Chapter 4

State-Space Planning

4.1 Introduction

The simplest classical planning algorithms are *state-space search algorithms.* These are search algorithms in which the search space is a subset of the state space: Each node corresponds to a state of the world, each arc corresponds to a state transition, and the current plan corresponds to the current path in the search space.

In this chapter, Section 4.2 discusses algorithms that search forward from the initial state of the world to try to find a state that satisfies the goal formula. Section 4.3 discusses algorithms that search backward from the goal formula to try to find the initial state. Section 4.4 describes an algorithm that combines elements of both forward and backward search. Section 4.5 describes a fast domain-specific forward-search algorithm.

4.2 Forward Search

One of the simplest planning algorithms is the Forward-search algorithm shown in Figure 4.1. The algorithm is nondeterministic (see Appendix A). It takes as input the statement $P = (O, s_0, g)$ of a planning problem $\mathcal{P}$. If $\mathcal{P}$ is solvable, then Forward-search(O, s_0, g) returns a solution plan; otherwise it returns failure.

The plan returned by each recursive invocation of the algorithm is called a *partial solution* because it is part of the final solution returned by the top-level invocation. We will use the term *partial solution* in a similar sense throughout this book.

Although we have written Forward-search to work on classical planning problems, the same idea can be adapted to work on any planning problem in which we can (1) compute whether or not a state is a goal state, (2) find the set of all actions applicable to a state, and (3) compute a successor state that is the result of applying an action to a state.

Example 4.1 As an example of how Forward-search works, consider the DWR problem whose initial state is the state s_5 shown in Figure 2.3 and whose goal formula is

```
Forward-search(O, s0, g)
    s ← s0
    π ← the empty plan
    loop
        if s satisfies g then return π
        applicable ← {a | a is a ground instance of an operator in O,
                          and precond(a) is true in s}
        if applicable = ∅ then return failure
        nondeterministically choose an action a ∈ applicable
        s ← γ(s, a)
        π ← π.a
```

Figure 4.1 A forward-search planning algorithm. We have written it using a loop, but it can easily be rewritten to use a recursive call instead (see Exercise 4.2).

$g = \{\text{at(r1, loc1)}, \text{loaded(r1, c3)}\}$. One of the execution traces of Forward-search chooses $a = \text{move(r1, loc2, loc1)}$ in the first iteration of the loop and $a = \text{load(crane1, loc1, c3, r1)}$ in the second iteration of the loop, producing the state s_6 shown in Figure 2.4. Since this state satisfies g, the execution trace returns:

$$\pi = \langle \text{move(r1, loc2, loc1)}, \text{load(crane1, loc1, c3, r1)} \rangle$$

There are many other execution traces, some of which are infinite. For example, one of them makes the following infinite sequence of choices for a:

move(r1, loc2, loc1)

move(r1, loc1, loc2)

move(r1, loc2, loc1)

move(r1, loc1, loc2)

...

■

4.2.1 Formal Properties

Proposition 4.1 Forward-search *is sound, so any plan* π *returned by* Forward-search(O, s_0, g) *is a solution for the planning problem* (O, s_0, g).

Proof The first step is to prove that at the beginning of every loop iteration:

$$s = \gamma(s_0, \pi)$$

For the first loop iteration, this is trivial since π is empty. If it is true at the beginning of the ith iteration, then because the algorithm has completed $i-1$ iterations, there are actions $a_1, \ldots, a_{i-1}$ such that $\pi = \langle a_1, \ldots, a_{i-1}\rangle$ and states $s_1, \ldots, s_{i-1}$ such that for $j = 1, \ldots, i-1$, $s_j = \gamma(s_{j-1}, a_j)$. If the algorithm exits at either of the return statements, then there is no $(i+1)$th iteration. Otherwise, in the last three steps of the algorithm, it chooses an action a_i that is applicable to s_{i-1}, assigns

$$\begin{aligned} s &\leftarrow \gamma(s_{i-1}, a_i) \\ &= \gamma(\gamma(s_0, \langle a_1, \ldots, a_{i-1}\rangle), a_i) \\ &= \gamma(s_0, \langle a_1, \ldots, a_i\rangle), \end{aligned}$$

and assigns $\pi \leftarrow \langle a_1, \ldots, a_i\rangle$. Thus $s = \gamma(s_0, \pi)$ at the beginning of the next iteration.

If the algorithm exits at the first return statement, then it must be true that s satisfies g. Thus, because $s = \gamma(s_0, \pi)$, it follows that π is a solution to (O, s_0, g). ■

Proposition 4.2 *Let $\mathcal{P} = (O, s_0, g)$ be a classical planning problem, and let Π be the set of all solutions to $\mathcal{P}$. For each $\pi \in \Pi$, at least one execution trace of* Forward-search(O, s_0, g) *will return π.*

Proof Let $\pi_0 = \langle a_1, \ldots, a_n\rangle \in \Pi$. We will prove that there is an execution trace such that for every positive integer $i \leq n+1$, $\pi = \langle a_1, \ldots, a_{i-1}\rangle$ at the beginning of the ith iteration of the loop (which means that the algorithm will return π_0 at the beginning of the $(n+1)$th iteration). The proof is by induction on i.

- If $i = 0$, then the result is trivial.
- Let $i > 0$, and suppose that at the beginning of the ith iteration, $s = \gamma(s_0, \langle a_1, \ldots, a_{i-1}\rangle)$. If the algorithm exits at either of the return statements, then there is no $(i+1)$st iteration, so the result is proved. Otherwise, $\langle a_1, \ldots, a_n\rangle$ is applicable to s_0, so $\langle a_1, \ldots, a_{i-1}, a_i\rangle$ is applicable to s_0, so a_i is applicable to $\gamma(s_0, \langle a_1, \ldots, a_{i-1}\rangle) = s$. Thus $a_i \in E$, so in the nondeterministic choice, at least one execution trace chooses $a = a_i$. This execution trace assigns

$$\begin{aligned} s &\leftarrow \gamma(s_0, \gamma(\langle a_1, \ldots, a_{i-1}\rangle), a_i) \\ &= \gamma(s_0, \langle a_1, \ldots, a_{i-1}, a_i\rangle) \end{aligned}$$

so $s = \gamma(s_0, \langle a_1, \ldots, a_{i-1}, a_i\rangle)$ at the beginning of the $(i+1)$st iteration. ■

One consequence of Proposition 4.2 is that Forward-search is complete. Another consequence is that Forward-search's search space is usually much larger than it needs to be. There are various ways to reduce the size of the search space by modifying the algorithm to *prune* branches of the search space (i.e., cut off search below these branches). A pruning technique is *safe* if it is guaranteed not to prune every solution; in this case the modified planning algorithm will still be complete. If we have some notion of plan optimality, then a pruning technique is *strongly safe* if there is at least one optimal solution that it doesn't prune. In this case, at least one trace of the modified planning algorithm will lead to an optimal solution if one exists.

Here is an example of a strongly safe pruning technique. Suppose the algorithm generates plans π_1 and π_2 along two different paths of the search space, and suppose π_1 and π_2 produce the same state of the world s. If π_1 can be extended to form some solution $\pi_1 \cdot \pi_3$, then $\pi_2 \cdot \pi_3$ is also a solution, and vice versa. Thus we can prune one of π_1 and π_2, and we will still be guaranteed of finding a solution if one exists. Furthermore, if we prune whichever of π_1 and π_2 is longer, then we will still be guaranteed of finding a shortest-length solution if one exists.

Although the above pruning technique can remove large portions of a search space, its practical applicability is limited due to the following drawback: it requires us to keep track of states along more than one path. In most cases, this will make the worst-case space complexity exponential.

There are safe ways to reduce the branching factor of Forward-search without increasing its space complexity, but most of them are problem-dependent. Section 4.5 is an example.

4.2.2 Deterministic Implementations

Earlier we mentioned that in order for a depth-first implementation of a nondeterministic algorithm to be complete, it needs to detect and prune all infinite branches. In the Forward-search algorithm, this can be accomplished by modifying the algorithm to incorporate a simple loop-checking scheme: Keep a record of the sequence $(s_0, s_1, \ldots, s_k)$ of states on the current path, and return failure whenever there is an $i < k$ such that $s_k = s_i$. This will prevent sequences of assignments such as the one described in Example 4.1 (see page 69), but there are some domains in which the second modification will prune infinite sequences sooner than the first one.

To show that the modification works correctly, we need to prove two things: (1) that it causes the algorithm to return failure on every infinite branch of the search space, and (2) that it does not cause the algorithm to return failure on every branch that leads to a shortest-length solution.

To prove (1), recall that classical planning problems are guaranteed to have only finitely many states. Thus, every infinite path must eventually produce some state s_k that is the same as a state s_i that previously occurred on that path—and whenever this occurs, the modified algorithm will return failure.

To prove (2), recall that if the modification causes the algorithm to return failure, then there must be an $i <, k$ such that $s_k = s_i$. If the current node in the search tree is part of a successful execution trace, then the sequence of states for that trace will be

$$\langle s_0, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_n \rangle,$$

where n is the length of the solution. Let that solution be $p = \langle a_1, \ldots, a_n \rangle$, where $s_{j+1} = \gamma(s_j, a_{j+1})$ for $j = 0, \ldots, n-1$. Then it is easy to prove that the plan $p' = \langle a_1, \ldots, a_{i-1}, a_k, a_{k+1}, \ldots, a_n \rangle$ is also a solution (see Exercise 4.3). Thus, p cannot be a shortest-length solution.

4.3 Backward Search

Planning can also be done using a backward search. The idea is to start at the goal and apply inverses of the planning operators to produce subgoals, stopping if we produce a set of subgoals satisfied by the initial state. The set of all states that are predecessors of states in S_g is:

$$\Gamma^{-1}(g) = \{s \mid \text{there is an action } a \text{ such that } \gamma^{-1}(g, a) \text{ satisfies } g\}$$

This is the basis of the Backward-search algorithm shown in Figure 4.2. It is easy to show that Backward-search is sound and complete; the proof is analogous to the proof for Forward-search.

Example 4.2 As an example of how Backward-search works, consider the same DWR problem given in Example 4.1 (see page 69). Recall that in this problem, the initial state is the state s_5 of Figure 2.3, and the goal formula is $g = \{$at(r1, loc1),

```
Backward-search(O, s0, g)
    π ← the empty plan
    loop
        if s0 satisfies g then return π
        relevant ← {a | a is a ground instance of an operator in O
                        that is relevant for g}
        if relevant = ∅ then return failure
        nondeterministically choose an action a ∈ applicable
        π ← a.π
        g ← γ^{-1}(g, a)
```

Figure 4.2 A nondeterministic backward search algorithm.

loaded(r1, c3)}, which is a subset of the state s_6 of Figure 2.4. In one of the execution traces of Backward-search it does the following:

In the first iteration of the loop, it chooses a = load(crane1, loc1, c3, r1) and then assigns:

$$
\begin{aligned}
g &\leftarrow \gamma^{-1}(g, a) \\
&= (g - \text{effects}^+(a)) \cup \text{precond}(a) \\
&= (\{\texttt{at(r1, loc1), loaded(r1, c3)}\} - \{\texttt{empty(crane1), loaded(r1, c3)}\}) \\
&\cup \{\texttt{belong(crane1, loc1), holding(crane1, c3), at(r1, loc1), unloaded(r1)}\} \\
&= \{\texttt{at(r1, loc1), belong(crane1, loc1), holding(crane1, c3), unloaded(r1)}\}
\end{aligned}
$$

In the second iteration of the loop, it chooses a = move(r1, loc2, loc1) and then assigns:

$$
\begin{aligned}
g &\leftarrow \gamma^{-1}(g, a) \\
&= (g - \text{effects}^+(a)) \cup \text{precond}(a) \\
&= (\{\texttt{at(r1,loc1), belong(crane1,loc1), holding(crane1,c3), at(r1,loc1),} \\
&\quad \texttt{unloaded(}r1\texttt{)}\} - \{\texttt{ at(r1,loc1), occupied(loc1) }\}) \\
&\cup \{\texttt{adjacent(loc2,loc1), at(r1,loc2), } \neg \texttt{ occupied(loc1)}\} \\
&= \{\texttt{belong(crane1, loc1), holding(crane1, c3),} \\
&\texttt{unloaded(r1), adjacent(loc2, loc1), at(r1, loc2), occupied(loc1)}\}
\end{aligned}
$$

This time g is satisfied by s_5, so the execution trace terminates at the beginning of the fourth interation and returns the plan:

$$\pi = \langle(\texttt{move(r1, loc2, loc1), load(crane1, loc1, c3, r1)}\rangle$$

There are many other execution traces, some of which are infinite. For example, one of them makes the following infinite sequence of assignments to a:

load(crane1, loc1, c3, r1)
unload(crane1, loc1, c3, r1)
load(crane1, loc1, c3, r1)
unload(crane1, loc1, c3, r1)
...

■

Let $g_0 = g$. For each integer $i > 0$, let g_i be the value of g at the end of the ith iteration of the loop. Suppose we modify Backward-search to keep a record of the

```
Lifted-backward-search(O, s0, g)
    π ← the empty plan
    loop
        if s0 satisfies g then return π
        relevant ← {(o,σ) | o is an operator in O that is relevant for g,
                        σ1 is a substitution that standardizes o's variables,
                        σ2 is an mgu for σ1(o) and the atom of g that o is
                        relevant for, and σ = σ2σ2}
        if relevant = ∅ then return failure
        nondeterministically choose a pair (o,σ) ∈ relevant
        π ← σ(o).σ(π)
        g ← γ^-1(σ(g),σ(o))
```

Figure 4.3 Lifted version of Backward-search. *mgu* is an abbreviation for *most general unifier*; see Appendix B for details.

sequence of goal formulas $(g_1, \ldots, g_k)$ on the current path and to backtrack whenever there is an $i < k$ such that $g_i \subseteq g_k$. Just as with Forward-search, it can be shown that this modification causes Backward-search to return failure on every infinite branch of the search space and that it does not cause Backward-search to return failure on every branch that leads to a shortest-length solution (see Exercise 4.5). Thus, the modification can be used to do a sound and complete depth-first implementation of Backward-search.

The size of *relevant* can be reduced by instantiating the planning operators only partially rather than fully. Lifted-backward-search, shown in Figure 4.3, does this. Lifted-backward-search is a straightforward adaptation of Backward-search. Instead of taking a ground instance of an operator $o \in O$ that is relevant for g, it standardizes[1] o's variables and then unifies it with the appropriate atom of g.

The algorithm is both sound and complete, and in most cases it will have a substantially smaller branching factor than Backward-search.

Like Backward-search, Lifted-backward-search can be modified in order to guarantee termination of a depth-first implementation of it while preserving its soundness and completeness. However, this time the modification is somewhat trickier. Suppose we modify the algorithm to keep a record of the sequence of goal formulas $(g_1, \ldots, g_k)$ on the current path and to backtrack whenever there is an $i < k$ such that $g_i \subseteq g_k$. This is not sufficient to guarantee termination. The problem is that this time, g_k need not be ground. There are infinitely many possible unground

1. *Standardizing* an expression means replacing its variable symbols with new variable symbols that do not occur anywhere else.

atoms, so it is possible to have infinite paths in which no two nodes are the same. However, if two different sets of atoms are unifiable, then they are essentially equivalent, and there are only finitely many possible nonunifiable sets of atoms. Thus, we can guarantee termination if we backtrack whenever there is an $i < k$ such that g_i unifies with a subset of g_k.

4.4 The STRIPS Algorithm

With all of the planning algorithms we have discussed so far, one of the biggest problems is how to improve efficiency by reducing the size of the search space. The STRIPS algorithm was an early attempt to do this. Figure 4.4 shows a ground version of the algorithm; STRIPS is the lifted version (see Exercise 4.11).

STRIPS is somewhat similar to Backward-search but differs from it in the following ways.

- In each recursive call of the STRIPS algorithm, the only subgoals eligible to be worked on are the preconditions of the last operator added to the plan. This reduces the branching factor substantially; however, it makes STRIPS incomplete.
- If the current state satisfies all of an operator's preconditions, STRIPS commits to executing that operator and will not backtrack over this commitment.

```
Ground-STRIPS(O, s, g)
    π ← the empty plan
    loop
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O,
                  and a is relevant for g}
        if A = ∅ then return failure
        nondeterministically choose any action a ∈ A
        π' ← Ground-STRIPS(O, s, precond(a))
        if π' = failure then return failure
        ;; if we get here, then π' achieves precond(a) from s
        s ← γ(s, π')
        ;; s now satisfies precond(a)
        s ← γ(s, a)
        π ← π.π'.a
```

Figure 4.4 A ground version of the STRIPS algorithm.

This prunes a large portion of the search space but again makes STRIPS incomplete.

As an example of a case where STRIPS is incomplete, STRIPS is unable to find a plan for one of the first problems a computer programmer learns how to solve: the problem of interchanging the values of two variables.

Even for problems that STRIPS solves, it does not always find the best solution. Here is an example.

Example 4.3 Probably the best-known planning problem that causes difficulty for STRIPS is the Sussman anomaly, which was described in Exercise 2.1. Figure 4.5 shows a DWR version of this problem. In the figure, the objects include one location (loc1), one crane (crane1), three containers (c1, c2, c3), and five piles (p1, p2, q1, q2, q3). Although STRIPS's search space for this problem contains infinitely many solutions (see Exercise 4.15), none of them are nonredundant. The shortest solutions that STRIPS can find are all similar to the following:

⟨ take(c3,loc,crane,c1)	
put(c3,loc,crane,q1)	
take(c1,loc,crane,p1)	
put(c1,loc,crane,c2)	(STRIPS has achieved on(c1,c2).)
take(c1,loc,crane,c2)	
put(c1,loc,crane,p1)	
take(c2,loc,crane,p2)	
put(c2,loc,crane,c3)	(STRIPS has achieved on(c2,c3) but needs to reachieve on(c1,c2).)
take(c1,loc,crane,p1)	
put(c1,loc,crane,c2)⟩	(STRIPS has now achieved both goals.)

■

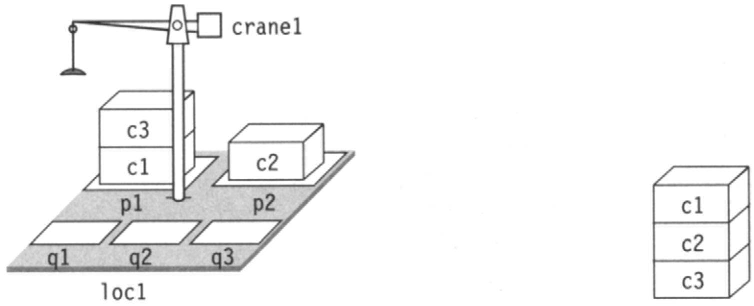
In both Example 4.3 and the problem of interchanging the values of two variables, STRIPS's difficulty involves *deleted-condition interactions*, in which the action chosen to achieve one goal has a side effect of deleting another previously achieved goal. For example, in the plan shown in Example 4.3, the action take(c1,loc,crane,c2) is necessary in order to help achieve on(c2,c3), but it deletes the previously achieved condition on(c1,c2).

One way to find the shortest plan for the Sussman anomaly is to interleave plans for different goals. The shortest plan for achieving on(c1,c2) from the initial state is:

⟨take(c3,loc,crane,c1), put(c3,loc,crane,q1), take(c1,loc,crane,p1), put(c1,loc,crane,c2)⟩

The shortest plan for achieving on(c2,c3) from the initial state is:

⟨take(c2,loc,crane,p2), put(c2,loc,crane,c3)⟩



s_0={in(c3,p1), top(c3,p1), in(c1,p1), on(c3,c1)
on(c1,pallet), in(c2,p2), top(c2,p2),
on(c2,pallet), top(pallet,q1), top(pallet,q2)
top(pallet,q3), empty(crane1)}

g={on(c1,c2)
on(c2,c3)}

Figure 4.5 A DWR version of the Sussman anomaly.

We can get the shortest plan for both goals by inserting the second plan between the second and third actions of the first plan.

Observations such as these led to the development of a technique called *plan-space planning*, in which the planning system searches through a space whose nodes are partial plans rather than states of the world, and a partial plan is a partially ordered sequence of partially instantiated actions rather than a totally ordered sequence. Plan-space planning is discussed in Chapter 5.

4.5 Domain-Specific State-Space Planning

This section illustrates how knowledge about a specific planning domain can be used to develop a fast planning algorithm that very quickly generates plans whose lengths are optimal or near optimal. The domain, which we call the *container-stacking domain*, is a restricted version of the DWR domain.

4.5.1 The Container-Stacking Domain

The language for the container-stacking domain contains the following constant symbols. There is a set of containers (c1, c2, ..., cn) and a set of piles (p1,p2, ..., pm, q1,q2, ..., ql), where n, m, and l may vary from one problem to

Table 4.1 Positions of containers in the initial state shown in Figure 4.5.

Container	*Position*	*Maximal?*	*Consistent with goal?*
c1	{on(c1,pallet)}	No	No: contradicts on(c1,c2)
c2	{on(c2,pallet)}	Yes	No: contradicts on(c2,c3)
c3	{on(c3,c1),on(c1,pallet)}	Yes	No: contradicts on(c1,c2)

another and $l \geq n$. There is one location loc, one crane crane, and a constant symbol pallet to represent the pallet at the bottom of each pile. The piles p1, ..., pm are the *primary piles*, and the piles q1, ..., ql are the *auxiliary piles*.

A container-stacking problem is any DWR problem for which the constant symbols are the ones just described and for which the crane and the auxiliary piles are empty in both the initial state and the goal. As an example, Figure 4.5 shows a container-stacking problem in which $n = 3$.

If s is a state, then a *stack* in s is any set of atoms $e \subseteq s$ of the form

$$\{\mathsf{in}(c_1, p), \mathsf{in}(c_2, p), \ldots, \mathsf{in}(c_k, p), \mathsf{on}(c_1, c_2), \mathsf{on}(c_2, c_3), \ldots, \mathsf{on}(c_{k-1}, c_k), \mathsf{on}(c_k, t)\},$$

where p is a pile, each c_i is a container, and t is the pallet. The top and bottom of e are c_1 and c_k, respectively. The stack e is *maximal* if it is not a subset of any other stack in s.

If s is a state and c is a container, then position(c, s) is the stack in s whose top is c. Note that position(c, s) is a maximal stack iff s contains the atom top(c, p); see Table 4.1 for examples.

From these definitions, it follows that in any state s, the position of a container c is consistent with the goal formula g only if the positions of all containers below c are also consistent with g. For example, in the container-stacking problem shown in Figure 4.5, consider the container c3. Because position(c1, s_0) is inconsistent with g and c3 is on c1, position(c1, s_0) is also inconsistent with g.

4.5.2 Planning Algorithm

Let $\mathcal{P}$ be a container-stacking problem in which there are m containers and n atoms. We can check whether $\mathcal{P}$ is solvable by checking whether g satisfies some simple consistency conditions. For example, g should not mention any containers not mentioned in s_0, g should not say that a container is sitting on two other containers at once, and so forth. It is not hard to write an algorithm to check these things in time $O(n \log n)$.

If $\mathcal{P}$ is solvable, then let $u_1, u_2, \ldots, u_k$ be all of the maximal stacks in g. It is easy to construct a plan that solves $\mathcal{P}$ by moving all containers to auxiliary pallets and then building each maximal stack from the bottom up. The length of this plan is at most $2m$ and it can be generated in time $O(n)$.

```
Stack-containers(O, s0, g):
    if g does not satisfy the consistency conditions then
        return failure        ;; the planning problem is unsolvable
    π ← the empty plan
    s ← s0
    loop
        if s satisfies g then return π
        if there are containers b and c at the tops of their piles such that
                position(c, s) is consistent with g and on(b, c) ∈ g
        then
            append actions to π that move b to c
            s ← the result of applying these actions to s
            ;; we will never need to move b again
        else if there is a container b at the top of its pile
                such that position(b, s) is inconsistent with g
                and there is no c such that on(b, c) ∈ g
        then
            append actions to π that move b to an empty auxiliary pile
            s ← the result of applying these actions to s
            ;; we will never need to move b again
        else
            nondeterministically choose any container c such that c is
                at the top of a pile and position(c, s) is inconsistent with g
            append actions to π that move c to an empty auxiliary pallet
            s ← the result of applying these actions to s
```

Figure 4.6 A fast algorithm for container stacking.

In general, the shortest solution length is likely to be much less than $2m$ because most of the containers will need to be moved only once or not at all. The problem of finding a shortest-length solution can be proved to be NP-hard, which provides strong evidence that it requires exponential time in the worst case. However, it is possible to devise algorithms that find, in low-order polynomial time, a solution whose length is either optimal or near optimal. One simple algorithm for this is the Stack-containers algorithm shown in Figure 4.6. Stack-containers is guaranteed to find a solution, and it runs in time $O(n^3)$, where n is the length of the plan it finds.

Unlike STRIPS, Stack-containers has no problem with deleted-condition interactions. For example, Stack-containers will easily find a shortest-length plan for the Sussman anomaly.

The only steps of Stack-containers that may cause the plan's length to be nonoptimal are the ones in the else clause at the end of the algorithm. However, these steps usually are not executed very often because the only time they are needed is when there is no other way to progress toward the goal.

4.6 Discussion and Historical Remarks

Although state-space search might seem like an obvious way to do planning, it languished for many years. For a long time, no good techniques were known for guiding the search; and without such techniques, a state-space search can search a huge search space. During the last few years, better techniques have been developed for guiding state-space search (see Part III of this book). As a result, some of the fastest current planning algorithms use forward-search techniques [33, 271, 414].

The container-stacking domain in Section 4.5 is a DWR adaptation of a well-known domain called the *blocks world*. The blocks world was originally developed by Winograd [555] as a test bed for his program for understanding natural language, but it subsequently has been used much more widely as a test bed for planning algorithms.

The planning problem in Example 4.3 (see page 77) is an adaptation of a blocks-world planning problem originally by Allen Brown [540], who was then a Ph.D. student of Sussman. Sussman popularized the problem [501]; hence it became known as the Sussman anomaly.

In Fikes and Nilsson's original version of STRIPS [189], each operator had a precondition list, add list, and delete list, and these were allowed to contain arbitrary well-formed formulas in first-order logic. However, in the presentation of STRIPS in Nilsson's subsequent textbook [426], the operators were restricted to a format equivalent to our classical planning operators.

Stack-containers is an adaption of Gupta and Nau's blocks-world planning algorithm [253]. Although our version of this algorithm runs in $O(n^3)$ time, Slaney and Thiébaux [481] describe an improved version that runs in linear time. They also describe another algorithm that runs in linear time and finds significantly better plans.

4.7 Exercises

4.1 Here is a simple planning problem in which the objective is to interchange the values of two variables v1 and v2.

$$s_0 = \{\text{value(v1,3), value(v2,5), value(v3,0)}\}$$

$$g = \{\text{value(v1,5), value(v2,3)}\}$$

assign(v, w, x, y)
  precond: value(v, x), value(w, y)
  effects: $\neg$ value(v, x), value(v, y)

If we run Forward-search on this problem, how many iterations will there be in the shortest execution trace? In the longest one?

4.2 Show that the algorithm shown in Figure 4.7 is equivalent to Forward-search, in the sense that both algorithms will generate exactly the same search space.

4.3 Prove property (2) of Section 4.2.2.

4.4 Prove that if a classical planning problem $\mathcal{P}$ is solvable, then there will always be an execution trace of Backward-search that returns a shortest-length solution for $\mathcal{P}$.

4.5 Prove that if we modify Backward-search as suggested in Section 4.3, the modified algorithm has the same property described in Exercise 4.4.

4.6 Explain why Lifted-backward-search needs to standardize its operators.

4.7 Prove that Lifted-backward-search is sound and complete.

4.8 Prove that Lifted-backward-search has the same property described in Exercise 4.4.

4.9 Prove that the search space for the modified version of Lifted-backward-search never has more nodes than the search space for the modified version of Backward-search.

4.10 Why did Exercise 4.9 refer to the modified versions of the algorithms rather than the unmodified versions?

4.11 Write STRIPS, the lifted version of the STRIPS algorithm in Figure 4.4.

4.12 Trace the operation of the STRIPS algorithm on the Sussman anomaly to create the plan given in Section 4.4. Each time STRIPS makes a nondeterministic choice, tell what the possible choices are. Each time it calls itself recursively, give the parameters and the returned value for the recursive invocation.

```
Recursive-forward-search(O, s0, g)
    if s satisfies g then return the empty plan
    active ← {a | a is a ground instance of an operator in O
                  and a's preconditions are true in s}
    if active = ∅ then return failure
    nondeterministically choose an action a1 ∈ active
    s1 ← γ(s, a1)
    π ← Recursive-forward-search(O, s1, g)
    if π ≠ failure then return a1.p
    else return failure
```

Figure 4.7 A recursive version of Forward-search.

4.13 In order to produce the plan given in Section 4.4, STRIPS starts out by working on the goal on(c1,c2). Write the plan STRIPS will produce if it starts out by working on the goal on(c2,c3).

4.14 Trace the operation of STRIPS on the planning problem in Exercise 4.1.

4.15 Prove that STRIPS's search space for the Sussman anomaly contains infinitely many solutions and that it contains paths that are infinitely long.

4.16 Redo Exercises 4.12 through 4.14 using your lifted version of STRIPS.

4.17 Our formulation of the container-stacking domain requires n auxiliary piles. Will the nth pile ever get used? Why or why not? How about the $(n-1)$th pile?

4.18 Show that if we modify the container-stacking domain to get rid of the auxiliary piles, then there will be problems whose shortest solution lengths are longer than before.

4.19 Suppose we modify the notation for the container-stacking domain so that instead of writing, e.g.,

```
in(a,p1), in(b,p1), top(a,p1), on(a,b), on(b,pallet),
in(c,p2), in(d,p2), top(c,p2), on(c,d), on(d,pallet),
```

we would write

```
clear(a), on(a,b), on(b,p1), clear(c), on(c,d), on(c,p2).
```

(a) Show that there is a one-to-one correspondence between each problem written in the old notation and an equivalent problem written in the new notation.

(b) What kinds of computations can be done more quickly using the old notation than using the new notation?

4.20 If P is the statement of a container-stacking problem, what is the corresponding planning problem in the blocks-world domain described in Exercise 3.5? What things prevent the two problems from being completely equivalent?

4.21 Trace the operation of Stack-containers on the Sussman anomaly to show that this algorithm finds the shortest solution.

4.22 Find a container-stacking problem for which Stack-containers will not always find a shortest-length solution. Hint: You probably will need at least 13 containers.