CHAPTER

14 PLANNING

The first of the AI "systems" that we will consider involve *planning*. A planning system is expected to analyze the situation in which an agent finds itself and then to develop a strategy for achieving the agent's goal—presumably by finding a series of actions that can be expected to have a desirable outcome.

Planning is one of a set of problems that are known as *synthesis* problems because they involve the development of a mental object of some form—a plan, a design for a circuit, what have you. Synthesis problems are to be contrasted with the problems that we have considered thus far, which are typically "classification" problems. In a diagnostic system, the aim is to classify the observed behavior of some device in order to determine which component is failing. The small set of possible outcomes (limited by the number of potentially faulty components in a diagnostic example) makes classification problems much easier than synthesis problems. After all, the number of possible circuits is tremendously larger than the number of components out of which the circuits are constructed. The number of possible plans is much greater than the number of single actions that might be taken in any given situation.

This is not to say that planning doesn't involve knowledge representation and search, of course. Information about the domain of action still needs to be communicated to the planning system, and search is still intrinsic to finding a solution to the planning problem. What we are trying to point out here is that the space of possible solutions is much larger for synthesis problems than it is for classification problems.

It is possible, incidentally, to view planning as classification—given the current situation, the aim is only to decide what single action to take next, as opposed to developing a general plan of action that will achieve the desired goal. But solving planning problems in this way seems fruitless—it's not terribly reasonable to store facts of the form, "If you want to get to Boston, the first thing you should do is get to a phone." (To call your travel agent, presumably.) You'll look pretty silly racing to a phone if you happen to be in an airport already (just go to the ticket counter), or if you're already in Boston! In addition, plans often need to be debugged; if you merely classify your situation based on the best action to take next and then something goes wrong, you won't really have any idea of how to fix it.[26]

Before we look at planning specifically, let me point out that there is yet another class of problems that are harder than either classification or synthesis problems—learning problems. By a *learning* problem I mean a problem where the idea is to discover a new and useful concept of some kind, like the *idea* of a "bird" if all you know about is animals, feathers, flying things, and so on.

The reason that learning is so hard is that the space of possible concepts is even larger than the space of possible plans. Roughly speaking, the space of possible solutions to a synthesis problem (for example, plans) grows exponentially in the number of actions one could take. But the number of possible concepts (basically normal-form expressions using existing predicates) grows doubly-exponentially in the size of the domain.

My own view is that classification problems are basically solved, at least in the sense that once we have determined the classification function to be used, the techniques available to implement such functions are well understood. Learning problems, on the other hand, are currently too hard for us because of the enormous search involved. (But there's still lots going on, as we'll see in the next chapter.) Planning problems, however, are "just right." If I had to predict the area of AI research that will make the most progress in the next ten years, I'd pick planning. There are a tremendous number of problems that are both interesting and tractable. Research in planning as a whole should show substantial progress as progress is made on the individual issues that we will discuss shortly.

There are other reasons that planning is interesting. For one thing, it is clearly a fundamental part of intelligent behavior. All of us are capable of developing strategies to achieve our goals, and our artifacts will need to be able to do the same.

Yet another thing that makes planning interesting is how hard it is. It seems like we should be able to write a planning system by just describing the domain to a theorem prover, and then asking it to prove that a plan exists. Since the proof can be expected to be constructive, this would enable us to use our existing declarative methods to solve planning problems.

As an example, imagine that we are trying to plan to achieve some simple goal, like making breakfast in the morning. A proof that we can make pancakes would presumably involve proving that we can get milk and flour, mix them in a bowl, and then cook the pancakes on the griddle—but the discovery of this proof is equivalent to solving the original planning

26 Actually, you won't even be bothered by the fact that something's gone wrong—with only a view of what action should be taken next and no view of why, you won't realize that anything's amiss!

problem. We have seen in a variety of places in Part III that computer-generated proofs tend to be constructive, so it seems as if the universality of first-order logic should imply that a planning system can be constructed in this fashion.

This is a nice idea, but it doesn't work. And the reasons it doesn't work will be the focus of this chapter. I'll spend some time talking about potential solutions to the problems we'll be discussing, but most of the material will be about the problems themselves. After all, I said in the introduction that I'd try to focus on topics that were well established in the AI literature—and in planning, the problems are much better established than the solutions are.

14.1 GENERAL-PURPOSE AND SPECIAL-PURPOSE PLANNERS

Before turning our attention to these general issues, I should spend a little more time on the possible reduction of planning problems to classification ones. Instead of classifying a situation in terms of, which action should I take next?, what if we attempt to classify it in terms of, which plan would allow me to achieve my goal?

A lot of the planning that we do is of just this form. I typically ride my bike to work if it's sunny, but drive if it's raining or cold. Either way, I know the route I expect to take; what I'm doing when I decide how to get to work is determining whether I want to invoke my existing bike-to-work plan or my existing drive-to-work plan.

Sometimes, of course, things aren't so simple. It was sunny last week, but I was having a new floor put in in the kitchen and that meant that the cat was locked in the garage. The cat hates being locked up; if I went to get my bicycle out of the garage, the cat would probably run away. Since my car was parked on the curb, I decided to drive to work instead of bicycling.

In the normal case, I have a simple set of rules that tells me to look at the weather and decide how to get to work. This is what is known as *special-purpose* planning; I invoke a planner that is capable of solving only a very restricted problem and then use the result. In the example involving my cat, I had to invoke a planner with a much deeper understanding of my household domain in order to decide what to do; it's unlikely that the special-purpose planner that I typically use could anticipate my difficulties. Planners that rely on deep knowledge about their domain are called *general-purpose* planners.

Special-purpose planners are classification systems; general-purpose planners are synthesis systems—and it is in general-purpose planning that all the problems show up. Although many special-purpose planners have been built and used in restricted domains, the point of this particular example is to remind you that the need to resort to general-purpose methods—to plan from first principles—is never very far away.

14.2 REASONING ABOUT ACTION

A simple planning problem is shown in Figure 14.1; our goal is to get block *B* onto block *C*. We will begin our discussion of planning by examining one of the issues underlying planning—how are we to describe actions themselves?

We start by identifying the vocabulary we will use when working in this domain, which is known as the *blocks world*. We introduce a predicate loc(b,l), indicating that the block b is at the location l. l might be a location on the table, or another block, and so on. Given this, our goal is to achieve loc(B,C).

There is a single action in this domain, that of moving a block to a new location; we denote this action by move(b,l). We will initially take move to be a predicate in our domain, so that move(b,l) means, "The block b has been moved to the location l." We will further assume that the nature of the domain is such that the move action succeeds only if both the block being moved and the target location are clear when the action is attempted.

Let's not worry about the preconditions for a moment, however. How are we to say that the result of moving b to l is that b is at l? It is tempting to write

$$\texttt{move}(b,l) \rightarrow \texttt{loc}(b,l) \tag{14.1}$$

but a moment's reflection shows that this is unsatisfactory. If, for example, we move b first to a location l_1 and then to a new location l_2, we would have

$$\texttt{move}(b,l_1) \wedge \texttt{move}(b,l_2)$$

and we can conclude from this and (14.1) that b is located at both l_1 *and* l_2!

The problem with (14.1) is that we can't really talk about the effects of actions without describing the situations in which the actions take place. So what we really want to say in (14.1) is that if we perform a move action in a situation s, then the result will hold in the next situation:

$$\texttt{move}(b,l,s) \rightarrow \texttt{loc}(b,l,\texttt{next}(s)) \tag{14.2}$$

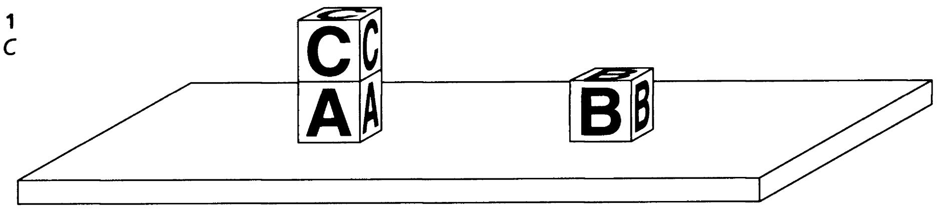

FIGURE 14.1
Move *B* to *C*

Note that we have added a third "situational" argument to the move and loc predicates.

It will be a bit more convenient if we rewrite (14.2) in a slightly different form. Rather than talk about the "next" situation, we will talk explicitly about the result of performing an action a in a situation s. Rather than the function next, we use a function result, where result(a,s) is the situation that results from performing the action a in the situation s. (14.2) now becomes

$$\texttt{loc}(b,l,\texttt{result}(\texttt{move}(b,l),s)) \qquad (14.3)$$

This axiom says explicitly that b will be located at l in the situation that results from moving b to l in s. We've changed move to a function here; move(b,l) takes a block b and a location l and produces the action of moving b to l. result is also a function.

Now: What about the preconditions? Let's introduce a new predicate clear, where clear(l,s) means that the location (or block) l is clear in the situation s. (14.3) finally becomes:

$$\texttt{clear}(b,s) \wedge \texttt{clear}(l,s) \rightarrow \texttt{loc}(b,l,\texttt{result}(\texttt{move}(b,l),s)) \qquad (14.4)$$

If, in some situation s, the block b and the location l are both clear, then b will be at l after we move it there.

Before turning to the problems with this kind of an axiomatization, let me introduce an alternative formulation of it. Rather than introduce a situational argument to all of the predicates in our domain, we can instead reify predicates like loc, making objects out of sentences such as loc(B,C). Instead of writing

$$\texttt{clear}(b,s)$$

to indicates that b is clear in s, we can write

$$\texttt{holds}(\texttt{clear}(b),s) \qquad (14.5)$$

where clear(b) is now an object of our domain instead of a sentence. What (14.5) says is that the *object* clear(b) holds in the situation s. If we do this, (14.4) becomes

$$\begin{aligned}&\texttt{holds}(\texttt{clear}(b),s) \wedge \texttt{holds}(\texttt{clear}(l),s)\\ &\qquad \rightarrow \texttt{holds}(\texttt{loc}(b,l),\texttt{result}(\texttt{move}(b,l),s))\end{aligned} \qquad (14.6)$$

We have already seen examples of reification in Chapter 13, where we used it to present a convenient description of inheritance hierarchies; (14.5) and (14.6) are another example of this technique.

Finally, let me introduce a little bit of terminology. A description like the one we have given of our domain, involving reified sentences like (14.6), uses what is known as the *situation calculus* because of the appearance of situational arguments in sentences like (14.6). The old predicates (like `clear`) that are functions in the new formulation are often called *fluents*. Thus `clear` is a fluent because `clear`(b) is an object in the reified description.

One difficulty with reification is that since we are treating expressions like `clear`(b) as objects, we have to be careful to avoid operating on them with logical operators. Thus we cannot write (14.6) as

$$\texttt{holds[clear}(b) \wedge \texttt{clear}(l),s] \rightarrow \texttt{holds[loc}(b,l),\texttt{result}(\texttt{move}(b,l),s)]$$

since conjunction is not defined as an operation on the *objects* `clear`(b) and `clear`(l).

One advantage of reification is that it allows us to quantify over the sentences (now objects) being reified. If we want to say that *nothing* holds in some situation s_{wow}, we could write this as

$$\forall f.\neg\texttt{holds}(f,s_{\text{wow}})$$

If we were not to reify the sentences in our domain, this axiom would involve quantification over predicates and would therefore not be a legitimate sentence of first-order logic. We will see a more interesting example of this use of reified sentences in Section 14.3.3.

The frame problem Suppose, then, that we decide to describe the `move` action using (14.6) or perhaps (14.4). Is this an adequate description?

Not yet. The reason is that we have no way to decide that if we move B to C in Figure 14.1, the block A will stay where it is.

This is a problem that we have already seen in the discussions of Sections 10.2.3 and 11.1.2. We observed there that any description of action needs some way to encode our knowledge that things typically stay the same from one situation to another. Making this observation precise is known as the *frame problem*, and we discuss some solutions that have been proposed in Section 14.3.

The qualification problem Even if we have found some way to address this difficulty, the resulting description is still inadequate, since there are typically "unmentioned" preconditions to most of the actions that we might consider. In our blocks world example, a precondition of `move`(b,l) is $b \neq l$, since it is impossible to put a block on top of itself. It seems, however, as if this fact should not force us to add a new precondition to the `move` action; we should be able to derive this precondition from the domain constraint

$$\forall b,s.\neg\texttt{holds}(\texttt{loc}(b,b),s)$$

In other words, given the fact that no block can ever be on top of itself, we should be able to conclude that any attempt to move it there will fail.

This is one aspect of the so-called *qualification problem:*

An action may fail because its success would involve the violation of a domain constraint.

Unfortunately, formalizing this idea precisely turns out to be very difficult.

The qualification problem refers to the fact that actions typically have preconditions that are not an explicit part of the domain description; as we've just seen, one way in which this can happen is when the precondition is a consequence of constraints on the domain in which the planner is working.

Another aspect of qualification is that actions often have preconditions that are unstated simply because they are unlikely to arise in practice. Consider an example from Chapter 11, where we noticed that you won't be able to start your car if someone has stuck a potato in the tail pipe. Is it reasonable to make the absence of the potato an explicit precondition to starting the car?[27]

There are simply too many potential preconditions of this sort for us to enumerate them in advance. Even if we could enumerate them, reasoning through these preconditions in order to decide whether or not a particular action will succeed could never be computationally viable in practice.

This sort of a qualification on the success of an action shares some features with the one mentioned earlier. Since a car with a blocked exhaust cannot run, we can derive the hidden precondition using whatever technique allowed us to realize that the action `move`(b,b) will always be unsuccessful.

But things are not so simple. Given an action `move`(b,l), we can presumably tell whether $b = l$ or not; when we go to start our car in the morning, do we know whether or not there is a potato in the tail pipe?

The conventional wisdom is that we nonmonotonically assume that actions succeed once their explicit preconditions have been satisfied. We assume that the action of starting our car in the morning will work, and conclude from that (if necessary) that there is no potato in the tail pipe.

But even this approach has problems. Certainly the conclusion is correct—we *do* assume that our car has no potato in the tail pipe. But what if our car is out of gas? Now the action of starting it has no chance of succeeding, but we still feel justified in concluding that the potato is missing.

We see from this that qualification assumptions such as the absence of

27 In the 1985 film *Beverly Hills Cop*, Eddie Murphy used a banana to block a car's exhaust. But in the 1977 paper "Epistemological Problems of Artificial Intelligence," John McCarthy discusses a potato. We use a potato in the interests of historical accuracy.

a potato are not really assumptions about actions (for example, that the action of starting the car will succeed); they're assumptions about the domain itself (for example, tail pipes are normally potato-free). This doesn't make the problem of identifying and describing these assumptions any easier, however.

The ramification problem Even if the frame and qualification difficulties are addressed, problems still remain. As an example, have a look at Figure 14.1 again. After moving *B* to *C*, how do we know that *B* is no longer located at its initial location?

This is the *ramification* problem; not only is it hard to specify exactly the things that *don't* change as actions take place and time goes by, it's hard to say what *does* change as these things happen. In the example we are considering, not only do we need to say that *B* isn't at its original location any more, we need to say that *C* is no longer clear.

Other examples are more subtle because the ramifications of successful actions can be situation-dependent. The action of moving an object typically doesn't change its color. But what about moving your car into the path of a spray paint gun? Cameras work because moving one object (the shutter) causes *another* object to change color (the film).

These, then, are the three problems in formalizing action:

1. The qualification problem. Describing precisely when actions do or do not succeed is more subtle than one might expect.
2. The frame problem. Describing precisely what stays the same when an action does succeed is more subtle than one might expect.
3. The ramification problem. Describing precisely what changes when an action succeeds is also more subtle than one might expect.

Other than that, no worries.

14.3 DESCRIPTIONS OF ACTION

In this section, we discuss some of the solutions that have been proposed in response to these difficulties.

14.3.1 Nondeclarative Methods

Probably the best-known approach is the one used by an implemented general-purpose planning system known as STRIPS. STRIPS uses a special-purpose data structure to describe actions and then manipulates this description using procedures designed specifically to solve planning problems. Because the planning process is procedural, it is reasonable to think of the STRIPS description as nondeclarative.

The data structure used by STRIPS describes actions in terms of *preconditions*, *add lists*, and *delete lists*. The preconditions of an action are those fluents that must hold in order for the action to be successful. The elements of the add list are the fluents that will hold after the action succeeds; the elements of the delete list are the fluents that cease to hold when the action succeeds. Any fluent not on an action's add or delete list is assumed to persist through the execution of that action. Since STRIPS is not a declarative system, it is free to maintain its description of the domain in any fashion it chooses; the representation used is simply a list of valid fluents. This list is modified destructively as actions occur.

It should be clear from this description that STRIPS makes no attempt to address the qualification problem; all of an action's preconditions need to be stated explicitly. The frame problem is handled via the assumption that unmentioned fluents persist from one situation to another; this is often known as the *STRIPS assumption*. The ramification problem is treated by explicitly listing all of the consequences of any action—both direct and "inferential."

Because the inferential consequences may be situation-dependent, STRIPS needs to split any action with potentially situation-dependent effects into a collection of action subtypes. In the example of the previous section, one would need the following actions:

```
          move-car-into-spray-paint
              move-car-elsewhere
    move-camera-shutter-with-film-in-camera
      move-camera-shutter-on-empty-camera
     move-other-than-car-or-camera-shutter
```

Things are clearly getting out of hand here; as the domain becomes more complicated, the number of action subtypes can be expected to grow exponentially with its size. Although STRIPS has a computational mechanism for dealing with the ramification problem, this mechanism is unlikely to be able to cope with domains of interesting size.

14.3.2 Monotonic Methods

The fact that the STRIPS representation of action is not declarative also has the consequence that information about actions cannot interact with other available information.

Think about my cat locked in the garage once again. Assuming that I hadn't previously thought about the problem of the cat escaping if I open the door, I won't know that "the cat is out" should be on the add list of the action "open the garage door with the cat locked in the garage."

The difficulty here is a result of the fact that the STRIPS action language is not one in which I can conveniently express all of my other information about the world—in this particular case, information about the fact that my cat hates being locked in the garage and so on.

A general-purpose planner like STRIPS is not as brittle as a special-purpose planner would be. But because STRIPS has no effective access to the declarative information already available to the system, some brittleness remains. Alternatively, since STRIPS makes specific assumptions about the form of the actions being described, it is unable to work with any actions that cannot be described in this way.

Here's another example. Like most other general-purpose planners, STRIPS assumes that the consequences of an action are well-defined. But what are the consequences of playing the lottery? It depends on whether you win or lose—and the planning system will hardly be able to predict this in advance! STRIPS also cannot handle simultaneous actions, actions that take more than a single unit of time to have effect, and so on. Some planning systems address some of these issues; other systems address others. But the limited expressive power of these methods remains.

Instead of trying to give a STRIPS-like description access to declarative information, an alternative approach is to use a legitimately declarative description of action, as we did in (14.4) and (14.6). The universality of first-order logic should now guarantee that we are capable of describing any possible action, and that this description interfaces easily with whatever other declarative information is available.

A description of action using first-order logic might be expected to deal with the ramification problem by saying, for example, that a block can only be in one place:

$$[\mathtt{holds}(\mathtt{loc}(b,l),s) \wedge \mathtt{holds}(\mathtt{loc}(b,l'),s)\ ] \rightarrow l = l' \qquad (14.7)$$

Given this axiom and the fact that the action of moving B to C is successful in Figure 14.1, it is possible to conclude that

$$\neg\mathtt{holds}(\mathtt{loc}(B,l_2),\mathtt{result}(\mathtt{move}(B,C),s_0)) \qquad (14.8)$$

where s_0 is the initial situation shown in the figure and l_2 is B's initial location.

The frame problem is not so simple, however. In order to specify exactly which fluents are preserved through the execution of which actions, one needs to break actions into subtypes as we did in the previous section. As an example, in order to describe situations in which moving an object doesn't change its color, we need to list explicitly exceptions involving cars, paint guns, and so on.[28] Just as the number of action subtypes grew exponentially in the size of the domain for a STRIPS system, so does the size of a monotonic axiomatization.

28 There is no *proof* that monotonic descriptions of action need to be this bad, and researchers continue to search for satisfactory monotonic axiomatizations. Section 14.6 contains pointers to some recent efforts.

14.3.3 Nonmonotonic Methods

What we would *like* to say, of course, is simply that fluents persist from one situation to the next if there is no reason to believe otherwise. So it would seem that nonmonotonic reasoning would be well-suited to the problem of describing action.

When we wanted to say that birds fly if there were no reason to believe otherwise, we wrote

$$\texttt{bird}(x) \wedge \neg \mathrm{ab}(x) \rightarrow \texttt{flies}(x)$$

If a bird is not abnormal, it can fly. We then used nonmonotonic techniques to ensure that when possible, we would conclude $\neg \mathrm{ab}(b)$ for any particular bird b.

For actions, what we would like to say is that if a fluent f holds in a situation s, then the fluent will continue to hold after we execute an action a:

$$\texttt{holds}(f,s) \wedge \neg \mathrm{ab}(a,f,s) \rightarrow \texttt{holds}(f,\texttt{result}(a,s)) \qquad (14.9)$$

(14.9) says exactly this: If f holds in s, and a is not abnormal in that it reverses the sense of f in s, then f will hold in $\texttt{result}(a,s)$. Note that the abnormality predicate appearing here has three arguments—whether or not the action causes f to cease to hold might depend on the action taken, the fluent involved, or the situation being considered.

Unfortunately, (14.9) turns out not to work. The rule (14.9) leads to multiple extensions that prevent our drawing all of the conclusions we would expect after a sequence of actions is executed; details of this are the topic of Exercise 5 at the end of this chapter. This difficulty can be addressed, however, by a suitable modification of the frame axiom (14.9); the idea is that it is not a *situation* like "the result of moving B onto C in the initial state" that should appear as an argument to ab, but a *description* of that situation, like "any state in which C is on B, B is on A, and A is on the table." This makes some intuitive sense—the fact that a fluent f is affected by the action a in the situation s should depend not on what s happens to be called, but should instead depend on the set of fluents f' for which $\texttt{holds}(f',s)$ is true.

14.4 SEARCH IN PLANNING

Having solved all of these formal problems in some way, we now need to deal with the huge *size* of the planning search space. In this section, we continue by examining some of the ideas that have been suggested in this regard.

14.4.1 Hierarchical Planning

Imagine that I am trying to get from Stanford to MIT.[29] I construct the tentative plan of driving to San Francisco airport, then flying to Logan (the airport in Boston), and finally driving from Logan to MIT.

The actions in this tentative plan aren't things I can execute directly; driving to San Francisco will involve getting to my car, starting it, and following some route between my starting location and the airport. Why, then, do I bother to form the three-step "high-level" plan that focuses the rest of my analysis?

The reason is exactly that: The high-level problem is far easier to solve than the low-level version that I must eventually confront. Once I have solved the high-level problem, my efforts when working on the low-level one will be simplified by my ability to focus on the intermediate goals of getting to and from the San Francisco and Boston airports.

This is a very general phenomenon; we form hierarchical plans all the time. In the example we are considering, my subplan of starting my car is itself hardly a primitive action. I will need to get the keys out of my pocket, insert them into the ignition, depress the clutch, turn the key, and so on. Depressing the clutch involves moving my foot to a certain location and then exerting a downward force.

One way to think of hierarchical planning is that we first solve an easier, more abstract planning problem and then use the solution to focus our attempts to solve the original problem. As we've described it, the "easy problem" is constructed by using a set of high-level actions to achieve our goal instead of doing so using the primitive actions allowed in our domain. Since the difficulty of a planning problem can be expected to be exponential in the depth of the search (that is, the length of the plan constructed) and each high-level action subsumes a sequence of low-level ones, the high-level plan should be short and constructing it should be easy.

Unfortunately, it is not always possible to flesh out the high-level plan to produce a viable low-level one. In our travel example, my car might be in the shop, there may be no available flights from San Francisco to Logan, or I may be unable to rent a car on my arrival in Boston. Note that the difficulties here are not things that will prevent my *executing* my low-level plan, but problems that will prevent my developing that plan in the first place. If I know my car to be in the shop, I will have difficulty finding a plan for driving to the airport. If there are no flights to Logan or no cars available in Boston, my travel agent will presumably inform me of the difficulty and I will need to modify my high-level plan in some way.

In some cases, it *is* possible to guarantee that the high-level plan has a low-level instantiation; we've already seen examples of this in our discussion of macro operators in Chapter 4. When we decide to flip two of

29 Why anyone would actually want to *do* this escapes me, but imagine it nevertheless.

the cubies in Rubik's cube en route to a solution, we know for sure that it will be possible to do so.

Finally, where does the high-level description of a domain come from in the first place? It can either be supplied by the user of the system or, more interestingly, be developed automatically. There are at least three ways in which this can be done.

In the first approach, the system can construct high-level actions by simply dropping some of the preconditions of lower-level ones. In the blocks world, perhaps we should first find a plan for achieving our goal by assuming that we can move blocks even if their intended destinations are not clear. After all, we can always clear these destinations off, and making this assumption will undoubtedly simplify the problem. This is the approach taken by an early hierarchical planner known as ABSTRIPS, but it appears not to be terribly useful on realistic problems.

The second approach involves examining previous planning efforts in order to identify "chunks" of low-level actions that should be grouped as potential high-level actions in future planning; this is the approach taken by a general problem-solving system called SOAR. However, since this is more a learning issue than a planning one *per se*, we will defer its discussion to the next chapter.

The final possibility is to use default information that is already in our domain to describe the high-level plan. Perhaps we have a default saying that one can typically drive anywhere close or fly between any two large cities; it is now this information that tentatively validates our high-level plan for getting to MIT.

14.4.2 Subgoal Ordering and Nonlinear Planning

Another simple blocks-world problem appears in Figure 14.2, where the goal is to get A on B and B on C. Which of these goals should we achieve first?

To a human planner, it's obvious that we have to get B on C first; towers have to be built from the bottom up. But an automated planner has no such intuition and is as likely to begin by putting A on B as otherwise.

To see why this is such a problem, imagine that we are trying to construct a larger tower involving the blocks $b_0, \ldots, b_n$. Of the $n!$ possible orders in which we might try to achieve the n subgoals of getting b_{i-1} onto b_i,

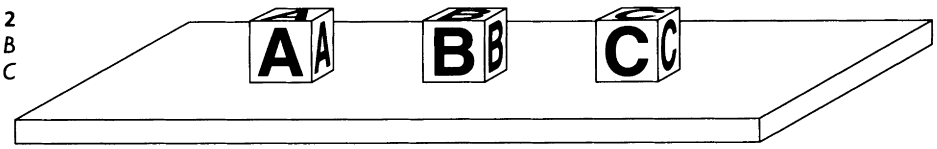

FIGURE 14.2 Get A on B and B on C

only one will be satisfactory. Is there any way to reduce the size of the search space so that these $n!$ attempts are not examined separately?

There is. Suppose that instead of viewing a "plan" as an ordered sequence of actions, we view it as a *partially* ordered sequence of actions. We don't commit to putting b_2 on b_3 before putting b_1 on b_2 until we realize that is the order in which these subgoals need to be achieved.

In more general terms, suppose that we know that our plan will include the two actions a_1 and a_2. We do not commit to either of these actions being before the other until information about the domain forces us to do so. In this particular example, a_1 would be the action of putting A on B, and a_2 the action of putting B on C. When we realize that the success of a_1 will cause the failure of one of the preconditions to a_2 (that B be clear), we add a constraint that the action a_1 must follow a_2 in the final plan. (Each of these might still be unordered with respect to a third action a_3, of course.)

Completely ordered plans are often called *linear* plans; the approach in which actions are initially unordered and constraints are established to order them is known as *nonlinear* planning.

The most successful nonlinear planning system is David Wilkins's SIPE. Unlike many other general-purpose planning systems, SIPE has been used in a practical application—the control of an Australian beer factory. SIPE's ability to handle the complexities involved in this application are a testimony to the power of nonlinear planning.[30]

In spite of the practical success of nonlinear planning systems, there are some important theoretical arguments against them. The first is that determining the consequences of a nonlinear plan can be NP-hard.

To see this, suppose that we have a nonlinear plan involving n actions and constraints among them. Of the $n!$ possible "linearizations" of the nonlinear plan, some of these will comply with the constraints and others won't. In general, if we want to know if some fluent f holds after the nonlinear plan has been executed, there is no more efficient approach than to work our way through each of the $n!$ possible linearizations and see if f holds after each one that complies with the partial ordering information.

The trade-off here is actually pretty obvious: Nonlinear plans are more expensive to compute with than their linear counterparts, but using them reduces the size of the search space. Whether or not they lead to overall performance improvements is in some sense an experimental question, and Wilkins's success with SIPE shows that the answer is yes in at least one interesting application.

In actuality, however, the technique people appear to use to reduce the size of the planning search space involves not nonlinearity but our ability

30 In order to get SIPE to work in this domain, Wilkins had to develop a parallel version of his planner, which he tentatively named P-SIPE. Wilkins's superiors at SRI, however, realized that "pissup" was Australian slang for a wild, drunken party, and they demanded that the system be called SIPE II instead.

to *debug* plans that fail for some reason. When we are unable to achieve our goal in Figure 14.2 of getting *A* on *B* and *B* on *C* by getting *A* on *B* first, we analyze the *reason* for the failure and use this to focus our subsequent planning efforts.

Planners need to debug the plans they construct whether they are linear or not—when it comes time to execute a plan in a real environment, environmental uncertainty will occasionally cause the plan to fail for an unexpected reason. It will almost invariably be more efficient to address the difficulty by debugging the original plan than by constructing a new one from scratch. Just as nonlinear planners appear to be more effective in practice than their linear counterparts, it seems likely that debugging planners will be more efficient still.

14.4.3 Subgoal Interaction and the Sussman Anomaly

The final problem we will discuss is shown in Figure 14.3; the initial situation is the same as that in some of our earlier examples, but the goal is now to get *B* on *C* and *A* on *B*. Which of these two goals are we to achieve first?

In fact, we can't achieve *either* subgoal first. If we begin by putting *B* on *C*, we'll have to take it off in order to get *A* onto *B*. If we begin by putting *A* on *B*, we'll have to take *it* off in order to get *B* onto *C*. The best plan here is to interleave the plan for getting *B* onto *C* (put it there) with the plan for getting *A* onto *B* (move *C* out of the way and then put *A* on *B*).

This problem is known as the *Sussman anomaly*. In general, it seems that subgoals *do* interact; when they do, we have no trouble achieving our overall goals by interleaving the plans for the subgoals. Automated planners need to behave similarly.

14.5 IMPLEMENTING A PLANNER

The fact that we have described planning in terms of problems as opposed to solutions may have given you the wrong impression; although planning is hard, progress is being made. So I'd like to end this chapter by describing a working planner in some detail. The system we'll present is not declar-

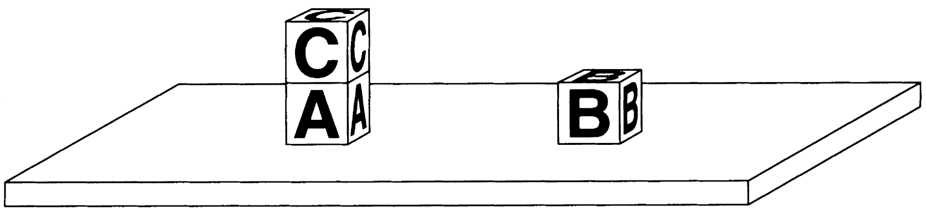

FIGURE 14.3 The Sussman anomaly: Get *B* on *C* and *A* on *B*

ative, or nonlinear, or any of those other fancy things, but it does get the job done.

Following STRIPS, we will describe an action a in terms of a set of preconditions $P(a)$, an add list $A(a)$, and a delete list $D(a)$. A situation S will be described simply by listing those fluents that hold in S. We can now make the following definitions:

DEFINITION 14.5.1 *Given a situation S and an action* a*, we will say that the action* succeeds *in that situation if and only if* $P(a) \subseteq S$*, so that every precondition to the action holds in the situation. The* result *of executing an action* a *that succeeds in the situation S is given by the situation*

$$\texttt{result}(a,S) = [S - D(a)] \cup A(a)$$

Given an action sequence $a_1, \ldots, a_n$ *and a situation S, we set* $S_0 = S$ *and* $S_i = \texttt{result}(a,S_{i-1})$ *for* $i = 1, \ldots, n$ *so that situation* n *is the result of applying action* a_n *in situation* $n - 1$*. We will say that the action sequence* succeeds *if every individual action succeeds, and that the action sequence* achieves the goal g *if* $g \in S_n$.

Our domain will be the blocks world as we have described it earlier in this chapter. There are two actions in this domain, `move` and `move-to-table`. The action `move(x,y,z)` moves the block x from the location y onto the block z; `move-to-table(x,y)` moves the block x from y onto the table. Here are the formal descriptions:

a	*P(a)*	*A(a)*	*D(a)*

`move(x,y,z)`	`clear(x)`
	`loc(x,y)`
	`clear(z)`
		`loc(x,z)`
		`clear(y)`
			`loc(x,y)`
			`clear(z)`

`move-to-table(x,y)`	`clear(x)`
	`loc(x,y)`
		`loc(x,table)`
		`clear(y)`
			`loc(x,y)`

Note the compromises we have had to make in order to use our simple STRIPS-like language: We have had to split the `move` action into two sub-actions, since the table is always assumed to be clear and there is no third precondition to the action of moving a block to the table. We have had to specify the blocks' starting locations as well as their ending locations, so that we know to include `clear(y)` in the add list of `move(x,y,z)`, saying that y will be clear after x is moved away. This change has also forced us to add additional preconditions to the `move` actions, saying that the blocks really are starting out where we expect them to be. Finally, we have had to treat `clear` explicitly, as opposed to defining a clear block as one with nothing else on it and letting the system deduce information about `clear` as a ramification of the `move` actions.

Rather than present an algorithm for planning using these ideas, we'll describe the search space associated with planning problems; the ideas of Part II can then be used to construct an implementation.

The nodes in our search space will be labelled with goals that our planner has yet to achieve; we'll construct our action sequences beginning with the last action (which presumably achieves our goal) and working backwards, adding actions that achieve the preconditions of later ones.

It is now tempting to describe the planning search space by labelling a node with a partial action sequence $\mathcal{A}$ and a set Z of still-to-be-achieved goals. Here is a more formal description:

DEFINITION 14.5.2 *Given an initial situation S and a set G of goals, the* STRIPS *search space associated to S and G is as follows:*

1. *The root node of the search space is ([],G), where the first element of the pair is an action sequence (empty, since we haven't thought of any actions yet) and the second element is a list of goals that have yet to be achieved.*
2. *A node $([a_1,\ldots,a_n],Z)$ is a goal node if $Z \subseteq S$, so that all of the remaining goals hold in the initial situation.*
3. *Given a node $([a_1,\ldots,a_n],Z)$, let a be any action that adds an element of Z, so that $A(a) \cap Z \neq \emptyset$. Then a successor node to $([a_1,\ldots,a_n],Z)$ is the node*

$$([a,a_1,\ldots,a_n],Z - A(a) \cup P(a))$$

where a has been added to the front of the action sequence, the goals that a adds have been removed from Z and a's preconditions have been added to Z as new subgoals.

As an example, we show in Figure 14.4 a solution to the planning

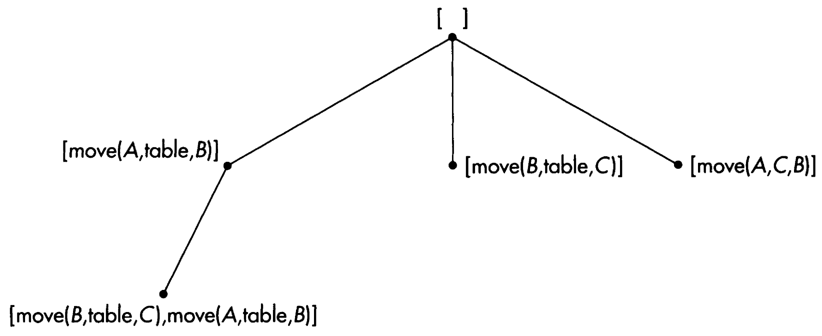

FIGURE 14.4 Solving the problem of Figure 14.2

problem of Figure 14.2. (In the interests of conserving space, only the actions are shown.) We begin with the root node

$$([],\{\texttt{loc}(A,B),\texttt{loc}(B,C)\})$$

Now adding the action `move(A,table,B)` gives us the node

$$([\texttt{move}(A,\texttt{table},B)],\{\texttt{clear}(A),\texttt{clear}(B),\texttt{loc}(A,\texttt{table}),\texttt{loc}(B,C)\})$$

where the goal `loc(A,B)` has been replaced with subgoals that are the preconditions to the `move` action. Next, we generate the node

$$([\texttt{move}(B,\texttt{table},C),\texttt{move}(A,\texttt{table},B)],$$
$$\{\texttt{clear}(A),\texttt{clear}(B),\texttt{clear}(C),\texttt{loc}(A,\texttt{table}),\texttt{loc}(B,\texttt{table})\})$$

in a similar way; since all of the new subgoals actually hold in the initial situation, we return `[move(B,table,C),move(A,table,B)]` as a solution to our planning problem.

Unfortunately, Definition 14.5.2 isn't right! In Figure 14.5, we show another "solution" to this problem that begins by putting A on B, and then puts B on C afterwards—but this plan won't work, since the action of moving B to C will fail if A is on top of B.

Our description of the problem has failed to take into account the fact that actions also delete things; one of the things they may delete is a solved precondition to another action. We need to modify Definition 14.5.2 as follows:

DEFINITION 14.5.3 *Given an initial situation S and a set G of goals, the* STRIPS *search space associated to S and G is as follows:*

1. *The root node of the search space is ([],G).*
2. *A node $([a_1,\ldots,a_n],Z)$ is a goal node if $Z \subseteq S$, so that all of the remaining goals hold in the initial situation.*

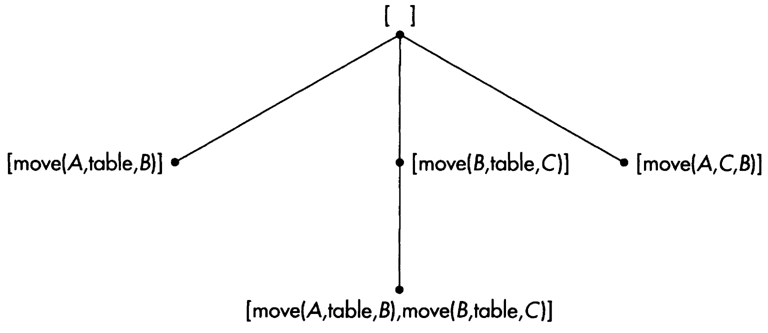

FIGURE 14.5 Not solving the problem of Figure 14.2

3. *Given a node* $([a_1,\ldots,a_n],Z)$, *let* a *be any action that adds an element of* Z, *so that* $A(a) \cap Z \neq \emptyset$, ***and deletes no goal in*** Z, ***so that*** $D(a) \cap Z = \emptyset$. *Then a successor node to* $([a_1,\ldots,a_n],Z)$ *is the node*

$$([a,a_1,\ldots,a_n],Z - A(a) \cup P(a))$$

where a *has been added to the front of the action sequence, the goals that* a *adds have been removed from* Z *and* a*'s preconditions have been added to* Z.

The effect of the change is to disallow any action that would delete a goal g that we were hoping to achieve earlier in the action sequence (that is, later in the planning process). If we want to take such an action, it must be followed by another action that achieves g.[31]

Given this change, we can no longer "derive" the bad plan of moving *A* onto *B* and then *B* onto *C*. If we make move (*B*, table, *C*) the final action of the plan, we get the search node

$$([\texttt{move}(B,\texttt{table},C)\ ],\{\texttt{clear}(B),\texttt{clear}(C),\texttt{loc}(B,\texttt{table}),\texttt{loc}(A,B)\}) \tag{14.10}$$

and now we cannot add the action move(*A*,*B*), since this deletes the subgoal clear(*B*). Expanding the node (14.10) will therefore require that we add an explicit action that takes place after move (*A*, table, *B*) and that clears off the top of *B*.

14.6 FURTHER READING

Planning is one of the most active research areas in AI and the amount of material published about it is enormous. Let me see if I can give you pointers to some of the more interesting stuff.

There are two areas of planning that have been left fairly untreated in this chapter: reactive planning and case-based planning. Reactive planners do indeed reduce planning to classification by computing in advance correct responses to any situation in which the planner might find itself; to act, we simply look up our current situation in the database and execute the associated action.

The advantage of this approach is that it is possible to bound the amount of time the system will spend thinking about a planning problem (it's just a database lookup); this is important if the system is to function in the real

31 In terms of conventional planning terminology, the troublesome action is said to *clobber* the goal g. The subsequent action that reinstates g is called a *white knight* [Chapman 1987].

world. The disadvantage is that the number of possible situations is simply too large for these ideas to be useful in practice. A debate on this question can be found in *AI Magazine* [Ginsberg 1989, and the replies to it]; I believe that the view these days is that although some measure of reactivity is needed by intelligent agents, complete prior analysis of all possible situations is impractical in fielded systems.

Related to reactive planning is *dynamic* planning, which concerns the problem of constructing plans that work in environments that change in unpredictable ways (as all real environments do, when it comes right down to it). GUARDIAN, for example, is a system that monitors critically ill medical patients [Hayes-Roth *et al.* 1989]; PHOENIX constructs plans for dealing with fires in Yellowstone National Park [Cohen *et al.* 1989]. TILEWORLD is a program that provides a dynamic testbed environment for researchers in this area [Pollack and Ringuette 1990].

In case-based planning [Hammond 1990], the basic idea is to respond to a planning problem by retrieving a solution from a library of "canned" plans. Although not exactly applicable to the situation in which the planner finds itself, the retrieved plan should at least be close, and is then debugged using domain-specific information to make it usable in the situation that is actually at hand. The problem is that plan debugging is hard and there has been very little progress made on it, at least so far.

With regard to the ideas that we have presented: The situation calculus was introduced by McCarthy and Hayes [1969] and Green [1969]. The frame problem first appears in McCarthy and Hayes [1969]; the qualification problem in McCarthy [1977]. The ramification problem has been around for a while, but is first referred to by this name by Finger [1987]. The blocks world was introduced in Terry Winograd's Ph.D. thesis, later republished [Winograd 1972].

The original reference for STRIPS is Fikes and Nilsson [1971]. The SIPE system, which we also discussed in the text, is described in more detail by Wilkins [1988]; other planning systems of considerable theoretical importance are TWEAK [Chapman 1987] and the work of McAllester and Rosenblitt [1991], both of which present clear formal descriptions of conditions under which fluents hold in nonlinear plans. All of these systems are nondeclarative.

A variety of authors have recently attempted to provide monotonic descriptions of action; the most important papers here are those by Elkan [1990], Reiter [1991], and Schubert [1990]. The basic idea in all of this work is to write monotonic axioms of the form, "If a fluent f has changed value, then an action a must have occurred." It is then possible to reason from the fact that a did not occur to the conclusion that f is unchanged.

Nonmonotonic descriptions of action currently seem the most promising; there was an enormous flurry of activity in the late 1980s when Hanks and McDermott [1987] introduced the "Yale shooting problem," which we describe in Exercise 5 at the end of this chapter. The solution sketched in the text is due to Baker [1991].

Hierarchical planners are introduced in Fikes *et al.* [1972]; recent work on the automatic construction of these hierarchies can be found in Knoblock [1990] and Ginsberg [1991b].

Finally, nonlinear planning began with Sacerdoti's work on NOAH [Sacerdoti 1977], and has continued with TWEAK and SIPE. It is shown in Dean and Boddy [1988b] that determining whether or not a fluent holds after the execution of a nonlinear plan can be exponentially difficult; a variety of authors (SIPE is typical) have dealt with this in practice by restricting the temporal language somewhat so that this determination can be made quickly. A general comparison of the efficiencies of linear and nonlinear planning can be found in Minton *et al.* [1991]. The Sussman anomaly first appears in Sussman [1975].

14.7 EXERCISES

1. Emanuel Lasker was the world chess champion from 1894–1921. When asked how many moves deep he looked when searching for a move, he replied, "Only one. But it's always the best move." Discuss this in the context of planning, game tree search, and classification problems.

2. In the text, we described as a disadvantage of reification the fact that one had to be careful not to use logical operators on the reified predicates; an advantage is that one can quantify over these predicates. What are the analogs of these observations in the reification examples of the previous chapter?

3. Prove that the number of subaction types needed by a STRIPS description of action can grow exponentially with domain size.

4. (a) What axioms are needed to describe the initial situation s_0 that appears in Figure 14.1?
   (b) Prove (14.8) from these axioms and others in the text.
   (c) What similar axioms would you need to conclude that C is not clear after moving B to C in Figure 14.1?

5. Consider a domain that contains two actions, `shoot` and `wait`. There are also two fluents, `loaded` and `alive`. The first of these means that our gun is loaded; the second means that our intended victim is alive.

If the gun is loaded, then shooting it has the intended effect:

$$\texttt{holds}(\texttt{loaded},s) \rightarrow \neg\texttt{holds}(\texttt{alive},\texttt{result}(\texttt{shoot},s))$$

Now suppose that s_0 is an initial situation in which the gun is loaded:

$$\texttt{holds}(\texttt{loaded},s_0)$$

(a) Assuming that we use the default description of the frame axiom given by (14.9), does

$$\neg\texttt{holds}(\texttt{alive},\texttt{result}(\texttt{shoot},\texttt{result}(\texttt{wait},s_0)))$$

hold in every extension of the default theory? (Hint: You may want to use the result of Exercise 9 in Chapter 11.)

(b) Does this seem intuitively reasonable?

This problem is known as the *Yale shooting problem* and first appeared in Hanks and McDermott [1987].

6. **Conditional planning**

(a) Suppose that instead of a move action, we have two actions `stack`(x,y) and `flip`(x,y). The action `stack` has no preconditions and results in a situation in which either x is on y or vice versa:

$$\begin{aligned}&\texttt{holds}[\texttt{loc}(x,y),\texttt{result}(\texttt{stack}(x,y),s)]\\ &\qquad \vee\ \texttt{holds}[\texttt{loc}(y,x),\texttt{result}(\texttt{stack}(x,y),s)]\end{aligned}$$

Meanwhile, the action `flip`(x,y) flips the blocks x and y provided that x is on y when it is attempted:

$$\texttt{holds}[\texttt{loc}(x,y),s] \rightarrow \texttt{holds}[\texttt{loc}(y,x),\texttt{result}(\texttt{flip}(x,y),s)]$$

Prove (using resolution or some other means) that there is a plan to get A onto B. What does this plan correspond to in more conventional terms?

(b) I live in a house with two bathrooms; suppose that a terrorist has planted a bomb in one of them. If I flush the toilet with the bomb in it, the bomb will be defused. I don't have time to flush both toilets, though; if I guess wrong, a timer on the bomb in the other toilet will cause it to explode.

i. Prove that there is a plan consisting of a single action that will defuse the bomb.

ii. Does the plan seem reasonable?

7. **Hierarchical planning and island-driven search.** Island-driven search was discussed in Exercise 11 in Chapter 3.
   (a) Show that hierarchical planning can be viewed as a special case of island-driven search.
   (b) We sometimes do not know that it will be possible to extend a plan from one level of the hierarchy to the next lower one. Is hierarchical planning likely to be computationally effective in these situations?
   (c) Give an example of a planning domain that shows this sort of behavior.

8. **Conjunctive subgoals and the frame axiom.** This problem concerns the blocks world as described by (14.6) and the (admittedly flawed!) frame axiom (14.9).
   (a) In the initial situation s_0, there are three blocks, A, B, and C. They are all clear and are located at locations l_1, l_2, and l_3. Describe this situation using the `holds` predicate.
   (b) Suppose that our goal is to get A on B and B on C. Not realizing that the subgoals interact, we try putting A on B first; it is no longer possible to put B on C.
      i. What axiom do we need to conclude that the second `move` action will fail after we put A on top of B?
      ii. Given the above axiom, what crucial instance of the frame axiom is violated for the action of moving A onto B in the initial situation s_0?
   (c) In general, suppose that we have a conjunctive goal $g_1 \wedge g_2$, and know that we can achieve g_1 because the preconditions $p_1, \ldots, p_m$ hold, and we can achieve g_2 because the preconditions $q_1, \ldots, q_n$ hold. If the action sequence that achieves g_1 is $a_1, \ldots, a_k$, what frame assumptions do we need to make in order to be able to conclude that we can still achieve g_2 after achieving g_1?

9. Imagine that we are trying to bolt two metal parts together and that they need to be positioned precisely relative to one another. If we put the bolt in and tighten it, we will be unable to position the parts after doing so; if we position the pieces first, we won't have a free hand to insert the bolt.

   The typical solution to this is to insert the bolt and tighten it only enough to produce some friction between the parts being connected. They are then positioned carefully and the bolt is tightened the rest of the way.

   What does all of this have to do with the Sussman anomaly?

10. Use Definition 14.5.3 to write a program that accepts as inputs a situation S, a group of action descriptions, an action sequence $\mathcal{A}$, and a

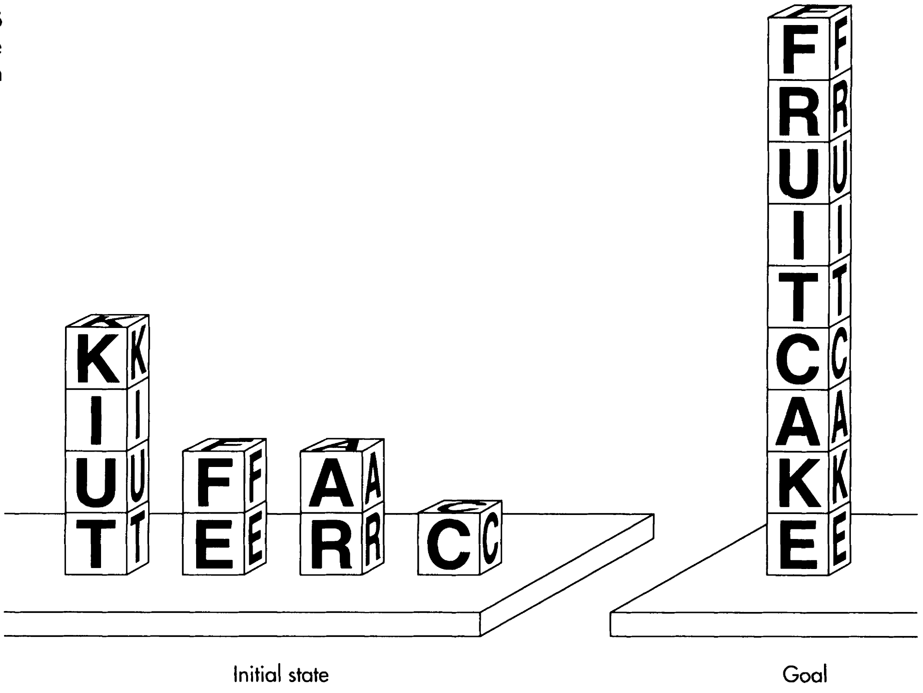

FIGURE 14.6
The fruitcake problem

goal g and determines whether or not the sequence succeeds in achieving the goal. Use the implementation to check a plan that solves the problem in Figure 14.6.

11. In the description of action in Section 14.5, why couldn't we have added `clear(table)` to our domain description and avoided splitting the `move` action in two?

12. Suppose that we were interested in constructing plans forward from the initial situation, instead of backward from our goal. In other words, a planning problem would consist of an initial situation S and a goal g; a node in the search space would be simply a successful action sequence

$$\mathcal{A} = [a_1, \ldots, a_n]$$

The successors to this node are obtained simply by adding a new action to the end of $\mathcal{A}$, where the new action is required to be successful in the situation resulting from executing the sequence $\mathcal{A}$ in the original situation S. A node is a goal node if the associated action sequence succeeds in achieving the goal g.

(a) What is the root node of the search space in this description?

(b) Describe the search space used by this approach to solve the problem appearing in Figure 14.2.

(c) How does the branching factor for this approach compare with that of an approach based on the goal-directed ideas of Section 14.5?

13. (a) Show in detail the search space the planner of Section 14.5 would use to solve the Sussman anomaly shown in Figure 14.3.

    (b) Show a path from the root node to the goal in the search space associated with Figure 14.6.

14. Suppose that the root node in the search space of Definition 14.5.3 is $([],G)$ and that we manage to reach the goal node $(\mathcal{A}, Z)$. Prove that the action sequence $\mathcal{A}$ is successful and that it achieves each goal $g \in G$.

15. (a) Find a goal G and an action sequence $\mathcal{A}$ that achieves it but does not appear in the search space given by Definition 14.5.3.

    (b) Prove that if $\mathcal{A}$ is an action sequence that achieves a goal G, there is always some subsequence of $\mathcal{A}$ that achieves G and does appear in the search space given by Definition 14.5.3.

16. Implement a planner based on the ideas of Section 14.5 and use it to solve some of the blocks-world problems we have described.