constant symbol. In this case, we would resolve the conflict by adding $z \neq B$, which means that future extensions to the plan can instantiate $z$ to any value except $B$. Anytime we apply a substitution to a plan, we must check that the inequalities do not contradict the substitution. For example, a substitution that includes $x/y$ conflicts with the inequality constraint $x \neq y$. Such conflicts cannot be resolved, so the planner must backtrack.

A more extensive example of POP planning with variables in the blocks world is given in Section 12.6.

**Heuristics for partial-order planning**

Compared with total-order planning, partial-order planning has a clear advantage in being able to decompose problems into subproblems. It also has a disadvantage in that it does not represent states directly, so it is harder to estimate how far a partial-order plan is from achieving a goal. At present, there is less understanding of how to compute accurate heuristics for partial-order planning than for total-order planning.

The most obvious heuristic is to count the number of distinct open preconditions. This can be improved by subtracting the number of open preconditions that match literals in the Start state. As in the total-order case, this overestimates the cost when there are actions that achieve multiple goals and underestimates the cost when there are negative interactions between plan steps. The next section presents an approach that allows us to get much more accurate heuristics from a relaxed problem.

The heuristic function is used to choose which plan to refine. Given this choice, the algorithm generates successors based on the selection of a single open precondition to work
on. As in the case of variable selection on constraint satisfaction algorithms, this selection has a large impact on efficiency. The most-constrained-variable heuristic from CSPs can be adapted for planning algorithms and seems to work well. The idea is to select the open condition that can be satisfied in the fewest number of ways. There are two special cases of this heuristic. First, if an open condition cannot be achieved by any action, the heuristic will select it; this is a good idea because early detection of impossibility can save a great deal of work. Second, if an open condition can be achieved in only one way, then it should be selected because the decision is unavoidable and could provide additional constraints on other choices still to be made. Although full computation of the number of ways to satisfy each open condition is expensive and not always worthwhile, experiments show that handling the two special cases provides very substantial speedups.

### 11.4 Planning Graphs

All of the heuristics we have suggested for total-order and partial-order planning 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 extract a solution directly from the planning graph, using a specialized algorithm such as the one called GRAPHPLAN.

A planning graph consists of a sequence of levels that correspond to time steps in the plan, where level 0 is the initial state. Each level contains a set of literals and a set of actions. Roughly speaking, the literals are all those that could be true at that time step, depending on the actions executed at preceding time steps. Also roughly speaking, the actions are all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold. We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, it might be optimistic about the minimum number of time steps required for a literal to become true. Nonetheless, this number of steps in the planning graph provides a good estimate of how difficult it is to achieve a given literal from the initial state. More importantly, the planning graph is defined in such a way that it can be constructed very efficiently.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned in Section 11.1, both STRIPS and ADL representations can be propositionalized. For problems with large numbers of objects, this could result in a very substantial blowup in the number of action schemata. Despite this, planning graphs have proved to be effective tools for solving hard planning problems.

We will illustrate planning graphs with a simple example. (More complex examples lead to graphs that won't fit on the page.) Figure 11.11 shows a problem, and Figure 11.12 shows its planning graph. We start with state level $S_0$, which represents the problem's initial state. We follow that with action level $A_0$, in which we place all the actions whose preconditions are satisfied in the previous level. Each action is connected to its preconditions in $S_0$ and its effects in $S_1$, in this case introducing new literals into $S_1$ that were not in $S_0$. 
Chapter 11. Planning

Figure 11.11 The "have cake and eat cake too" problem.

Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level $S_2$. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

The planning graph needs a way to represent inaction as well as action. That is, it needs the equivalent of the frame axioms in situation calculus that allow a literal to remain true from one situation to the next if no action alters it. In a planning graph this is done with a set of persistence actions. For every positive and negative literal $C$, we add to the problem a persistence action with precondition $C$ and effect $C$. Figure 11.12 shows one "real" action, $Eat(Cake)$ in $A_0$, along with two persistence actions drawn as small square boxes.

Level $A_0$ contains all the actions that could occur in state $S_0$, but just as importantly it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure 11.12 indicate mutual exclusion (or mutex) links. For example, $Eat(Cake)$ is mutually exclusive with the persistence of either $Have(Cake)$ or $¬Eaten(Cake)$. We shall see shortly how mutex links are computed.

Level $S_1$ contains all the literals that could result from picking any subset of the actions in $A_0$. It also contains mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex: depending on the choice of actions in $A_0$, one or the other, but not both, could be the result. In other words, $S_1$ represents multiple states, just as regression state-space search does, and the mutex links are constraints that define the set of possible states.

We continue in this way, alternating between state level $S_i$ and action level $A_i$, until we reach a level where two consecutive levels are identical. At this point, we say that the graph
Section 11.4. Planning Graphs

has leveled off. Every subsequent level will be identical, so further expansion is unnecessary.

What we end up with is a structure where every level contains all the actions that are applicable in \( S_i \), along with constraints saying which pairs of actions cannot both be executed. 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. The complexity of constructing the planning graph is a low-order polynomial in the number of actions and literals, whereas the state space is exponential in the number of literals.

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.

**Planning graphs for heuristic estimation**

A planning graph, once constructed, is a rich source of information about the problem. For example, a literal that does not appear in the final level of the graph cannot be achieved by any plan. This observation can be used in backward search as follows: any state containing an unachievable literal has a cost \( h(n) = \infty \). Similarly, in partial-order planning, any plan with an unachievable open condition has \( h(n) = \infty \).

This idea can be made more general. We can estimate the cost of achieving any goal literal as the level at which it first appears in the planning graph. We will call this the level cost of the goal. In Figure 11.12, \( Have(Cake) \) has level cost 0 and \( Eaten(Cake) \) has level cost 1. It is easy to show (Exercise 11.9) that these estimates are admissible for the individual goals. The estimate might not be very good, 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 actions except persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

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 very accurate. The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this is inadmissible but works very well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 11.2. For our problem, the heuristic estimate for the conjunctive goal $\text{Have(Cake)} \land \text{A Eaten(Cake)}$ will be $0 + 1 = 1$, whereas the correct answer is 2. Moreover, if we eliminated the $\text{Bake(Cake)}$ action, the estimate would still be 1, but the conjunctive goal would be impossible. 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 $\text{Bake(Cake)}$. It dominates the max-level heuristic and works extremely well on tasks in which there is a good deal of interaction among subplans.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently soluble. 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 that 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 the 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 Section 5.2.)

**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 11.13) has two main steps, which alternate within a loop. First, it checks whether all the goal literals are present in the current level with no mutex links between any pair of them. If this is the case, then a solution might exist within the current graph, so the algorithm tries to extract that solution. Otherwise, it expands the graph by adding the actions for the current level and the state literals for the next level. The process continues until either a solution is found or it is learned that no solution exists.
**Section 11.4. Planning Graphs**

**Figure 11.13** The `GRAPHPLAN` algorithm. `GRAPHPLAN` alternates between a solution extraction step and a graph expansion step. `EXTRACT-SOLUTION` looks for whether a plan can be found, starting at the end and searching backwards. `EXPAND-GRAPH` adds the actions for the current level and the state literals for the next level.

Let us now trace the operation of `GRAPHPLAN` on the spare tire problem from Section 11.1. The entire graph is shown in Figure 11.14. The first line of `GRAPHPLAN` initializes the planning graph to a one-level \( S_0 \) graph consisting of the five literals from the initial state. The goal literal \( \text{At}(\text{Spare, Axle}) \) 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 the three actions whose preconditions exist at level \( S_0 \) (i.e., all the actions except \( \text{PutOn}(\text{Spare, Axle}) \)), 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.

**Figure 11.14** The planning graph for the spare tire problem after expansion to level \( S_2 \). Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.
At(Spare, Axle) is still not present in $S_1$, so again we do not call EXTRACT-SOLUTION. The call to EXPAND-GRAPH gives us the planning graph shown in Figure 11.14. 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. In essence, EXTRACT-SOLUTION solves a Boolean CSP whose variables are the actions at each level, and the values for each variable are in or out of the plan. We can use standard CSP algorithms for this, or we can define EXTRACT-SOLUTION as a 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 $S_C$ 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 ~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 $A_1$.
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, choose the action with the easiest preconditions first. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Termination of GRAPHPLAN

So far, we have skated over the question of termination. If a problem has no solution, can we be sure that GRAPHPLAN will not loop forever, extending the planning graph at each iteration? The answer is yes, but the full proof is beyond the scope of this book. Here, we outline just the main ideas, particularly the ones that shed light on planning graphs in general.

The first step is to notice 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 literals' increasing; 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 \( i \) nor actions that cannot be executed at level \( i \). We can see that "mutexes decrease monotonically" is true if you consider that these invisible literals and actions are mutex with everything.

The proof is a little complex, but 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.
Because the actions and literals increase and the mutexes decrease, and because there are only a finite number of actions and literals, every planning graph will eventually level off—all subsequent levels will be identical. Once a graph has leveled off, if it is missing one of the goals of the problem, or if two of the goals are mutex, then the problem can never be solved, and we can stop the GRAPHPLAN algorithm and return failure. If the graph levels off with all goals present and nonmutex, but EXTRACT-SOLUTION fails to find a solution, then we might have to extend the graph again a finite number of times, but eventually we can stop. This aspect of termination is more complex and is not covered here.

11.5 Planning with Propositional Logic

We saw in Chapter 10 that planning can be done by proving a theorem in situation calculus. That theorem says that, given the initial state and the successor-state axioms that describe the effects of actions, the goal will be true in a situation that results from a certain action sequence. As early as 1969, this approach was thought to be too inefficient for finding interesting plans. Recent developments in efficient reasoning algorithms for propositional logic (see Chapter 7) have generated renewed interest in planning as logical reasoning.

The approach we take in this section is based on testing the satisfiability of a logical sentence rather than on proving a theorem. We will be finding models of propositional sentences that look like this:

\[ \text{initial state} \land \text{all possible action descriptions} \land \text{goal} \]

The sentence will contain proposition symbols corresponding to every possible action occurrence; a model that satisfies the sentence will assign true to the actions that are part of a correct plan and false to the others. An assignment that corresponds to an incorrect plan will not be a model, because it will be inconsistent with the assertion that the goal is true. If the planning problem is unsolvable, then the sentence will be unsatisfiable.

Describing planning problems in propositional logic

The process we will follow to translate STRIPS problems into propositional logic is a textbook example (so to speak) of the knowledge representation cycle: We will begin with what seems to be a reasonable set of axioms, we will find that these axioms allow for spurious unintended models, and we will write more axioms.

Let us begin with a very simple air transport problem. In the initial state (time 0), plane \( P_1 \) is at \( SFO \) and plane \( P_2 \) is at \( JFK \). The goal is to have \( P_1 \) at \( JFK \) and \( P_2 \) at \( SFO \); that is, the planes are to change places. First, we will need distinct proposition symbols for assertions about each time step. We will use superscripts to denote the time step, as in Chapter 7. Thus, the initial state will be written as

\[ \text{At}(P_1, SFO)^0 \land \text{At}(P_2, JFK)^0 \cdot \]

(Remember that \( \text{At}(P_1, SFO)^0 \) is an atomic symbol.) Because propositional logic has no closed-world assumption, we must also specify the propositions that are not true in the initial
state. If some propositions are unknown in the initial state, then they can be left unspecified (the **open world assumption**). In this example we specify:

\[ \neg At(P_1, JFK)^0 \land \neg At(P_2, SFO)^0 \]

The goal itself must be associated with a particular time step. Since we do not know a priori how many steps it takes to achieve the goal, we can try asserting that the goal is true in the initial state, time \( T = 0 \). That is, we assert \( At(P_1, JFK)^0 \land At(P_2, SFO)^0 \). If that fails, we try again with \( T = 1 \), and so on until we reach the minimum feasible plan length. For each value of \( T \), the knowledge base will include only sentences covering the time steps from \( 0 \) up to \( T \). To ensure termination, an arbitrary upper limit, \( T_{\text{max}} \), is imposed. This algorithm is shown in Figure 11.15. An alternative approach that avoids multiple solution attempts is discussed in Exercise 11.17.

The next issue is how to encode action descriptions in propositional logic. The most straightforward approach is to have one proposition symbol for each action occurrence; for example, \( \text{Fly}(P_1, SFO, JFK)^0 \) is true if plane \( P_1 \) flies from \( SFO \) to \( JFK \) at time \( 0 \). As in Chapter 7, we write propositional versions of the successor-state axioms developed for the situation calculus in Chapter 10. For example, we have

\[ \text{At}(P_1, JFK)^1 \iff (At(P_1, JFK)^0 \land \neg ((\text{Fly}(P_1, JFK, SFO)^0 \land At(P_1, JFK)^0)) \lor (\text{Fly}(P_1, SFO, JFK)^0 \land At(P_1, SFO)^0)) \]

That is, plane \( P_1 \) will be at \( JFK \) at time \( i \) if it was at \( JFK \) at time \( 0 \) and didn't fly away, or it was at \( SFO \) at time \( 0 \) and flew to \( JFK \). We need one such axiom for each plane, airport, and time step. Moreover, each additional airport adds another way to travel to or from a given airport and hence adds more disjuncts to the right-hand side of each axiom.

With these axioms in place, we can run the satisfiability algorithm to find a plan. There ought to be a plan that achieves the goal at time \( T = 1 \), namely, the plan in which the two
planes swap places. Now, suppose the KB is

\[
\text{\textit{initial state A successor-state axioms A \textit{goal}}}^{1},
\]

which asserts that the goal is true at time \( T = 1 \). You can check that the assignment in which

\[
\text{\text{Fly}}(P_1, SFO, JFK)^0 \text{ and } \text{Fly}(P_2, JFK, SFO)^0
\]

are true and all other action symbols are false is a model of the KB. So far, so good. Are there other possible models that the satisfiability algorithm might return? Indeed, yes. Are all these other models satisfactory plans? Alas, no. Consider the rather silly plan specified by the action symbols

\[
\text{Fly}(P_1, SFO, JFK)^0 \text{ and } \text{Fly}(P_1, JFK, SFO)^0 \text{ and } \text{Fly}(P_2, JFK, SFO)^0
\]

This plan is silly because plane \( P_1 \) starts at \( SFO \), so the action \( \text{Fly}(P_1, JFK, SFO)^0 \) is infeasible. Nonetheless, the plan \textit{is} a model of the sentence in Equation (11.2). That is, it is consistent with everything we have said so far about the problem. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (11.1)) say about actions whose preconditions are not satisfied. The axioms do predict correctly that nothing will happen when such an action is executed (see Exercise 11.15), but they do \textit{not} say that the action cannot be executed! To avoid generating plans with illegal actions, we must add \textbf{precondition axioms} stating that an action occurrence requires the preconditions to be satisfied. For example, we need

\[
\text{Fly}(P_1, JFK, SFO)^0 \Rightarrow \text{At}(P_1, JFK)^0.
\]

Because \( \text{At}(P_1, JFK)^0 \) is stated to be false in the initial state, this axiom ensures that every model also has \( \text{Fly}(P_1, JFK, SFO)^0 \) set to false. With the addition of precondition axioms, there is exactly one model that satisfies all of the axioms when the goal is to be achieved at time 1, namely the model in which plane \( P_1 \) flies to \( JFK \) and plane \( P_2 \) flies to \( SFO \). Notice that this solution has two parallel actions, just as with GRAPHPLAN or POP.

More surprises emerge when we add a third airport, \( LAX \). Now, each plane has two actions that are legal in each state. When we run the satisfiability algorithm, we find that a model with \( \text{Fly}(P_1, SFO, JFK)^6 \) and \( \text{Fly}(P_2, JFK, SFO)^6 \) and \( \text{Fly}(P_2, JFK, LAX)^6 \) satisfies all the axioms. That is, the successor-state and precondition axioms allow a plane to fly to two destinations at once! The preconditions for the two flights by \( P_2 \) are satisfied in the initial state; the successor-state axioms say that \( P_2 \) will be at \( SFO \) and \( LAX \) at time 1; so the goal is satisfied. Clearly, we must add more axioms to eliminate these spurious solutions. One approach is to add \textbf{action exclusion axioms} that prevent simultaneous actions. For example, we can insist on complete exclusion by adding all possible axioms of the form

\[
-(\text{Fly}(P_2, JFK, SFO)^6 \text{ A Fly}(P_2, JFK, LAX))
\]

These axioms ensure that no two actions can occur at the same time. They eliminate all spurious plans, but also force every plan to be totally ordered. This loses the flexibility of partially ordered plans; also, by increasing the number of time steps in the plan, computation time may be lengthened.

---

\(^6\) Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.
Instead of complete exclusion, we can require only partial exclusion—that is, rule out simultaneous actions only if they interfere with each other. The conditions are the same as those for mutex actions: two actions cannot occur simultaneously if one negates a precondition or effect of the other. For example, \( \text{Fly}(P_2, JFK, SFO)^0 \) and \( \text{Fly}(P_2, JFK, LAX)^0 \) cannot both occur, because each negates the precondition of the other; on the other hand, \( \text{Fly}(P_1, SFO, JFK)^0 \) and \( \text{Fly}(P_2, JFK, SFO)^0 \) can occur together because the two planes do not interfere. Partial exclusion eliminates spurious plans without forcing a total ordering.

Exclusion axioms sometimes seem a rather blunt instrument. Instead of saying that a plane cannot fly to two airports at the same time, we might simply insist that no object can be in two places at once:

\[
\forall p, x, y, t \colon x \neq y \Rightarrow \neg(At(p, x)^t \land At(p, y)^t).
\]

This fact, combined with the successor-state axioms, implies that a plane cannot fly to two airports at the same time. Facts such as this are called state constraints. In propositional logic, of course, we have to write out all the ground instances of each state constraint. For the airport problem, the state constraint suffices to rule out all spurious plans. State constraints are often much more compact than action exclusion axioms, but they are not always easy to derive from the original STRIPS description of a problem.

To summarize, planning as satisfiability involves finding models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and either the action exclusion axioms or the state constraints. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a valid plan for the original problem—that is, every linearization of the plan is a legal sequence of actions that reaches the goal.

**Complexity of propositional encodings**

The principal drawback of the propositional approach is the sheer size of the propositional knowledge base that is generated from the original planning problem. For example, the action schema \( \text{Fly}(p, a_1, a_2) \) becomes \( T \times (\text{Planes} \times \text{Airports})^2 \) different proposition symbols. In general, the total number of action symbols is bounded by \( T \times |\text{Act}| \times |O|^P \), where \( |\text{Act}| \) is the number of action schemata, \( |O| \) is the number of objects in the domain, and \( P \) is the maximum arity (number of arguments) of any action schema. The number of clauses is larger still. For example, with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom has 583 million clauses.

Because the number of action symbols is exponential in the arity of the action schema, one answer might be to try to reduce the arity. We can do this by borrowing an idea from semantic networks (Chapter 10). Semantic networks use only binary predicates; predicates with more arguments are reduced to a set of binary predicates that describe each argument separately. Applying this idea to an action symbol such as \( \text{Fly}(P_1, SFO, JFK)^0 \), we obtain three new symbols:

- \( \text{Fly}_1(P_1)^0 \) : plane \( P_1 \) flew at time 0
- \( \text{Fly}_2(SFO)^0 \) : the origin of the flight was \( SFO \)
- \( \text{Fly}_3(JFK)^0 \) : the destination of the flight was \( JFK \).
This process, called **symbol splitting**, eliminates the need for an exponential number of symbols. Now we only need \( T \times (Act \times P \times O) \).

Symbol splitting by itself can reduce the number of symbols, but does not automatically reduce the number of axioms in the KB. That is, if each action symbol in each clause were simply replaced by a conjunction of three symbols, then the total size of the KB would remain roughly the same. Symbol splitting actually does reduce the size of the KB because some of the split symbols will be irrelevant to certain axioms and can be omitted. For example, consider the successor-state axiom in Equation (11.1), modified to include \( LAX \) and to omit action preconditions (which will be covered by separate precondition axioms):

\[
At(P_1, JFK) \equiv (At(P_1, JFK) \land \neg Fly(P_1, JFK, SFO) \land \neg Fly(P_1, JFK, LAX)) \lor Fly(P_1, SFO, JFK) \lor Fly(P_1, LAX, JFK).
\]

The first condition says that \( P_1 \) will be at JFK if it was there at time 0 and didn't fly from JFK to any other city, no matter which one; the second says it will be there if it flew to JFK from another city, no matter which one. Using the split symbols, we can simply omit the argument whose value does not matter:

\[
At(P_1, JFK) \equiv (At(P_1, JFK) \land \neg Fly_{\text{a}}(P_1) \land \neg Fly_{\text{b}}(JFK)) \lor Fly_{\text{a}}(P_1) \lor Fly_{\text{b}}(LAX, JFK).
\]

Notice that \( SFO \) and LAX are no longer mentioned in the axiom. More generally, the split action symbols now allow the size of each successor-state axiom to be independent of the number of airports. Similar reductions occur with the precondition axioms and action exclusion axioms (see Exercise 11.16). For the case described earlier with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom is reduced from 583 million clauses to 9,360 clauses.

There is one drawback: the split-symbol representation does not allow for parallel actions. Consider the two parallel actions \( Fly_{\text{a}}(P_1, SFO, JFK) \) and \( Fly_{\text{b}}(P_2, JFK, SFO) \). Converting to the split representation, we have

\[
Fly_{\text{a}}(P_1) \land Fly_{\text{b}}(SFO) \land Fly_{\text{b}}(JFK) \land Fly_{\text{a}}(LAX) \land Fly_{\text{b}}(SFO).
\]

It is no longer possible to determine what happened! We know that \( P_1 \) and \( P_2 \) flew, but we cannot identify the origin and destination of each flight. This means that a complete action exclusion axiom must be used, with the drawbacks noted previously.

Planners based on satisfiability can handle large planning problems—for example, finding optimal 30-step solutions to blocks-world planning problems with dozens of blocks. The size of the propositional encoding and the cost of solution are highly problem-dependent, but in most cases the memory required to store the propositional axioms is the bottleneck. One interesting finding from this work has been that backtracking algorithms such as DPLL are often better at solving planning problems than local search algorithms such as WALKSAT. This is because the majority of the propositional axioms are Horn clauses, which are handled efficiently by the unit propagation technique. This observation has led to the development of hybrid algorithms combining some random search with backtracking and unit propagation.
Planning is an area of great current interest within AI. One reason for this is that it combines the two major areas of AI we have covered so far: search and logic. That is, 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 to both improvements in performance amounting to several orders of magnitude in the last decade and 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 \( p \) primitive propositions in a domain, then there are \( 2^p \) states. For complex domains, \( p \) can grow quite large. Consider that objects in the domain have properties (\textit{Location}, \textit{Color}, etc.) and relations (\textit{At}, \textit{On}, \textit{Between}, etc.). With \( d \) objects in a domain with ternary relations, we get \( 2^{d^3} \) states. We might conclude that, in the worst case, planning is hopeless.

Against such pessimism, the divide-and-conquer approach can be a powerful weapon. In the best case—full decomposability of the problem—divide-and-conquer offers an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. Partial-order planners deal with this with causal links, a powerful representational approach, but unfortunately each conflict must be resolved with a choice (put the conflicting action before or after the link), and the choices can multiply exponentially. GRAPHPLAN avoids these choices during the graph construction phase, using mutex links to record conflicts without actually making a choice as to how to resolve them. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. How well this works depends on the SAT solver used.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has \textbf{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 \textit{Table}), then the subgoals are serializable bottom to top: if we first achieve \( C \) on \textit{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 domain without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner which 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.
There is more than one way to control combinatorial explosions. We saw in Chapter 5 that there are many techniques for controlling backtracking in constraint satisfaction problems (CSPs), such as dependency-directed backtracking. All of these techniques can be applied to planning. For example, extracting a solution from a planning graph can be formulated as a Boolean CSP whose variables state whether a given action should occur at a given time. The CSP can be solved using any of the algorithms in Chapter 5, such as min-conflicts. A closely related method, used in the BLACKBOX system, is to convert the planning graph into a CNF expression and then extract a plan by using a SAT solver. This approach seems to work better than SATPLAN, presumably because the planning graph has already eliminated many of the impossible states and actions from the problem. It also works better than GRAPHPLAN, presumably because a satisfiability search such as WALKSAT has much greater flexibility than the strict backtracking search that GRAPHPLAN uses.

There is no doubt that planners such as GRAPHPLAN, SATPLAN, and BLACKBOX have moved the field of planning forward, both by raising the level of performance of planning systems and by clarifying the representational and combinatorial issues involved. These methods are, however, inherently propositional and thus are limited in the domains they can express. (For example, logistics problems with a few dozen objects and locations can require gigabytes of storage for the corresponding CNF expressions.) It seems likely that first-order representations and algorithms will be required if further progress is to occur, although structures such as planning graphs will continue to be useful as a source of heuristics.

11.7 SUMMARY

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

- Planning systems are problem-solving algorithms that operate on explicit propositional (or first-order) 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.
- The STRIPS language describes actions in terms of their preconditions and effects and describes the initial and goal states as conjunctions of positive literals. The ADL language relaxes some of these constraints, allowing disjunction, negation, and quantifiers.
- State-space search can operate in the forward direction (progression) or the backward direction (regression). Effective heuristics can be derived by making a subgoal independence assumption and by various relaxations of the planning problem.
- Partial-order planning (POP) algorithms explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal. They are particularly effective on problems amenable to a divide-and-conquer approach.
Section 11.7. Summary

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, or 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.

The GRAPHPLAN algorithm processes the planning graph, using a backward search to extract a plan. It allows for some partial ordering among actions.

- The SATPLAN algorithm translates a planning problem into propositional axioms and applies a satisfiability algorithm to find a model that corresponds to a valid plan. Several different propositional representations have been developed, with varying degrees of compactness and efficiency.
- 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. STRIPS used a version of the QA3 theorem proving system (Green, 1969b) as a subroutine for establishing the truth of preconditions for actions. Lifschitz (1986) offers precise definitions and an analysis of the STRIPS language. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and a survey of its relationship to more recent planning efforts.

The action representation used by STRIPS has been far more influential than its algorithmic approach. Almost all planning systems since then have used one variant or another of the STRIPS language. Unfortunately, the proliferation of variants has made comparisons needlessly difficult. With time came a better understanding of the limitations and tradeoffs among formalisms. The Action Description Language, or ADL, (Pednault, 1986) relaxed some of the restrictions in the STRIPS language 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 STRIPS, ADL, and other languages. PDDL has been used as the standard language for the planning competitions at the AIPS conference, beginning in 1998.

Planners in the early 1970s generally worked with 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 11.11), 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.

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 by using logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time. INTERPLAN (Tate, 1975a, 1975b) also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

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.\footnote{Some confusion exists over terminology. Many authors use the term \textit{nonlinear} to mean partially ordered. This is slightly different from Sacerdoti's original usage referring to interleaved plans.}

Partial-order planning dominated the next 20 years of research, yet for much of that time, the field was not widely understood. TWEAK (Chapman, 1987) was a logical reconstruction and simplification of planning work of this time; his formulation was clear enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various formulations of the planning problem. Chapman's work led to what was arguably the first simple and readable description of a complete partial-order planner (McAllester and Rosenblitt, 1991). An implementation of McAllester and Rosenblitt's algorithm called SNLP (Soderland and Weld, 1991) was widely distributed and allowed many researchers to understand and experiment with partial-order planning for the first time. The POP algorithm described in this chapter is based on SNLP.

Weld's group also developed UCPOP (Penberthy and Weld, 1992), the first planner for problems expressed in ADL. UCPOP incorporated the number-of-unsatisfied-goals heuristic. It ran somewhat faster than SNLP, but was seldom able to find plans with more than a dozen or so steps. Although improved heuristics were developed for UCPOP (Joslin and Pollack, 1994; Gerevini and Schubert, 1996), partial-order planning fell into disrepute in the 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a rehabilitation is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN and is competitive with the fastest state-space planners.

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
Section 11.7. Summary

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) give a very thorough analysis of heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes by Subbarao Kambhampati. 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), searched planning graphs using a local search technique inspired by WALKSAT.

Planning as satisfiability and the SATPLAN algorithm were proposed by Kautz and Selman (1992), who were inspired by the surprising success of greedy local search for satisfiability problems. (See Chapter 7.) Kautz et al. (1996) also investigated various forms of propositional representations for STRIPS axioms, 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).

The resurgence of interest in state-space planning was pioneered by Drew McDermott’s UNPOP program (1996), which was the first to suggest a distance heuristic based on a relaxed problem with delete lists ignored. 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) were the first to make state-space search practical for large planning problems. The most successful state-space searcher to date is Hoffmann’s (2000) FAST_FORWARD or FF, winner of the AIPS 2000 planning competition. FF uses a simplified planning graph heuristic with a very fast search algorithm that combines forward and local search in a novel way.

Most recently, there has been interest in the representation of plans as binary decision diagrams, a compact description of finite automata 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.
Weld (1994, 1999) provides two excellent surveys of modern planning algorithms. 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. Readings in Planning (Allen et al., 1990) is a comprehensive anthology of many of the best earlier articles in the field, including several good surveys. Yang (1997) provides a book-length overview of partial-order planning techniques.

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 (AIPS), the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

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

11.2 Given the axioms from Figure 11.2, what are all the applicable concrete instances of \( Fly(p, from, to) \) in the state described by

\[
At(P_1, JFK) \land At(P_2, SFO) \land Plane(P_1) \land Plane(P_2) \\
\land Airport(JFK) \land Airport(SFO) ?
\]

11.3 Let us consider how we might translate a set of STRIPS schemata into the successor-state axioms of situation calculus. (See Chapter 10.)

- Consider the schema for \( Fly(p, from, to) \). Write a logical definition for the predicate \( FlyPrecond(p, from, to, s) \), which is true if the preconditions for \( Fly(p, from, to) \) are satisfied in situation \( s \).
- 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.
- 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.
- Finally, develop a general and precisely specified procedure for carrying out the translation from a set of STRIPS schemata to a set of successor-state axioms.

11.4 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. Grasping results in holding 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 down STRIPS-style definitions of the six actions.

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 sf situation calculus. Can this goal be solved by a STRIPS-style system?

d. Your axiom for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the Push operator is applied. Is this an example of the ramification problem or the qualification problem? Fix your problem description to account for heavy objects.

11.5 Explain why the process for generating predecessors in backward search does not need to add the literals that are negative effects of the action.

11.6 Explain why dropping negative effects from every action schema in a STRIPS problem results in a relaxed problem.

11.7 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?

d. Consider a partial-order planner that combines the method in part (c) with the standard method of adding actions to achieve open conditions. Would the resulting algorithm be the same as part (b)?

11.8 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 11.2.

11.9 Prove the following assertions about planning graphs:

- A literal that does not appear in the final level of the graph cannot be achieved.
- The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

11.10 We contrasted forward and backward state-space search planners 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.
Figure 11.16 shows a blocks-world problem 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 in STRIPS notation 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.

![Start State and Goal State](Image)

**Figure 11.16** The “Sussman anomaly” blocks-world planning problem.

11.12 Consider the problem of putting on one’s shoes and socks, as defined in Section 11.3. Apply GRAPHPLAN to this problem and show the solution obtained. Now add actions for putting on a coat and a hat. Show the partial order plan that is a solution, and show that there are 180 different linearizations of the partial-order plan. What is the minimum number of different planning graph solutions needed to represent all 180 linearizations?

11.13 The original STRIPS program was designed to control Shakey the robot. Figure 11.17 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 was never dexterous enough to climb on a box or toggle a switch, but the STRIPS planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y)$, which requires that Shakey be at $x$ and that $x$ and $y$ are locations in the same room. 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: $Push(b, x, y)$. We will need the predicate $Box$ and constants for the boxes.
- Climb onto a box: $ClimbUp(b)$; climb down from a box: $ClimbDown(b)$. We will need the predicate $On$ and the constant $Floor$.
- Turn a light switch on: $TurnOn(s)$; turn it off: $TurnOff(s)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.
Describe Shakey's six actions and the initial state from Figure 11.17 in STRIPS notation. Construct a plan for Shakey to get Box 2 into Room 2.

11.14 We saw that planning graphs can handle only propositional actions. What if we want to use planning graphs for a problem with variables in the goal, such as \( \text{At}(P_1, x) \land \text{At}(P_2, x) \), where \( x \) ranges over a finite domain of locations? How could you encode such a problem to work with planning graphs? (Hint: remember the Finish action from POP planning. What preconditions should it have?)

11.15 Up to now we have assumed that actions are only executed in the appropriate situations. Let us see what propositional successor-state axioms such as Equation (11.1) 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

\[
\text{initial state } A \text{ successos-state axioms } A \models \text{goal?}
\]

c. With first-order successor-state axioms in situation calculus (as in Chapter 10), is it possible to prove that a plan containing illegal actions will achieve the goal?

11.16 Giving examples from the airport domain, explain how symbol-splitting reduces the size of the precondition axioms and the action exclusion axioms. Derive a general formula for the size of each axiom set in terms of the number of time steps, the number of action schemata, their arities, and the number of objects.

11.17 In the SATPLAN algorithm in Figure 11.15, each call to the satisfiability algorithm asserts a goal \( g^T \), where \( T \) ranges from 0 to \( T_{\text{max}} \). Suppose instead that the satisfiability algorithm is called only once, with the goal \( g^c \lor g^1 \lor \ldots \lor g^{T_{\text{max}}} \).

a. Will this always return a plan if one exists with length less than or equal to \( T_{\text{max}} \)?

b. Does this approach introduce any new spurious "solutions"?

c. Discuss how one might modify a satisfiability algorithm such as \text{WALKSAT} \ so that it finds short solutions (if they exist) when given a disjunctive goal of this form.
12 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 tasks such as scheduling Hubble Space Telescope observations, operating factories, and handling the logistics for military campaigns are more complex; they extend the basics in terms both of the representation language and of the way the planner interacts with the environment. This chapter shows how. Section 12.1 describes planning and scheduling with time and resource constraints, and Section 12.2 describes planning with predefined subplans. Sections 12.3 to 12.6 present a series of agent architectures designed to deal with uncertain environments. Section 12.7 shows how to plan when the environment contains other agents.

12.1 TIME, SCHEDULES, AND RESOURCES

The STRIPS representation talks about what actions do, but, because the representation is based on situation calculus, it cannot talk about how long an action takes or even about when an action occurs, except to say that it is before or after another action. For some domains, we would like to talk about when actions begin and end. For example, in the cargo delivery domain, we might like to know when the plane carrying some cargo will arrive, not just that it will arrive when it is done flying.

Time is of the essence in the general family of applications called job shop scheduling. Such tasks require completing a set of jobs, each of which consists of a sequence of actions, where each action has a given duration and might require some resources. The problem is to determine a schedule that minimizes the total time required to complete all the jobs, while respecting the resource constraints.

An example of a job shop scheduling problem is given in Figure 12.1. This is a highly simplified automobile assembly problem. There are two jobs: assembling cars $C_1$ and $C_2$. Each job consists of three actions: adding the engine, adding the wheels, and inspecting the
Figure 12.1 A job shop scheduling problem for assembling two cars. The notation \( \text{Duration}(d) \) means that an action takes \( d \) minutes to execute. \( \text{Engine}(E_1, C_1, 60) \) means that \( E_1 \) is an engine that fits into chassis \( C_1 \) and takes 60 minutes to install.

results. The engine must be put in first (because having the front wheels on would inhibit access to the engine compartment) and of course the inspection must be done last.

The problem in Figure 12.1 can be solved by any of the planners we have already seen. Figure 12.2 (if you ignore the numbers) shows the solution that the partial-order planner POP would come up with. To make this a scheduling problem rather than a planning problem, we must now determine when each action should begin and end, based on the durations of actions as well as their ordering. The notation \( \text{Duration}(d) \) in the effect of an action (where \( d \) must be bound to a number) means that the action takes \( d \) minutes to complete.

Given a partial ordering of actions with durations, as in Figure 12.2, we can apply the critical path method (CPM) to determine the possible start and end times of each action. A path through 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 12.2.)

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. In the figure, the critical path is shown with bold lines. To complete the whole plan in the minimal total time, the actions on the critical path must be executed with no delay between them. Actions that are off the critical path have some leeway—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 12.2 that the whole plan will take 85 minutes, that each action on the critical path has 0 slack (this will always be the case) and that each of the actions in the assembly of \( C_1 \) have a 15-minute window in which they can be started. Together the \( ES \) and \( LS \) times for all the actions constitute a schedule for the problem.
The following formulas serve as a definition for $ES$ and $LS$ and also as the outline of a dynamic programming algorithm to compute them:

$$ES(Start) = 0.$$  
$$ES(B) = \max_{A < B} ES(A) + Duration(A).$$  
$$LS(Finish) = ES(Finish).$$  
$$LS(A) = \min_{A < B} LS(B) - Duration(A).$$

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 backwards from the $Finish$ action. The details are left as an exercise.

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, the problem of finding a minimum-duration schedule, given a partial ordering on the actions, is quite easy to solve.

Scheduling with resource constraints

Real scheduling problems are complicated by the presence of constraints on resources. For example, adding an engine to a car requires an engine hoist. If there is only one hoist, then we cannot simultaneously add engine $E_1$ to car $C_1$ and engine $E_2$ to car $C_2$; hence, the schedule shown in Figure 12.2 would be infeasible. The engine hoist is an example of a reusable resource—a resource that is "occupied" during the action but that becomes available again when the action is finished. Notice that reusable resources cannot be handled in our standard description of actions in terms of preconditions and effects, because the amount of resource available is unchanged after the action is completed. For this reason, we augment our representation to include a field of the form $\text{RESOURCE}: R(k)$, which means that $k$ units of resource $R$ are required by the action. The resource requirement is both a prerequisite—the action cannot be performed if the resource is unavailable—and a temporary effect, in the sense that the availability of resource $r$ is reduced by $k$ for the duration of the action. Figure 12.3 shows how to extend the engine assembly problem to include three resources: an engine hoist for installing engines, a wheel station for putting on the wheels, and two inspectors. Figure 12.4 shows the solution with the fastest completion time, 115 minutes. This is 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 representation of resources as numerical quantities, such as $\text{Inspectors}(2)$, rather than as named entities, such as $\text{Inspector}(I_1)$ and $\text{Inspector}(I_2)$, 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 schedule is proposed that has 10 concurrent $\text{Inspect}$ 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 $10^9$ ways of assigning inspectors to $\text{Inspect}$ actions, to no avail.

Despite their advantages, resource constraints make scheduling problems more complicated by introducing additional interactions among actions. Whereas unconstrained scheduling using the critical-path method is easy, finding a resource-constrained schedule with the earliest possible completion time is NP-hard. This complexity is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a

---

1 In contrast, consumable resources, such as screws for assembling the engine, can be handled within the standard framework; see Exercise 12.2.
Section 12.1. Time, Schedules, and Resources

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 Part II. One simple but popular heuristic is the minimum slack algorithm. It schedules actions in a greedy fashion. On each iteration, it considers the unscheduled actions that have had all their predecessors scheduled and schedules the one with the least slack for the earliest possible start. It then updates the ES and LS times for each affected action and repeats. The heuristic

\[\text{Init(Chassis}(C_1) \land \text{Chassis}(C_2)\]
\[\quad \land \text{Engine}(E_1, C_1; 30) \land \text{Engine}(E_2, C_2; 60)\]
\[\quad \land \text{Wheels}(W_1, C_1; 30) \land \text{Wheels}(W_2, C_2; 15)\]
\[\quad \land \text{EngineHoists}(1) \land \text{WheelStations}(1) \land \text{Inspectors}(2)\]
\[\text{Goal}(\text{Done}(C_1) \land \text{Done}(C_2))\]

\[\text{Action(AddEngine}(e, c),\]
\[\quad \text{PRECOND: Engine}(e, c, d) \land \text{Chassis}(c) \land \neg \text{EngineIn}(c),\]
\[\quad \text{EFFECT: EngineIn}(c) \land \text{Duration}(d),\]
\[\quad \text{RESOURCE: EngineHoists}(1)\]
\[\text{Action(AddWheels}(w, c),\]
\[\quad \text{PRECOND: Wheels}(w, c, d) \land \text{Chassis}(c) \land \neg \text{EngineIn}(c),\]
\[\quad \text{EFFECT: WheelsOn}(c) \land \text{Duration}(d),\]
\[\quad \text{RESOURCE: WheelStations}(1)\]
\[\text{Action(Inspect}(c),\]
\[\quad \text{PRECOND: EngineIn}(c) \land \text{WheelsOn}(c),\]
\[\quad \text{EFFECT: Done}(c) \land \text{Duration}(10),\]
\[\quad \text{RESOURCE: Inspectors}(1)\]

Figure 12.3 Job shop scheduling problem for assembling two cars, with resources. The available resources are one engine assembly station, one wheel assembly station, and two inspectors. The notation RESOURCE; r means that the resource r is used during execution of an action, but becomes free again when the action is complete.

Figure 12.4 A solution to the job shop scheduling problem with resources from Figure 12.3. The left-hand margin lists the three resources, and actions are shown aligned horizontally with the resources they consume. There are two possible schedules, depending on which assembly uses the engine station first; we’ve shown the optimal solution, which takes 115 minutes.
is based on the same principle as the most-constrained-variable 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 12.4.

The approach we have taken in this section is "plan first, schedule later": that is, we divided the overall problem into a planning phase in which actions are selected and partially ordered 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. This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. When there are severe resource constraints, however, it could be that some legal plans will lead to much better schedules than others. In that case, 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 11 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way that they detect conflicts with causal links. Heuristics can be modified to estimate the total completion time of a plan, rather than just the total cost of the actions. This is currently an active area of research.

12.2 Hierarchical Task Network Planning

One of the most pervasive ideas for dealing with complexity is hierarchical decomposition. 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 that 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. In the best case—when high-level solutions always turn out to have satisfactory low-level implementations—hierarchical methods can result in linear-time instead of exponential-time planning algorithms.

This section describes a planning method based on hierarchical task networks or HTNs. The approach we take combines ideas from both partial-order planning (Section 11.3) and the area known as "HTN planning." In HTN planning, the initial plan, which describes the problem, is viewed as a very high-level description of what is to be done—for example, building a house. Plans are refined by applying action decompositions. Each action decomposition reduces a high-level action to a partially ordered set of lower-level actions. Action decompositions, therefore, embody knowledge about how to implement actions. For example, building a house might be reduced to obtaining a permit, hiring a contractor, doing the construction, and paying the contractor. (Figure 12.5 shows such a decomposition.) The process continues until only primitive actions remain in the plan. Typically, the primitive actions will be actions that the agent can execute automatically. For a general contractor,
"install landscaping" might be primitive because it simply involves calling the landscaping contractor. For the landscaping contractor, actions such as "plant rhododendrons here" might be considered primitive.

In "pure" HTN planning, plans are generated only by successive action decompositions. The HTN therefore views planning as a process of making an activity description more concrete, rather than (as in the case of state-space and partial-order planning) a process of constructing an activity description, starting from the empty activity. It turns out that every STRIPS action description can be turned into an action decomposition (see Exercise 12.6), and that partial-order planning can be viewed as a special case of pure HTN planning. For certain tasks, however---especially "novel" conjunctive goals—the pure HTN viewpoint is rather unnatural, and we prefer to take a hybrid approach in which action decompositions are used as plan refinements in partial-order planning, in addition to the standard operations of establishing an open condition and resolving conflicts by adding ordering constraints. (Viewing HTN planning as an extension of partial-order planning has the additional advantage that we can use the same notational conventions instead of introducing a whole new set.) We begin by describing action decomposition in more detail. Then we explain how the partial-order planning algorithm must be modified to handle decompositions, and finally we discuss issues of completeness, complexity, and practicality.

**Representing action decompositions**

General descriptions of action decomposition methods are stored in a plan library, from which they are extracted and instantiated to fit the needs of the plan being constructed. Each method is an expression of the form \( \text{Decompose}(a, d) \). This says that an action \( a \) can be decomposed into the plan \( d \), which is represented as a partial-order plan, as described in Section 11.3.

Building a house is a nice, concrete example, so we will use it to illustrate the concept of action decomposition. Figure 12.5 depicts one possible decomposition of the \( \text{BuildHouse} \) action into four lower-level actions. Figure 12.6 shows some of the action descriptions for the domain, as well as the decomposition for \( \text{BuildHouse} \) as it would appear in the plan library. There might be other possible decompositions in the library.

The Start action of the decomposition supplies all those preconditions of actions in the plan that are not supplied by other actions. We call these the **external preconditions**. In our example, the external preconditions of the decomposition are Land and Money. Similarly, the **external effects**, which are the preconditions of Finish, are all those effects of actions in the plan that are not negated by other actions. In our example, the external effects of \( \text{BuildHouse} \) are House and \( \neg \text{Money} \). Some HTN planners also distinguish between **primary effects**, such as House, and **secondary effects**, such as \( \neg \text{Money} \). Only primary effects may be used to achieve goals, whereas both kinds of effects might cause conflicts with other actions; this can greatly reduce the search space.\(^2\)

---

\(^2\) It could also prevent the discovery of unexpected plans. For example, a person facing bankruptcy proceedings can eliminate all liquid assets (i.e., achieve \( \neg \text{Money} \)) by buying or building a house. This plan is useful because current law precludes the seizure of a primary residence by creditors.
Figure 12.5 One possible decomposition for the BuildHouse action.

---

Action(BuyLand, PRECOND: Money, EFFECT: Land $\rightarrow$ Money)
Action(GetLoan, PRECOND: GoodCredit, EFFECT: Money, A Mortgage)
Action(BuildHouse, PRECOND: Land, EFFECT: House)

Action(GetPermit, PRECOND: Land, EFFECT: Permit)
Action(HireBuilder, EFFECT: Contract)
Action(Construction, PRECOND: Permit A Contract, EFFECT: HouseBuilt A $\rightarrow$ Permit)
Action(PayBuilder, PRECOND: Money A HouseBuilt, EFFECT: $\neg$ Money A House A $\rightarrow$ Contract)

Decompose(BuildHouse)

Plan(STEPS: {$S_1$: GetPermit, $S_2$: HireBuilder, $S_3$: Construction, $S_4$: PayBuilder})

ORDERINGS: {$S_1 \prec S_2 \prec S_3 \prec S_4 \prec Finish, Start \prec S_2 \prec S_3$}

LINKS: {$Start \xrightarrow{Land} S_1, Start \xrightarrow{Money} S_2, S_1 \xrightarrow{Permit} S_3, S_2 \xrightarrow{Contract} S_3, S_3 \xrightarrow{HouseBuilt} S_4, S_4 \xrightarrow{House} Finish, S_4 \xrightarrow{Money} Finish$}

Figure 12.6 Action descriptions for the house-building problem and a detailed decomposition for the BuildHouse action. The descriptions adopt a simplified view of money and an optimistic view of builders.

A decomposition should be a correct implementation of the action. A plan d implements an action a correctly if d is a complete and consistent partial-order plan for the problem of achieving the effects of a given the preconditions of a. Obviously, a decomposition will be correct if it is the result of running a sound partial-order planner.

A plan library could contain several decompositions for any given high-level action; for example, there might be another decomposition for BuildHouse that describes a process whereby the agent builds a house from rocks and turf with its own bare hands. Each de-
composition should be a correct plan, but it could have additional preconditions and effects beyond those stated in the high-level action description. For example, the decomposition for \textit{BuildHouse} in Figure 12.5 requires \textit{Money} in addition to \textit{Land} and has the effect \textit{$\neg Money}.

The self-build option, on the other hand, doesn't require money, but does require a ready supply of \textit{Rocks} and \textit{Turf}, and could result in a \textit{BadBack}.

Given that a high-level action, such as \textit{BuildHouse}, may have several possible decompositions, it is inevitable that its STRIPS action description will hide some of the preconditions and effects of those decompositions. The preconditions of the high-level action should be the intersection of the external preconditions of its decompositions, and the effects should be the intersection of the external effects of the decompositions. Put another way, the high-level preconditions and effects are guaranteed to be a subset of the true preconditions and effects of every primitive implementation.

Two other forms of information hiding should be noted. First, the high-level description completely ignores all \textit{internal effects} of the decompositions. For example, our \textit{BuildHouse} decomposition has temporary internal effects \textit{Permit} and \textit{Contract}.\footnote{\textit{Construction} negates the \textit{Permit}, otherwise the same permit could be used to build many houses. Unfortunately, \textit{Construction} does not terminate the \textit{Contract} because we have to \textit{PayBuilder} first.} Second, the high-level description does not specify the intervals "inside" the activity during which the high-level preconditions and effects must hold. For example, the \textit{Land} precondition needs to be true (in our very approximate model) only until \textit{Get Permit} is performed, and \textit{House} is true only after \textit{PayBuilder} is performed.

Information hiding of this kind is essential if hierarchical planning is to reduce complexity; we need to be able to reason about high-level actions without worrying about myriad details of the implementations. There is, however, a price to pay. For example, conflicts might exist between internal conditions of one high-level action and internal actions of another, but these is no way to detect this from the high-level descriptions. This issue has significant implications for HTN planning algorithms. In a nutshell, whereas primitive actions can be treated as point events by the planning algorithm, high-level actions have temporal extent within which all sorts of things can be going on.

\section*{Modifying the planner for decompositions}

We now show how to modify POP to incorporate HTN planning. We do that by modifying the POP successor function (page 390) to allow decomposition methods to be applied to the current partial plan \textit{P}. The new successor plans are formed by first selecting some nonprimitive action \textit{a'} in \textit{P} and then, for any \textit{Decompose\{a, d\}} method from the plan library such that \textit{a} and \textit{a'} unify with substitution \textit{8}, replacing \textit{a'} with \textit{d = SUBST\{0, d\}}.

Figure 12.7 shows an example. At the top, there is a plan \textit{P} for getting a house. The high-level action, \textit{d = BuildHouse}, is selected for decomposition. The decomposition \textit{d} is selected from Figure 12.5, and \textit{BuildHouse} is replaced by this decomposition. An additional step, \textit{GetLoan}, is then introduced to resolve the new open condition, \textit{Money}, that is created by the decomposition step. Replacing an action with its decomposition is a bit like transplant surgery: we have to take the new subplan out of its packaging (the \textit{Start} and \textit{Finish} steps),
insert it, and hook everything up properly. There might be several ways to do this. To be more precise, we have for each possible decomposition $d'$.

1. First, the action $a'$ is removed from $P$. Then, for each step $s$ in the decomposition $d'$, we need to choose an action to fill the role of $s$ and add it to the plan. It can be either a new instantiation of $s$ or an existing step $s'$ from $P$ that unifies with $s$. For example, the decomposition of a *MakeWine* action might suggest that we *BuyLand*; possibly, we can use the same *BuyLand* action that we already have in the plan. We call this **subtask sharing**.

   In Figure 12.7, there are no sharing opportunities, so new instances of the actions are created. Once the actions have been chosen, all the internal constraints from $d'$ are copied over—for example, that *GetPermit* is ordered before *Construction* and that there is a causal link between these two steps supplying the *Permit* precondition of *Construction*. This completes the task of replacing $d'$ with the instantiation of $d'$. 

2. The next step is to hook up the ordering constraints for $d$ in the original plan to the steps in $d'$. First, consider an ordering constraint in $P$ of the form $B \prec a'$. How should $B$ be ordered with respect to the steps in $d'$? The most obvious solution is that $B$ should come before every step in $d'$, and that can be achieved by replacing every constraint of the form $Start \prec s$ in $d'$ with a constraint $B \prec s$. On the other hand, this approach might be too strict! For example, *BuyLand* has to come before *BuildHouse*, but there is no need for *BuyLand* to come before *HireBuilder* in the expanded plan. Imposing an overly strict ordering might prevent some solutions from being found. Therefore, the best solution is for each ordering constraint to record the **reason** for the constraint; then, when a high-level action is expanded, the new ordering constraints can be as relaxed as possible, consistent with the reason for the original constraint. Exactly the same considerations apply when we are replacing constraints of the form $d \prec C$. 

---

**Figure 12.7** Decomposition of a high-level action within an existing plan. The *BuildHouse* action is replaced by the decomposition from Figure 12.5. The external precondition *Land* is supplied by the existing causal link from *BuyLand*. The external precondition *Money* remains open after the decomposition step, so we add a new action, *GetLoan*.  

---
3. The final step is to hook up causal links. If \( B \) \( \rightarrow_{P} a' \) was a causal link in the original plan, replace it by a set of causal links from \( B \) to all the steps in \( d' \) with preconditions \( p \) that were supplied by the \textit{Start} step in the decomposition \( d \) (i.e., to all the steps in \( d' \) for which \( p \) is an \textit{external} precondition). In the example, the causal link \( \text{BuyLand} \overset{\text{Land}}{\rightarrow} \text{BuildHouse} \) is replaced by the link \( \text{BuyLand} \overset{\text{Land}}{\rightarrow} \text{Permit} \). (The \textit{Money} precondition for \textit{PayBuilder} in the decomposition becomes an open condition, because no action in the original plan supplies \textit{Money} to \textit{BuildHouse}.) Similarly, for each causal link \( a' \rightarrow_{P} C \) in the plan, replace it with a set of causal links to \( C \) from whichever step in \( d' \) supplies \( p \) to the \textit{Finish} step in the decomposition \( d \) (i.e., from the step in \( d' \) that has \( p \) as an \textit{external} effect). In the example, we replace the link \( \text{BuildHouse} \overset{\text{House}}{\rightarrow} \text{Finish} \) with the link \( \text{PayBuilder} \overset{\text{House}}{\rightarrow} \text{Finish} \).

This completes the additions required for generating decompositions in the context of the POP planner.\(^4\)

Additional modifications to the POP algorithm are required because of the fact that high-level actions \textit{hide information} about their final primitive implementations. In particular, the original POP algorithm backtracks with failure if the current plan contains an irresolvable conflict—that is, if an action conflicts with a causal link but cannot be ordered either before or after the link. (Figure 11.9 shows an example.) With high-level actions, on the other hand, apparently irresolvable conflicts can sometimes be resolved by \textit{decomposing} the conflicting actions and interleaving their steps. An example is given in Figure 12.8. Thus, it may be the case that a complete and consistent primitive plan can be obtained by decomposition \textit{even when no complete and consistent high-level plan exists}. This possibility means that a complete HTN planner must forgo many pruning opportunities that are available to a standard POP planner. Alternatively, we can prune anyway and hope that no solution is missed.

\section*{Discussion}

Let's begin with the bad news: pure HTN planning (where the only allowable plan refinement is decomposition) is undecidable, \textit{even though the underlying state space is finite}! This might seem very depressing, since the point of HTN planning is to gain efficiency. The difficulty arises because action decompositions can be recursive—for example, going for a walk can be implemented by taking a step and then going for a walk—so HTN plans can be arbitrarily long. In particular, the shortest HTN solution could be arbitrarily long, so that there is no way to terminate the search after any fixed time. There are, however, at least three ways to look on the bright side:

1. We can rule out recursion, which very few domains require. In that case, all HTN plans are of finite length and can be enumerated.
2. We can bound the length of solutions we care about. Because the state space is finite, a plan that has more steps than there are states in the state space \textit{must} include a loop that visits the same state twice. We would lose \textit{little} by ruling out HTN solutions of this kind, and we would control the search.

---

\(^4\) There are some additional minor modifications required for handling conflict resolution with high-level actions; the interested reader can consult the papers cited at the end of the chapter.
Figure 12.8 The *Gift of the Magi* problem, taken from the O. Henry story, shows an inconsistent abstract plan that nevertheless can be decomposed into a consistent solution. Part (a) shows the problem: A poor couple has only two prized possessions—he a gold watch and she her beautiful long hair. Each plans to buy a present to make the other happy. He decides to trade his watch to buy a silver comb for her hair, and she decides to sell her hair to get a gold chain for his watch. In (b) the partial plan is inconsistent, because there is no way to order the "Give Comb" and "Give Chain" abstract steps without a conflict. (We assume that the "Give Comb" action has the precondition Hair, because if the wife doesn't have her long hair, the action won't have the intended effect of making her happy, and similarly for the "Give Chain" action.) In (c) we decompose the "Give Comb" step with an "installment plan" method. In the first step of the decomposition, the husband takes possession of the comb and gives it to his wife, while agreeing to deliver the watch in payment at a later date. In the second step, the watch is handed over and the obligation is fulfilled. A similar method decomposes the "Give Chain" step. As long as both giving steps are ordered before the delivery steps, this decomposition solves the problem. (Note that it relies on the problem being defined so that the happiness of using the chain with the watch or the comb with the hair persists even after the possessions are surrendered.)

3. We can adopt the hybrid approach that combines POP and HTN planning. Partial-order planning by itself suffices to decide whether a plan exists, so the hybrid problem is clearly decidable.

We need to be a little bit careful with the third answer. POP can string together primitive actions in arbitrary ways, so we might find ourselves with solutions that are very hard to understand and do not have the nice, hierarchical organization of HTN plans. An appropriate compromise is to control the hybrid search so that action decompositions are preferred over adding new actions, although not to such an extent that arbitrarily long HTN plans are generated before any primitive actions can be added. One way to do this is to use a cost function
that gives a discount for actions introduced by decomposition; the larger the discount, the more the search will resemble pure HTN planning and the more hierarchical the solution will be. Hierarchical plans are usually much easier to execute in realistic settings, and are easier to fix when things go wrong.

Another important characteristic of HTN plans is the possibility of subtask sharing. Recall that subtask sharing means using the same action to implement two different steps in plan decompositions. If we disallow subtask sharing, then each instantiation of a decomposition $d'$ can be done in only one way, rather than many, thereby greatly pruning the search space. Usually, this pruning saves some time and at worst leads to a solution that is slightly longer than optimal. In some cases, however, it can be more problematic. For example, consider the goal "enjoy a honeymoon and raise a family." The plan library might come up with "get married and go to Hawaii" for the first subgoal and "get married and have two children" for the second. Without subtask sharing, the plan will include two distinct marriage actions, often considered highly undesirable.

An interesting example of the costs and benefits of subtask sharing occurs in optimizing compilers. Consider the problem of compiling the expression $\tan(x) - \sin(x)$. Most compilers accomplish this by merging two separate subroutine calls in a trivial way: all the steps of $\tan$ come before any of the steps of $\sin$. But consider the following Taylor series approximations for $\sin$ and $\tan$:

$$\tan(x) \approx x + \frac{x^3}{3} + 2\frac{x^5}{15} + \frac{2^3 x^7}{315}; \quad \sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}.$$  

An HTN planner with subtask sharing could generate a more efficient solution, because it could choose to implement many steps of the $\sin$ computation with existing steps from $\tan$. Most compilers do not do this kind of interprocedural sharing because it would take too much time to consider all the possible shared plans. Instead, most compilers generate each independently, and then perhaps modify the result with a peephole optimizer.

Given all the additional complications caused by the introduction of action decompositions, why do we believe that HTN planning can be efficient? The actual sources of complexity are hard to analyze in practice, so we consider an idealized case. Suppose, for example, that we want to construct a plan with $n$ actions. For a nonhierarchical, forward state-space planner with $b$ allowable actions at each state, the cost is $O(b^n)$. For an HTN planner, let us suppose a very regular decomposition structure: each nonprimitive action has $d$ possible decompositions, each into $k$ actions at the next lower level. We want to know how many different decomposition trees there are with this structure. Now, if there are $n$ actions at the primitive level, then the number of levels below the root is $\log_k n$, so the number of internal decomposition nodes is $1 + k + k^2 + \ldots + k^{\log_k n - 1} = (n - 1)/(k - 1)$. Each internal node has $d$ possible decompositions, so there are $d^{(n-1)/(k-1)}$ possible regular decomposition trees that could be constructed. Examining this formula, we see that keeping $d$ small and $k$ large can result in huge savings: essentially we are taking the kth root of the nonhierarchical cost, if $b$ and $d$ are comparable. On the other hand, constructing a plan library that has a small number of long decompositions, but nonetheless allows us to solve any problem, is not always possible. Another way of saying this is that long macros that are usable across a wide range of problems are extremely precious.
Another, and perhaps better, reason for believing that HTN planning is efficient is that it works in practice. Almost all planners for large-scale applications are HTN planners, because HTN planning allows the human expert to provide the crucial knowledge about how to perform complex tasks so that large plans can be constructed with little computational effort. 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 millions of steps.

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.

12.3 Planning and Acting in Nondeterministic Domains

So far we have considered only classical planning domains that are fully observable, static, and deterministic. Furthermore, we have assumed that the action descriptions are correct and complete. In these circumstances, an agent can plan first and then execute the plan "with its eyes closed." In an uncertain environment, on the other hand, an agent must use its percepts to discover what is happening while the plan is being executed and possibly modify or replace the plan if something unexpected happens.

Agents have to deal with both incomplete and incorrect information. Incompleteness arises because the world is partially observable, nondeterministic, or both. For example, the door to the office supply cabinet might or might not be locked; one of my keys might or might not open the door if it is locked; and I might or might not be aware of these kinds of incompleteness in my knowledge. Thus, my model of the world is weak, but correct. On the other hand, incorrectness arises because the world does not necessarily match my model of the world; for example, I might believe that my key opens the supply cabinet, but I could be wrong if the locks have been changed. Without the ability to handle incorrect information, an agent can end up being as unintelligent as the dung beetle (page 37), which attempts to plug up its nest with a ball of dung even after the ball has been removed from its grasp.

The possibility of having complete or correct knowledge depends on how much indeterminacy there is in the world. With bounded indeterminacy, actions can have unpredictable
effects, but the possible effects can be listed in the action description axioms. For example, when we flip a coin, it is reasonable to say that the outcome will be Heads or Tails. An agent can cope with bounded indeterminacy by making plans that work in all possible circumstances. With unbounded indeterminacy, on the other hand, the set of possible preconditions or effects either is unknown or is too large to be enumerated completely. This would be the case in very complex or dynamic domains such as driving, economic planning, and military strategy. An agent can cope with unbounded indeterminacy only if it is prepared to revise its plans and/or its knowledge base. Unbounded indeterminacy is closely related to the qualification problem discussed in Chapter 10—the impossibility of listing all the preconditions required for a real-world action to have its intended effect.

There are four planning methods for handling indeterminacy. The first two are suitable for bounded indeterminacy, and the second two for unbounded indeterminacy:

◊ Sensorless planning: Also called conformant planning, this method constructs standard, sequential plans that are to be executed without perception. The sensorless planning algorithm must ensure that the plan achieves the goal in all possible circumstances, regardless of the true initial state and the actual action outcomes. Sensorless planning relies on coercion—the idea that the world can be forced into a given state even when the agent has only partial information about the current state. Coercion is not always possible, so sensorless planning is often inapplicable. Sensorless problem solving, involving search in belief state space, was described in Chapter 3.

◊ Conditional planning: Also known as contingency planning, this approach deals with bounded indeterminacy by constructing a conditional plan with different branches for the different contingencies that could arise. Just as in classical planning, the agent plans first and then executes the plan that was produced. The agent finds out which part of the plan to execute by including sensing actions in the plan to test for the appropriate conditions. In the air transport domain, for example, we could have plans that say "check whether SFO airport is operational. If so, fly there; otherwise, fly to Oakland." Conditional planning is covered in Section 12.4.

◊ Execution monitoring and replanning: In this approach, the agent can use any of the preceding planning techniques (classical, sensorless, or conditional) to construct a plan, but it also uses execution monitoring to judge whether the plan has a provision for the actual current situation or need to be revised. Replanning occurs when something goes wrong. In this way, the agent can handle unbounded indeterminacy. For example, even if a replanning agent did not envision the possibility of SFO's being closed, it can recognize that situation when it occurs and call the planner again to find a new path to the goal. Replanning agents are covered in Section 12.5.

◊ Continuous planning: All the planners we have seen so far are designed to achieve a goal and then stop. A continuous planner is designed to persist over a lifetime. It can handle unexpected circumstances in the environment, even if these occur while the agent is in the middle of constructing a plan. It can also handle the abandonment of goals and the creation of additional goals by goal formulation. Continuous planning is described in Section 12.6.
Let's consider an example to clarify the differences among the various kinds of agents. The problem is this: given an initial state with a chair, a table, and some cans of paint, with everything of unknown color, achieve the state where the chair and table have the same color.

A **classical planning** agent could not handle this problem, because the initial state is not fully specified—we don't know what color the furniture is.

A **sensorless planning** agent must find a plan that works without requiring any sensors during plan execution. The 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). Coercion is most appropriate when propositions are expensive or impossible to perceive. For example, doctors often prescribe a broad-spectrum antibiotic rather than using the conditional plan of doing a blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic. They do this because the delays and costs involved in performing the blood tests are usually too great.

A **conditional planning** agent can generate a better plan: first sense the color of the table and chair; if they are already the same then the plan is done. If not, sense the labels on the paint cans; if there is a can that is the same color as one piece of furniture, then apply the paint to the other piece. Otherwise paint both pieces with any color.

A **replanning** agent could generate the same plan as the conditional planner, or it could generate fewer branches at first and fill in the others at execution time as needed. It could also deal with incorrectness of its action descriptions. For example, suppose that the \texttt{Paint} (\texttt{obj}, \texttt{color}) action is believed to have the deterministic effect \texttt{Color} (\texttt{obj}, \texttt{color}). A conditional planner would just assume that the effect has occurred once the action has been executed, but a replanning agent would check for the effect, and if it were not true (perhaps because the agent was careless and missed a spot), it could then replan to repaint the spot. We will return to this example on page 441.

A **continuous planning** agent, in addition to handling unexpected events, can revise its plans appropriately if, say, we add the goal of having dinner on the table, so that the painting plan must be postponed.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of conditional plan branches designed to handle punctures or crashes; on the other hand, most car drivers never consider these possibilities, so they respond to punctures and crashes as replanning agents. In general, agents create conditional plans only for those contingencies that have important consequences and a nonnegligible chance of going wrong. Thus, a car driver contemplating a trip across the Sahara desert might do well to consider explicitly the possibility of breakdowns, whereas a trip to the supermarket requires less advance planning.

The agents we describe in this chapter are designed to handle indeterminacy, but are not capable of making tradeoffs between the probability of success and the cost of plan construction. Chapter 16 provides additional tools for dealing with these issues.
12.4 CONDITIONAL PLANNING

Conditional planning is a way to deal with uncertainty by checking what is actually happening in the environment at predetermined points in the plan. Conditional planning is simplest to explain for fully observable environments, so we will begin with that case. The partially observable case is more difficult, but more interesting.

Conditional planning in fully observable environments

Full observability means that the agent always knows the current state. If the environment is nondeterministic, however, the agent will not be able to predict the outcome of its actions. A conditional planning agent handles nondeterminism by building into the plan (at planning time) conditional steps that will check the state of the environment (at execution time) to decide what to do next. The problem, then, is how to construct these conditional plans.

We will use as our example domain the venerable vacuum world, whose state space, for the deterministic case, is laid out on page 65. Recall that the available actions are Left, Right, and Suck. We will need some propositions to define the states: let AtL (AtR) be true if the agent is in the left (right) state,\(^5\) and let CleanL (CleanR) be true if the left (right) state is clean. The first thing we need to do is augment the STRIPS language to allow for nondeterminism. To do this, we allow actions to have disjunctive effects, meaning that the action can have two or more different outcomes whenever it is executed. For example, suppose that moving Left sometimes fails. Then the normal action description

\[
\text{Action(Left, PRECOND:AtR, EFFECT:AtL \land \neg AtR)}
\]

must be modified to include a disjunctive effect:

\[
\text{Action(Left, PRECOND:AtR, EFFECT:AtL \lor AtR)}. 
\] (12.1)

We will also find it useful to allow actions to have conditional effects, wherein the effect of the action depends on the state in which it is executed. Conditional effects appear in the Effect slot of an action, and have the syntax “when <condition>: <effect>.” For example, to model the Suck action, we would write

\[
\text{Action(Suck, PRECOND:, EFFECT: ([when AtL: CleanL] \land [when AtR: CleanR])].}
\]

Conditional effects do not introduce indeterminacy, but they can help to model it. For example, suppose we have a devious vacuum cleaner that sometimes dumps dirt on the destination square when it moves, but only if that square is clean. This can be modeled with a description such as

\[
\text{Action(Left, PRECOND:AtR, EFFECT:AtL \lor (AtL \land [when CleanL:\neg CleanL])},
\]

which is both disjunctive and conditional.\(^6\) To create conditional plans, we need conditional steps. We will write these using the syntax “if <test> then plan_A else plan_B,” where

---

\(^5\) Obviously, AtR is true iff \neg AtL is true, and vice versa. We use two propositions mainly to improve readability.

\(^6\) The conditional effect when CleanL: \neg CleanL may look a little odd. Remember, however, that here CleanL refers to the situation before the action and \neg CleanL refers to the situation after the action.
<test> is a Boolean function of the state variables. For example, a conditional step for the vacuum world might be, "If At L A Clean L then Right else Suck." The execution of such a step proceeds in the obvious way. By nesting conditional steps, plans become trees.

We want conditional plans that work regardless of which action outcomes actually occur. We have seen this problem before, in a different guise. In two-player games (Chapter 6), we want moves that will win regardless of which moves the opponent makes. For this reason, nondeterministic planning problems are often called games against nature.

Let us consider a specific example in the vacuum world. The initial state has the robot in the right square of a clean world; because the environment is fully observable, the agent knows the full state description, At R A Clean L A Clean R. The goal state has the robot in the left square of a clean world. This would be quite trivial, were it not for the "double Murphy" vacuum cleaner that sometimes deposits dirt when it moves to a clean destination square and sometimes deposits dirt if Suck is applied to a clean square.

A "game tree" for this environment is shown in Figure 12.9. Actions are taken by the robot in the "state" nodes of the tree, and nature decides what the outcome will be at the "chance" nodes, shown as circles. A solution is a subtree that (1) has a goal node at every leaf, (2) specifies one action at each of its "state" nodes, and (3) includes every outcome branch at each of its "chance" nodes. The solution is shown in bold lines in the figure; it corresponds to the plan [Left if At L A Clean L A Clean R then Right else Suck]. (For now, because we are using a state-space planner, the tests in conditional steps will be complete state descriptions.)

For exact solutions of games, we use the minimax algorithm (Figure 6.3). For conditional planning, there are typically two modifications. First, MAX and MIN nodes can become

![Figure 12.9](attachment:sdvd.png)

The first two levels of the search tree for the "double Murphy" vacuum world. State nodes are OR nodes where some action must be chosen. Chance nodes, shown as circles, are AND nodes where every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution is shown in bold lines.
Section 12.4.  Conditional Planning 435

function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(INITIAL-STATE[problem], problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
    if GOAL-TEST(problem)(state) then return the empty plan
    if state is on path then return failure
    for each action, state-set in SUCCESSORS[problem](state) do
        plan ← AND-SEARCH(state-set, problem, [state | path])
        if plan = failure then return [action | plan]
    return failure

function AND-SEARCH(state-set, problem, path) returns a conditional plan, or failure
    for each sᵢ in state-set do
        planᵢ ← OR-SEARCH(sᵢ, problem, path)
        if planᵢ = failure then return failure
    return [if s₁ then plan₁, else if s₂ then plan₂, else ... if sₙ₋₁ then planₙ₋₁, else planₙ]

Figure 12.10 An algorithm for searching AND-OR graphs generated by nondeterministic environments. We assume that SUCCESSORS returns a list of actions, each associated with a set of possible outcomes. The aim is to find a conditional plan that reaches a goal state in all circumstances.

OR and AND nodes. Intuitively, the plan needs to take some action at every state it reaches, but must handle every outcome for the action it takes. Second, the algorithm needs to return a conditional plan rather than just a single move. At an OR node, the plan is just the action selected, followed by whatever comes next. At an AND node, the plan is a nested series of if-then-else steps specifying subplans for each outcome; the tests in these steps are the complete state descriptions.\(^7\)

Formally speaking, the search space we have defined is an AND-OR graph. In Chapter 7, AND-OR graphs showed up in propositional Horn clause inference. Here, the branches are actions rather than logical inference steps, but the algorithm is the same. Figure 12.10 gives a recursive, depth-first algorithm for AND-OR graph search.

One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic planning problems (e.g., if an action sometimes has no effect, or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is no solution from the current state; it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some other path from the root. Exercise 12.15 investigates this issue.

\(^7\) Such plans could also be written using a case construct.
Figure 12.11  The first level of the search graph for the "triple Murphy" vacuum world, where we have shown cycles explicitly. All solutions for this problem are cyclic plans.

The plans returned by AND-OR-GRAPH-SEARCH contain conditional steps that test the entire state description to decide on a branch. In many cases, we can get away with less exhaustive tests. For example, the solution plan in Figure 12.9 could be written simply as [Left, if CleanL then [] else Suck]. This is because the single test, CleanL, suffices to divide the states at the AND-node into two singleton sets, so that after the test the agent knows exactly what state it is in. In fact, a series of if–then–else tests of single variables always suffices to divide a set of states into singletons, provided that the state is fully observable. We could, therefore, restrict the tests to be of single variables without loss of generality.

There is one final complication that often arises in nondeterministic domains: things don't always work the first time, and one has to try again. For example, consider the "triple Murphy" vacuum cleaner, which (in addition to its previously stated habits) sometimes fails to move when commanded—for example, Left can have the disjunctive effect $AtL \lor AtR$, as in Equation (12.1). Now the plan [Left, if CleanL then [] else Suck] is no longer guaranteed to work. Figure 12.11 shows part of the search graph; clearly, there are no longer any acyclic solutions, and AND-OR-GRAPH-SEARCH would return with failure. There is, however, a cyclic solution, which is to keep trying Left until it works. We can express this solution by adding a label to denote some portion of the plan and using that label later instead of repeating the plan itself. Thus, our cyclic solution is

$$[L_1: \text{Left, if } AtR \text{ then } L_1 \text{ else if } CleanL \text{ then } [] \text{ else Suck}].$$

(A better syntax for the looping part of this plan would be "while AtR do Left.") The modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 12.16. The key realization is that a loop in the state space back to a state $L$ translates to a loop in the plan back to the point where the subplan for state $L$ is executed.

Now we have the ability to synthesize complex plans that look more like programs, with conditionals and loops. Unfortunately, these loops are, potentially, infinite loops. For example, nothing in the action representation for the triple Murphy world says that Left will eventually succeed. Cyclic plans are therefore less desirable than acyclic plans, but they may be considered solutions, provided that every leaf is a goal state and a leaf is reachable from every point in the plan.
Conditional planning in partially observable environments

The preceding section dealt with fully observable environments, which have the advantage that conditional tests can ask any question at all and be sure of getting an answer. In the real world, partial observability is much more common. In the initial state of a partially observable planning problem, the agent knows only a certain amount about the actual state. The simplest way to model this situation is to say that the initial state belongs to a state set; the state set is a way of describing the agent's initial belief state.\(^8\)

Suppose that a vacuum-world agent knows that it is in the right-hand square and that the square is clean, but it cannot sense the presence or absence of dirt in other squares. Then as far as it knows it could be in one of two states: the left-hand square might be either clean or dirty. This belief state is marked \(A\) in Figure 12.112. The figure shows part of the AND–OR graph for the "alternate double Murphy" vacuum world, in which dirt can sometimes be left behind when the agent leaves a clean square.\(^9\) If the world were fully observable, the agent could construct a cyclic solution of the form "Keep moving left and right, sucking up dirt whenever it appears, until both squares are clean and I'm in the left square." (See Exercise 12.16.) Unfortunately, with local dirt sensing, this plan is unexecutable, because the truth value of the test "both squares are clean" cannot be determined.

Let us look at how the AND–OR graph is constructed. From belief state \(A\), we show the outcome of moving Left. (The other actions make no sense.) Because the agent can leave dirt behind, the two possible initial worlds become four possible worlds, as shown in \(B\) and \(C\). The worlds form two distinct belief states, classified by the available sensor information.\(^10\) In \(B\), the agent knows \(CleanL\); in \(C\) it knows \(\neg CleanL\). From \(C\), cleaning up the dirt moves the agent to \(B\). From \(B\), moving Right might or might not leave dirt behind, so there are again four possible worlds, divided according to the agent's knowledge of \(CleanR\) (back to \(A\)) or \(\neg CleanR\) (belief state \(D\)).

In sum, nondeterministic, partially observable environments give us an AND–OR graph of belief states. Conditional plans can be found, therefore, using exactly the same algorithm as in the fully observable case, namely AND-OR-GRAPH-SEARCH. Another way to understand what's going on is to see that the agent's belief state is always fully observable—if it always knows what it knows. "Standard fully observable problem solving is just a special case in which every belief state is a singleton set containing exactly one physical state.

Are we done? Not quite! We still need to decide how belief states should be represented, how sensing works, and how action descriptions should be written in this new setting.

There are three basic choices for belief states:

1. Sets of full state descriptions. For example, the initial belief state in Figure 12.12 is

\[
\{(AtR A \text{CleanR} A \text{CleanL}), (AtR A \text{CleanR} A \neg \text{CleanL})\}.
\]

This representation is simple to work with, but very expensive: if there are \(n\) Boolean

---

\(^8\) These concepts are introduced in Section 3.6, which the reader might wish to consult before proceeding.

\(^9\) Parents with young children will be familiar with this phenomenon. Apologies to others, as usual.

\(^10\) Notice that they are not classified by whether there is dirt left behind when the agent moves. Branching in belief-state space is caused by alternative knowledge outcomes, not alternative physical outcomes.
propositions defining the state, then a belief state can contain $O(2^n)$ physical state descriptions, each of size $O(n)$. Exponentially large belief states will occur whenever the agent knows only a fraction of the propositions—the less it knows, the more possible states it might be in.

2. Logical sentences that capture exactly the set of possible worlds in the belief state. For example, the initial state can be written as

$$AtR \land CleanR.$$  

Clearly, every belief state can be captured exactly by a single logical sentence; if we have to, we can use the disjunction of all the conjunctive state descriptions, but our example shows that more compact sentences could exist.

One drawback with general logical sentences is that, because there are many different, logically equivalent sentences that describe the same belief state, repeated state checking in the graph search algorithm can require general theorem proving. For this
reason, we would like a canonical representation for sentences in which every belief state corresponds to exactly one sentence. One such representation uses a conjunction of literals ordered by proposition name—\( AtR \land CleanR \) is an example. This is just the standard state representation under the open-world assumption from Chapter 11. Not all logical sentences can be written in such form—for example, there is no way to represent \( AtL \lor CleanR \)—but many domains can be handled.

3. Knowledge propositions describing the agent's knowledge. (See also Section 7.7.) For the initial state, we have

\[
K(AtR) \land K(CleanR)
\]

Here, \( K \) stands for "knows that" and \( K(P) \) means that the agent knows that \( P \) is true. With knowledge propositions, we use the closed-world assumption—if a knowledge proposition does not appear in the list, it is assumed false. For example, \( \neg K(CleanL) \) and \( \neg K(\neg CleanL) \) are implicit in the sentence above, so it captures the fact that the agent is ignorant of the truth value of \( CleanL \).

It turns out that the second and third options are roughly equivalent, but we will use the third option, knowledge propositions, because it gives a more vivid description of sensing and because we already know how to write STRIPS descriptions with the closed-world assumption.

In both options, each proposition symbol can appear in one of three ways: positive, negative, or unknown. Therefore, there are exactly \( 3^n \) possible belief states that can be described this way. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are \( 2^n \) physical states, so there are \( 2^{2^n} \) belief states—far more than \( 3^n \), so options 2 and 3 are quite restricted as representations of belief states. This is currently believed to be inevitable, because any scheme capable of representing every possible belief state will require \( O(\log_2(2^{2^n})) = O(2^n) \) bits to represent each one in the worst case. Our simple schemes require only \( O(n) \) bits to represent each belief state, trading expressiveness for compactness. In particular, if an action occurs, one of whose preconditions is unknown, then the resulting belief state will not be exactly representable and the action outcome becomes unknown.

Now we need to decide how sensing works. There are two choices here. We can have automatic sensing, which means that at every time step the agent gets all the available percepts. The example in Figure 12.12 assumes automatic sensing of location and local dirt. Alternatively, we can insist on active sensing, which means that percepts are obtained only by executing specific sensory actions such as CheckDirt and CheckLocation. We will treat each kind of sensing in turn.

Let us now write an action description using knowledge propositions. Suppose the agent moves Left in the alternate-double-Murphy world with automatic local dirt sensing; according to the rules for that world, the agent might or might not leave dirt behind if the square was clean. As a physical effect, this would be disjunctive; but as a knowledge effect, it

---

11 The best-known canonical representation for a general propositional sentence is the binary decision diagram, or BDD (Bryant, 1992).
12 This is the same notation used for circuit-based agents in Chapter 7. Some authors use it to mean "knows whether \( P \) is true." Translating between the two representations is straightforward.
simply deletes the agent's knowledge of \( \text{CleanR} \). The agent will also know whether \( \text{CleanL} \) is true, one way or the other, because of local dirt sensing, and it will know that it is \( \text{AtL} \):

\[
\text{Action}(\text{Left}, \text{PRECOND: AtR}, \text{EFFECT: } \text{K}(\text{AtL}) \land \lnot \text{K}(\text{AtR}) \land \text{when} \text{ CleanR} \land \lnot \text{K}(\text{CleanR}) \land \\
\text{when} \text{ CleanL} \land \text{K}(\text{CleanL}) \land \text{when} \lnot \text{CleanL} \land \text{K}(\lnot \text{CleanL})).
\]

Notice that the preconditions and \textbf{when} conditions are plain propositions, not knowledge propositions. This is as it should be, because the outcomes of actions do depend on the actual world, but how do we check the truth of those conditions when all we have is the belief state? If the agent knows a proposition, say \( \text{K}(\text{AtR}) \), in the current belief state, then the proposition must be true in the current physical state, and indeed the action is applicable. If the agent doesn't know a proposition—for example, the \textbf{when} condition \text{CleanL}—then the belief state must include worlds in which \text{CleanL} is true and worlds in which \text{CleanL} is false. It is this that gives rise to multiple belief states resulting from the action. Thus, if the initial state is \( \text{K}(\text{AtL}) \land \text{K}(\text{CleanL}) \), then after the move \text{Left}, the two outcome belief states are \( \{ \text{K}(\text{AtL}), \text{K}(\text{CleanL}) \} \) and \( \{ \text{K}(\text{AtL}), \text{K}(\lnot \text{CleanL}) \} \). In both cases, the truth value of \text{CleanL} is known, so the \text{CleanL} test can be used in the plan.

With active sensing (as opposed to automatic sensing), the agent gets new percepts only by asking for them. Thus, after moving \text{Left}, the agent will not know whether the left-hand square is dirty, so the last two conditional effects no longer appear in the action description in Equation (12.2). To find out whether the square is dirty, the agent can \text{CheckDirt}:

\[
\text{Action}(\text{CheckDirt}, \text{EFFECT: } \text{when} \text{ AtL } \land \text{A Cleaning } \land \text{K}(\text{CleanL}) \land \text{when} \text{ AtL } \land \lnot \text{CleanL} \land \text{K}(\lnot \text{CleanL}) \land \\
\text{when} \text{ AtR } \land \text{A Cleaning } \land \text{K}(\text{CleanR}) \land \text{when} \text{ AtR } \land \lnot \text{CleanR} \land \text{K}(\lnot \text{CleanR})).
\]

It is easy to show that \text{Left} followed by \text{CheckDirt} in the active sensing setting results in the same two belief states as \text{Left} did in the automatic sensing setting. With active sensing, it is always the case that physical actions map a belief state into a single successor belief state. Multiple belief states can be introduced only by sensory actions, which provide specific knowledge and hence allow conditional tests to be used in plans.

We have described a general approach to conditional planning based on state-space \textit{AND–OR} search. The approach has proved to be quite effective on some test problems, but other problems are intractable. Theoretically, it can be shown that conditional planning belongs to a harder complexity class than classical planning. Recall that the definition of the class \textit{NP} is that a candidate solution can be checked to see whether it really is a solution in polynomial time. This is true for classical plans (at least, for those of polynomial size) so the problem of classical planning is in \textit{NP}. But in conditional planning a candidate must be checked to see whether, for all possible states, there exists \textit{some} path through the plan that satisfies the goal. Checking the "all/some" combination cannot be done in polynomial time, so conditional planning is harder than \textit{NP}. The only way out is to ignore some of the possible contingencies during the planning phase and to handle them only when they actually occur. This is the approach we pursue in the next section.
An execution monitoring agent checks its percepts to see whether everything is going according to plan. Murphy's law tells us that even the best-laid plans of mice, men, and conditional planning agents frequently fail. The problem is unbounded indeterminacy—some unanticipated circumstance will always arise for which the agent's actions descriptions are incorrect. Therefore, execution monitoring is a necessity in realistic environments. We will consider two kinds of execution monitoring: a simple, but weak form called action monitoring, whereby the agent checks the environment to verify that the next action will work, and a more complex but more effective form called plan monitoring, in which the agent verifies the entire remaining plan.

A replanning agent knows what to do when something unexpected happens: call a planner again to come up with a new plan to reach the goal. To avoid spending too much time planning, this is usually done by trying to repair the old plan—to find a way from the current unexpected state back onto the plan.

As an example, let us return to the double Murphy vacuum world in Figure 12.9. In this world, moving into a clean square sometimes deposits dirt in that square; but what if the agent doesn't know that or doesn't worry about it? Then it will come up with a very simple solution: \texttt{[Left]} If no dirt is dumped on arrival when the plan is actually executed, then the agent will detect the achievement of the goal. Otherwise, because the \texttt{CleanL} precondition of the implicit \texttt{Finish} step is not satisfied, the agent will generate a new plan: \texttt{[Suck]} Execution of this plan always succeeds.

Together, execution monitoring and replanning form a general strategy that can be applied to both fully and partially observable environments, and to a variety of planning representations including state-space, partial-order, and conditional plans. One simple approach to state-space planning is shown in Figure 12.13. The planning agent starts with a goal and creates an initial plan to achieve it. The agent then starts executing actions one by one. The replanning agent, unlike our other planning agents, keeps track of both the remaining unexecuted plan segment \texttt{plan} and the complete original plan \texttt{whole-plan}. It uses action monitoring: before carrying out the next action of \texttt{plan}, the agent examines its percepts to see whether any preconditions of the plan have unexpectedly become unsatisfied. If they have, the agent will try to get back on track by replanning a sequence of actions that should take it back to some point in the \texttt{whole-plan}.

Figure 12.14 provides a schematic illustration of the process. The replanner notices that the preconditions of the first action in \texttt{plan} are not satisfied by the current state. It then calls the planner to come up with a new subplan called \texttt{repair} that will get from the current situation to some state \texttt{s} on \texttt{whole-plan}. In this example, the state \texttt{s} happens to be one step back from the current remaining \texttt{plan}. (That is why we keep track of the whole plan, rather than just the remaining plan.) In general, we choose \texttt{s} to be as close as possible to the current state. The concatenation of \texttt{repair} and the portion of \texttt{whole-plan} from \texttt{s} onward, which we call \texttt{continuation}, makes up the new \texttt{plan}, and the agent is ready to resume execution.
function REPLANNING-AGENT( percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
  plan, a plan, initially []
  whole-plan, a plan, initially []
  goal, a goal

TELL(KB, MAKE-PERCEPT-SENTENCE( percept, t))
current ← STATE-DESCRIPTION(KB, t)
if plan = [] then
  whole-plan ← plan.append(PLANNER(current, goal, KB))
if PRECONDITIONS(FIRST(plan)) not currently true in KB then
  candidates ← SORT(whole-plan, ordered by distance to current)
  find state s in candidates such that
  failure ≠ repair ← PLANNER(current, s, KB)
  continuation ← the tail of whole-plan starting at s
  whole-plan ← plan.append(continuation)
return POP(plan)

Figure 12.13  An agent that does action monitoring and replanning. It uses a complete state-space planning algorithm called PLANNER as a subroutine. If the preconditions of the next action are not met, the agent loops through the possible points p in whole-plan, trying to find one that PLANNER can plan a path to. This path is called repair. If PLANNER succeeds in finding a repair, the agent appends repair and the tail of the plan after p, to create the new plan. The agent then returns the first step in the plan.

Figure 12.14  Before execution, the planner comes up with a plan, here called whole-plan, to get from S to G. The agent executes the plan until the point marked E. Before executing the remaining plan, it checks preconditions as usual and finds that it is actually in state O rather than state E. It then calls its planning algorithm to come up with repair, which is a plan to get from O to some point P on the original whole-plan. The new plan now becomes the concatenation of repair and continuation (the resumption of the original whole-plan).
Now let's return to the example problem of achieving a chair and table of matching color, this time via replanning. We'll assume a fully observable environment. In the initial state the chair is blue, the table is green, and there is a can of blue paint and a can of red paint.

That gives us the following problem definition:

\[
\begin{align*}
\text{Init} & : \text{Color}(\text{Chair, Blue}) \land \text{Color}(\text{Table, Green}) \\
& \land \text{ContainsColor}(\text{BC, Blue}) \land \text{PaintCan(BC)}) \\
& \land \text{ContainsColor}(\text{RC, Red}) \land \text{PaintCan(RC)}) \\
\text{Goal} & : \text{Color}(\text{Chair, x}) \land \text{Color}(\text{Table, x}) \\
\text{Action} & : \text{Paint}(\text{object, color}) \\
& \quad \text{PRECOND} : \text{HavePaint(\text{color})} \\
& \quad \text{EFFECT} : \text{Color}(\text{object, color}) \\
\text{Action} & : \text{Open}(\text{can}) \\
& \quad \text{PRECOND} : \text{PaintCan(\text{\textit{can}})} \land \text{ContainsColor(\text{can, color})} \\
& \quad \text{EFFECT} : \text{HavePaint(\text{color})} \\
\end{align*}
\]

The agent's \textit{PLANNER} should come up with the following plan:

\[(\text{Start} \text{Open(BC)}; \text{Paint(Table, Blue)}; \text{Finish})\]

Now the agent is ready to execute the plan. Assume that all goes well as the agent opens the blue paint and applies it to the table. The agents from previous sections would declare victory at this point, having completed the steps in the plan. But the execution monitoring agent must first check the precondition of the \textit{Finish} step, which says that the two pieces must have the same color. Suppose the agent perceives that they do not have the same color, because it missed a spot of green on the table. The agent then needs to figure out a position in \textit{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 \textit{Paint} action, so the agent chooses the empty sequence for repair and makes its \textit{plan} be the same \{(\text{Paint, Finish})\} sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the \textit{Paint} action is retried. This behavior will loop until the table 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.

Action monitoring is a very simple method of execution monitoring but it can sometimes lead to less than intelligent behavior. For example, suppose that the agent constructs a plan to solve the painting problem by painting the chair and table red. Then it opens the can of red paint and finds that there is only enough paint for the chair. Action monitoring would not detect failure until after the chair has been painted, at which point \textit{HavePaint(\text{Red})} becomes false. What we really need to do is detect failure whenever the state is such that the remaining plan no longer works. \textit{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. In some cases, it can rescue the agent from disaster when the doomed plan would have led to a dead end from which the goal would be unachievable.

\[\text{Plan monitoring makes our agent smarter than a dung beetle. (See page 37.) Our agent would notice that the} \]
It is relatively straightforward to modify a planning algorithm so that it annotates the plan at each point with the preconditions for success of the remaining plan. If we extend plan monitoring to check whether the current state satisfies the plan preconditions at any future point, rather than just the current point, then plan monitoring will also be able to take advantage of serendipity—that is, 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.

So far, we have described monitoring and replanning in fully observable environments. Things can become much more complicated when the environment is partially observable. First, things can go wrong without the agent's being able to detect it. Second, "checking preconditions" could require the execution of sensing actions, which have to be planned for—either at planning time, which takes us back to conditional planning, or at execution time. In the worst case, the execution of a sensing action could require a complex plan that itself requires monitoring and hence further sensing actions, and so on. If the agent insists on checking every precondition, it might never get around to actually doing anything. The agent should prefer to check those variables that are important, have a good chance of going wrong, and are not too expensive to perceive. This allows the agent to respond appropriately to important threats, but not waste time checking to see whether the sky is falling.

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, even with unbounded indeterminacy?" then the answer is no, because the agent could inadvertently arrive at a dead end, as described for online search in Section 4.5. For example, the vacuum agent might not know that its batteries can run out. Let's rule out dead ends; that is, let's assume that the agent can construct a plan to reach the goal from any state in the environment. If we 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. The replanning agent therefore has capabilities analogous to those of the conditional planning agent. In fact, we can modify a conditional planner so that it constructs only a partial solution plan that includes steps of the form "if <test> then plan-A else replan." Under the assumptions we have made, such a plan can be a correct solution to the original problem; it might also be much cheaper to construct than a full conditional plan.

Trouble occurs when the agent's repeated attempts to reach the goal are futile—when they are blocked by some precondition or effect that it doesn't know about. For example, if the agent has the wrong card key to its hotel room, no amount of inserting and removing it is going to open the door. One solution is to choose randomly from among the set of possible repair plans, rather than trying the same one each time. In this case, the repair plan of going to the front desk and getting a card key to the room would be a useful alternative. Given that the agent might not be able to distinguish between the truly nondeterministic case and the futile case, some variation in repairs is a good idea in general.

---

14 Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp. (See page 37.)
Another solution to the problem of incorrect action descriptions is learning. After a few tries, a learning agent should be able to modify the action description that says that the key opens the door. At that point, the replanner will automatically come up with an alternative plan, such as getting a new key. This kind of learning is described in Chapter 21.

Even with all these potential improvements, the replanning agent still has a few shortcomings. It cannot perform in real-time environments, and there is no bound on the amount of time it will spend replanning and thus no bound on the time it takes to decide on an action. Also, it cannot formulate new goals of its own or accept new goals in addition to its current goals, so it cannot be a long-lived agent in a complex environment. These shortcomings will be addressed in the next section.

12.6 CONTINUOUS PLANNING

In this section, we design an agent that persists indefinitely in an environment. Thus it is not a "problem solver" that is given a single goal and then plans and acts until the goal is achieved; rather, it lives through a series of ever-changing goal formulation, planning, and acting phases. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process in a continuous planning agent.

The agent is thought of as always being part of the way through executing a plan—the grand plan of living its life. Its activities include executing some steps of the plan that are ready to be executed, refining the plan to satisfy open preconditions or resolve conflicts, and modifying the plan in the light of additional information obtained during execution. Obviously, when it first formulates a new goal, the agent will have no actions ready to execute, so it will spend a while generating a partial plan. It is quite possible, however, for the agent to begin execution before the plan is complete, especially when it has independent subgoals to achieve. The continuous planning agent monitors the world continuously, updating its world model from new percepts even if its deliberations are still continuing.

We will first go through an example and then describe the agent program, which we will call CONTINUOUS-POP-AGENT because it uses partial-order plans to represent its intended activities. To simplify the presentation, we will assume a fully observable environment. The same techniques can be extended to the partially observable case.

The example we will use is a problem from the blocks world domain (Section 11.1). The start state is shown in Figure 12.15(a). The action we will need is Move(x, y), which moves block x onto block y, provided that both are clear. Its action schema is

\[
\text{Action(Move}(x, y)\text{),}
\]

\[
\text{PRECOND: Clear}(x) \land \text{Clear}(y) \land \text{On}(x, z),
\]

\[
\text{EFFECT: On}(x, y) \land \text{Clear}(z) \land \neg \text{On}(z, x) \land \neg \text{Clear}(y).
\]

The agent first needs to formulate a goal for itself. We won't discuss goal formulation here, but instead we will assume that somehow the agent was told (or decided on its own) to achieve the goal On(C, D) \land On(D, B). The agent starts planning for this goal. Unlike
all our other agents, which would shut off their percepts until the planner returns a complete solution to this problem, the continuous planning agent builds the plan incrementally, with each increment taking a bounded amount of time. After each increment, the agent returns *NoOp* as its action and checks its percepts again. We assume that the percepts don’t change and the agent quickly constructs the plan shown in Figure 12.16. Notice that although the preconditions of both actions are satisfied by *Start*, there is an ordering constraint putting *Move*(\( D \rightarrow B \)) before *Move*(\( C \rightarrow D \)). This is needed to ensure that *Clear*(\( D \)) remains true until *Move*(\( D \rightarrow B \)) is completed. Throughout the continuous planning process, *Start* is always used as the label for the current state. The agent updates the state after each action.

The plan is now ready to be executed, but before the agent can take action, nature intervenes. An external agent (perhaps the agent’s teacher getting impatient) moves \( D \) onto \( B \) and the world is now in the state shown in Figure 12.15(b). The agent perceives this, recognizes that *Clear*(\( B \)) and *On*(\( D \rightarrow G \)) are no longer true in the current state, and updates its model of the current state accordingly. The causal links that were supplying the preconditions *Clear*(\( B \)) and *On*(\( D \rightarrow G \)) for the *Move*(\( D \rightarrow B \)) action become invalid and must be removed from the plan. The new plan is shown in Figure 12.17. At all times, *Start* represents the current state, so this *Start* is different from the one in the previous figure. Notice that the plan is now incomplete: two of the preconditions for *Move*(\( D \rightarrow B \)) are open, and its precondition *On*(\( D \rightarrow y \)) is now uninstantiated, because there is no longer any reason to assume the that move will be from \( G \).
Now the agent can take advantage of the "helpful" interference by noticing that the causal link $Move(D, B) \xrightarrow{On(D, B)} Finish$ can be replaced by a direct link from Start to Finish. This process is called **extending** a causal link and is done whenever a condition can be supplied by an earlier step instead of a later one without causing a new conflict.

Once the old causal link from $Move(D, B)$ to Finish is removed, $Move(D, B)$ no longer supplies any causal links at all. It is now a **redundant step**. All redundant steps, and any links supplying them, are dropped from the plan. This gives the plan in Figure 12.18.

Now the step $Move(C, D)$ is ready to be executed, because all of its preconditions are satisfied by the Start step, no other steps are necessarily before it, and it does not conflict with any other link in the plan. The step is removed from the plan and executed. Unfortunately, the agent is clumsy and drops $C$ onto A instead of D, giving the state shown in Figure 12.15(c). The new plan state is shown in Figure 12.19. Notice that although there are now no actions in the plan, there is still an open condition for the Finish step.

---

**Figure 12.17** After someone else moves $D$ onto $B$, the unsupported links supplying $Clear(B)$ and $On(D, G)$ are dropped, producing this plan.

**Figure 12.18** The link supplied by $Move(D, B)$ has been replaced by one from Start, and the now-redundant step $Move(D, B)$ has been dropped.

**Figure 12.19** After $Move(C, D)$ is executed and removed from the plan, the effects of the Start step reflect the fact that $C$ ended up on $A$ instead of the intended $D$. The goal precondition $On(C, D)$ is still open.
The agent decides to plan for the open condition. Once again, \( Move(C, D) \) will satisfy the goal condition. Its preconditions are satisfied by new causal links from the Start step. The new plan appears in Figure 12.20.

Once again, \( Move(C, D) \) is ready for execution. This time it works, resulting in the goal state shown in Figure 12.15(d). Once the step is dropped from the plan, the goal condition \( On(C, D) \) becomes open again. Because the Start step is updated to reflect the new world state, however, the goal condition can be satisfied immediately by a link from the Start step. This is the normal course of events when an action is successful. The final plan state is shown in Figure 12.21. Because all the goal conditions are satisfied by the Start step and there are no remaining actions, the agent is now free to remove the goals from Finish and formulate a new goal.

From this example, we can see that continuous planning is quite similar to partial-order planning. On each iteration, the algorithm finds something about the plan that needs fixing—a so-called plan flaw—and fixes it. The POP algorithm can be seen as a flaw-removal algorithm where the two flaws are open preconditions and causal conflicts. The continuous planning agent, on the other hand, addresses a much broader range of flaws:

- **Missing goal:** The agent can decide to add a new goal or goals to the Finish state. (Under continuous planning, it might make more sense to change the name of Finish to Infinity, and of Start to Current, but we will stick with tradition.)
- **Open precondition:** Add a causal link to an open precondition, choosing either a new or an existing action (as in POP).
- **Causal Conflict:** Given a causal link \( A \xrightarrow{P} B \) and an action \( C \) with effect \( \neg p \), choose an ordering constraint or variable constraint to resolve the conflict (as in POP).
• **Unsupported link:** If there is a causal link \( \text{Start} \xrightarrow{p} \text{A} \) where \( p \) is no longer true in \( \text{Start} \), then remove the link. (This prevents us from executing an action whose preconditions are false.)

• **Redundant action:** If an action \( \text{A} \) supplies no causal links, remove it and its links. (This allows us to take advantage of serendipitous events.)

• **Unexecuted action:** If an action \( \text{A} \) (other than Finish) has its preconditions satisfied in \( \text{Start} \), has no other actions (besides \( \text{Start} \)) ordered before it, and conflicts with no causal links, then remove \( \text{A} \) and its causal links and return it as the action to be executed.

• **Unnecessary historical goal:** If there are no open preconditions and no actions in the plan (so that all causal links go directly from \( \text{Start} \) to \( \text{Finish} \)), then we have achieved the current goal set. Remove the goals and the links to them to allow for new goals.

The **Continuous-Pop-Agent** is shown in Figure 12.22. It has a cycle of "perceive, remove flaw, act." It keeps a persistent plan in its knowledge base, and on each turn it removes one flaw from the plan. It then takes an action (although often the action will be \( \text{NoOp} \)) and repeats the loop. This agent can handle many of the problems listed in the discussion of the replanning agent on page 445. In particular, it can act in real time, it handles serendipity, it can formulate its own goals, and it can handle unexpected events that affect future plans.

```plaintext
function Continuous-Pop-Agent(percet) returns an action
    static: plan, a plan, initially with just Start, Finish
    action ← NoOp (the default)
    EFFECTS[Start] = UPDATE(EFFECTS[Start], percept)
    REMOVE-FLAW(plan) //possibly updating action
    return action

Figure 12.22  Continuous-Pop-Agent, a continuous partial-order planning agent. After receiving a percept, the agent removes a flaw from its constantly updated plan and then returns an action. Often it will take many steps of flaw-removal planning, during which it returns NoOp, before it is ready to take a real action.
```

### 12.7 MultiAgent Planning

So far we have dealt with **single-agent environments**, in which our agent is alone. When there are other agents in the environment, our agent could simply include them in its model of the environment, without changing its basic algorithms. In many cases, however, that would lead to poor performance because dealing with other agents is not the same as dealing with nature. In particular, nature is (one assumes) indifferent to the agent's intentions,\(^\text{15}\) whereas other agents are not. This section introduces multiagent planning to handle these issues.

\(^{15}\) Residents of the United Kingdom, where the mere act of planning a picnic guarantees rain, might disagree.
We saw in Chapter 2 that multiagent environments can be cooperative or competitive. We will begin with a simple cooperative example: team planning in doubles tennis. Plans can be constructed that specify actions for both players on the team; we will describe techniques for constructing such plans efficiently. Efficient plan construction is useful, but does not guarantee success; the agents have to agree to use the same plan! This requires some form of coordination, possibly achieved by communication.

Cooperation: Joint goals and plans

Two agents playing on a doubles tennis team have the joint goal of winning the match, which gives rise to various subgoals. Let's suppose that at one point in the game, they have the joint goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net. We can represent this notion as a multiagent planning problem, as shown in Figure 12.23.

This notation introduces two new features. First, \( \text{Agents}(A, B) \) declares that there are two agents, A and B, who are participating in the plan. (For this problem the opposing players are not considered agents.) Second, each action explicitly mentions the agent as a parameter, because we need to keep track of which agent does what.

A solution to a multiagent planning problem is a joint plan consisting of actions for each agent. A joint plan is a solution if the goal will be achieved when each agent performs its assigned actions. The following plan is a solution to the tennis problem:

\[
\text{PLAN 1:} \\
A : \ [\text{Go}(A, \text{[RightBaseline]}), \text{Hit}(A, \text{Ball})] \\
B : \ [\text{NoOp}(B), \text{NoOp}(B)].
\]

If both agents have the same knowledge base, and if this is the only solution, then everything would be fine; the agents could each determine the solution and then jointly execute it. Unfortunately for the agents (and we will soon see why it's unfortunate), there is another plan...

---

**Figure 12.23** The doubles tennis problem. Two agents are playing together and can be in one of four locations: \([\text{LeftBaseline}], [\text{RightBaseline}], [\text{LeftNet}]\) and \([\text{RightNet}]\). The ball can be returned if exactly one player is in the right place.
that satisfies the goal just as well as the first:

\[
\text{PLAN 2:} \\
A : \langle \text{Go}(A, \{\text{Left}, \text{Net}\}) \text{NoOp}(A) \rangle \\
B : \langle \text{Go}(B, \{\text{Right}, \text{baseline}\}) \text{Hit}(B, \text{Ball}) \rangle
\]

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 probably collide with each other; no one returns the ball and the net may remain uncovered. Hence, the existence of correct joint plans does not mean that the goal will be achieved. The agents need a mechanism for coordination to reach the same joint plan; moreover, it must be common knowledge (see Chapter 10) among the agents that some particular joint plan will be executed.

**Multibody planning**

This section concentrates on the construction of correct joint plans, deferring the coordination issue for the time being. We call this multibody planning; it is essentially the planning problem faced by a single centralized agent that can dictate actions to each of several physical entities. In the truly multiagent case, it enables each agent to figure out what the possible joint plans are that would succeed if executed jointly.

Our approach to multibody planning will be based on partial-order planning, as described in Section 11.3. We will assume full observability, to keep things simple. There is one additional issue that doesn't arise in the single-agent case: the environment is no longer truly static, because other agents could act while any particular agent is deliberating. Therefore, we need to be concerned about synchronization. For simplicity, we will assume that each action takes the same amount of time and that actions at each point in the joint plan are simultaneous.

At any point in time, each agent is executing exactly one action (perhaps including \text{NoOp}). This set of concurrent actions is called a joint action. For example, a joint action in the tennis domain (page 450) with two agents \(A\) and \(B\) is \(\langle \text{NoOp}(A), \text{Hit}(B, \text{Ball}) \rangle\). A joint plan consists of a partially ordered graph of joint actions. For example, Plan 2 for the tennis problem can be represented as this sequence of joint actions:

\[
\langle \text{Go}(A, \{\text{Left}, \text{Net}\}) \rangle \langle \text{Go}(B, \{\text{Right}, \text{baseline}\}) \rangle \\
\langle \text{NoOp}(A), \text{Hit}(B, \text{Ball}) \rangle
\]

We could do planning using the regular POP algorithm, applied to the set of all possible joint actions. The only problem is the size of this set: with 10 actions and 5 agents we get \(10^5\) joint actions. It would be tedious to specify the preconditions and effects of each action correctly, and inefficient to do planning with such a large set.

An alternative is to define joint actions implicitly, by describing how each individual action interacts with other possible actions. This will be simpler, because most actions are independent of most others; we need list only the few actions that actually interact. We can do that by augmenting the usual STRIPS or ADL action descriptions with one new feature: a concurrent action list. This is similar to the precondition of an action description except that rather than describing state variables, it describes actions that must or must not be executed.
concurrently. For example, the *Hit* action could be described as follows:

\[
\text{Action}(\text{Hit}(A,Ball)), \\
\text{CONCURRENT}: \neg \text{Hit}(B,Ball) \\
\text{PRECOND}: \text{Approaching}(\text{Ball},[x,y]) \land \text{At}(A,[x,y]) \\
\text{EFFECT}: \text{Returned}(\text{Ball}).
\]

Here, we have the prohibited-concurrency constraint that, during the execution of the *Hit* action, there can be no other *Hit* action by another agent. We can also require concurrent action, for example when two agents are needed to carry a cooler full of beverages to the tennis court. The description for this action says that agent *A* cannot execute a *Carry* action unless there is another agent *B* who is simultaneously executing a *Carry* of the same cooler:

\[
\text{Action}(\text{Carry}(A,\text{cooler,here,there})), \\
\text{CONCURRENT}: \text{Carry}(B,\text{cooler,here,there}) \\
\text{PRECOND}: \text{At}(A,\text{here}) \land \text{At}(\text{cooler,here}) \land \neg \text{At}(A,\text{here}) \land \neg \text{At}(\text{cooler,here}) \\
\text{EFFECT}: \text{At}(A,\text{there}) \land \text{At}(\text{cooler,there}) \land \neg \text{At}(A,\text{here}) \land \neg \text{At}(\text{cooler,here}).
\]

With this representation, it is possible to create a planner that is very close to the POP partial-order planner. There are three differences:

1. In addition to the temporal ordering relation *A* \(\prec\) *B*, we allow *A* \(\equiv\) *B* and *A* \(\preceq\) *B*, meaning "concurrent" and "before or concurrent," respectively.
2. When a new action has required concurrent actions, we must instantiate those actions, using new or existing actions in the plan.
3. Prohibited concurrent actions are an additional source of constraints. Each constraint must be resolved by constraining conflicting actions to be before or after.

This representation gives us the equivalent of POP for multibody domains. 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.

**Coordination mechanisms**

The simplest method by which a group of agents can ensure agreement on a joint plan is to adopt a *convention* prior to engaging in joint activity. A convention is any constraint on the selection of joint plans, beyond the basic constraint that the joint plan must work if all agents adopt it. For example, the convention "stick to your side of the court" would cause the doubles partners to select plan 2, whereas the convention "one player always stays at the net" would lead them to plan 1. Some conventions, such as driving on the proper side of the road, are so widely adopted that they are considered *social laws*. Human languages can also be viewed as conventions.

The conventions in the preceding paragraph are domain-specific and can be implemented by constraining the action descriptions to rule out violations of the convention. A more general approach is to use domain-independent conventions. For example, if each agent runs the same multibody planning algorithm with the same inputs, it can follow the convention of executing the first feasible joint plan found, confident that the other agents will come
Conventions can also arise through evolutionary processes. For example, colonies of social insects execute very elaborate joint plans, which are facilitated by the common genetic makeup of the individuals in the colony. Conformity can also be enforced by the fact that deviation from conventions reduces evolutionary fitness, so that any feasible joint plan can become a stable equilibrium. 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 all individuals speak the same language. One final example appears in the flocking behavior of birds. We can obtain a reasonable simulation if each bird (sometimes called a boid or boid) executes the following three rules:

1. Separation: Steer away from neighbors when you start to get too close.
2. Cohesion: Steer towards the average position of the neighbors.
3. Alignment: Steer towards the average orientation (heading) of the neighbors.

If all the birds execute the same policy, the flock exhibits the emergent behavior of flying as a pseudo-rigid body with roughly constant density that does not disperse over time. As with insects, there is no need for each agent to possess the joint plan that models the actions of other agents.

Typically, conventions are adopted to cover a universe of individual multiagent planning problems, rather than being developed anew for each problem. This can lead to inflexibility and breakdown, as can be seen sometimes in doubles tennis when the ball is roughly equidistant between the two partners. In the absence of an applicable convention, agents can use communication to achieve common knowledge of a feasible joint plan. For example, a doubles 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. In our tennis problem, 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.

The burden for ensuring that the agents arrive at a successful joint plan can be placed either on the agent designers or on the agents themselves. In the former case, before the agents begin to plan, the agent designer should prove that the agents' policies and strategies will be successful. The agents themselves can be reactive if that works for the environment they exist in, and they need not have explicit models about the other agents. In the latter case, the agents are deliberative; they must prove or otherwise demonstrate that their plan will be effective, taking the other agents' reasoning into account. For example, in an environment with two logical agents A and B, they could both have the following definition:

\[ \forall p, s \text{ Feasible}(p, s) \iff \text{CommonKnowledge}(\{A, B\}. \text{Achieves}(p, s, \text{Goal})) \]

This says that in any situation s, the plan p is a feasible joint plan in that situation if it is common knowledge among the agents that p will achieve the goal. We need further axioms
to establish common knowledge of a joint intention to execute a particular joint plan; only then can agents begin to act.

**Competition**

Not all multiagent environments involve cooperative agents. Agents with conflicting utility functions are in competition with each other. One example of this is two-player zero-sum games, such as chess. We saw in Chapter 6 that a chess-playing agent needs to consider the opponent's possible moves for several steps into the future. That is, an agent in a competitive environment must (a) recognize that there are other agents, (b) compute some of the other agent's possible plans, (c) compute how the other agent's plans interact with its own plans, and (d) decide on the best action in view of these interactions. So competition, like cooperation, requires a model of the other agent's plans. On the other hand, there is no commitment to a joint plan in a competitive environment.

Section 12.4 drew the analogy between games and conditional planning problems. The conditional planning algorithm in Figure 12.10 constructs plans that work under worst-case assumptions about the environment, so it can be applied in competitive situations where the agent is concerned only with success and failure. When the agent and its opponents are concerned about the cost of a plan, then minimax is appropriate. As yet, there has been little work on combining minimax with methods, such as POP and HTN planning, that go beyond the state-space search model used in Chapter 6. We will return to the question of competition in Section 17.6, which covers game theory.

**12.8 Summary**

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

- 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 decomposition rules. This makes it feasible to 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.

- Incomplete information can be dealt with by planning to use sensing actions to obtain the information needed. Conditional 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 sensorless and conditional plans can be constructed by search in the space of belief states.

- Incorrect information results in unsatisfied preconditions for actions and plans. Execution monitoring detects violations of the preconditions for successful completion of the plan.
- A replanning agent uses execution monitoring and splices in repairs as needed.
- A continuous planning agent creates new goals as it goes and reacts in real time.
- Multiagent planning is necessary when there are other agents in the environment with which to cooperate, compete, or coordinate. Multibody planning constructs joint plans, using an efficient decomposition of joint action descriptions, but must be augmented with some form of coordination if two cooperative agents are to agree on which joint plan to execute.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with continuous time was first dealt with by DEVISER (Vere, 1983). The issue of systematic representation of time in plans was addressed 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. 0-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, 0-PLAN has been applied to software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars. 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 (Descotte 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.

After an initial flurry of theoretical work in the late 1980s, temporal planning made a comeback recently, when new algorithms and increased processing power made it feasible to attack practical applications. 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).

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-ERSI (Fuchs et al., 1990), both based on 0-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 (Muscetola et al., 1998) became the first autonomous planner-scheduler to control a spacecraft when it flew onboard the Deep Space One probe in 1999. The literature on job-shop scheduling in operations research is surveyed by Vaessens et al. (1996); theoretical results are presented by Martin and Shmoys (1996).

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 0-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. Other authors (Ingerson and Steel, 1988; Young et al., 1994; Barrett and Weld, 1994; Kambhampati et al., 1998) have proposed the hybrid approach taken in this chapter, in which decompositions are just another form of refinement that can be used in partial-order 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.

The unpredictability and partial observability of real environments was recognized early on in robotics projects that used planning techniques, including Shakey (Fikes et al., 1972) and FREDDY (Michie, 1974). The problem received more attention after the publication of McDermott’s (1978a) influential article, Planning and Acting.

Early planners, which lacked conditionals and loops, did not explicitly recognize the concept of conditional planning; but nevertheless they sometimes resorted to a coercive style in response to environmental uncertainty. Sacerdoti’s NOAH used coercion in its solution to the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state. 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. We describe this idea in the context of robotics. (See Figure 25.17.)
Goldman and Boddy (1996) introduced the term conformant planning for sensorless planners that handle uncertainty by coercing the world into known states, 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, the fastest belief-state conformant planners, such as HSCP (Bertoli et al., 2001a), use binary decision diagrams (BDDs) (Bryant, 1992) to represent belief states and are up to five orders of magnitude faster than CGP.

WARPLAN-C (Warren, 1976), a variant of WARPLAN, was one of the earliest planners to use conditional actions. Olawski and Gini (1990) lay out the major issues involved in conditional planning.

The conditional planning approach described in the chapter is based on 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 a BDD-based approach that constructs conditional plans with loops. C-BURIDANDraper et al., 1994) handles conditional planning for actions with probabilistic outcomes, a problem also addressed under the heading of POMDPs (Chapter 17).

There is a close relation between conditional planning and automated program synthesis; a number of references appear in Chapter 9. The two fields have been pursued separately, because of the enormous difference in cost between execution of machine instructions and execution of actions by robot vehicles or manipulators. Linden (1991) attempts explicit cross-fertilization between the two fields.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for domains involving uncertainty. Search-based techniques led to search in belief space (Bonet and Geffner, 2000); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 1999) and to planning using quantified Boolean logic (Rintanen, 1999); partial order planning led to UWL (Etzioni et al., 1992). CNLP (Peot and Smith, 1992), and CASSANDRA (Pryor and Collins, 1996). GRAPHPLAN led to Sensory Graphplan or SGP (Weld et al., 1998), but a full probabilistic GRAPHPLAN has yet to be developed.

The earliest major treatment of execution monitoring was PLANEX (Fikes et al., 1972), which worked with the STRIPS planner to control the robot Shakey. PLANEX used triangle tables—essentially an efficient storage mechanism for the plan preconditions at each point in the plan—to allow recovery from partial execution failure without complete replanning. Shakey's model of execution is discussed further in Chapter 25. The NASI 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. It used theorem proving to reason about these complex actions.

SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1991, 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 and job-shop scheduling for an Australian beer factory. Another study used SiPE to plan the construction of multistory buildings, one of the most complex domains ever tackled by a planner.

IPEM (Integrated Planning, Execution, and Monitoring) (Ambros-Ingerson and Steel, 1988) was the first system to integrate partial-order planning and execution to yield a continuous planning agent. Our CONTINUOUS-POP-AGENT combines ideas from IPEM, the PUCCINI planner (Golden, 1998), and the CYPRESS system (Wilkins et al., 1995).

In the mid-1980s, it was believed by some that partial-order planning and related techniques could never run fast enough to generate effective behavior for an agent in the real world (Agre and Chapman, 1987). Instead, reactive planning systems were proposed; in their basic form, these are reflex agents, possibly with internal state, that can be implemented with any of a variety of representations for condition–action rules. Brooks's (1986) subsumption architecture (see Chapters 7 and 25) used layered finite-state machines in legged and wheeled robots to control their locomotion and avoid obstacles. Pengei (Agre and Chapman, 1987) was able to play a (fully observable) video game using Boolean circuits combined with a "visual" representation of current goals and the agent's internal state.

"Universal plans" (Schoppers, 1987) 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. A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Ginsberg (1989) made a spirited attack on universal plans, including intractability results for some formulations of the reactive planning problem. Schoppers (1989) made an equally spirited reply.

As is often the case, a hybrid approach resolves the controversy. Using well-designed hierarchies, HTN planners, such as PRS (Georgeff and Lansky, 1987) and RAP (Firby, 1996), as well as continuous planning agents, can achieve reactive response times and complex long-range planning behavior in many problem domains.

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) provided a formalization of multiagent planning in first-order logic, while Pednault (1986) gave 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). Our presentation of multibody partial-order planning is based on the work of Boutilier and Brafman (2001).

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 dissolution, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. (Weiss, 1999) is a book-length overview of multiagent systems.

The boid model on page 453 is due to Reynolds (1987), who won an Academy Award for its application to flocks of bats and swarms of penguins in Batman Returns.
12.1 Examine carefully the representation of time and resources in Section 12.1.

a. Why is it a good idea to have $\text{Duration}(d)$ be an effect of an action, rather than having a separate field in the action of the form $\text{Duration: } d$? (Hint: Consider conditional effects and disjunctive effects.)

b. Why is $\text{Resource: } m$ a separate field in the action, rather than being an effect?

12.2 A consumable resource is a resource that is (partially) used up by an action. For example, attaching engines to cars requires screws. The screws, once used, are not available for other attachments.

a. Explain how to modify the representation in Figure 12.3 so that there are 100 screws initially, engine $E_1$ requires 40 screws, and engine $E_2$ requires 50 screws. The $+$ and $-$ function symbols may be used in effect literals for resources.

b. Explain how the definition of conflict between causal links and actions in partial-order planning must be modified to handle consumable resources.

c. Some actions—for example, resuppling the factory with screws or refueling a car—can increase the availability of resources. A resource is monotonically non-increasing if no action increases it. Explain how to use this property to prune the search space.

12.3 Give decompositions for the $\text{HireBuilder}$ and $\text{GetPermit}$ steps in Figure 12.7, and show how the decomposed subplans connect into the overall plan.

12.4 Give an example in the house-building domain of two abstract subplans that cannot be merged into a consistent plan without sharing steps. (Hint: Places where two physical parts of the house come together are also places where two subplans tend to interact.)

12.5 Some people say an advantage of HTN planning is that it can solve problems like “take a round trip from Los Angeles to New York and back” that are hard to express in non-HTN notations because the start and goal states would be the same ($\text{At(LA)}$). Can you think of a way to represent and solve this problem without HTNs?

12.6 Show how a standard STRIPS action description can be rewritten as an HTN decomposition, using the notation $\text{Achieve}(p)$ to denote the activity of achieving the condition p.

12.7 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 copies the contents of a memory location, while 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 operator 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 using a temporary variable.

12.8 Consider the following argument: In a framework that allows uncertain initial states, **disjunctive effects** are just a notational convenience, not a source of additional representational power. For any action schema \( a \) with disjunctive effect \( P \lor Q \), we could always replace it with the conditional effects \( \text{when } R: P \land \neg \text{when } R: Q \), which in turn can be reduced to two regular actions. The proposition \( R \) stands for a random proposition that is unknown in the initial state and for which there are no sensing actions. Is this argument correct? Consider separately two cases, one in which only one instance of action schema \( a \) is in the plan, the other in which more than one instance is.

12.9 Why can’t conditional planning deal with unbounded indeterminacy?

12.10 In the blocks world we were forced to introduce two STRIPS actions, **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.

12.11 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.

12.12 Write out the full description of Suck for the double Murphy vacuum cleaner that sometimes deposits dirt when it moves to a clean destination square and sometimes deposits dirt if Suck is applied to a clean square.

12.13 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.

12.14 The following quotes are from the backs of shampoo bottles. Identify each as an unconditional, conditional, or execution monitoring plan. (a) "Lather. Rinse. Repeat." (b) "Apply shampoo to scalp and let it remain for several minutes. Rinse and repeat if necessary." (c) "See a doctor if problems persist."

12.15 The AND-OR-GRAPH-SEARCH algorithm in Figure 12.10 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were to store every visited state and check against that list. (See GRAPH-SEARCH in Figure 3.19 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (Hint: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels to avoid having multiple copies of subplans.

12.16 Explain precisely how to modify the AND-OR-GRAPH-SEARCH algorithm to generate a cyclic plan if no acyclic plan exists. You will need to deal with three issues: labeling
the plan steps so that a cyclic plan can point back to an earlier part of the plan, modifying OR-SEARCH so that it continues to look for acyclic plans after finding a cyclic plan, and augmenting the plan representation to indicate whether a plan is cyclic. Show how your algorithm works on (a) the triple Murphy vacuum world, and (b) the alternate double Murphy vacuum world. You might wish to use a computer implementation to check your results. Can the plan for case (b) be written using standard loop syntax?

12.17 Specify in full the belief state update procedure for partially observable environments. That is, the method for computing the new belief state representation (as a list of knowledge propositions) from the current belief-state representation and an action description with conditional effects.

12.18 Write action descriptions, analogous to Equation (12.2), for the Right and Suck actions. Also write a description for CheckLocation, analogous to Equation (12.3). Repeat using the alternative set of propositions from Exercise 12.11.

12.19 Look at the list on page 445 of things that the replanning agent can't do. Sketch an algorithm that can handle one or more of them.

12.20 Consider the following problem: A patient arrives at the doctor's office with symptoms that could have been caused either by dehydration or by disease D (but not both). There are two possible actions: Drink, which unconditionally cures dehydration, and Medicate, which cures disease D, but has an undesirable side-effect if taken when the patient is dehydrated. Write the problem description in PDDL, and diagram a sensorless plan that solves the problem, enumerating all relevant possible worlds.

12.21 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.