11 PLANNING

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

PLANNING AGENT

In this chapter, we introduce the basic ideas involved in planning systems. We begin by specifying a simple **planning agent** that is very similar to a problem-solving agent (Chapter 3) in that it constructs plans that achieve its goals, and then executes them. Section 11.2 explains the limitations of the problem-solving approach, and motivates the design of planning systems. The planning agent differs from a problem-solving agent in its representations of goals, states, and actions, as described in Section 11.4. The use of explicit, logical representations enables the planner to direct its deliberations much more sensibly. The planning agent also differs in the way it represents and searches for solutions. The remainder of the chapter describes in detail the basic **partial-order planning** algorithm, which searches through the space of plans to find one that is guaranteed to succeed. The additional flexibility gained from the partially ordered plan representation allows a planning agent to handle quite complicated domains.

11.1 A SIMPLE PLANNING AGENT

When the world state is accessible, an agent can use the percepts provided by the environment to build a complete and correct model of the current world state. Then, given a goal, it can call a suitable planning algorithm (which we will call IDEAL-PLANNER) to generate a plan of action. The agent can then execute the steps of the plan, one action at a time.

The algorithm for the simple planning agent is shown in Figure 11.1. This should be compared with the problem-solving agent shown in Figure 3.1. The planning algorithm IDEAL-PLANNER can be any of the planners described in this chapter or Chapter 12. We assume the existence of a function STATE-DESCRIPTION, which takes a percept as input and returns an initial state description in the format required by the planner, and a function MAKE-GOAL-QUERY, which is used to ask the knowledge base what the next goal should be. Note that the agent must deal with the case where the goal is infeasible (it just ignores it and tries another), and the case

```
function SIMPLE-PLANNING-AGENT(percept) returns an action
   static: KB, a knowledge base (includes action descriptions)
           p, a plan, initially NoPlan
           t, a counter, initially 0, indicating time
   local variables: G, a goal
                    current, a current state description

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   current ← STATE-DESCRIPTION(KB, t)
   if p = NoPlan then
      G ← ASK(KB, MAKE-GOAL-QUERY(t))
      p ← IDEAL-PLANNER(current, G, KB)
   if p = NoPlan or p is empty then action ← NoOp
   else
      action ← FIRST(p)
      p ← REST(p)
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

Figure 11.1 A simple planning agent. The agent first generates a goal to achieve, and then constructs a plan to achieve it from the current state. Once it has a plan, it keeps executing it until the plan is finished, then begins again with a new goal.

where the complete plan is in fact empty, because the goal is already true in the initial state. The agent interacts with the environment in a minimal way—it uses its percepts to define the initial state and thus the initial goal, but thereafter it simply follows the steps in the plan it has constructed. In Chapter 13, we discuss more sophisticated agent designs that allow more interaction between the world and the planner during plan execution.

11.2 FROM PROBLEM SOLVING TO PLANNING

Planning and problem solving are considered different subjects because of the differences in the representations of goals, states, and actions, and the differences in the representation and construction of action sequences. In this section, we first describe some of the difficulties encountered by a search-based problem-solving approach, and then introduce the methods used by planning systems to overcome these difficulties.

Recall the basic elements of a search-based problem-solver:

- **Representation of actions.** Actions are described by programs that generate successor state descriptions.
- **Representation of states.** In problem solving, a complete description of the initial state is

given, and actions are represented by a program that generates complete state descriptions, Therefore, all state representations are complete. In most problems, a state is a simple data structure: a permutation of the pieces in the eight puzzle, the position of the agent in a route-finding problem, or the position of the six people and the boat in the missionaries and cannibals problem. State representations are used only for successor generation, heuristic function evaluation, and goal testing.

- **Representation of goals.** The only information that a problem-solving agent has about its goal is in the form of the goal test and the heuristic function. Both of these can be applied to states to decide on their desirability, but they are used as "black boxes." That is, the problem-solving agent cannot "look inside" to select actions that might be useful in achieving the goal.

- **Representation of plans.** In problem solving, a solution is a sequence of actions, such as "Go from Arad to Sibiu to Fagaras to Bucharest." During the construction of solutions, search algorithms consider only unbroken sequences of actions beginning from the initial state (or, in the case of bidirectional search, ending at a goal state).

Let us see how these design decisions affect an agent's ability to solve the following simple problem: "Get a quart of milk and a bunch of bananas and a variable-speed cordless drill." Treating this as a problem-solving exercise, we need to specify the initial state: the agent is at home but without any of the desired objects, and the operator set: all the things that the agent can do. We can optionally supply a heuristic function: perhaps the number of things that have not yet been acquired.

Figure 11.2 shows a very small part of the first two levels of the search space for this problem, and an indication of the path toward the goal. The actual branching factor would be in the thousands or millions, depending on how actions are specified, and the length of the solution could be dozens of steps. Obviously, there are too many actions and too many states to consider. The real difficulty is that the heuristic evaluation function can only choose among states to decide which is closer to the goal; it cannot eliminate actions from consideration. Even if the evaluation function could get the agent into the supermarket, the agent would then resort to a guessing game. The agent makes guesses by considering actions—buying an orange, buying tuna fish, buying corn flakes, buying milk—and the evaluation function ranks these guesses—bad, bad, bad, good. The agent then knows that buying milk is a good thing, but has no idea what to try next and must start the guessing process all over again.

The fact that the problem-solving agent considers sequences of actions starting from the initial state also contributes to its difficulties. It forces the agent to decide first what to do in the initial state, where the relevant choices are essentially to go to any of a number of other places. Until the agent has figured out *how* to obtain the various items—by buying, borrowing, leasing, growing, manufacturing, stealing—it cannot really decide where to go. The agent therefore needs a more flexible way of structuring its deliberations, so that it can work on whichever part of the problem is most likely to be solvable given the current information.

The first key idea behind planning is to *"open up" the representation of states, goals, and actions.* Planning algorithms use descriptions in some formal language, usually first-order logic or a subset thereof. States and goals and goals are represented by sets of sentences, and actions are represented by logical descriptions of preconditions and effects. This enables the planner to

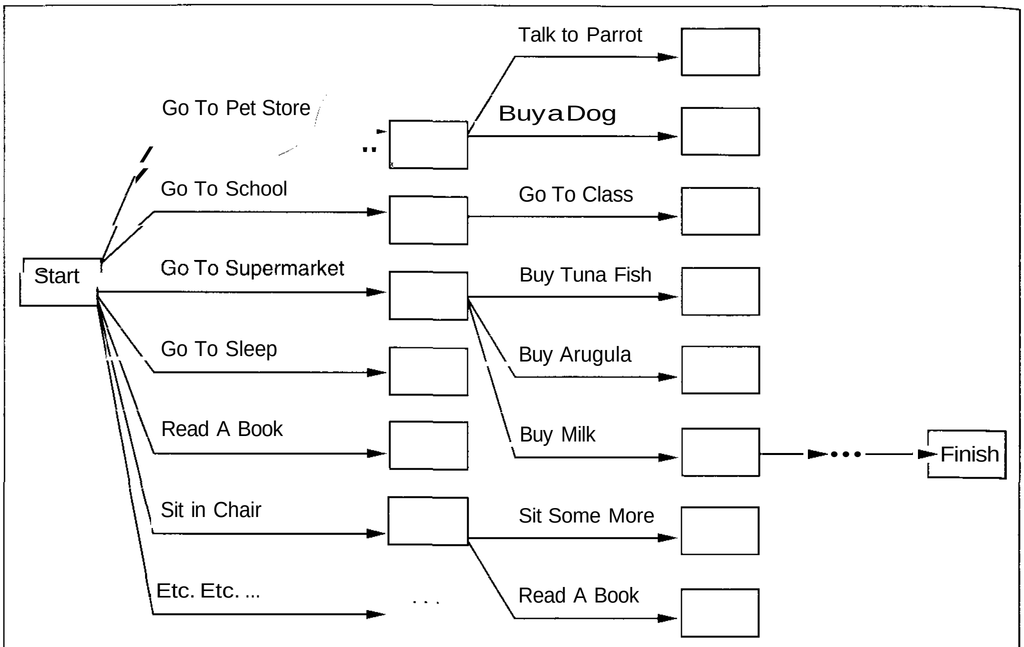

Figure 11.2 Solving a shopping problem with forward search through the space of situations in the world.

make direct connections between states and actions. For example, if the agent knows that the goal is a conjunction that includes the conjunct *Have*(*Milk*), and that *Buy*(*x*) achieves *Have*(*x*), then the agent knows that it is worthwhile to consider a plan that includes *Buy*(*Milk*). It need not consider irrelevant actions such as *Buy*(*WhippingCream*) or *GoToSleep*.

The second key idea behind planning is that *the planner is free to add actions to the plan wherever they are needed, rather than in an incremental sequence starting at the initial state.* For example, the agent may decide that it is going to have to *Buy*(*Milk*), even before it has decided where to buy it, how to get there, or what to do afterwards. There is no necessary connection between the order of planning and the order of execution. By making "obvious" or "important" decisions first, the planner can reduce the branching factor for future choices and reduce the need to backtrack over arbitrary decisions. Notice that the representation of states as sets of logical sentences plays a crucial role in making this freedom possible. For example, when adding the action *Buy*(*Milk*) to the plan, the agent can represent the state in which the action is executed as, say, *At*(*Supermarket*). This actually represents an entire class of states—states with and without bananas, with and without a drill, and so on. Search algorithms that require complete state descriptions do not have this option.

The third and final key idea behind planning is that *most parts of the world are independent of most other parts.* This makes it feasible to take a conjunctive goal like "get a quart of milk *and* a bunch of bananas *and* a variable-speed cordless drill" and solve it with a divide-and-conquer strategy. A subplan involving going to the supermarket can be used to achieve the first two conjuncts, and another subplan (e.g., either going to the ·hardware store or borrowing from a neighbor) can be used to achieve the third. The supermarket subplan can be further divided into a milk subplan and a bananas subplan. We can then put all the subplans together to solve the whole problem. This works because there is little interaction between the two subplans: going to the supermarket does not interfere with borrowing from a neighbor, and buying milk does not interfere with buying bananas (unless the agent runs out of some resource, like time or money).

Divide-and-conquer algorithms are efficient because it is almost always easier to solve several small sub-problems rather than one big problem. However, divide-and-conquer fails in cases where the cost of combining the solutions to the sub-problems is too high. Many puzzles have this property. For example, the goal state in the eight puzzle is a conjunctive goal: to get tile 1 in position A *and* tile 2 in position *B and* ... up to tile 8. We could treat this as a planning problem and plan for each subgoal independently, but the reason that puzzles are "tricky" is that it is difficult to put the subplans together. It is easy to get tile 1 in position A, but getting tile 2 in position *B* is likely to move tile 1 out of position. For tricky puzzles, the planning techniques in this chapter will not do any better than problem-solving techniques of Chapter 4. Fortunately, the real world is a largely benign place where subgoals tend to be nearly independent. If this were not the case, then the sheer size of the real world would make successful problem solving impossible.

11.3 PLANNING IN SITUATION CALCULUS

Before getting into planning techniques in detail, we present a formulation of planning as a logical inference problem, using situation calculus (see Chapter 7). A planning problem is represented in situation calculus by logical sentences that describe the three main parts of a problem:

- **Initial state:** An arbitrary logical sentence about a situation S_0. For the shopping problem, this might be[1]

  $$At(Home, S_0) \text{ A } \neg Have(Milk, S_0) \text{ A } \neg Have(Bananas, S_0) \text{ A } \neg Have(Drill, S_0)$$

- **Goal state:** A logical query asking for suitable situations. For the shopping problem, the query would be

  $$3\, s\;\; At(Home, s) \text{ A } Have(Milk, s) \text{ A } Have(Bananas, s) \text{ A } Have(Drill, s)$$

- **Operators:** A set of descriptions of actions, using the action representation described in Chapter 7. For example, here is a successor-state axiom involving the *Buy*(*Milk*)action:

  $$\begin{array}{lcl} \text{V } a, s\;\; Have(Milk, Result(a, s)) & \Leftrightarrow & [(a = Buy(Milk) \text{ A } At(Supermarket, s) \\ & \text{V} & \;\; (Have(Milk, s) \wedge a \neq Drop(Milk))] \end{array}$$

Recall that situation calculus is based on the idea that actions transform states: *Result*(*a*, *s*) names the situation resulting from executing action *a* in situation *s*. For the purposes of planning, it

[1] A better representation might be along the lines of $\neg\exists\, m\;\; Milk(m) \text{ A } Have(m, So)$, but we have chosen the simpler notation to facilitate the explanation of planning methods. Notice that partial state information is handled automatically by a logical representation, whereas problem-solving algorithms required a special multiple-state representation.

will be useful to handle action sequences as well as single actions. We will use $Result'(l,s)$ to mean the situation resulting from executing the sequence of actions / starting in s. $Result'$ is defined by saying that an empty sequence of actions has no effect on a situation, and the result of a nonempty sequence of actions is the same as applying the first action, and then applying the rest of the actions from the resulting situation:

$$\forall s \;\; Result'([\,], s) = s$$
$$\forall a, p, s \;\;\; Result'([a|p], s) = Result'(p, Result(a, s))$$

A solution to the shopping problem is a plan p that when applied to the start state S_0 yields a situation satisfying the goal query. In other words, a p such that

$$At(Home, Result'(p, S_0)) \text{ A } Have(Milk, Result'(p, S_0)) \text{ A } Have(Bananas, Result'(p, S_0))$$
$$\text{A } Have(Drill, Result'(p, S_0))$$

If we hand this query to ASK, we end up with a solution such as

$$p = [Go(SuperMarket), Buy(Milk), Buy(Banana),$$
$$Go(HardwareStore), Buy(Drill), \;Go(Home)]$$

From the theoretical point of view, there is little more to say. We have a formalism for expressing goals and plans, and we can use the well-defined inference procedure of first-order logic to find plans. It is true that there are some limitations in the expressiveness of situation calculus, as discussed in Section 8.4, but situation calculus is sufficient for most planning domains.

Unfortunately, a good theoretical solution does not guarantee a good practical solution. We saw in Chapter 3 that problem solving takes time that is exponential in the length of the solution in the worst case, and in Chapter 9, we saw that logical inference is only semidecidable. If you suspect that planning by unguided logical inference would be inefficient, you're right. Furthermore, the inference procedure gives us no guarantees about the resulting plan p other than that it achieves the goal. In particular, note that if p achieves the goal, then so do $[Nothing|p]$ and $[A, A^{-1} \backslash p]$, where *Nothing* is an action that makes no changes (or at least no relevant changes) to the situation, and A^{-1} is the inverse of A (in the sense that $s = Result(A^{-1}, Result(A, s))$). So we may end up with a plan that contains irrelevant steps if we use unguided logical inference.

To make planning practical we need to do two things: (1) Restrict the language with which we define problems. With a restrictive language, there are fewer possible solutions to search through. (2) Use a special-purpose algorithm called a **planner** rather than a general-purpose theorem prover to search for a solution. The two go hand in hand: every time we define a new problem-description language, we need a new planning algorithm to process the language. The remainder of this chapter and Chapter 12 describe a series of planning languages of increasing complexity, along with planning algorithms for these languages. Although we emphasize the algorithms, it is important to remember that *we are always dealing with a logic: a formal language with a well-defined syntax, semantics, and proof theory.* The proof theory says what can be inferred about the results of action sequences, and therefore what the legal plans are. The algorithm enables us to find those plans. The idea is that the algorithm can be designed to process the restricted language more efficiently than a resolution theorem prover.

PLANNER

11.4 BASIC REPRESENTATIONS FOR PLANNING

The "classical" approach that most planners use today describes states and operators in a restricted language known as the STRIPS language,[2] or in extensions thereof. The STRIPS language lends itself to efficient planning algorithms, while retaining much of the expressiveness of situation calculus representations.

Representations for states and goals

In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated. For example, the initial state for the milk-and-bananas problem might be described as

$$At(Home) \text{ A } \neg Have(Milk) \text{ A } \neg Have(Bananas) \text{ A } \neg Have(Drill) \text{ A } \cdots$$

As we mentioned earlier, a state description does not need to be complete. An incomplete state description, such as might be obtained by an agent in an inaccessible environment, corresponds to a set of possible complete states for which the agent would like to obtain a successful plan. Many planning systems instead adopt the convention—analogous to the "negation as failure" convention used in logic programming—that if the state description does not mention a given positive literal then the literal can be assumed to be false.

Goals are also described by conjunctions of literals. For example, the shopping goal might be represented as

$$At(Home) \text{ A } Have(Milk) \text{ A } Have(Bananas) \text{ A } Have(Drill)$$

Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as

$$At(x) \wedge Sells(x, Milk)$$

As with goals given to theorem provers, the variables are assumed to be existentially quantified. However, one must distinguish clearly between a goal given to a planner and a query given to a theorem prover. The former asks for a sequence of actions that *makes the goal true if executed,* and the latter asks whether the query sentence *is true* given the truth of the sentences in the knowledge base.

Although representations of initial states and goals are used as inputs to planning systems, it is quite common for the planning process itself to maintain only implicit representations of states. Because most actions change only a small part of the state representation, it is more efficient to keep track of the changes. We will see how this is done shortly.

[2] Named after a pioneering planning program known as the STanford Research Institute Problem Solver. There are two unfortunate things about the name STRIPS. First, the organization no longer uses the name "Stanford" and is now known as SRI International. Second, the program is what we now call a planner, not a problem solver, but when it was developed in 1970, the distinction had not been articulated. Although the STRIPS planner has long since been superseded, the STRIPS language for describing actions has been invaluable, and many "STRIPS-like" variants have been developed.

Representations for actions

Our STRIPS operators consist of three components:

ACTION DESCRIPTION

- **The action description** is what an agent actually returns to the environment in order to do something. Within the planner it serves only as a name for a possible action.

PRECONDITION

- **The precondition** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.

EFFECT

- **The effect** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.[3]

Here is an example of the syntax we will use for forming a STRIPS operator for going from one place to another:

$$Op(\text{ACTION:}Go(there), \text{PRECOND:}At(here) \wedge Path(here, there),$$
$$\text{EFFECT:}At(there) \wedge \neg At(here))$$

(We will also use a graphical notation to describe operators, as shown in Figure 11.3.) Notice that there are no explicit situation variables. Everything in the precondition implicitly refers to the situation immediately before the action, and everything in the effect implicitly refers to the situation that is the result of the action.

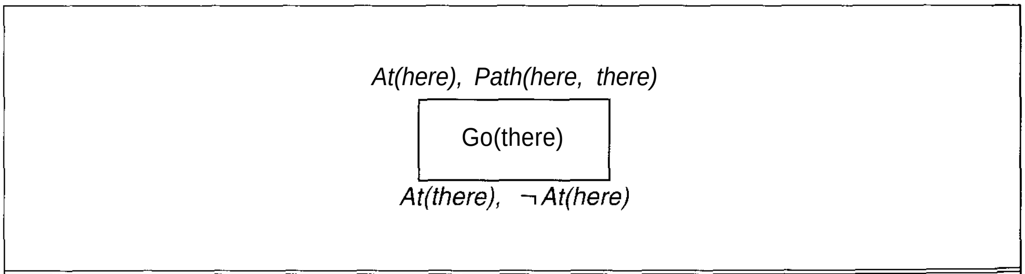

Figure 11.3 Diagrammatic notation for the operator *Go(there)*. The preconditions appear above the action, and the effects below.

OPERATOR SCHEMA

An operator with variables is known as an **operator schema,** because it does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables. Usually, only fully instantiated operators can be executed; our planning algorithms will ensure that each variable has a value by the time the planner is done. As with state descriptions, the language of preconditions and effects is quite restricted. The precondition must be a conjunction of positive literals, and the effect must be a conjunction of positive and/or negative literals. All variables are assumed universally quantified, and there can be no additional quantifiers. In Chapter 12, we will relax these restrictions.

APPLICABLE

We say that an operator o is **applicable** in a state s if there is some way to instantiate the variables in o so that every one of the preconditions of o is true in s, that is, if $Precond(o) \subseteq s$. In the resulting state, all the positive literals in $Effect(o)$ hold, as do all the literals that held in s,

[3] The original version of STRIPS divided the effects into an **add list and a delete list.**

except for those that are negative literals in *Effect*(*o*). For example, if the initial situation includes the literals

$$At(Home),\ Path(Home, Supermarket), \ldots$$

then the action *Go*(*Supermarket*) is applicable, and the resulting situation contains the literals

$$\neg At(Home),\ At(Supermarket),\ Path(Home, Supermarket), \ldots$$

Situation Space and Plan Space

In Figure 11.2, we showed a search space of *situations* in the world (in this case the shopping world). A path through this space from the initial state to the goal state constitutes a plan for the shopping problem. If we wanted, we could take a problem described in the STRIPS language and solve it by starting at the initial state and applying operators one at a time until we reached a state that includes all the literals in the goal. We could use any of the search methods of Part II. An algorithm that did this would clearly be considered a problem solver, but we could also consider it a planner. We would call it a **situation space** planner because it searches through the space of possible situations, and a **progression** planner because it searches forward from the initial situation to the goal situation. The main problem with this approach is the high branching factor and thus the huge size of the search space.

SITUATION SPACE

PROGRESSION

One way to try to cut the branching factor is to search backwards, from the goal state to the initial state; such a search is called **regression** planning. This approach is *possible* because the operators contain enough information to regress from a partial description of a result state to a partial description of the state before an operator is applied. We cannot get complete descriptions of states this way, but we don't need to. The approach is *desirable* because in typical problems the goal state has only a few conjuncts, each of which has only a few appropriate operators, whereas the initial state usually has many applicable operators. (An operator is appropriate to a goal if the goal is an effect of the operator.) Unfortunately, searching backwards is complicated somewhat by the fact that we often have to achieve a conjunction of goals, not just one. The original STRIPS algorithm was a situation-space regression planner that was incomplete (it could not always find a plan when one existed) because it had an inadequate way of handling the complication of conjunctive goals. Fixing this incompleteness makes the planner very inefficient.

REGRESSION

In summary, the nodes in the search tree of a situation-space planner correspond to situations, and the path through the search tree is the plan that will be ultimately returned by the planner. Each branch point adds another step to either the beginning (regression) or end (progression) of the plan.

An alternative is to search through the space of *plans* rather than the space of *situations*. That is, we start with a simple, incomplete plan, which we call a **partial plan.** Then we consider ways of expanding the partial plan until we come up with a complete plan that solves the problem. The operators in this search are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. The solution is the final plan, and the path taken to reach it is irrelevant.

PARTIAL PLAN

Operations on plans come in two categories. **Refinement operators** take a partial plan and add constraints to it. One way of looking at a partial plan is as a representation for a set

REFINEMENT OPERATORS

of complete, fully constrained plans. Refinement operators eliminate some plans from this set, but they never add new plans to it. Anything that is not a refinement operator is a **modification operator.** Some planners work by constructing potentially incorrect plans, and then "debugging" them using modification operators. In this chapter, we use only refinement operators.

MODIFICATION OPERATOR

Representations for plans

If we are going to search through a space of plans, we need to be able to represent them. We can settle on a good representation for plans by considering partial plans for a simple problem: putting on a pair of shoes. The goal is the conjunction of *RightShoeOn* A *LeftShoeOn,* the initial state has no literals at all, and the four operators are

> *Op*(ACTION:*RightShoe*,PRECOND:*RightSockOn*, EFFECT: *RightShoeOn)*
> *Op*(ACTION:*RightSock*, EFFECT:*RightSockOn*)
> *Op*(ACTION:*LeftShoe*,PRECOND:*LeftSockOn*, *EFFECT. LeftShoeOn)*
> *Op*(ACTION:*LeftSock*,EFFECT: *LeftSockOn*)

LEAST COMMITMENT

A partial plan for this problem consists of the two steps *RightShoe* and *LeftShoe.* But which step should come first? Many planners use the principle of **least commitment,** which says that one should only make choices about things that you currently care about, leaving the other choices to be worked out later. This is a good idea for programs that search, because if you make a choice about something you don't care about now, you are likely to make the wrong choice and have to backtrack later. A least commitment planner could leave the ordering of the two steps unspecified. When a third step, *RightSock*, is added to the plan, we want to make sure that putting on the right sock comes before putting on the right shoe, but we do not care where they come with respect to the left shoe. A planner that can represent plans in which some steps are ordered (before or after) with respect to each other and other steps are unordered is called a **partial order** planner. The alternative is a **total order** planner, in which plans consist of a simple list of steps. A totally ordered plan that is derived from a plan *P* by adding ordering constraints is called a **linearization of *P.***

PARTIAL ORDER

TOTAL ORDER

LINEARIZATION

The socks-and-shoes example does not show it, but planners also have to commit to bindings for variables in operators. For example, suppose one of your goals is *Have*(*Milk*),and you have the action *Buy*(*item*, *store).* A sensible commitment is to choose this action with the variable *item* bound to *Milk.* However, there is no good reason to pick a binding for *store,* so the principle of least commitment says to leave it unbound and make the choice later. Perhaps another goal will be to buy an item that is only available in one specialty store. If that store also carries milk, then we can bind the variable *store* to the specialty store at that time. By delaying the commitment to a particular store, we allow the planner to make a good choice later. This strategy can also help prune out bad plans. Suppose that for some reason the branch of the search space that includes the partially instantiated action *Buy(Milk, store)* leads to a failure for some reason unrelated to the choice of store (perhaps the agent has no money). If we had committed to a particular store, then the search algorithm would force us to backtrack and consider another store. But if we have not committed, then there is no choice to backtrack over and we can discard this whole branch of the search tree without having to enumerate any of the stores. Plans in which every variable is bound to a constant are called **fully instantiated plans.**

FULLY INSTANTIATED PLANS

PLAN

In this chapter, we will use a representation for plans that allows for deferred commitments about ordering and variable binding. A **plan** is formally defined as a data structure consisting of the following four components:

- A set of plan steps. Each step is one of the operators for the problem.
- A set of step ordering constraints. Each ordering constraint is of the form $S_i \prec Sj$, which is read as "S_i before S_j"and means that step S_i must occur sometime before step Sj (but not necessarily immediately before).[4]
- A set of variable binding constraints. Each variable constraint is of the form $v = x$, where v is a variable in some step, and x is either a constant or another variable.
- **A set** of **causal links**.[5] A causal link is written as $S_i \xrightarrow{c} S_j$ and read as "S_i achieves c for Sj" Causal links serve to record the purpose(s) of steps in the plan: here a purpose of S_i is to achieve the precondition c of Sj.

CAUSAL LINKS

The initial plan, before any refinements have taken place, simply describes the unsolved problem. It consists of two steps, called *Start* and *Finish,* with the ordering constraint *Start* X *Finish.* Both *Start* and *Finish* have null actions associated with them, so when it is time to execute the plan, they are ignored. The *Start* step has no preconditions, and its effect is to add all the propositions that are true in the initial state. The *Finish* step has the goal state as its precondition, and no effects. By defining a problem this way, our planners can start with the initial plan and manipulate it until they come up with a plan that is a solution. The shoes-and-socks problem is defined by the four operators given earlier and an initial plan that we write as follows:

$Plan(\text{STEPS}:\{\ S_1: Op(\text{ACTION}:Start),$
$\quad S_2: Op(\text{ACTION}:Finish,$
$\quad\quad \text{PRECOND}:RightShoeOn \text{ A } LeftShoeOn)\},$
$\text{ORDERINGS}: \{S_1 \prec S_2\},$
$\text{BINDINGS}: \{\},$
$\text{LINKS}: \{\})$

As with individual operators, we will use a graphical notation to describe plans (Figure 11.4(a)). The initial plan for the shoes-and-socks problem is shown in Figure 11.4(b). Later in the chapter we will see how this notation is extended to deal with more complex plans.

Figure 11.5 shows a partial-order plan that is a solution to the shoes-and-socks problem, and six linearizations of the plan. This example shows that the partial-order plan representation is powerful because it allows a planner to ignore ordering choices that have no effect on the correctness of the plan. As the number of steps grows, the number of possible ordering choices grows exponentially. For example, if we added a hat and a coat to the problem, which interact neither with each other nor with the shoes and socks, then there would still be one partial plan that represents all the solutions, but there would be 180 linearizations of that partial plan. (Exercise 11.1 asks you to derive this number).

[4] We use the notation $A \prec B \prec C$ to mean $(A \text{ X } B) \text{ A } (B \prec C)$.

[5] Some authors call causal links **protection intervals.**

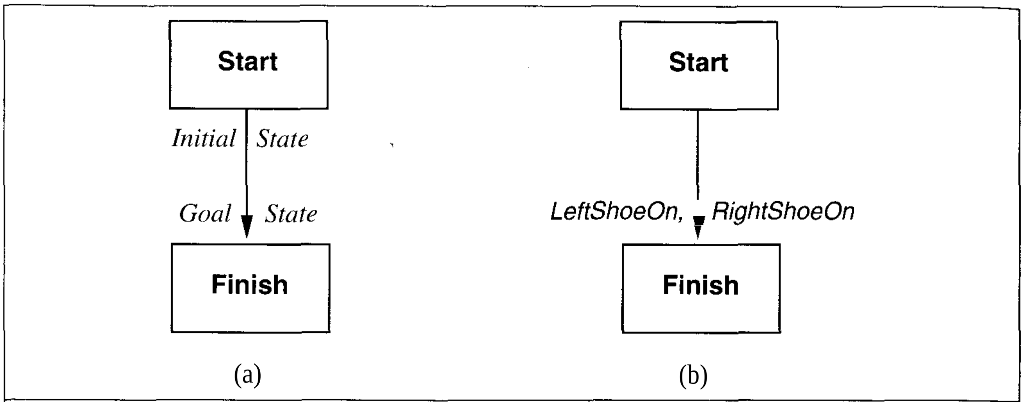

Figure 11.4 (a) Problems are defined by partial plans containing only *Start* and *Finish* steps. The initial state is entered as the effects of the *Start* step, and the goal state is the precondition of the *Finish* step. Ordering constraints are shown as arrows between boxes. (b) The initial plan for the shoes-and-socks problem.

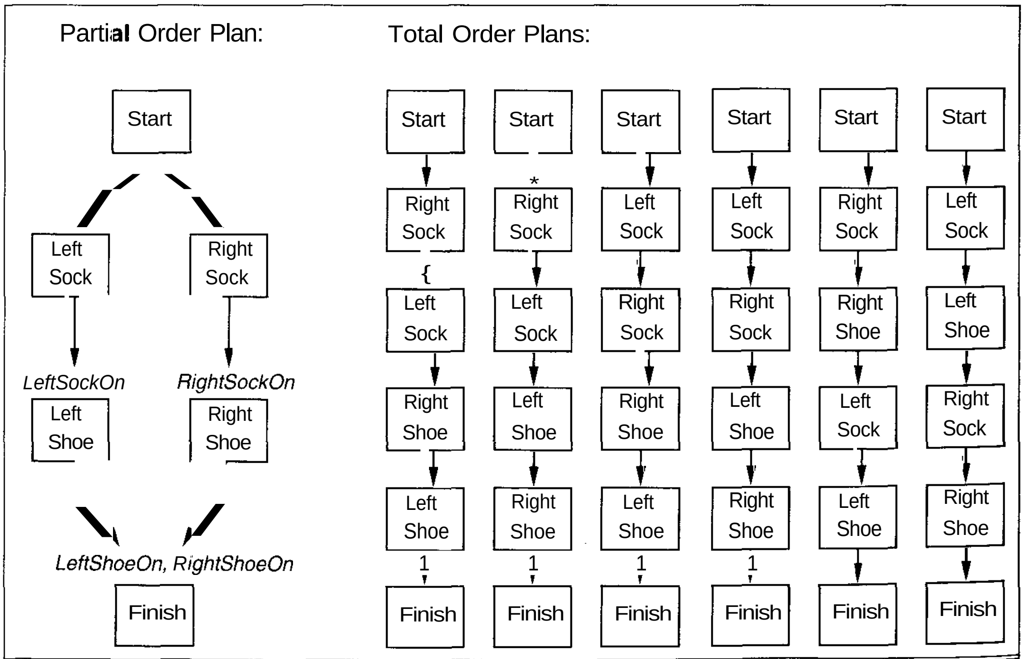

Figure 11.5 A partial-order plan for putting on shoes and socks (including preconditions on steps), and the six possible linearizations of the plan.

Solutions

A solution is a plan that an agent can execute, and that guarantees achievement of the goal. If we wanted to make it really easy to check that a plan is a solution, we could insist that only fully instantiated, totally ordered plans can be solutions. But this is unsatisfactory for three reasons. First, for problems like the one *in* Figure 11.5, it is more natural for the planner to return a partial-order plan than to arbitrarily choose one of the many linearizations of it. Second, some agents are capable of performing actions in parallel, so it makes sense to allow solutions with parallel actions. Lastly, when creating plans that may later be combined with other plans to solve larger problems, it pays to retain the flexibility afforded by the partial ordering of actions.

SOLUTION

Therefore, we allow partially ordered plans as solutions using a simple definition: a **solution is a complete, consistent** plan. We need to define these terms.

COMPLETE PLAN

ACHIEVED

A **complete plan** is one in which every precondition of every step is **achieved** by some other step. A step achieves a condition if the condition is one of the effects of the step, and if no other step can possibly cancel out the condition. More formally, a step S_i achieves a precondition c of the step Sj if (1) $S_i \prec Sj$ and c G EFFECTS(S_i); and (2) there is no step S_k such that ($\neg c$) G EFFECTS(S_k), where $S_i \prec S_k \prec Sj$ in some linearization of the plan.

CONSISTENT PLAN

A **consistent plan** is one in which there are no contradictions in the ordering or binding constraints. A contradiction occurs when both $S_i \prec Sj$ and $S_j \prec S_i$ hold or both $v = A$ and $v = B$ hold (for two different constants *A* and *B).* Both $\prec$ and = are transitive, so, for example, a plan with $S_1 \prec S_2$, $S_2 \prec S_3$, and $S_3 \prec S_1$ is inconsistent.

The partial plan in Figure 11.5 is a solution because all the preconditions are achieved. From the preceding definitions, it is easy to see that any linearization of a solution is also a solution. Hence the agent can execute the steps in any order consistent with the constraints, and still be assured of achieving the goal.

11.5 A PARTIAL-ORDER PLANNING EXAMPLE

In this section, we sketch the outline of a partial-order regression planner that searches through plan space. The planner starts with an initial plan representing the start and finish steps, and on each iteration adds one more step. If this leads to an inconsistent plan, it backtracks and tries another branch of the search space. *To keep the search focused, the planner only considers adding steps that serve to achieve a precondition that has not yet been achieved.* The causal links are used to keep track of this.

We illustrate the planner by returning to the problem of getting some milk, a banana, and a drill, and bringing them back home. We will make some simplifying assumptions. First, the *Go* action can be used to travel between any two locations. Second, the description of the *Buy* action ignores the question of money (see Exercise 11.2). The initial state is defined by the following operator, where *HWS* means hardware store and *SM* means supermarket:

Op(ACTION:*Start*, EFFECT:*At*(*Home*)A *Sells*(*HWS*, *Drill)*
A *Sells*(*SM*,*Milk*), *Sells*(*SM*,*Banana*))

The goal state is defined by a *Finish* step describing the objects to be acquired and the final destination to be reached:

> *Op*(ACTION:*Finish*,
> PRECOND:*Have*(*Drill*)A *Have*(*Milk*) A *Have*(*Banana*) A *At*(*Home*))

The actions themselves are defined as follows:

> *Op*(ACTION:*Go*(*there*), PRECOND:*At*(*here*),
> EFFECT:*At*(*there*) A ¬*At*(*here*))
> *Op*(ACTION:*Buy*(*x*), PRECOND:*At*(*store*)A *Sells*(*store*, *x*),
> EFFECT: *Have*(*x*))

Figure 11.6 shows a diagram of the initial plan for this problem. We will develop a solution to the problem step by step, showing at each point a figure illustrating the partial plan at that point in the development. As we go along, we will note some of the properties we require for the planning algorithm. After we finish the example, we will present the algorithm in detail.

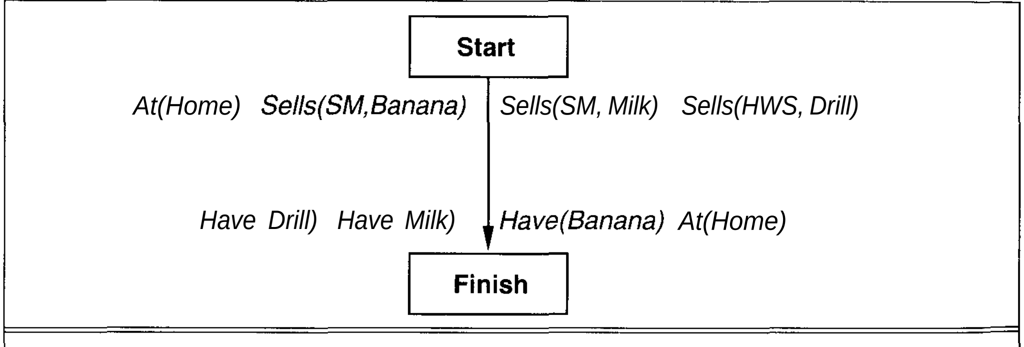
Start
At(Home) Sells(SM,Banana) Sells(SM, Milk) Sells(HWS, Drill)
Have Drill) Have Milk) Have(Banana) At(Home)
Finish

Figure 11.6 The initial plan for the shopping problem.

The first thing to notice about Figure 11.6 is that there are many possible ways in which the initial plan can be elaborated. Some choices will work, and some will not. As we work out the solution to the problem, we will show some correct choices and some incorrect choices. For simplicity, we will start with some correct choices. In Figure 11.7 (top), we have selected three *Buy* actions to achieve three of the preconditions of the *Finish* action. In each case there is only one possible choice because the operator library offers no other way to achieve these conditions.

The bold arrows in the figure are causal links. For example, the leftmost causal link in the figure means that the step *Buy*(*Drill*)was added in order to achieve the *Finish* step's *Have*(*Drill*) precondition. The planner will make sure that this condition is maintained by **protecting** it: if a step might delete the *Have*(*Drill*)condition, then it will not be inserted between the *Buy*(*Drill*) step and the *Finish* step. Light arrows in the figure show ordering constraints. By definition, all actions are constrained to come after the *Start* action. Also, all causes are constrained to come before their effects, so you can think of each bold arrow as having a light arrow underneath it.

The second stage in Figure 11.7 shows the situation after the planner has chosen to achieve the *Sells* preconditions by linking them to the initial state. Again, the planner has no choice here because there is no other operator that achieves *Sells*.

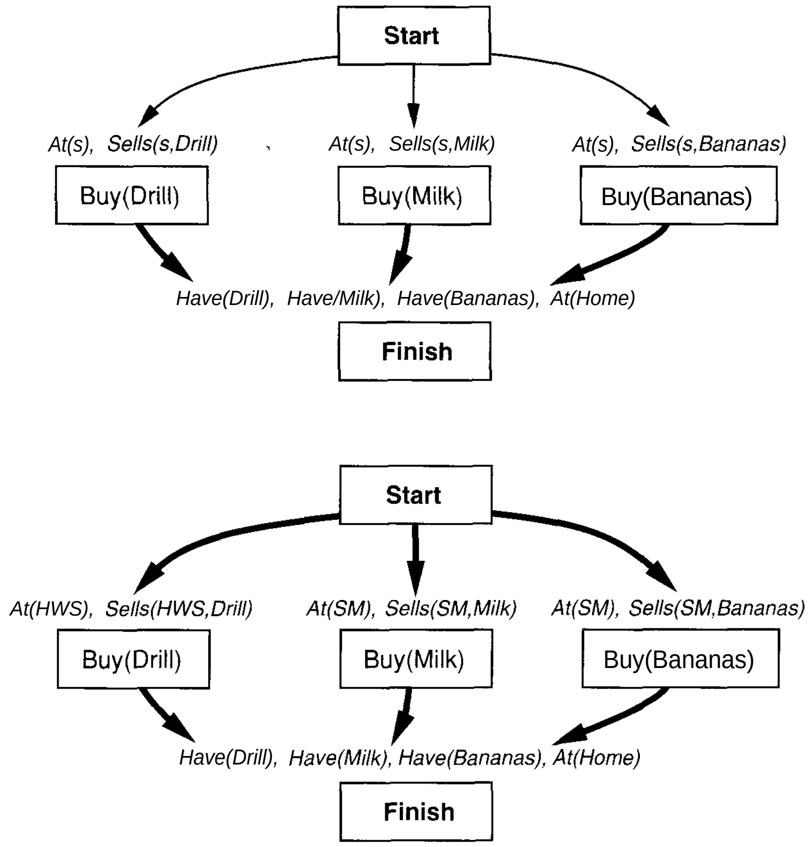

Figure 11.7 Top: A partial plan that achieves three of the four preconditions of *Finish.* The heavy arrows show causal links. Bottom: Refining the partial plan by adding causal links to achieve the *Sells* preconditions of the *Buy* steps.

Although it may not seem like we have done much yet, this is actually quite an improvement over what we could have done with the problem-solving approach. First, out of all the things that one can buy, and all the places that one can go, we were able to choose just the *right Buy* actions and just the right places, without having to waste time considering the others. Then, once we have chosen the actions, we need not decide how to order them; a partial-order planner can make that decision later.

In Figure 11.8, we extend the plan by choosing two *Go* actions to get us to the hardware store and supermarket, thus achieving the *At* preconditions of the *Buy* actions.

So far, everything has been easy. A planner could get this far without having to do any search. Now it gets harder. The two *Go* actions have unachieved preconditions that interact with each other, because the agent cannot be *At* two places at the same time. Each *Go* action has a precondition *At(x),* where *x* is the location that the agent was at before the *Go* action. Suppose

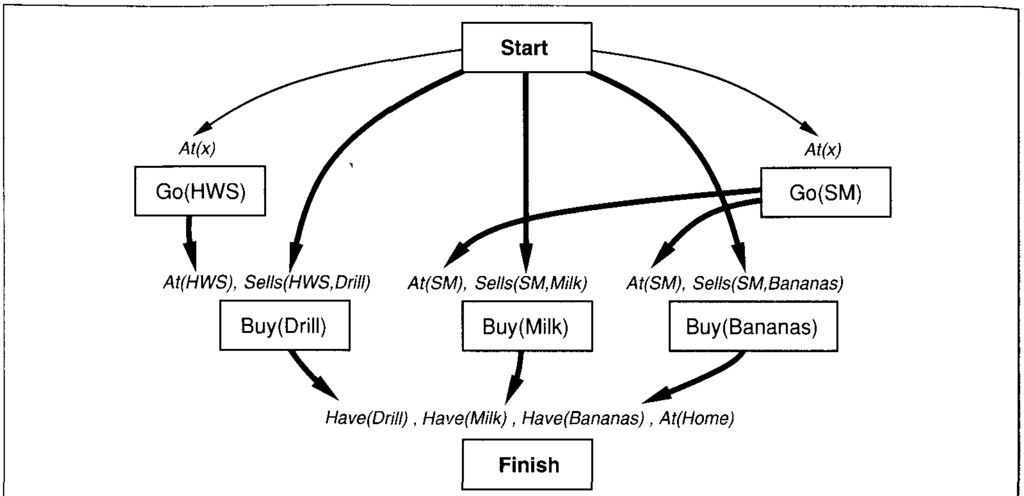
Start
At(x)
At(x)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish

Figure 11.8 A partial plan that achieves A? preconditions of the three *Buy* actions.

the planner tries to achieve the preconditions of *Go*(*HWS*)and *Go*(*SM*)by linking them to the *At*(*Home*)condition in the initial state. This results in the plan shown in Figure 11.9.

Unfortunately, this will lead to a problem. The step *Go*(*HWS*)adds the condition*At*(*HWS*), but it also deletes the condition *At*(*Home*). So if the agent goes to the hardware store, it can no longer go from home to the supermarket. (That is, unless it introduces another step to go back home from the hardware store—but the causal link means that the start step, not some other step,

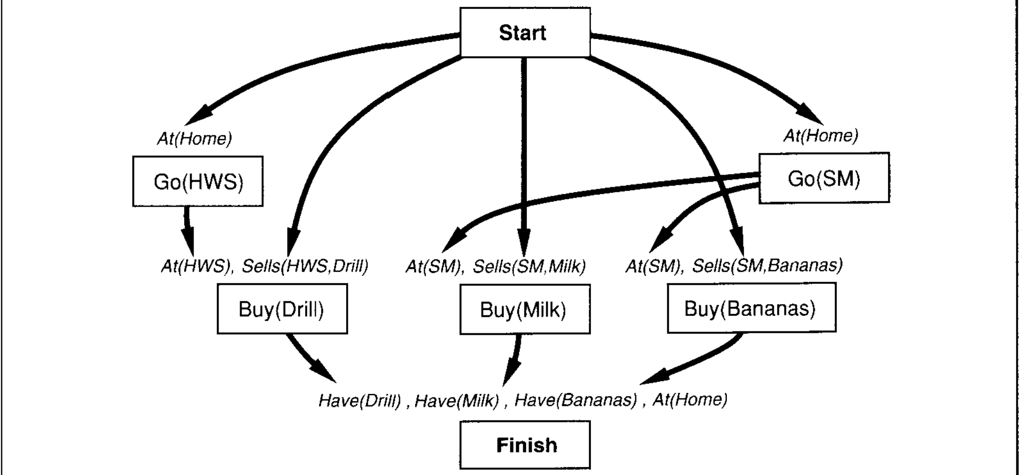
Start
At(Home)
At(Home)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish

Figure 11.9 A flawed plan that gets the agent to the hardware store and the supermarket.

achieves the *At*(*Home*) precondition.) On the other hand, if the agent goes to the supermarket first, then it cannot go from home to the hardware store.

At this point, we have reached a dead end in the search for a solution, and must back up and try another choice. The interesting part is seeing how *a planner could notice that this partial plan is a dead end without wasting a lot of time on it.* The key is that the the causal links in a partial plan are **protected links**. A causal link is protected by ensuring that **threats**—that is, steps that might delete (or **clobber**) the protected condition—are ordered to come before or after the protected link. Figure 11.10(a) shows a threat: The causal link $S_1 \xrightarrow{c} S_2$ is threatened by the new step S_3 because one effect of S_3 is to delete c. The way to resolve the threat is to add ordering constraints to make sure that S_3 does not intervene between S_1 and S_2. If S_3 is placed before S_1 this is called **demotion** (see Figure 11.10(b)), and if it is placed after S_2, it is called **promotion** (see Figure 11.10(c)).

PROTECTED LINKS

THREATS

DEMOTION

PROMOTION

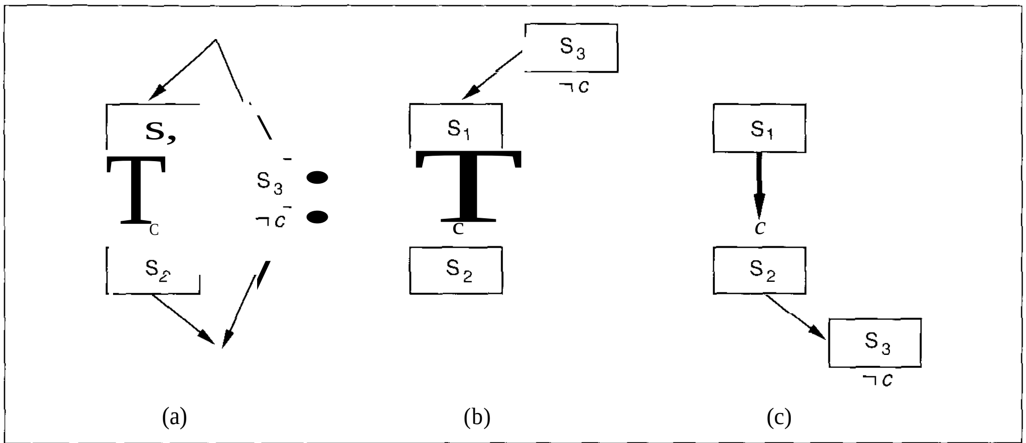

Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

In Figure 11.9, there is no way to resolve the threat that each *Go* step poses to the other. Whichever *Go* step comes first will delete the *At(Home)* condition on the other step. Whenever the planner is unable to resolve a threat by promotion or demotion, it gives up on the partial plan and backs up to a try a different choice at some earlier point in the planning process.

Suppose the next choice is to try a different way to achieve the *At*(x) precondition of the *Go(SM)* step, this time by adding a causal link from *Go(HWS)* to *Go*(*SM*). In other words, the plan is to go from home to the hardware store and then to the supermarket. This introduces another threat. Unless the plan is further refined, it will allow the agent to go from the hardware store to the supermarket without first buying the drill (which was why it went to the hardware store in the first place). However much this might resemble human behavior, we would prefer our planning agent to avoid such forgetfulness. Technically, the *Go(SM)* step threatens the *At*(*HWS*) precondition of the *Buy* (*Drill*) step, which is protected by a causal link. The threat is resolved by constraining *Go*(*SM*) to come after *Buy*(*Drill*). Figure 11.11 shows this.

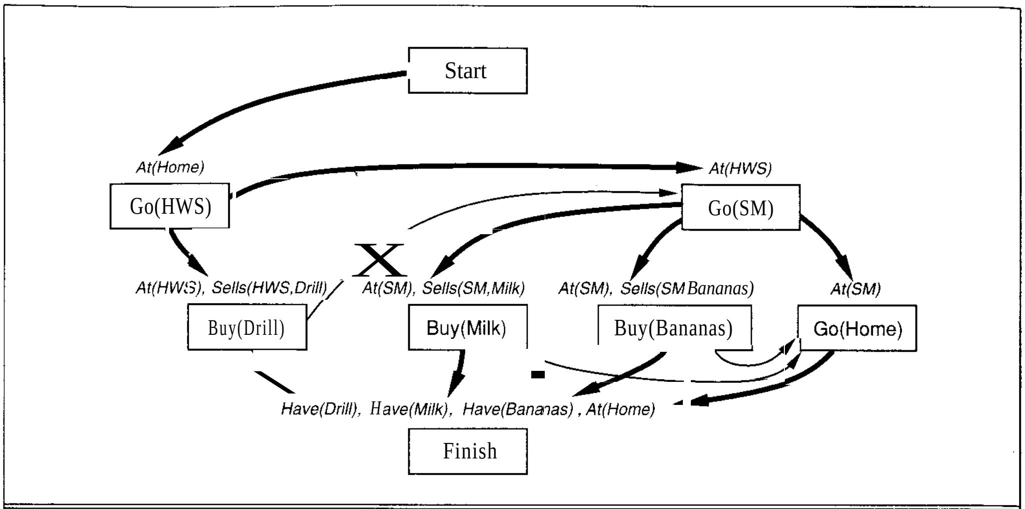

Figure 11.11 Causal link protection in the shopping plan. The $Go(HWS) \xrightarrow{At(HWS)} Buy(Drill)$ causal link is protected ordering the $Go(SM)$ step after $Buy(Drill)$, and the $Go(SM) \xrightarrow{At(SM)} Buy(Milk/Bananas)$ link is protected by ordering $Go(Home)$ after $Buy(Milk)$ and $Buy(Bananas)$.

Only the $At(Home)$ precondition of the *Finish* step remains unachieved. Adding a $Go(Home)$ step achieves it, but introduces an $At(x)$ precondition that needs to be achieved.[6] Again, the protection of causal links will help the planner decide how to do this:

- If it tries to achieve $At(x)$ by linking to $At(Home)$ in the initial state, there will be no way to resolve the threats caused by $go(HWS)$ and $Go(SM)$.
- If it tries to link $At(x)$ to the $Go(HWS)$ step, there will be no way to resolve the threat posed by the $Go(SM)$ step, which is already constrained to come after $Go(HWS)$.
- A link from $Go(SM)$ to $At(x)$ means that x is bound to *SM*, so that now the *Go(Home)* step deletes the *At(SM)* condition. This results in threats to the *At(SM)* preconditions of $Buy(Milk)$ and $Buy(Bananas)$, but these can be resolved by ordering $Go(Home)$ to come after these steps (Figure 11.11).

Figure 11.12 shows the complete solution plan, with the steps redrawn to reflect the ordering constraints on them. The result is an almost totally ordered plan; the only ambiguity is that $Buy(Milk)$ and $Buy(Bananas)$ can come in either order.

Let us take stock of what our partial-order planner has accomplished. It can take a problem that would require many thousands of search states for a problem-solving approach, and solve it with only a few search states. Moreover, the least commitment nature of the planner means it only needs to search at all in places where subplans interact with each other. Finally, the causal links allow the planner to recognize when to abandon a doomed plan without wasting a lot of time expanding irrelevant parts of the plan.

[6] Notice that the $Go(Home)$ step also has the effect $\neg At(x)$, meaning that the step will delete an *At* condition for some location yet to be decided. This is a **possible threat** to protected conditions in the plan such as $At(SM)$, but we will not worry about it for now. Possible threats are dealt with in Section 11.7.

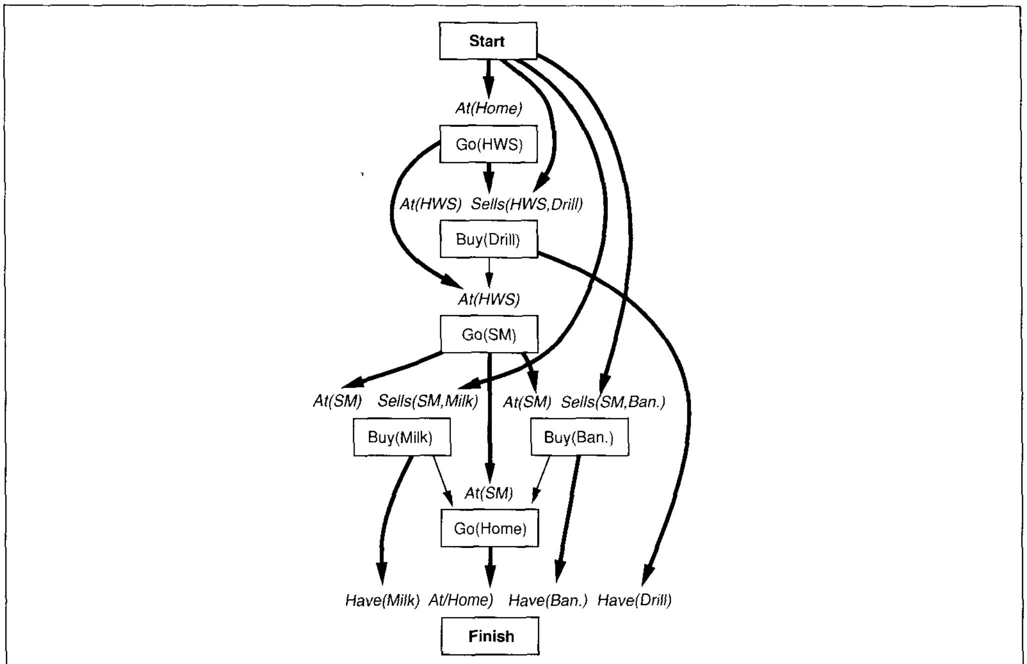

Figure 11.12 A solution to the shopping problem.

11.6 A PARTIAL-ORDER PLANNING ALGORITHM

In this section, we develop a more formal algorithm for the planner sketched in the previous section. We call the algorithm POP, for Partial-Order Planner. The algorithm appears in Figure 11.13. (Notice that POP is written as a *nondeterministic* algorithm, using **choose and fail** rather than explicit loops. Nondeterministic algorithms are explained in Appendix B.)

POP starts with a minimal partial plan, and on each step extends the plan by achieving a precondition c of a step S_{need}. It does this by choosing some operator—either from the existing steps of the plan or from the pool of operators—that achieves the precondition. It records the causal link for the newly achieved precondition, and then resolves any threats to causal links. The new step may threaten an existing causal link or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in SELECT-SUBGOAL is *not* a candidate for backtracking. The reason is that every precondition needs to be considered eventually, and the handling of preconditions is commutative: handling c_1 and then c_2 leads to exactly the same set of possible plans as handling c_2 and then c_1. So we

```
function POP(initial, goal, operators) returns plan

  plan ← MAKE-MINIMAL-PLAN(initial, goal)
  loop do
      if SOLUTION?(plan) then return plan
      $S_{need}$, c ← SELECT-SUBGOAL(plan)
      CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)
      RESOLVE-THREATS(plan)
  end
```

```
function SELECT-SUBGOAL(plan) returns $S_{need}$, c

  pick a plan step $S_{need}$ from STEPS(plan)
      with a precondition c that has not been achieved
  return $S_{need}$, c
```

```
procedure CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)

  choose a step $S_{add}$ from operators or STEPS(plan) that has c as an effect
  if there is no such step then fail
  add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(plan)
  add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(plan)
  if $S_{add}$ is a newly added step from operators then
      add $S_{add}$ to STEPS(plan)
      add $Start \prec S_{add} \prec Finish$ to ORDERINGS(plan)
```

```
procedure RESOLVE-THREATS(plan)

  for each $S_{threat}$ that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(plan) do
      choose either
          Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(plan)
          Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(plan)
      if not CONSISTENT(plan) then fail
  end
```

Figure 11.13 The partial-order planning algorithm, POP.

can just pick a precondition and move ahead without worrying about backtracking. The pick we make affects only the speed, and not the possibility, of finding a solution.

Notice that POP is a regression planner, because it starts with goals that need to be achieved and works backwards to find operators that will achieve them. Once it has achieved all the preconditions of all the steps, it is done; it has a solution. POP *is sound and complete.* Every plan it returns is in fact a solution, and if there is a solution, then it will be found (assuming a breadth-first or iterative deepening search strategy). At this point, we suggest that the reader return to the example of the previous section, and trace through the operation of POP in detail.

11.7 PLANNING WITH PARTIALLY INSTANTIATED OPERATORS

The version of POP in Figure 11.13 outlines the algorithm, but leaves some details unspecified. In particular, it does not deal with variable binding constraints. For the most part, all this entails is being diligent about keeping track of binding lists and unifying the right expressions at the right time. The implementation techniques of Chapter 10 are applicable here.

There is one substantive decision to make: in RESOLVE-THREATS, should an operator that has the effect, say, $\neg At(x)$ be considered a threat to the condition $At(Home)$? Currently we can distinguish between threats and non-threats, but this is a **possible threat.** There are three main approaches to dealing with possible threats:

POSSIBLE THREAT

- **Resolve now with an equality constraint:** Modify RESOLVE-THREATS so that it resolves all possible threats as soon as they are recognized. For example, when the planner chooses the operator that has the effect $\neg At(x)$, it would add a binding such as $x = HWS$ to make sure it does not threaten $At(Home)$.
- **Resolve now with an inequality constraint:** Extend the language of variable binding constraints to allow the constraint $x \neq Home$. This has the advantage of being a lower commitment—it does not require an arbitrary choice for the value of x—but it is a little more complicated to implement, because the unification routines we have used so far all deal with equalities, not inequalities.
- **Resolve later:** The third possibility is to ignore possible threats, and only deal with them when they become *necessary* threats. That is, RESOLVE-THREATS would not consider $\neg At(x)$, to be a threat to $At(Home)$. But if the constraint $x = Home$ were ever added to the plan, then the threat would be resolved (by promotion or demotion). This approach has the advantage of being low commitment, but has the disadvantage of making it harder to decide if a plan is a solution.

Figure 11.14 shows an implementation of the changes to CHOOSE-OPERATOR, along with the changes to RESOLVE-THREATS that are necessary for the third approach. It is certainly possible (and advisable) to do some bookkeeping so that RESOLVE-THREATS will not need to go through a triply nested loop on each call.

When partially instantiated operators appear in plans, the criterion for solutions needs to be refined somewhat. In our earlier definition (page 349), we were concerned mainly with the question of partial ordering; a solution was defined as a partial plan such that all linearizations are guaranteed to achieve the goal. With partially instantiated operators, we also need to ensure that all instantiations will achieve the goal. We therefore extend the definition of achievement for a step in a plan as follows:

> A step Si **achieves** a precondition c of the step S_j if (1) $S_i \prec S_j$ and S_i has an effect that necessarily unifies with c; and (2) there is no step S_k such that $S_i \prec S_k \prec Sj$ in some linearization of the plan, and S_k has an effect that possibly unifies with $\neg c$.

The POP algorithm can be seen as constructing a proof that the each precondition of the goal step is achieved. CHOOSE-OPERATOR comes up with the S_i that achieves (1), and RESOLVE-THREATS makes sure that (2) is satisfied by promoting or demoting possible threats. The tricky part is that

```
procedure CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)

  choose a step $S_{add}$ from operators or STEPS(plan) that has $c_{add}$ as an effect
        such that $u$ = UNIFY(C, $c_{add}$, BINDINGS(plan))
  if there is no such step
     then fail
  add $u$ to BINDINGS(plan)
  add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(plan)
  add $S_{add} \prec S_{need}$ to ORDERINGS(plan)
  if $S_{add}$ is a newly added step from operators then
     add $S_{add}$ to STEPS(plan)
     add Start $\prec S_{add} \prec$ Finish to ORDERINGS(plan)
```

```
procedure RESOLVE-THREATS(plan)

  for each $S_i \xrightarrow{c} S_j$ in LINKS(plan) do
     for each $S_{threat}$ in STEPS(plan) do
        for each $c'$ in EFFECT($S_{threat}$) do
           if SUBST(BINDINGS(plan), c) = SUBST(BINDINGS(plan), $\neg c'$) then
              choose either
                 Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(plan)
                 Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(plan)
           if not CONSISTENT(plan)
              then fail
        end
     end
  end
```

Figure 11.14 Support for partially instantiated operators in POP.

if we adopt the "resolve-later" approach, then there will be possible threats that are not resolved away. We therefore need some way of checking that these threats are all gone before we return the plan. It turns out that if the initial state contains no variables and if every operator mentions all its variables in its precondition, then any complete plan generated by POP is guaranteed to be fully instantiated. Otherwise we will need to change the function SOLUTION? to check that there are no uninstantiated variables and choose bindings for them if there are. If this is done, then POP is guaranteed to be a sound planner in all cases.

It is harder to see that POP is complete—that is, finds a solution whenever one exists—but again it comes down to understanding how the algorithm mirrors the definition of achievement. The algorithm generates every possible plan that satisfies part (1), and then filters out those plans that do not satisfy part (2) or that are inconsistent. Thus, if there is a plan that is a solution, POP will find it. So if you accept the definition **of solution** (page 349), you should accept that POP is a sound and complete planner.

11.8 KNOWLEDGE ENGINEERING FOR PLANNING

The methodology for solving problems with the planning approach is very much like the general knowledge engineering guidelines of Section 8.2:

- Decide what to talk about.
- Decide on a vocabulary of conditions (literals), operators, and objects.
- Encode operators for the domain.
- Encode a description of the specific problem instance.
- Pose problems to the planner and get back plans.

We will cover each of these five steps, demonstrating them in two domains.

The blocks world

What to talk about: The main consideration is that operators are so restricted in what they can express (although Chapter 12 relaxes some of the restrictions). In this section we show how to define knowledge for a classic planning domain: the blocks world. This domain consists of a set of cubic blocks sitting on a table. 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 only pick up 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 make two stacks, one with block A on B, and the other with C on D.

Vocabulary: The objects in this domain are the blocks and the table. They are represented by constants. We will use $On(b,x)$ to indicate that block b is on x, where x is either another block or the table. The operator for moving block b from a position on top of x to a position on top y will be $Move(b,x,y)$. Now one of the preconditions on moving b is that no other block is on it. In first-order logic this would be $\neg\exists x \ \ On(x,b)$ or alternatively $\forall x \ \ \neg On(x,b)$. But our language does not allow either of these forms, so we have to think of something else. The trick is to invent a predicate to represent the fact that no block is on b, and then make sure the operators properly maintain this predicate. We will use $Clear(x)$ to mean that nothing is on x.

Operators: The operator $Move$ moves a block b from x to y if both b and y are clear, and once the move is made, x becomes clear but y is clear no longer. The formal description of $Move$ is as follows:

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

Unfortunately, this operator does not maintain $Clear$ properly when x or y is the table. When $x = Table$, this operator has the effect $Clear(Table)$, but the table should not become clear, and when $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear to move a block onto it. To fix this, we do two things. First, we introduce another operator to move a block *b* from *x* to the table:

> *Op*(ACTION:*MoveToTable*(*b*, *x*),
> PRECOND:*On*(*b*, *x*) A *Clear*(*b*),
> EFFECT: *On*(*b*, *Table*) A *Clear*(*x*)A ¬*On*(*b*, *x*))

Second, we take the interpretation of *Clear*(*x*)to be "there is a clear space on *x* to hold a block." Under this interpretation, *Clear(Table)* will always be part of the initial situation, and it is proper that *Move*(*b*, *Table*,*y*) has the effect *Clear*(*Table*). The only problem is that nothing prevents the planner from using *Move*(*b*, *x*, *Table*) instead of *MoveToTable*(*b*, *x*). We could either 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)* A *Block*(*y*)to the precondition of *Move.*

Finally, there is the problem of spurious operations like *Move*(*B*, *C*, *C*), which should be a no-op, but which instead has contradictory effects. It is common to ignore problems like this, because they tend not to have any effect on the plans that are produced. To really fix the problem, we need to be able to put inequalities in the precondition: $b \neq x \neq y$.

Shakey's world

The original STRIPS program was designed to control Shakey,[7] a robot that roamed the halls of SRI in the early 1970s. It turns out that most of the work on STRIPS involved simulations where the actions performed were just printing to a terminal, but occasionally Shakey would actually move around, grab, and push things, based on the plans created by STRIPS. Figure 11.15 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.

Shakey can move from place to place, push movable objects (such as boxes), climb on and off of rigid objects (such as boxes), and turn light switches on and off. We will develop the vocabulary of literals along with the operators:

1. Go from current location to location *y*: *Go(y)*
   This is similar to the *Go* operator used in the shopping problem, but somewhat restricted. The precondition *At*(*Shakey*, *x*) establishes the current location, and we will insist that *x* and *y* be *In* the same room: *In*(*x*, *r*) A *In*(*y*, *r*). To allow Shakey to plan a route from room to room, we will say that the door between two rooms is *In* both of them.
2. Push an object *b* from location *x* to location *y*: *Push*(*b*, *x*, *y*)
   Again we will insist that the locations be in the same room. We introduce the predicate *Pushable(b)*, but otherwise this is similar to *Go.*
3. Climb up onto a box: *Climb*(*b*).
   We introduce the predicate *On* and the constant *Floor*, and make sure that a precondition of *Go* is *On*(*Shakey*, *Floor*). For *Climb*(*b*), the preconditions are that Shakey is A? the same place as *b*, and *b* must be *Climbable.*

7 Shakey's name comes from the fact that its motors made it a little unstable when it moved.

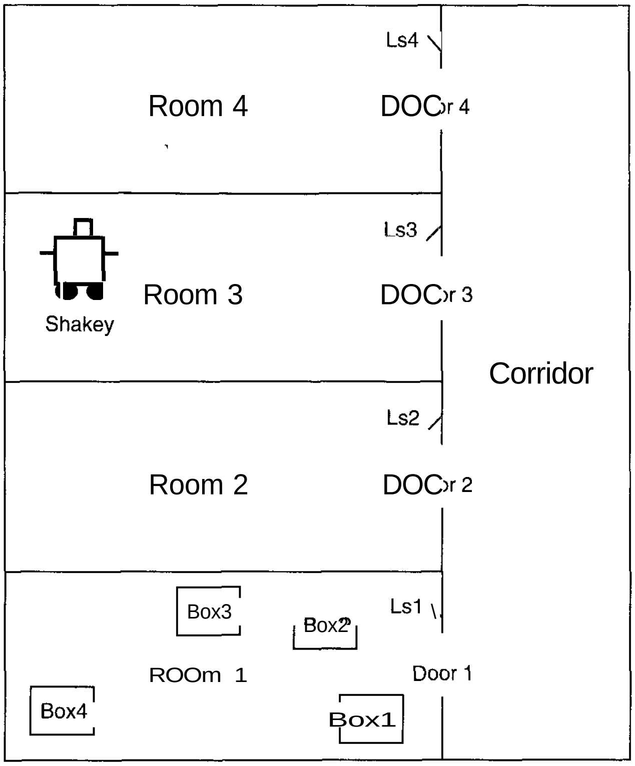

Figure 11.15 Shakey's world.

4. Climb down from a box: *Down*(*b*).
   This just undoes the effects of a *Climb*.
5. Turn a light switch on: *TurnOn*(*ls*).
   Because Shakey is short, this can only be done when Shakey is on top of a box that is at the light switch's location.[8]
6. Turn a light switch off: *TurnOff*(*ls*).
   This is similar to *TurnOn*. Note that it would not be possible to represent toggling a light switch as a STRIPS action, because there are no conditionals in the language to say that the light becomes on if it was off and off if it was on. (Section 12.4 will add conditionals to the language.)

In situation calculus, we could write an axiom to say that every box is pushable and climbable. But in STRIPS, we have to include individual literals for each box in the initial state. We also have to include the complete map of the world in the initial state, in terms of what objects are *In* which

[8] Shakey was never dextrous enough to climb on a box or toggle a switch, but STRIPS was capable of finding plans using these actions.

rooms, and which locations they are *At.* We leave this, and the specification of the operators, as an exercise.

In conclusion, it is possible to represent simple domains with STRIPS operators, but it requires ingenuity in coming up with the right set of operators and predicates in order to stay within the syntactic restrictions that the language imposes.

11.9 SUMMARY

In this chapter, we have defined the planning problem and shown that situation calculus is expressive enough to deal with it. Unfortunately, situation calculus planning using a general-purpose theorem prover is very inefficient. Using a restricted language and special-purpose algorithms, planning systems can solve quite complex problems. Thus, planning comes down to an exercise in finding a language that is just expressive enough for the problems you want to solve, but still admits a reasonably efficient algorithm. The points to remember are as follows:

- Planning agents use lookahead to come up with actions that will contribute to goal achievement. They differ from problem-solving agents in their use of more flexible representations of states, actions, goals, and plans.
- The STRIPS language describes actions in terms of their preconditions and effects. It captures much of the expressive power of situation calculus, but not all domains and problems that can be described in the STRIPS language.
- It is not feasible to search through the space of situations in complex domains. Instead we search through the space of plans, starting with a minimal plan and extending it until we find a solution. For problems in which most subplans do not interfere with each other, this will be efficient.
- The principle of least commitment says that a planner (or any search algorithm) should avoid making decisions until there is a good reason to make a choice. Partial-ordering constraints and uninstantiated variables allow us to follow a least-commitment approach.
- The causal link is a useful data structure for recording the purposes for steps. Each causal link establishes a protection interval over which a condition should not be deleted by another step. Causal links allow early detection of unresolvable conflicts in a partial plan, thereby eliminating fruitless search.
- The POP algorithm is a sound and complete algorithm for planning using the STRIPS representation.
- The ability to handle partially instantiated operators in POP reduces the need to commit to concrete actions with fixed arguments, thereby improving efficiency.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The roots of AI planning lie partly in problem solving through state-space search and associated techniques such as problem reduction and means-ends analysis, especially as embodied in Newell and Simon's GPS, and partly in theorem proving and situation calculus, especially as embodied in the QA3 theorem proving system (Green, 1969b). Planning has also been historically motivated by the needs of robotics. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these three influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI International. Its overall control structure was modeled on that of GPS, and it used a version of QA3 as a subroutine for establishing preconditions for actions. Lifschitz (1986) offers careful criticism and formal analysis of the STRIPS system. Bylander (1992) shows simple planning in the fashion of STRIPS 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.

LINEAR
NONLINEAR
NONINTERLEAVED

For several years, terminological confusion has reigned in the field of planning. Some authors (Genesereth and Nilsson, 1987) use the term **linear** to mean what we call totally ordered, and **nonlinear** for partially ordered. Sacerdoti (1975), who originated the term, used "linear" to refer to a property that we will call **noninterleaved.** Given a set of subgoals, a noninterleaved planner can find plans to solve each subgoal, but then it can only combine them by placing all the steps for one subplan before or after all the steps of the others. Many early planners of the 1970s were noninterleaved, and thus were incomplete—they could not always find a solution when one exists. This was forcefully driven home by the Sussman Anomaly (see Exercise 11.4), found during experimentation with the HACKER system (Sussman, 1975). (The anomaly was actually found by Alien Brown, not by Sussman himself, who thought at the time that assuming linearity to begin with was often a workable approach.) HACKER introduced the idea of protecting subgoals, and was also an early example of plan learning.

Goal regression planning, in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals, 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 using 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, 1975b; Tate, 1975a) also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

The construction of partially ordered plans (then called **task networks)** was pioneered by the NOAH planner (Sacerdoti, 1975; Sacerdoti, 1977), and thoroughly investigated in Tate's (1977) NONLIN system, which also retained the clear conceptual structure of its predecessor INTERPLAN. INTERPLAN and NONLIN provide much of the grounding for the work described in this chapter and the next, particularly in the use of causal links to detect potential protection violations. NONLIN was also the first planner to use an explicit algorithm for determining the truth or falsity of conditions at various points in a partially specified plan.

TWEAK (Chapman, 1987) formalizes a generic, partial-order planning system. Chapman provides detailed analysis, including proofs of completeness and intractability (NP-hardness and undecidability) of various formulations of the planning problem and its subcomponents. The POP algorithm described in the chapter is based on the SNLP algorithm (Soderland and Weld, 1991), which is an implementation of the planner described by McAllester and Rosenblitt (1991). Weld contributed several useful suggestions to the presentation in this chapter.

A number of important papers on planning were presented at the Timberline workshop in 1986, and its proceedings (Georgeff and Lansky, 1986) are an important source. *Readings in Planning* (Alien *el al.*, 1990) is a comprehensive anthology of many of the best articles in the field, including several good survey articles. *Planning and Control* (Dean and Wellman, 1991) is a good general introductory textbook on planning, and is particularly remarkable because it makes a particular effort to integrate classical AI planning techniques with classical and modern control theory, metareasoning, and reactive planning and execution monitoring. Weld (1994), provides an excellent survey of modern planning algorithms.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences, but there are also specialized conferences devoted exclusively to planning, like the Timberline workshop, the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, or the International Conferences on AI Planning Systems.

EXERCISES

11.1 Define the operator schemata for the problem of putting on shoes and socks and a hat and coat, assuming that there are no preconditions for putting on the hat and coat. Give a partial-order plan that is a solution, and show that there are 180 different linearizations of this solution.

11.2 Let us consider a version of the milk/banana/drill shopping problem in which money is included, at least in a simple way.

a. Let CC denote a credit card that the agent can use to buy any object. Modify the description of *Buy* so that the agent has to have its credit card in order to buy anything.
b. Write a *PickUp* operator that enables the agent to *Have* an object if it is portable and at the same location as the agent.
c. Assume that the credit card is at home, but *Have*(*CC*) is initially false. Construct a partially ordered plan that achieves the goal, showing both ordering constraints and causal links.
d. Explain in detail what happens during the planning process when the agent explores a partial plan in which it leaves home without the card.

11.3 There are many ways to characterize planners. For each of the following dichotomies, explain what they mean, and how the choice between them affects the efficiency and completeness of a planner.

a. Situation space vs. plan space.
b. Progressive vs. regressive.

c. Refinement vs. debugging.
d. Least commitment vs. more commitment.
e. Bound variables vs. unbound variables.
f. Total order vs. partial order.
g. Interleaved vs. noninterleaved.
h. Unambiguous preconditions vs. ambiguous preconditions.
i. Systematic vs. unsystematic.

SUSSMAN ANOMALY

11.4 Figure 11.16 shows a blocks-world planning problem known as the **Sussman anomaly.** The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Encode the problem using STRIPS operators, and use POP to solve it.

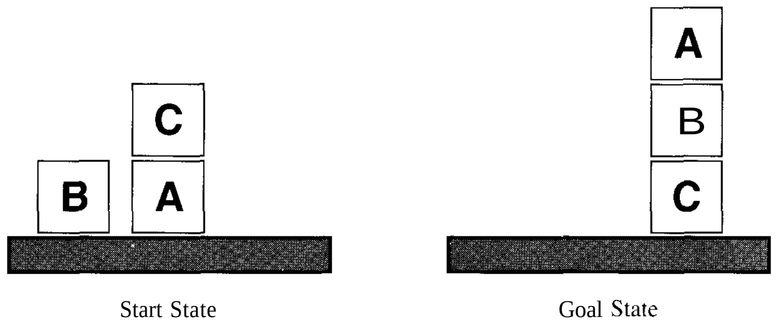

Figure 11.16 The "Sussman anomaly" blocks-world planning problem.

11.5 Suppose that you are the proud owner of a brand new time machine. That means that you can perform actions that affect situations in the past. What changes would you have to make to the planners in this chapter to accommodate such actions?

11.6 The POP algorithm shown in the text is a regression planner, because it adds steps whose effects satisfy unsatisfied conditions in the plan. Progression planners add steps whose preconditions are satisfied by conditions known to be true in the plan. Modify POP so that it works as a progression planner, and compare its performance to the original on several problems of your choosing.

11.7 In this exercise, we will look at planning in Shakey's world.

a. Describe Shakey's six actions in situation calculus notation.
b. Translate them into the STRIPS language.
c. Either manually or using a partial-order planner, construct a plan for Shakey to get *Box2* into *Room*2 from the starting configuration in Figure 11.15.
d. Suppose Shakey has *n* boxes in a room and needs to move them all into another room. What is the complexity of the planning process in terms of n?

11.8 POP is a nondeterministic algorithm, and has a choice about which operator to add to the plan at each step and how to resolve each threat. Can you think of any domain-independent heuristics for ordering these choices that are likely to improve POP's efficiency? Will they help in Shakey's world? Are there any additional, domain-dependent heuristics that will improve the efficiency still further?

11.9 In this exercise we will consider the monkey-and-bananas problem, in which there is a monkey in a room with some bananas hanging out of reach from the ceiling, but 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, *Climb* onto an object, and *Grasp* 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 in predicate calculus.
b. Write down STRIPS-style definitions of the four actions, providing at least the obvious preconditions.
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 the box is necessarily at C) in the language of 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 frame problem or the qualification problem?