# 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. This chapter is organized as follows:

- 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

```
Forward-search(O, s_0, g)
    s ← s_0
    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).

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 that are 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 $\text{DWR}_1$ problem whose initial state is the state $s_1$ of Figure 2.2 and Example 2.10, and whose formula is $g = \mathsf{at}(\mathsf{r1}, \mathsf{loc1}), \mathsf{loaded}(\mathsf{r1}, \mathsf{c3})$. One of the execution traces of Forward-search does the following. In the first iteration of the loop, it chooses

$$a = \mathsf{move}(\mathsf{r1}, \mathsf{loc2}, \mathsf{loc1}),$$

producing the state $s_5$ of Figure 2.3 and Example 2.13. In the second iteration, it chooses

$$a = \mathsf{load}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1}),$$

producing the state $s_6$ of Figure 2.4 and Example 2.14. Since this state satisfies $g$, the execution trace returns

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

There are many other execution traces, some of which are infinite. For

example, one of them makes the following infinite sequence of choices for $a$:

$$\begin{array}{l}\mathsf{move}(\mathsf{r1},\mathsf{loc2},\mathsf{loc1});\\ \mathsf{move}(\mathsf{r1},\mathsf{loc1},\mathsf{loc2});\\ \mathsf{move}(\mathsf{r1},\mathsf{loc2},\mathsf{loc1});\\ \mathsf{move}(\mathsf{r1},\mathsf{loc1},\mathsf{loc2});\\ \ldots\end{array}$$

□

### 4.2.1 Formal Properties

**Proposition 4.2** Forward-search *is sound, any plan* $\pi$ *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 $\pi$ is empty. If it is true at the beginning of the $i$'th iteration, then since 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, since $s = \gamma(s_0, \pi)$, it follows that $\pi$ is a solution to $(O, s_0, g)$. □

**Proposition 4.3** *Let* $\mathcal{P} = (O, s_0, g)$ *be a classical planning problem, and let* $\Pi$ *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* $\pi$.

**Proof.** Let $\pi_0 = \langle a_1, \ldots, a_n \rangle \in \Pi$. We will prove that there is a nondeterministic trace such that for for every positive integer $i \leq n+1$, $\pi = \langle a_1, \ldots, a_{i-1} \rangle$ at the beginning of the $i$'th iteration of the loop (which means that the algorithm will return $\pi_0$ 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 $i$'th 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 4.3 is that Foreward-search is complete. Another consequence is that Foreward-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 pruning technique is *strongly safe* if there is at least one optimal solution that it doesn$\widehat{\text{p}}$rune. 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 $\pi_1$ and $\pi_2$ along two different paths of the search space, and suppose $\pi_1$ and $\pi_2$ produce the same state of the world $s$. If $\pi_1$ can be extended to form some solution $\pi_1\pi_3$, then $\pi_2\pi_3$ is also a solution, and vice versa. Thus we can prune one of $\pi_1$ and $\pi_2$, and we will still be guaranteed of finding a solution if one exists. Furthermore, if the plan that we prune is whichever of $\pi_1$ and $\pi_2$ 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 draw-

back: 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 gives 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 will need to detect and prune all infinite branches. In the Forward-search algorithm, this can be accomplished by keeping a record of the sequence $(s_0, s_1, \ldots, s_k)$ of states on the current path, and modifying the algorithm to return failure whenever there is an $i < k$ such that $s_k = s_i$. Even better is to modify the algorithm to return failure whenever there is an $i < k$ such that $s_k \subseteq s_i$. Either modification will prevent sequences of assignments such as the one described in Example 4.1, but there are some domains in which the second modification will prune infinite sequences sooner than the first one.

To show that the second 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 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 any successful nondeterministic 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.

```
Backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        applicable ← {a | a is a ground instance of an operator in O
                          that is relevant for g}
        if applicable = ∅ then return failure
        nondeterministically choose an action a ∈ applicable
        π ← a.π
        g ← γ^{-1}(g, a)
```

Figure 4.2: Nondeterministic backward search.

## 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 that is 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 4.4** As an example of how Backward-search works, consider the same $\text{DWR}_1$ problem given in Example 4.1. Recall that in this problem, the initial state is the state $s_1$ of Figure 2.2, and the goal formula is $g = \{\mathsf{at}(\mathsf{r1},\mathsf{loc1}), \mathsf{loaded}(\mathsf{r1},\mathsf{c3})\}$, which is a subset of the state $s_6$ of Figure 2.4. One of the execution traces of Backward-search does the following:

In the first iteration of the loop, it chooses $a = \mathsf{load}(\mathsf{crane1},\mathsf{loc1},\mathsf{c3},\mathsf{r1})$, and then assigns

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

In the second iteration of the loop, it chooses $a = \mathsf{move}(\mathsf{r1}, \mathsf{loc2}, \mathsf{loc1})$, and then assigns

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

In the third iteration of the loop, it chooses $a = \mathsf{take}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{c1}, \mathsf{p1})$. This time we will omit the details of computing $g \leftarrow \gamma^{-1}(g, a)$, except to say that the resulting value of $g$ is satisfied by $s_1$, so that the execution trace terminates at the beginning of the fourth interation, and returns the plan

$$\pi = \langle \mathsf{take}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{c1}, \mathsf{p1}), (\mathsf{move}(\mathsf{r1}, \mathsf{loc2}, \mathsf{loc1}), \mathsf{load}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1}) \rangle.$$

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

$\mathsf{load}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1})$;
$\mathsf{unload}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1})$;
$\mathsf{load}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1})$;
$\mathsf{unload}(\mathsf{crane1}, \mathsf{loc1}, \mathsf{c3}, \mathsf{r1})$;
...

□

Let $g_0 = g$. For each integer $i > 0$, let $g_i$ be the value of $g$ at the end of the $i$'th iteration of the loop. Suppose we modify Backward-search 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 $g_i \subseteq g_k$. 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 *active* 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

```
Lifted-backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        relevant ← {(o, σ) | o is an operator in O that is relevant for g,
                     σ_1 is a substitution that standardizes o's variables,
                     σ_2 is an mgu for σ_1(o) and the atom of g that o is
                     relevant for, and σ = σ_2σ_1}
        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.

Backward-search. Instead of taking a ground instance of an operator $o \in O$ that is relevant for $g$, it standardizes $o$'s variables[1] and then unifies it [2] 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 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 non-unifiable 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 $g_k$.

[1] *Standardizing* an expression means replacing its variable symbols with new variable symbols that do not occur anywhere else. One of the exercises deals with why standardizing is needed here.

[2] *mgu* is an abbreviation for *most general unifier*; see Appendix B for details.

```
STRIPS(O, s_0, 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 o is relevant for g}
        if A = ∅ then return failure
        nondeterministically choose any action a ∈ A
        π' ← STRIPS(O, s_0, 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 nondeterministic version of the STRIPS algorithm.

## 4.4 The STRIPS Algorithm

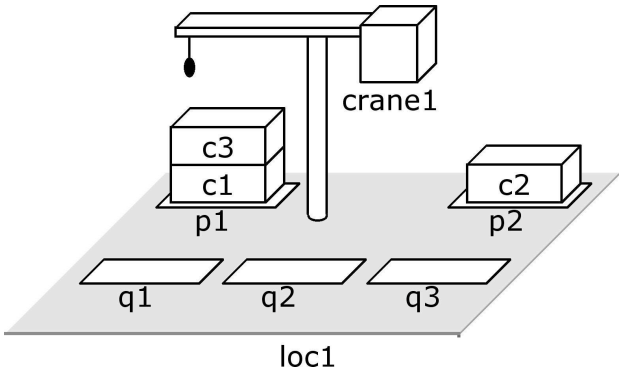
With all of the planning algorithms we have discussed so far, one of the biggest problems is how to improve efficiency by reducing the size of the search space. The STRIPS algorithm was an early attempt to do this. Figure 4.4 shows a nondeterministic version of the algorithm. In our version, every partial plan is ground, but it is easy to write a lifted version (see Exercise 4.15).

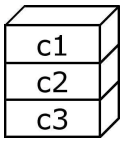
STRIPS is somewhat similar to Backward-search, but differs from it in the following ways:

1. In each recursive call of the STRIPS algorithm, the only subgoals that are eligible to be worked on are the preconditions of the last previous operator that was added to the plan. This reduces the branching factor substantially; however, it makes STRIPS incomplete.

2. If the current state satisfies all of an operator's preconditions, STRIPS commits to executing that operator, and will not backtrack over this commitment. This prunes off 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 that a computer programmer learns




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

$g = \{\mathsf{on(c1,c2)}, \mathsf{on(c2,c3)}\}$

Figure 4.5: A DWR version of the Sussman anomaly.

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 4.5** 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 loc, one crane crane, 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 4.14), none of them are irredundant. 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 re-achieve on(c1,c2) |
| take(c1,loc,crane,p1), | |
| put(c1,loc,crane,c2). | STRIPS has now achieved both goals |

□

In both Example 4.5 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 above, 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),

and the shortest plan for achieving on(c1,c2) from the initial state is

take(c2,loc,crane,p2), put(c2,loc,crane,c3).

We can get the shortest plan for both goals by inserting the second plan between the first and second lines 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 very 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 $\mathsf{c1}, \mathsf{c2}, \ldots, \mathsf{c}n$ and a set of piles $\mathsf{p1}, \mathsf{p2}, \ldots, \mathsf{p}m, \mathsf{q1}, \mathsf{q2}, \ldots, \mathsf{q}l$, where $m, n, l$ may vary from one problem to another and $l \geq n$. There is one location loc, one crane crane, and a constant

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) |

symbol pallet to represent the pallet at the bottom of each pile. The piles $\mathsf{p}1, \ldots, \mathsf{p}m$ are the *primary piles*, and the piles $\mathsf{q}1, \ldots, \mathsf{q}l$ are the *auxiliary piles*.

A container-stacking problem is any DWR problem for which the constant symbols are the ones described above, 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 if and only if $s$ contains the atom $\mathsf{top}(c, p)$; see Table 4.1 for examples.

From the above 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. Since position$(\mathsf{c}1, s_0)$ is inconsistent with $g$ and c3 is on c1, position$(\mathsf{c}1, s_0)$ 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. In time $O(n \log n)$ one can check whether or not $\mathcal{P}$ is solvable, by checking whether or not $g$ is consistent, and whether or not $g$ mentions any containers not mentioned in $s_0$. If $g$ is inconsistent or mentions a container not mentioned in $s_0$, then clearly $\mathcal{P}$ is not solvable.

```
Stack-containers(O, s_0, g):
    if g is inconsistent or refers to any containers not in s_0 then
        return failure        ;; the planning problem is unsolvable
    π ← the empty plan
    s ← s_0
    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
                g contains on(b, c)
        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.

Suppose $g$ is consistent and only mentions containers that are also mentioned in $s_0$, and 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 takes time $O(n)$ to produce it.

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 guaranteed to find a solution, and it runs in time $O(n^3)$, where $n$ is the length of the plan that 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 non-optimal 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 that 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 3 of this book). As a result, some of the fastest current planning algorithms use forward-search techniques [30, 263, 402].

The container-stacking domain in 4.5 is a DWR adaptation of a well known domain called the blocks world. The blocks world was originally developed by Winograd [545] as a test bed for his natural-language understanding program, but it subsequently has been used much more widely as a test bed for planning algorithms.

The planning problem in Example 4.5 is an adaptation of a blocks-world planning problem originally by Allen Brown [532], who was then a Ph.D. student of Sussman. Sussman was the one who popularized the problem [492]; hence it became known as the Sussman anomaly.

In Fikes and Nilsson's original version of STRIPS [180], 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 [414], the operators were restricted to a format that is equivalent to our classical planning operators.

Stack-containers is an adaption of Gupta and Nau's blocks-world planning algorithm [241]. Although our version of this algorithm runs in $O(n^3)$

time, Slaney and Thiébaux [470] describe an improved version of it that runs in linear time, and they also describe another algorithm that also 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:

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

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

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.

```
Recursive-forward-search(O, s_0, 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 a_1 ∈ active
    s_1 ← γ(s, a_1)
    π ← Recursive-forward-search(s_1, g, O)
    if π ≠ failure then return a_1.p
    else return failure
```

Figure 4.7: A recursive version of Forward-search.

**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 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 grounded-backward-search.

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

**4.11** 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.

**4.12** 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).

**4.13** Trace the operation of STRIPS on the planning problem in Exercise 4.7.

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

**4.15** Write a lifted version of the STRIPS algorithm.

**4.16** Redo Exercise 4.11 through 4.13 using your lifted version of STRIPS.

**4.17** Our formulation of the container-stacking domain requires $n$ auxiliary piles. Will the $n$'th pile ever get used? Why or why not? How about the $n-1$'st 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 length is longer than before.

**4.19** Suppose we modify the notation for the container-stacking domain so that instead of writing, for example,

$$\mathsf{in}(\mathsf{a},\mathsf{p1}),\mathsf{in}(\mathsf{b},\mathsf{p1}),\mathsf{top}(\mathsf{a},\mathsf{p1}),\mathsf{on}(\mathsf{a},\mathsf{b}),\mathsf{on}(\mathsf{b},\mathsf{pallet}),$$
$$\mathsf{in}(\mathsf{c},\mathsf{p2}),\mathsf{in}(\mathsf{d},\mathsf{p2}),\mathsf{top}(\mathsf{c},\mathsf{p2}),\mathsf{on}(\mathsf{c},\mathsf{d}),\mathsf{on}(\mathsf{d},\mathsf{pallet})$$

we would instead write

$$\mathsf{clear}(\mathsf{a}),\mathsf{on}(\mathsf{a},\mathsf{b}),\mathsf{on}(\mathsf{b},\mathsf{p1}),\mathsf{clear}(\mathsf{c}),\mathsf{on}(\mathsf{c},\mathsf{d}),\mathsf{on}(\mathsf{c},\mathsf{p2})$$

(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 3.6? What things prevent the two problems from being completely equivalent?

**4.21** Show that Stack-containers will always find a shortest-length solution for the Sussman anomaly.

**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 thirteen containers.

# Chapter 5

# Plan-Space Planning

## 5.1 Introduction

In the previous chapter, we addressed planning as the search for a path in the graph $\Sigma$ of a state-transition system. For state-space planning, the search space is given directly by $\Sigma$. Nodes are states of the domain; arcs are state transitions or actions; a plan is a sequence of actions corresponding to a path form the initial state to a goal state.

We shall be considering in this chapter a more elaborate search space that is not $\Sigma$ anymore. It is a space where nodes are *partially specified plans*. Arcs are *plan refinement operations* intended to further complete a partial plan, i.e., to achieve an open goal or to remove a possible inconsistency. Intuitively, a refinement operation avoids adding to the partial plan any constraint which is not strictly needed for addressing the refinement purpose. This is called the *least commitment principle*. Planning starts from an initial node corresponding to an empty plan. The search aims at a final node containing a solution plan that achieves correctly the required goals.

Plan-space planning differs from state-space planning not only in its search space, but also in its definition of a solution plan. Plan-space uses a more general plan structure than a sequence of actions. Here planning is considered as two separate operations (i) the choice of actions, and (ii) the ordering of the chosen actions such as to achieve the goal. A plan is defined as a set of planning operators together with ordering constraints and binding constraints; it may not correspond to a sequence of actions.

The search space is detailed in Section 5.2. Properties of solution plans are analyzed in Section 5.3; correctness conditions with respect to the semantics of state transition systems are established. Algorithms for plan-space

planning are proposed in Section 5.4. Several extensions are considered in Section 5.5.The chapter ends with a discussion and exercises.

## 5.2 The Search Space of Partial Plans

Generally speaking, a plan is a set of actions organized into some structure, e.g., a sequence. A partial plan can be defined as any subset of actions that keeps some useful part of this structure, e.g., a subsequence for state-space planning. All planning algorithms seen up to now extend step by step a partial plan. However, these were particular partial plans. Their actions are sequentially ordered. The total order reflects the intrinsic constraints of the actions in the partial plan, but also the particular search strategy of the planning algorithm. The former constraints are needed: a partial plan that is just an unstructured collection of actions would be meaningless since the relevance of a set of actions depends strongly on their organization. However the constraints reflecting the search strategy of the algorithm are not needed. There can be advantages in avoiding them.

To find out what is needed in a partial plan, let us develop an informal planning step on a simple example. Assume that we already have a partial plan; let us refine it by adding a new action and let us analyze how the partial plan should be updated. We'll come up with four ingredients: adding actions, adding ordering constraints, adding causal relationships, and adding variable binding constraints.

**Example 5.1** In the *DWR* domain, consider the problem where a robot r1 has to move a container c1 from pile p1 at location l1 to pile p2 and location l2 (see Figure 5.1). Initially r1 is unloaded at a location l3. There are empty cranes k1 and k2 at location l1 and l2 respectively. Pile p1 at location l1 contains only container c1, pile p2 at location l2 is empty. All locations are adjacent. □

For the above example, let us consider a partial plan which contains only the two following actions:

- take(k1,c1,p1,l1): crane k1 picks up container c1 from pile p1 at location l1;
- load(k1,c1,r1,l1): crane k1 loads container c1 on robot r1 at location l1.

**Adding actions.** There is nothing in this partial plan that guarantees that robot r1 is already at location l1. Proposition at(r1,l1) required as a

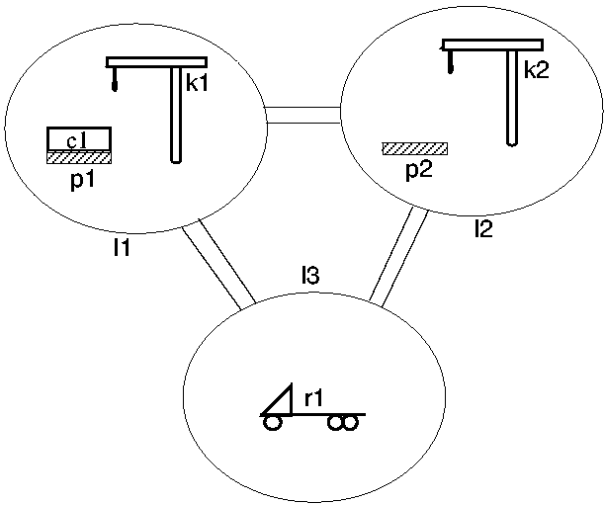



Figure 5.1: A simple DWR problem.

precondition by action load is a *subgoal* in this partial plan. Let us add to this plan the following action:

- move(r1,$l$,l1): robot r1 moves to the required location l1.

**Adding ordering constraints.** This additional action achieves its purpose only if it is constrained to come *before* the load action. Should the move action come *before* or *after* the take action? Both options are feasible. The partial plan does not commit unnecessarily at this stage to either ones. This follows a strategy, called the *least commitment principle*, of not adding a constraint to a partial plan unless strictly needed. Here, the only needed constraint is to have move before load. Other constraints, that may be needed later-on while planning, or even while executing the plan, may order move and take or may permit to run them concurrently.

**Adding causal links.** In the partial plan, we have added one action and one ordering constraint to another action already in the plan. Is that enough? Not quite. Since there is no explicit notion of a current state, an ordering constraint does not say that the robot stays at location l1 until the load action is performed. While pursuing the refinement of the partial plan, the planner may find other reasons to move the robot elsewhere, forgetting about the rationale for moving it to l1 in the first place. Hence, we'll be encoding explicitly in the partial plan the reason why action move was added

to it: to satisfy the subgoal at(r1,l1) required by action load.

This relationship between the two actions move and load with respect to proposition at(r1,l1), is called a *causal link*. The former action is called the *provider* of the proposition, the later is its *consumer*.[1] The precise role of a causal link is to state that a precondition of an action is *supported* by another action. Consequently, a precondition without a causal link is not supported. It will considered to be an open *subgoal* in the partial plan.

A provider has to be ordered before a consumer, but not necessarily strictly before in a sequence: other actions may take place in between. Hence, a causal link does not prevent interference between the *provider* and the *consumer* that may be due to other actions introduced later-on. Note that a causal link goes with an ordering constraint, but we may have ordering constraints without causal links: none of these relations is redundant with respect to the other.

**Adding variable binding constraints.** A final item in the partial plan that goes with the refinement we are considering is that of variable binding constraints. Planning operators are generic schema with variable arguments and parameters. Operators are added in a partial plan with a systematic variable renaming. We should make sure that the new operator move concerns the same robot r1 and the same location l1 as those in operators take and load. What about location $l$ the robot will be coming from? At this stage there is no reason to bind this variable to a constant. The same rationale of *least commitment* applies here as in the ordering between move and take. The variable $l$ is kept unbounded. Later-on, while further refining the partial plan, we may find it useful to bind it to the initial position of the robot or to some other locations. To sum up, we have added to the partial plan: *i)* an action, *ii)* an ordering constraint, *iii)* a causal link, and *iv)* variable binding constraints. These are exactly the four ingredients of partial plans.

**Definition 5.2** A partial plan is a tuple $\pi = (A, \prec, B, L)$ where:

- $A = \{a_1, ..., a_k\}$ is a set of partially instantiated planning operators;
- $\prec$ is a set of ordering constraints on $A$ of the form $(a_i \prec a_j)$;
- $B$ is a set of binding constraints on the variables of actions in $A$ of the form $x = y$, $x \neq y$, or $x \in D_x$, $D_x$ being a subset of the domain of $x$;

[1] Not in the sense of consuming a resource: load does not change this precondition.

- $L$ is a set of causal links of the form $\langle a_i \xrightarrow{p} a_j \rangle$, such that $a_i$ and $a_j$ are actions in $A$, the constraint $(a_i \prec a_j)$ is in $\prec$, proposition $p$ is an effect of $a_i$ and a precondition of $a_j$, and the binding constraints for variables of $a_i$ and $a_j$ appearing in $p$ are in $B$.

□

A plan space is an implicit directed graph whose vertices are partial plans and whose edges correspond to refinement operations. An outgoing edge from a vertex $\pi$ in the plan space is a *refinement operation* that transforms $\pi$ into a successor partial plan $\pi'$. A refinement operation consists of one or more of the following steps:

- adding an action to $A$;
- adding an ordering constraint to $\prec$;
- adding a binding constraint to $B$;
- adding a causal link to $L$.

Planning in a plan space is a search in that graph of a path from an initial partial plan denoted $\pi_0$ to a node recognized as a solution plan. At each step, the planner has to choose and apply a refinement operation to the current plan $\pi$ in order to achieve the specified goals. We now describe how goals and initial states are specified in the plan space.

Partial plans represent only actions and their relationships, states are not explicit. Hence, goals and initial states have also to be coded within the partial plan format as particular actions. Since preconditions are subgoals, the propositions corresponding to the goal $g$ are represented as the preconditions of a dummy action, call it $a_\infty$, that has no effect. Similarly, the propositions in the initial state $s_0$ are represented as the effects of a dummy action, $a_0$, that has no precondition. The initial plan $\pi_0$ is defined as the set $\{a_0, a_\infty\}$, together with the ordering constraint $(a_0 \prec a_\infty)$, but with no binding constraint and no causal link. The initial plan $\pi_0$, as well as any current partial plan, represent goals and subgoals as preconditions without causal links.

**Example 5.3** Let us illustrate two partial plans corresponding to the previous example (figure 5.1). The goal of having container c1 in pile p2 can be expressed simply as: in(c1,p2). The initial state is :

{adjacent(l1,l2), adjacent(l1,l3), adjacent(l2, l3),
adjacent(l2, l1), adjacent(l3, l1), adjacent(l3, l2),
attached(p1,l1), attached(p2, l2), belong(k1,l1), belong(k2, l2),

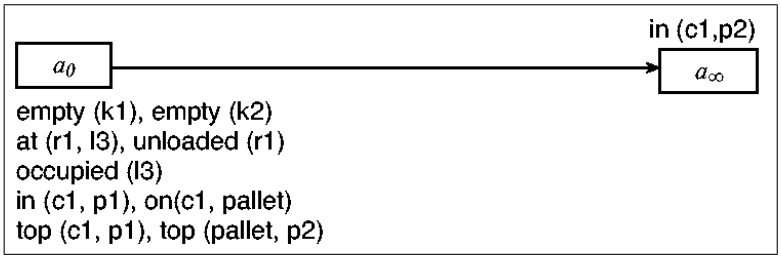



Figure 5.2: Initial plan $\pi_0$.

empty(k1) , empty(k2) , at(r1,l3), unloaded(r1) , occupied(l3),
in(c1,p1) , on(c1,pallet) , top(c1,p1) , top(pallet, p2) }

The first three lines describe rigid properties, i.e., invariant properties on the topology of locations, of piles and cranes; the last two define the specific initial conditions considered in this example.

A graphical representation of the initial plan $\pi_0$ corresponding to this problem is drawn in figure 5.2. The partial plan discussed earlier with the 3 actions take, load and move is given in figure 5.3. In these figures, each box is an action with preconditions above and effects bellow the box; to simplify, rigid properties are not shown in the effects of $a_0$ ; solid arrows are ordering constraints; dashed arrows are causal links; binding constraints are implicit or shown directly in the arguments. □

To summarize, a partial plan $\pi$ is a structured collection of actions that provides the causal relationships for the actions in $\pi$, as well as their intrinsic ordering and variable binding constraints. A partial plan also provides subgoals in $\pi$ as preconditions without causal links. A plan-space planner refines partial plan by further ordering and constraining its actions, or by adding new actions "anywhere" in the partial plan, as long as the constraint in $\pi$ remain satisfiable. A partial plan enables to neatly decouple two main issues in classical planning:

- which actions need to be done in order to meet the goals, and
- how to organize these actions.

It is convenient to view a partial plan $\pi$ as representing a particular set of plans. It is the set of all sequences of actions corresponding to a path from the initial state to a goal state that can be obtained by refinement operations on $\pi$, that is by adding to $\pi$ operators, ordering and binding constraints. A refinement operation on $\pi$ reduces the set of plans associated with $\pi$ to a smaller subset. This view will be formalized in the next section.

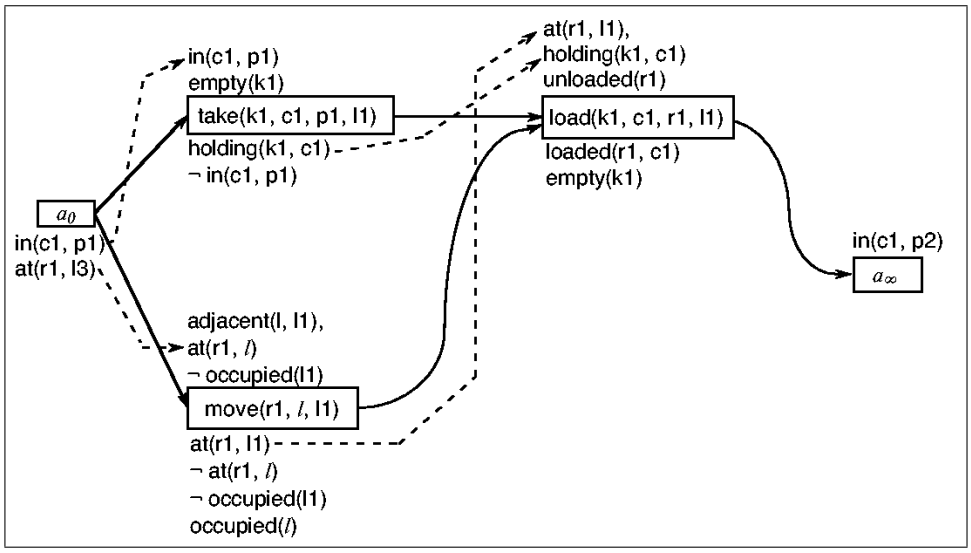



Figure 5.3: A partial plan.

## 5.3 Solution Plans

In order to define planning algorithms in the plan space, we need to formally specify what is a solution plan in the plan space. A solution plan for a problem $P = (\Sigma, s_0, g)$ has been formally defined, with respect to the semantics of state transition systems, as a sequence of ground actions from $s_0$ to a state in $g$ (see Section 2.3.3). We now have to take into account that actions in a partial plan $\pi = (A, \prec, B, L)$ are only partially ordered and partially instantiated.

A consistent partial order $\prec$ corresponds to the set of all sequences of totally ordered actions of $A$ that satisfy the partial order. Technically, these sequences are the topological orderings of the partial order $\prec$. There can be an exponential number of them. Note that a partial order defines a directed graph; it is consistent when this graph is loop free.

Similarly, for a consistent set of binding constraints $B$, there are many sequences of totally instantiated actions of $A$ that satisfy $B$. These sequences correspond to all the ways of instantiating every unbounded variable $x$ to a value in its domain $D_x$ allowed by the constraints. The set $B$ is consistent if every binding of a variable $x$ with a value in the allowed domain $D_x$ is consistent with the remaining constraints. Note that this is a strong consistency requirement.

**Definition 5.4** A partial plan $\pi = (A, \prec, B, L)$ is a *solution plan* for problem $P = (\Sigma, s_0, g)$ if

(i) its ordering constraints $\prec$ and binding constraints $B$ are consistent; and

(ii) every sequence of totally ordered and totally instantiated actions of $A$ satisfying $\prec$ and $B$ is a sequence that defines a path in the state transition system $\Sigma$ from the initial state $s_0$ corresponding to effects of action $a_0$ to a state containing all goal propositions in $g$ given by preconditions of $a_\infty$.

□

According to this definition, a solution plan corresponds to a set of sequences of actions, each being a path from the initial state to a goal state. This definition does not provide a computable test for verifying plans: it is not feasible to check all instantiated sequences of actions of $A$. We need a practical condition for characterizing the set of solutions. We already have a hint. Remember that a subgoal is a precondition without a causal link. Hence, a plan $\pi$ meets its initial goals, and all the subgoals due to the preconditions of its actions, if it has a causal link for every precondition. However this is not sufficient: $\pi$ may not be constrained enough to guarantee that all possible sequences define a solution path in graph $\Sigma$.

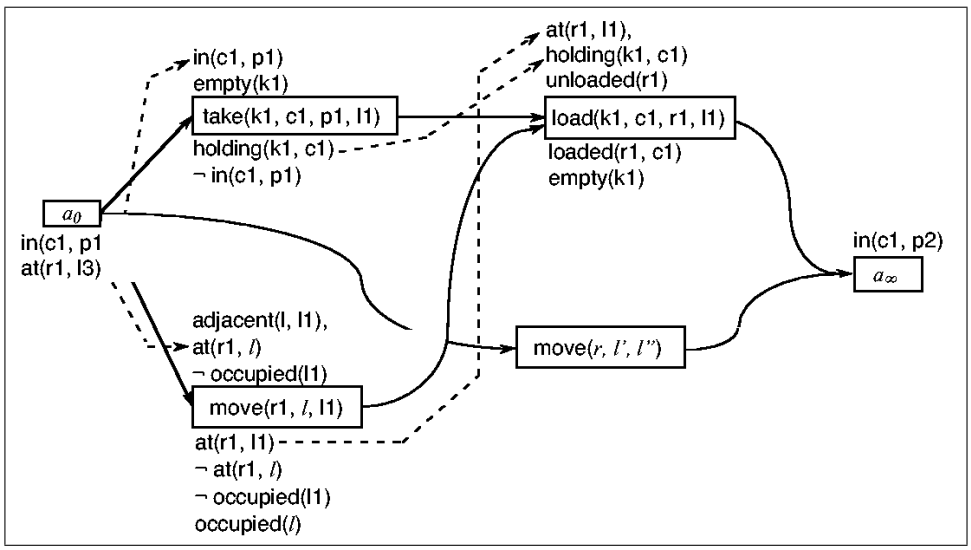



Figure 5.4: A plan containing an incorrect sequence.

**Example 5.5** Consider the previous example where we had a causal link

from action move(r1,$l$, l1) to action load(k1,c1,r1,l1), with respect to proposition at(r1,l1). Assume that the final plan contains another action move(r, $l'$,$l$"), without any ordering constraint that requires this action to be ordered before the previous move(r1,$l'$,l1) or after action load, and without any binding constraint that requires $r$ to be distinct from r1, nor $l'$ from l1 (Figure 5.4).

Among the set of totally ordered and instantiated sequences of actions of such a plan, there is at least one sequence that contains the subsequence

⟨move(r1,$l$, l1), ..., move(r1,l1,$l$"), ..., load(k1,c1,r1,l1)⟩

which is not a correct solution: the precondition at(r1,l1) is not satisfied in the state preceding action load. □

**Definition 5.6** [Threats] An action $a_k$ in a plan $\pi$ is a *threat* on a causal link $\langle a_i \xrightarrow{p} a_j \rangle$ iff:

- $a_k$ has an effect $\neg q$ that is possibly inconsistent with $p$, i.e. $q$ and $p$ are unifiable;
- the ordering constraints $(a_i \prec a_k)$ and $(a_k \prec a_j)$ are consistent with $\prec$; and
- the binding constraints for the unification of $q$ and $p$ are consistent with $B$.

□

**Definition 5.7** [Flaws] A *flaw* in a plan $\pi = (A, \prec, B, L)$ is either:

- a subgoal, i.e., a precondition of an action in $A$ without a causal link, or
- a threat, that is an action that may interfere with a causal link.

□

**Proposition 5.8** *A partial plan $\pi = (A, \prec, B, L)$ is a solution to the planning problem $P = (\Sigma, s_0, g)$ if $\pi$ has no flaw and if the sets of ordering constraints $\prec$ and binding constraints $B$ are consistent.*

**Proof** Let us prove the proposition inductively on the number of actions in $A$.
*Base step*: for $A = \{a_0, a_1, a_\infty\}$, there is just a single sequence of totally ordered actions. Since $\pi$ has no flaw, every precondition of $a_1$ is satisfied by a causal link with $a_0$, that is the initial state, and every goal or precondition of $a_\infty$ is satisfied by a causal link with $a_0$ or $a_1$.
*Induction step*: assume the proposition to hold for any plan having $(n-1)$

actions. Consider a plan $\pi$ with $n$ actions and without a flaw. Let $A_i = \{a_{i1}, ..., a_{ik}\}$ be a subset of actions whose only predecessor in the partial order $\prec$ is $a_0$. Every totally ordered sequence of actions of $\pi$ satisfying $\prec$ starts necessarily with some action $a_i$ from this subset. Since $\pi$ has no flaw, all preconditions of $a_i$ are met by a causal link with $a_0$. Hence every instance of action $a_i$ is applicable to the initial state corresponding to $a_0$.

Let $[a_0, a_i]$ denotes the first state of $\Sigma$ after the execution of $a_i$ in state $s_0$, and $\pi' = (A', \prec', B', L')$ be the remaining plan. That is, $\pi'$ is obtained from $\pi$ by replacing $a_0$ and $a_i$ with a single action (in the plan space notation) that has no precondition and whose effects correspond to the first state $[a_0, a_i]$, and by adding to $B$ the binding constraints due to this first instance of $a_i$. Let us prove that $\pi'$ is a solution.

- $\prec'$ is consistent since no ordering constraint has been added from $\pi$ to $\pi'$;
- $B'$ is consistent since new binding constraints where consistent with $B$;
- $\pi'$ has no threat since no action has been added in $\pi$ which had no threat;
- $\pi'$ has no subgoal: every precondition of $\pi$ that was satisfied by a causal link with an action $a \neq a_0$ is still supported by the same causal link in $\pi'$. Consider a precondition $p$ supported by a causal link $\langle a_0 \xrightarrow{p} a_j \rangle$, since $a_i$ was not a threat in $\pi$ then effects of any consistent instance of $a_i$ do not interfere with $p$. Hence condition $p$ is satisfied in the first state $[a_0, a_i]$. The causal link in $\pi'$ is now $\langle [a_0, a_i] \xrightarrow{p} a_j \rangle$.

Hence $\pi'$ has $(n-1)$ actions without a flaw: by induction it is a solution plan. □

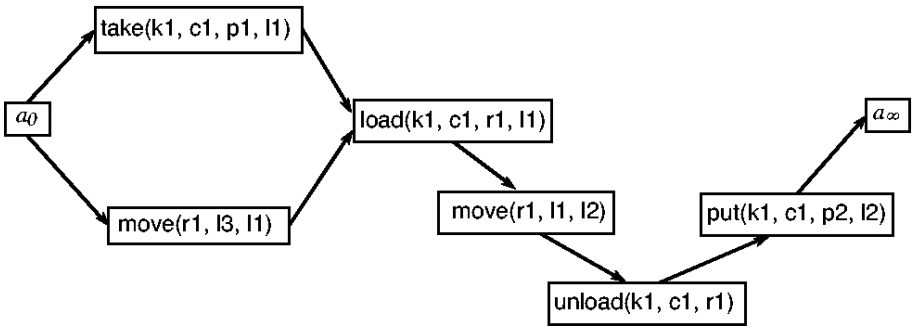



Figure 5.5: A solution plan.

**Example 5.9** Let us augment the previous example with actions:

- move(r1,l1,l2)
- unload(c1,r1,G2)
- put(c1,p2, G2)

The corresponding plan (Figure 5.5, where causal links are not shown) is a solution. □

Finally let us remark that Definition 5.4 and Proposition 5.8 allow for two types of flexibility in a solution plan: actions may not be totally ordered and they may not be totally instantiated. This remark applies also to $a_\infty$, since all actions in a partial have the same syntactical status. In other words, the flexibility introduced allows to handle partially instantiated goals. Recall that variables in a goal are existentially quantified. For example, any state where there is a container $c$ in the pile p2 meets a goal such as in($c$, p2) (see Exercise 5.4).

## 5.4 Algorithms for Plan Space Planning

### 5.4.1 PSP Procedure

The characterization of the set of solution plans gives the elements needed for the design of planning algorithms in the plan space. Since $\pi$ is a solution when it has no flaw, the main principle is to refine $\pi$, while maintaining $\prec$ and $B$ consistent, until it has no flaw. The basic operations for refining a partial plan $\pi$ towards a solution plan are the following:

- find the flaws of $\pi$, i.e., its subgoals and its threats;
- select one such a flaw;
- find ways for resolving it;
- choose a resolver for the flaw;
- refine $\pi$ according to that resolver.

The resolvers of a flaw are defined such as to maintain the consistency of $\prec$ and $B$ in any refinement of $\pi$ with a resolver. When there is no flaw in $\pi$, then the conditions of Proposition 5.8 are met, and $\pi$ is a solution plan. Symmetrically, when a flaw has no resolver, then $\pi$ cannot be refined into a solution plan.

Let us specify the corresponding procedure as a recursive nondeterministic schema, called PSP (for Plan Space Planning). The pseudo code of PSP is given in figure 5.6. The variables and procedures used in PSP are:

- $flaws$: denotes the set of all flaws in $\pi$ provided by procedures OpenGoals and Threats; $\phi$ is a particular flaw in this set;
- $resolvers$: denotes the set of all possible ways of resolving the current flaw $\phi$ in plan $\pi$; it is given by procedure Resolve; the resolver $\rho$ is an element of this set;
- $\pi'$: is a new plan obtained by refining $\pi$ according to resolver $\rho$ using for that procedure Refine.

```
PSP(π)
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then return(π)
    select any flaw φ ∈ flaws
    resolvers ← Resolve(φ, π)
    if resolvers = ∅ then return(failure)
    nondeterministically choose a resolver ρ ∈ resolvers
    π' ← Refine(ρ, π)
    return(PSP(π'))
end
```

Figure 5.6: PSP Procedure

The PSP procedure is called initially with the initial plan $\pi_0$. Each successful recursion is a refinement of the current plan according to a given resolver. The choice of the resolver is a *nondeterministic* step. The correct implementation of the "nondeterministically choose" step is the following:

- when a recursive call on a refinement with the chosen resolver returns a failure, then another recursion is performed with a new resolver;
- when all resolvers have been tried out unsuccessfully then a failure is returned from that recursion level back to a previous choice point, this is equivalent to an empty set of resolvers.

Note that the selection of a flaw (step select) is a deterministic step. All flaws need to be solved before reaching a solution plan. The order in which flaws are processed is very important for the efficiency of the procedure, but it is unimportant for its soundness and completeness. Before getting into the properties of PSP, let us detail the 4 procedures that is uses.

OpenGoals($\pi$): finds all subgoals in $\pi$. These are preconditions not supported by a causal link. This procedure is efficiently implemented with an *agenda*

data structure. For each new action $a$ in $\pi$, all preconditions of $a$ are added to the agenda; for each new causal link in $\pi$ the corresponding proposition is removed from the agenda.

Threats($\pi$): finds every action $a_k$ that is a threat on some causal link $\langle a_i \xrightarrow{p} a_j \rangle$. This can be done by testing all triples of actions in $\pi$, which takes $O(n^3)$. Here also an incremental processing is more efficient. For each new action in $\pi$, all causal links not involving that action are tested (in $O(n^2)$). For each new causal link, all actions in $\pi$, but those of the causal link, are tested (in $O(n)$, $n$ being the current number of actions in $\pi$).

Resolve($\phi, \pi$): finds all ways of solving a flaw $\phi$.

- If $\phi$ is a subgoal for a precondition $p$ of some action $a_j$, then its resolvers are either of the following:
  - A causal link $\langle a_i \xrightarrow{p} a_j \rangle$ if there is an action $a_i$ already in $\pi$ whose effect can provide $p$. This resolver contains 3 elements: the causal link, the ordering constraint $(a_i \prec a_j)$ if consistent with $\prec$, and the binding constraints to unify $p$ with the effect of $a_i$.
  - A new action $a$ that can provide $p$. This resolver contains $a$ together with the corresponding causal link, the ordering and binding constraints, including the constraints $(a_0 \prec a \prec a_\infty)$ required for a new action. Note that there is no need here to check the consistency of these constraints with $\prec$ and $B$.
- If $\phi$ is a threat on causal link $\langle a_i \xrightarrow{p} a_j \rangle$ by an action $a_k$ that as an effect $\neg q$, and $q$ is unifiable with $p$, then its resolvers are any of the following:[2]
  - The constraint $(a_k \prec a_i)$, if consistent with $\prec$, i.e., ordering $a_k$ before the causal link,
  - The constraint $(a_j \prec a_k)$, if consistent with $\prec$, i.e., ordering $a_k$ after the causal link,
  - A binding constraint consistent with $B$ that makes $q$ and $p$ non-unifiable.

Note that another way of addressing a threat is to choose for a causal link an alternate provider $a'_i$ that has no threat. Replacing $a_i$ with $a'_i$ as the provider for $a_j$ can be done through backtracking.

Refine($\rho, \pi$): refines the partial plan $\pi$ with the elements in the resolver, adding to $\pi$ an ordering constraint, one or several binding constraints, a

[2]These three resolvers are called respectively *promotion*, *demotion* and *separation*.

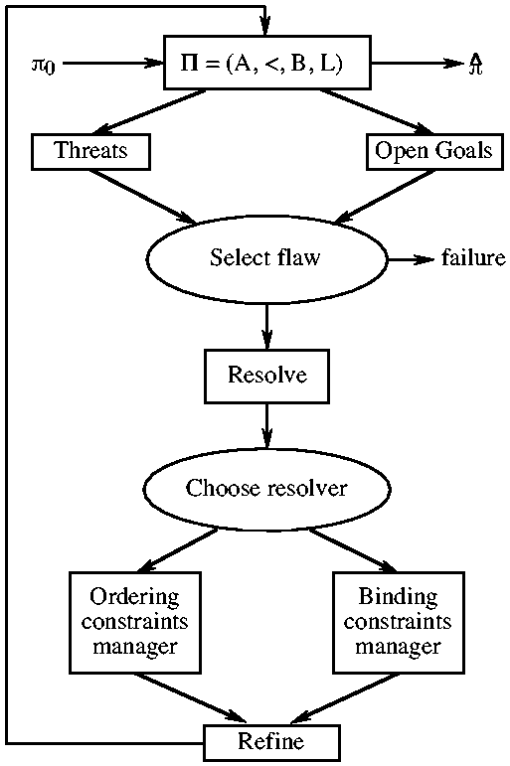



Figure 5.7: Organization of PSP.

causal link, and/or a new action. This procedure is straightforward: no testing needs to be done, since we have checked, while finding a resolver, that the corresponding constraints are consistent with $\pi$. Refine has just to maintain incrementally the set of subgoals in the agenda and the set of threats.

The two previous procedures perform several query and update operations on the two sets of constraints $\prec$ and $B$. It is preferable to have these operations carried out by specific *constraints managers*. Let us describe them briefly.

The Ordering constraint manager handles query and update operations. The former include querying whether a constraint $(a \prec a')$ is consistent with $\prec$, or asking for all actions in $A$ that can be ordered after some action $a$. The update operations consist of adding a consistent ordering constraint, and removing one or several ordering constraints, which is useful for backtracking

on a refinement. The alternatives for performing these operations are either of the following:

- To maintain as an explicit data structure only input constraints, in that case updates are trivial, in $O(1)$, whereas queries require a search in $O(n^2)$, $n$ being the number of constraints in $\prec$.
- To maintain the transitive closure of $\prec$. This makes queries easy, in $O(1)$, but requires an $O(n^2)$ propagation for each new update; a removal is performed on the input constraints plus a complete propagation.

In planning there are usually more queries than updates, hence the later alternative is preferable.

The Binding constraint manager handles three types of constraints on variables over finite domains: (i) unary constraints $x \in D_x$, (ii) binary constraints $x = y$, and (iii) $x \neq y$. Types (i) and (ii) are easily managed through a Union-Find algorithm in time linear in the number of query and update operations, whereas type (iii) rises a general NP-complete CSP problem.[3]

The global organization of the PSP procedure is depicted in Figure 5.7. The correct behavior of PSP is based on the nondeterministic step for choosing a resolver for a flaw and for backtracking over that choice, when needed, through all possible resolvers. The order on which resolvers are tried out is important for the efficiency of the algorithm. This choice has to be heuristically guided.

The order in which the flaws are processed (the step select for selecting the next flaw to be resolved) is also very important for the performance of the algorithm, even if no backtracking is needed at this point. A heuristic function is again essential for guiding the search.

**Proposition 5.10** *The PSP procedure is sound and complete: whenever $\pi_0$ can be refined into a solution plan, PSP($\pi_0$) returns such a plan.*

**Proof** Let us first prove the soundness, then the completeness.

In the initial $\pi_0$, the sets of constraints in $\prec$ and $B$ are obviously consistent. The Resolve procedure is such that every refinement step uses a resolver that is consistent with $\prec$ and $B$. Consequently, the successful termination step of PSP returns a plan $\pi$ that has no flaw and whose sets of constraints in $\prec$ and $B$ are consistent. According to Proposition 5.8, $\pi$ is

[3]For example, the well known NP-complete graph coloring problem is directly coded with type (iii) constraints.

a solution plan, hence PSP is sound. In order to prove the completeness of this nondeterministic procedure, we must show that there is at least one of its execution traces that returns a solution, when there is one. Let us prove it inductively on the length $k$ of a solution plan.
*Base step*, for $k = 0$: if the empty plan is a solution to the problem, then $\pi_0$ has no flaw and PSP returns immediately this empty plan.
*Induction step*: assume that PSP is complete for planning problems that have solutions of length $(k-1)$, and assume that the problem at hand $P = (O, s_0, g)$ admits a solution $\langle a_1, \ldots, a_k \rangle$ of length $k$. Since $a_k$ is relevant for the goal $g$, then there is at least one trace of PSP that chooses (and partially instantiate) an operator $a'_k$ to address the flaws in $\pi_0$, such that $a_k$ is an instance of $a'_k$. Let $[a'_k, a_\infty]$ be the set of goal propositions (in plan-space notation) corresponding to $\gamma^{-1}(g, a'_k)$. The next recursion of PSP takes place on a partial plan $\pi_1$ that has three actions $\{a_0, a'_k, a_\infty\}$. $\pi_1$ is equivalent to the initial plan of a problem from the state $s_0$ to a goal given by $[a'_k, a_\infty]$. This problem admit $\langle a_1, \ldots, a_{k-1} \rangle$ as a solution of length $(k-1)$. By induction, the recursion of PSP on $\pi_1$ has an execution trace which finds this solution. □

An important remark is in order. Even with the restrictive assumption of a finite transition system (assumption **A0** in Section 1.5) the plan space is *not finite*. A *deterministic* implementation of the PSP procedure will maintain the completeness only if it guarantees to explore all partial plans, up to some length. It has to rely, for example, on an iterative deepening search, such as IDA*, where a bound on the length of the sought solution is progressively increased. Otherwise, the search may keep exploring deeper and deeper a single path in the search space, adding indefinitely new actions to the current partial plan and never backtracking. Note that a search with an A* algorithm is also feasible as long as the refinement cost from one partial plan to the next is not null.

### 5.4.2 PoP Procedure

The PSP procedure is a generic schema that can be instantiated into several variants. Let us describe briefly one of them, algorithm PoP, which corresponds to a popular implementation for a planner in the plan space.

The pseudo code for PoP is given in figure 5.8. It is specified as a nondeterministic recursive procedure. It has two arguments: $\pi = (A, \prec, B, L)$ the current partial plan, and *agenda* which is a set of couples $\langle a, p \rangle$, $a$ being an action in $A$ and $p$ a precondition of $a$ which is a subgoal. The

```
PoP(π, agenda)          ;; where π = (A, ≺, B, L)
   if agenda = ∅ then return(π)
   select any pair ⟨a_j, p⟩ in and remove it from agenda
   relevant ← Providers(p, π)
   if relevant = ∅ then return(failure)
   nondeterministically choose an action a_i ∈ relevant
   L ← L ∪ {⟨a_i —p→ a_j⟩}
   update B with the binding constraints of this causal link
   if a_i is a new action in A then do:
      update A with a_i
      update ≺ with (a_i ≺ a_j), (a_0 ≺ a_i ≺ a_∞)
      update agenda with all preconditions of a_i
   for each threat on ⟨a_i —p→ a_j⟩ or due to a_i do:
      resolvers ← set of resolvers for this threat
      if resolvers = ∅ then return(failure)
      nondeterministically choose a resolver in resolvers
      add that resolver to ≺ or to B
   return(PoP(π, agenda))
end
```

Figure 5.8: Procedure PoP

arguments of the initial call are $\pi_0$ and an agenda containing $a_\infty$ with all its preconditions.

PoP uses a procedure called Providers that finds all actions, either in $A$ or new instances of planning operators of the domain, that have an effect $q$ unifiable with $p$, this set of actions is denoted *relevant* for the current goal.

There is a main difference between PSP and PoP. PSP processes the two types of flaws in a similar way: at each recursion it selects heuristically a flaw from any type to pursue the refinement. The PoP procedure has a distinct control for subgoals and for threats. At each recursion, it first refines with respect to a subgoal, then it proceeds by solving all threats due to the resolver of that subgoal. Consequently, there are two nondeterministic steps: (i) the choice of a relevant action for solving a subgoal, and (ii) the choice of an ordering or a binding constraints for solving a threat.

Backtracking for these two nondeterministic steps is performed chronologically, i.e., by going over all resolvers for a threat, and if none of them works then backtracking to another resolver for the current subgoal. Note that a resolver for a threat is a single constraint to be added to either $\prec$ or

$B$.

## 5.5 Extensions

The PoP procedure is a simplified version of a planner called UCPOP that implements many of the extensions of the classical representation introduced in Section 2.4. Let us mention here briefly how plan-space planning can handle some of these extensions.

**Conditional Operators.** Recall that a conditional operator has, in addition to normal preconditions and effects, some conditional effects that depend on whether an associated antecedent condition holds in the state $s$ to which the operator is applied. In order to handle a conditional operator two changes in the PoP procedure are required:

- The Providers procedure may find an action that provides $p$ conditionally. If this is the action chosen in *relevant* then the antecedent on which $p$ conditionally depends need to be added to the *agenda*, along with the other unconditional preconditions of the action.
- An action $a_k$ can be a "conditional threat" on a causal link $\langle a_i \xrightarrow{p} a_j \rangle$, when the threatening effect is a conditional effect of $a_k$. In that case, there is another set of possible resolvers for this threat. In addition to ordering and binding constraints, here one may also solve the threat by adding to the agenda, as a precondition of $a_k$, the negation of a proposition in the antecedent condition of the threat. By making the antecedent condition false, one is sure that $a_k$ cannot be a threat to the causal link. Note that there can be several such resolvers.

**Disjunctive Preconditions.** They correspond to a syntactical facility that allows to specify in a concise way several operators into a single one. They are in principle "easy" to handle although they lead to an exponentially larger search space. Whenever the pair selected in the agenda corresponds to a disjunctive precondition, a nondeterministic choice takes place. One disjunct is chosen for which relevant providers are sought, and the procedure is pursued. The other disjuncts of the precondition are left as a possible backtrack point.

**Quantified Conditional Effects.** Under the assumption of a finite $\Sigma$ a quantified expression can be expanded into a finite ground expression.

This expansion is made easier when each object in the domain is typed (see Section 2.4).

## 5.6 Plan Space Versus State Space Planning

The search space is a critical aspect for the types of plans that can be synthesized and for the performance of a planner. Here are some of the ways in which plan-space planning compares to state-space planning:

- A partial plan separates the choice of the actions that need to be done from how to organize them into a plan except from the intrinsic ordering and binding constraints of the chosen actions. In state-space planning, the ordering of the sequence of actions reflects these constraints as well as the control strategy of the planner.
- The plan structure and the rationale for the plan's components are explicit in the plan space: causal links give the role of each action with respect to goals and preconditions. It is easy to explain a partial plan to a user. A sequence generated by a state-space planner is less explicit with regards to this causal structure.
- The state space is finite (under assumption $A0$), while the plan space is not, as discussed earlier.
- Intermediate states are explicit in state-space planning, while there are no states in the plan space.
- Nodes of the search space are more complex in the plan space than states. Refinement operations for a partial plan take significantly longer to compute than state transitions or state regressions.

Despite the extra cost for each node, plan-space planning appeared for a while to be a significant improvement over state-space planning, leading to a search exploring smaller spaces for several planning domains [49, 536].

However, partial plans have an important drawback: the notion of explicit states along a plan is lost. Recent state-space planner have been able to significantly benefit from this advantage by making a very efficient use of domain-specific heuristics and control knowledge. This has enabled state-space planning to scale up to very large problems. There are some attempts to generalize these heuristics and control techniques to the plan-space planning [412]. However, it appears to be harder to control plan-space planners

as efficiently as state-space ones because of the lack of explicit states.[4] In summary, plan-space planners are not today competitive enough in classical planning with respect to computational efficiency. Nevertheless, plan-space planners keep other advantages:

- They build partially ordered and partially instantiated plans that are more explicit and flexible for execution.
- They provide an open planning approach for handling several extensions to the classical framework, such as time, resources and information gathering actions. In particular planning with temporal and resource constraints can be brought as natural generalization of the PSP schema (see chapter 14).
- They allow distributed planning and multi-agent planning to be addressed very naturally, since different types of plan-merging operations are easily defined and handled on the partial plan structure.

A natural question then arises: is it possible to blend state-space and plan-space into an approach that keeps the best of the two words? The planner called FLECS [519] provides an affirmative answer. The idea in FLECS is to combine the least-commitment principle with what is called an *eager commitment strategy*. The former chooses new actions and constraints in order to solve flaws in a partial plan $\pi$, as it is done PSP. The latter maintains a current state $s$; whenever there is in $\pi$ an action $a$ applicable to $s$ that has no predecessor in $\prec$ except $a_0$, it chooses to progress from $s$ to the successor state $\gamma(s, a)$. Flaws are assessed with respect of this new current state. At each recursion, FLECS introduces a flexible choice point between eager or least commitment: it chooses nondeterministically either to progress on the current state or to solve some flaw in the current partial plan. The termination condition is an empty set of flaws. In a way, FLECS applies to plan-space planning the idea of the STRIPS procedure in state-space planning (see Exercise 5.10). However, unlike STRIPS, the procedure is sound and complete, providing the flexible choice between eager or least commitment is a backtrack point. Several interesting heuristics for guiding this choice, and in particular an abstraction driven heuristic can be considered [516].

[4] For more detail, see Part III about the heuristics control issues, and in particular Section 9.4 on heuristics for plan-space planning.

## 5.7 Discussion and Historical Remarks

The shift from state-space to plan-space planning is usually attributed to Sacerdoti [449], who developed a planner called NOAH [448] that has also been a seminal contribution to task reduction or Hierarchical Task Network (HTN) planning (see Chapter 11). Interestingly, the structure of a partial plan emerged progressively from another contribution to HTN planning, the NONLIN planner [494], which introduced causal links.[5] NONLIN raised the issue of *linear* versus *nonlinear* planning, that stayed for a while as an important and confusing debate issue in the field. The *linear* adjective for a planner referred confusingly in different papers either to the structure of the planner's current set of actions (a sequence instead of a partial order), or to its search strategy that addresses one goal after the previous one has been completely solved.

The issue of interacting goals in a planning problem and how to handle them efficiently has been a motivating concern for the study of the plan-space. Starting from [493] and the so-called *Sussman anomaly* (see Example 4.5), several authors such as [157, 259, 282, 48] discussed this issue.

In the context of plan-space planning, [438] introduced a distinction between problems with fully independent goals, *serializable* goals, and arbitrarily interacting goals. The first category is the easiest to handle. In the second category, there is an ordering for solving the goals without violating the previously solved ones. If the goals are addressed in this correct order, then the planning complexity grows linearly in the number of goals. This goal dependence hierarchy is further refined in the context of plan-space planning by [49] where the authors introduce a planner-dependent notion of *trivially* and *laboriously serializable* goals; according to their analysis plan-space planners have the advantage of more often leading to trivial serializable goals that are easily solved.

The truth criterion in a partial plan has been another major debate issue in the field. In state-space planning, it is trivial to check whether some condition is true or not in some current state. But plan-space planning does not keep explicit states. Hence it is less easy to verify if a proposition is true before or after the execution of an action in a partially ordered and partially instantiated set of actions. The so-called *Modal Truth Criterion* (MTC) [114] provided a necessary and sufficient condition for the truth of a proposition at some point in a partial plan $\pi$. A planner called TWEAK relied at each recursion on the evaluation of this MTC criterion for synthe-

[5] See [288] for a comparative analysis of plan-space and HTN planning.

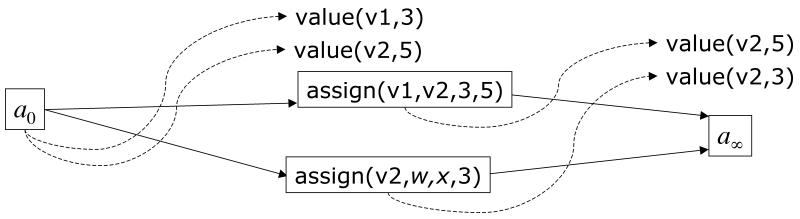
sizing a plan in the plan-space. It was shown that if $\pi$ contains actions with conditional effects, then the evaluation of the MTC is an NP-hard problem. This complexity result led to a belief that plan-space planning with extended representation is impractical. However, this belief is incorrect since planning does not require a necessary and sufficient truth condition. It only has to enforce a sufficient truth condition, which basically corresponds in PSP to the identification and resolution of flaws, performed in polynomial time. A detailed analysis of the MTC in planning appears in [294].

The UCPOP planner of Penberthy and Weld [424, 536] has been a major contribution to plan-space planning. It builds on the advances brought by SNLP [369], an improved formalization TWEAK, to propose a proven sound and complete planner able to handle most of the extensions to the classical representation introduced in the Action Description Language ADL of Pednault [421, 422]. The well documented open source Lisp implementation of UCPOP [47] offers several enhancements such as domain axioms and control rules. The latter even incorporate some of the learning techniques developed in the state-space planner PRODIGY [378, 516] for acquiring control knowledge [47].

The work on UCPOP opened to way to several other extensions such as handling incomplete information and sensing actions [426, 175, 226] or managing extended goals with protection conditions that guarantee a plan meeting some safety requirements [538] (see Part V). Issues such as the effects of domain features on the performance of the planner [317] and the role of ground actions (that is an early commitment as opposed to the late commitment strategy) [555] have been studied. Recent implementations, such as HCPOP [208], offer quite effective search control and pruning capabilities. The study of the relationship between state-space and plan-space exemplified in the FLECS planner [519] follows on several algorithmic contributions such as [377, 379], and on studies that relate the performance of PSP to specific features of the planning domain [317]. The work of Kambhampati [287, 293] introduces a much wider perspective and a nice formalization that takes into account many design issues such as multiple contributors to a goal or systematicity [286], i.e. whether a planner is non redundant and does not visit a partial-plan in the space more than once.

## 5.8 Exercises

**5.1** Here is a partial plan generated by PSP for the variable-interchange problem described in Exercise 4.7:




(a) How many threats are there?

(b) How many children (immediate successors) would this partial plan have?

(c) How may different solution plans can be reached from this partial plan?

(d) How many different irredundant solution plans can be reached from this partial plan?

(e) If we start PSP running from this plan, can it ever find any redundant solutions?

(f) Trace the operation of PSP starting from this plan, along whichever execution trace finds the solution that has the smallest number of actions.

**5.2** Trace the PSP procedure step by step on Example 5.1, from Figure 5.2 to figure 5.5.

**5.3** Trace the operation of PSP on the Sussman anomaly (Example 4.5).

**5.4** Trace the PSP procedure on the problem that has a single partially instantiated goal in(*c*, p2) with an initial state similar to that of Figure 5.1 except that location l1 has two piles p0 and p1, p0 has a single container c0 and p1 has a container c1.

**5.5** Let $\mathcal{P}$ be a planning problem in which no operator has any negative effects.

(a) What part(s) of PSP will be unneeded to solve $\mathcal{P}$?

(b) Suppose we run PSP deterministically, with a best-first control strategy. When, if at all, will PSP have to backtrack?

**5.6** Consider the following "painting problem." We have a can of red paint c1, a can of blue paint c2, two paint brushes r1 and r2, and four unpainted blocks b1, b2, b3, b4. We want to make b1 and b2 red, and make b3 and b4 blue. Here is a classical formulation of the problem:

$s_0$ ={can(c1), can(c2), color(c1,red), color(c2,blue), brush(r1), brush(r2), dry(r1), dry(r2), block(b1), block(b2), dry(b1), dry(b2), block(b3), block(b4), dry(b3), dry(b4)}
$g$ ={color(b1,red), color(b2,red), color(b3,blue), color(b4,blue)}

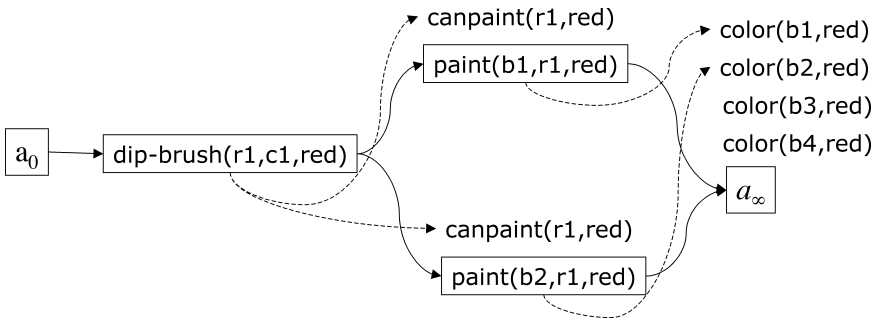
dip-brush(r,c,k)
  precond: brush(r), can(c), color(c,k)
  effects: ¬dry(r), canpaint(r,k)

paint(b,r,k)
  precond: block(b), brush(r), canpaint(r,k)
  effects: ¬dry(b), color(b,k), ¬canpaint(r,k)

(a) In the paint operator, what is the purpose of the effect ¬canpaint(r,k)?

longer be

(b) Starting from the initial state, will PSP ever generate the following partial plan? Explain why or why not. would either constrain paint(b2,r1,red) to come after paint(b1,r1,red) or vice versa,




(c) What threats are there in the partial plan?

vice versa.

(d) Starting from the partial plan, resolve all of the open goals without resolving any threats. What threats are there in the plan now?

(e) If we start PSP running with this plan as input, will PSP generate any successors? Explain why or why not.

(f) If we start PSP running with this plan as input, will PSP find a solution? Explain why or why not.

**5.7** Dan wants to wash his clothes with a washing machine wm, wash his dishes in a dishwasher dw, and bathe in a bathtub bt. The water supply doesn't have enough pressure to do more than one of these at once. Here is a classical representation of the problem:

**Initial state:** status(dw,ready), status(wm,ready), status(bt,ready), clean(dan,0), clean(clothes,0), clean(dishes,0), loc(dishes,dw), loc(clothes,wm), loc(dan,bt), use(water,0)

**Goal formula:** clean(clothes,1), clean(dishes,1), clean(dan,1)

**Operators:**

start-fill($x$)
- precond: status($x$,ready), use(water,0)
- effects: status($x$,fill), use(water,1)

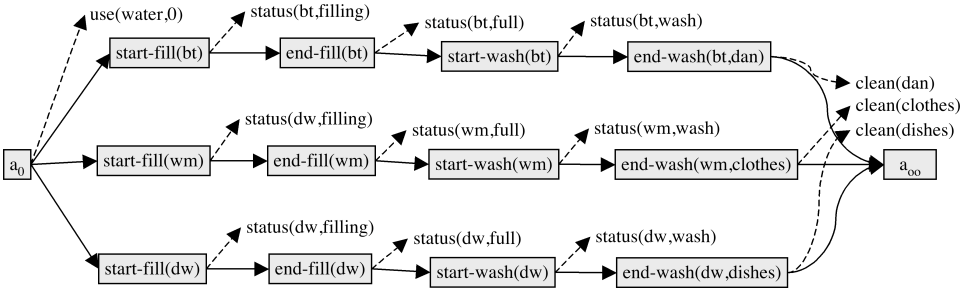
end-fill($x$)
- precond: status($x$,fill)
- effects: status($x$,full), use(water,0)

start-wash($x$)
- precond: status($x$,full)
- effects: status($x$,wash)

end-wash($x, y$)
- precond: status($x$,wash)
- effects: status($x$,ready), clean($y$,1)

replacing each atom $p(x, y)$ with $p(x) = y$.

(a) Let $\pi_1$ be the following partial plan. What threats are in $\pi_1$?




(b) What flaws are in $\pi_1$ other than threats?

(c) How many different solutions can PSP find if we start it out with the plan $\pi_1$, and do not allow it to add any new actions to the plan? Explain your answer.

can resolve threats to the three different use(water,0) preconditions.

(d) How many different solutions can PSP find if we do allow it to add new actions to $\pi_1$? Explain your answer.

actions to establish the unestablished preconditions.

**5.8** Let $P = (O, s_0, g)$ and $P' = (O, s_0, g')$ be the statements of two planning problems having the same operators and initial state. Let $B$ and $B'$ be the search spaces for PSP on $P$ and $P'$, respectively.

(a) If $g \subseteq g'$, then is $B \subseteq B'$?

(b) If $B \subseteq B'$ then is $g \subseteq t'$?

(c) Under what conditions, if any, can we guarantee that $B$ is finite?

(d) How does your answer to part (c) change if we run PSP deterministically with a breadth-first control strategy?

(e) How does your answer to part (c) change if we run PSP deterministically with a depth-first control strategy?

**5.9** Run the HCPOP plan-space planner [208] on several problems of the DWR domain of Example 2.11. At which point in the number of containers, locations, or robots, the planner is overwhelmed?

**5.10** Discuss the commonalities and differences of FLECS and STRIPS. Why is the former complete while the latter is not?

# Part II

# Neoclassical Planning

# Introduction to Neoclassical Planning

Neoclassical planning, like classical planning, is also concerned with restricted state-transition systems. We will be using here the classical representations developed in Chapter 2. However, at a time when classical planning appeared to be stalled for expressiveness as well as for complexity reasons, the techniques discussed in this part, that we qualify as *neoclassical*,[6] led to a revival of the research on classical planning problems. The development of neoclassical techniques brought new search spaces and search algorithms for planning that allowed directly, or indirectly through improvement of classical techniques, a significant scale-up on the size of classical problems that could be solved.

The main differences between classical and neoclassical techniques are the following:

- In classical planning, every node of the search space is a partial plan, i.e., a sequence of actions in the state-space, or a partially ordered set of actions in the plan-space; any solution reachable form that node contains entirely *all* the actions this partial plan.
- In neoclassical planning, every node of the search space can be viewed as a set of several partial plans. This set is either explicit or implicit in the data structures that make a search node, but it is evident in the fact that in the neoclassical approaches, not every action in a node appears in a solution plan reachable from that node.[7]

[6] *Neoclassic : of or relating to a revival or adaptation of the classical style, especially in literature, art, or music* [Webster New Collegiate Dictionary]. Neoclassical has referred to slightly different meanings in planning literature depending on one's views of where and when the revival took place.

[7] Because of this property, neoclassical planning approaches have sometimes been called *disjunctive-refinement approaches* [297].

We will come back to this common feature in Part III, once the reader has become familiar with the neoclassical techniques. Three such techniques will be studied here:

- *planning-graph* techniques, which are based on a powerful reachability structure for a planning problem, called a planning graph, which is used to efficiently organize and constrain the search space;
- *propositional satisfiability* techniques, which encode a planning problem into a SAT problem and then relies on efficient SAT procedures for finding a solution, among which complete methods based on the Davis-Putnam procedure, and pseudo-random local search methods;
- *constraint satisfaction* techniques, which similarly enable to encode a planning problem into a constraint satisfaction problem, and that also bring to the field a variety of efficient methods, in particular filtering and constraint propagation for disjunctive refinements in the plan-space or within the planning-graph approaches.

These three techniques are described in Chapters 6, 7, and 8, respectively.

# Chapter 6

# Planning-Graph Techniques

## 6.1 Introduction

The *planning-graph techniques* developed in this chapter rely on the classical representation scheme.[1] These techniques introduce a very powerful search space, called a "*planning graph*", which departs significantly from the two search spaces presented earlier, the state space (Chapter 4) and the plan space (Chapter 5).

State space planners provide a plan as a sequence of actions. Plan space planners synthesize a plan as a partially ordered set of of actions, any sequence meeting the constraints of the partial order is a valid plan. Planning-graph approaches take a middle ground. Their output is a sequence of subsets of actions, e.g., $\langle\{a_1, a_2\}, \{a_3, a_4\}, \{a_5, a_6, a_7\}\rangle$, which represents all sequences starting with $a_1$ and $a_2$ in any order, followed by $a_3$ and $a_4$ in any order, followed by $a_5$, $a_6$ and $a_7$ in any order. A sequence of subsets of actions is obviously more general than a sequence of actions: there are $2 \times 2 \times 6 = 24$ sequences of actions in the previous example. However, a sequence of subsets is less general than an partial order. It can be expressed immediately as a partial order, but the converse is false, e.g., a plan with 3 actions $a_1, a_2$ and $a_3$ and a single ordering constraint $a_1 \prec a_3$ cannot be structured as a sequence of subsets, unless an additional constraint is added.

We have seen that the main idea behind plan-space planning is the *least commitment principle*: that is to refine a partial plan, one flaw at a time, by adding only the ordering and binding constraints needed for solving that

[1] The Graphplan algorithm assumes, for notational convenience, a somewhat restricted but theoretically equivalent representation, with no negated literals in preconditions of operators nor in goals; this restriction is easily relaxed.

flaw. Planning-graph approaches on the other hand make strong commitments while planning: actions are considered fully instantiated and at specific steps. These approaches rely instead on two powerful and interrelated ideas, that are:

- *reachability analysis*, and
- *disjunctive refinement*.

Disjunctive refinement consists of addressing one or several flaws through a disjunction of resolvers. Since in general flaws are not independent and their resolvers may interfere, dependency relations are posted as constraints to be dealt with at a later stage.

Disjunctive refinement may not appear right away to the reader as the main motivation in planning-graph techniques. However, reachability analysis is clearly a driving mechanism for these approaches. Let us detail its principles before getting into planning graph algorithms.

## 6.2 Planning Graphs

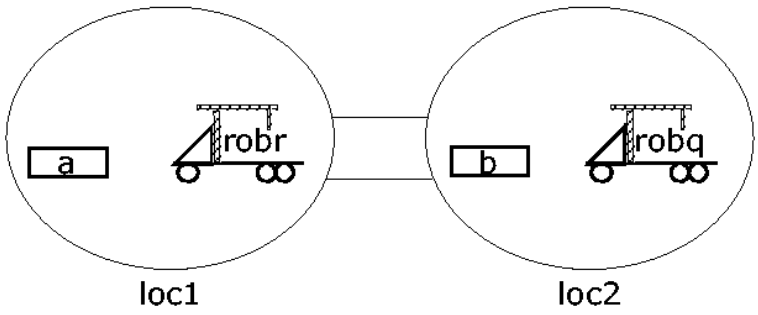
### 6.2.1 Reachability Trees

Given a set $A$ of actions, a state $s$ is *reachable* from some initial state $s_0$ if there is a sequence of actions of $A$ that defines a path from $s_0$ to $s$. Reachability analysis consists of analyzing which states can be reached from $s_0$ in some number of steps, and how to reach them. Reachability can be computed *exactly*, through a *reachability tree* that gives $\hat{\Gamma}(s_0)$, or it can be *approximated* through a structure called a *planning graph* that will be developed in this section. Let us first introduce an example.

**Example 6.1** Consider a *Simplified DWR* domain with no piles and no cranes where robots can load and unload autonomously containers, and where locations may contain an unlimited number of robots. In this domain, let us define a problem (see Figure 6.1) with two locations loc1 and loc2, two containers conta and contb and two robots robr and robq. Initially, robr and conta are in location loc1, robq and contb are in loc2. The goal is to have conta in loc2 and contb in loc1. Here, the set $A$ has twenty actions corresponding to the instances of the three operators in Figure 6.1.

To simplify the forthcoming figures, let us denote ground instances of predicates by propositional symbols:

- $r_1$ and $r_2$ stand for at(robr, loc1) and at(robr, loc2);




move$(r, l, l')$ ;; robot $r$ at location $l$ moves to a connected location $l'$
precond : at$(r, l)$, adjacent$(l, l')$
effects : at$(r, l')$, $\neg$ at$(r, l)$

load$(c, r, l)$ ;; robot $r$ loads container $c$ at location $l$
precond : at$(r, l)$, in$(c, l)$, unloaded$(r)$
effects : loaded$(r, c)$, $\neg$ in$(c, l)$, $\neg$ unloaded$(r)$

unload(c, r, l) ;; robot $r$ unloads container $c$ at location $l$
precond : at$(r, l)$, loaded$(r, c)$
effects : unloaded$(r)$, in$(c, l)$, $\neg$ loaded$(r, c)$

Figure 6.1: A simplified DWR problem.

- $q_1$ and $q_2$ stand for at(robq, loc1) and at(robq, loc2);
- $a_1$, $a_2$, $a_r$ and $a_q$ stand respectively for container conta in location loc1, in location loc2, loaded on robr or loaded on robq;
- $b_1$, $b_2$, $b_r$ and $b_q$ stand for the possible positions of container contb;
- $u_r$ and $u_q$ stand for unloaded(robr) and unloaded(robq).

Let us also denote the twenty actions in $A$ as follows:

- Mr12 is the action move(robr, loc1, loc2), Mr21 the opposite move, Mq12 and Mq21 are the move actions of robot robq;
- Lar1 is the action load(conta, robr, loc1), Lar2, Laq1 and Laq2 are the other load actions for conta in loc2 and with robq respectively; Lbr1, Lbr2, Lbq1 and Lbq2 are the load actions for contb; and
- Uar1, Uar2, Uaq1, Uaq2, Ubr1, Ubr2, Ubq1, Ubq2 are the unload actions.

The reachability tree for this domain, partially developed down to level 2

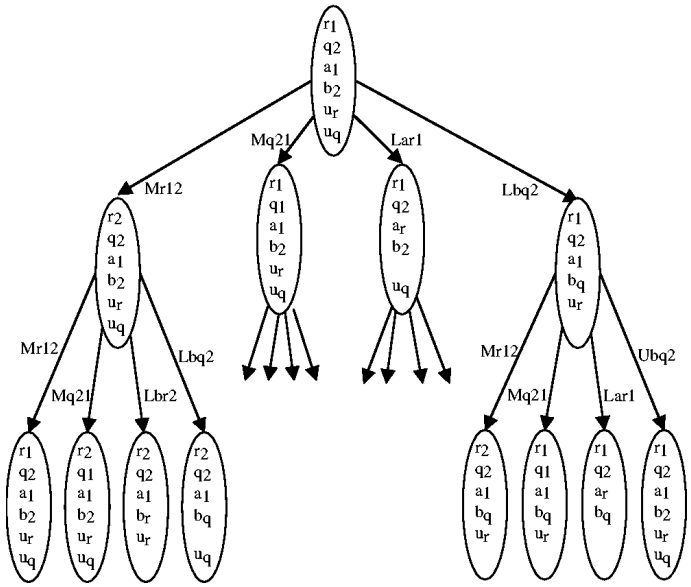



Figure 6.2: Reachability tree.

from the initial state $\{r_1, q_2, a_1, b_2, u_r, u_q\}$, is shown in figure 6.2. □

A reachability tree is a tree $T$ whose nodes are states of $\Sigma$ and whose edges corresponds to actions of $\Sigma$. The root of $T$ is the state $s_0$. The children of a node $s$ are all the states in $\Gamma(s)$. A complete reachability tree from $s_0$ gives $\hat{\Gamma}(s_0)$. A reachability tree developed down to depth $d$ solves *all* planning problems with $s_0$ and $A$, for *every* goal that is reachable in $d$ or fewer actions: a goal is reachable from $s_0$ in at most $d$ steps if and only if it appears in some node of the tree. Unfortunately, a reachability tree blows up in $O(k^d)$ nodes, where $k$ is the number of valid actions per state.

Since some nodes can be reached by different paths, the reachability tree can be factorized into a graph. Figure 6.3 illustrates such a reachability graph down to level 2 for the previous example (omitting for clarity most of back arcs from a node to its parents). However, even this reachability graph would be of a very large impractical size, as large as the number of states in the domain.

### 6.2.2 Reachability with Planning Graphs

A major contribution of the Graphplan planner is a relaxation for the reachability analysis. The approach provides an incomplete condition of reacha-

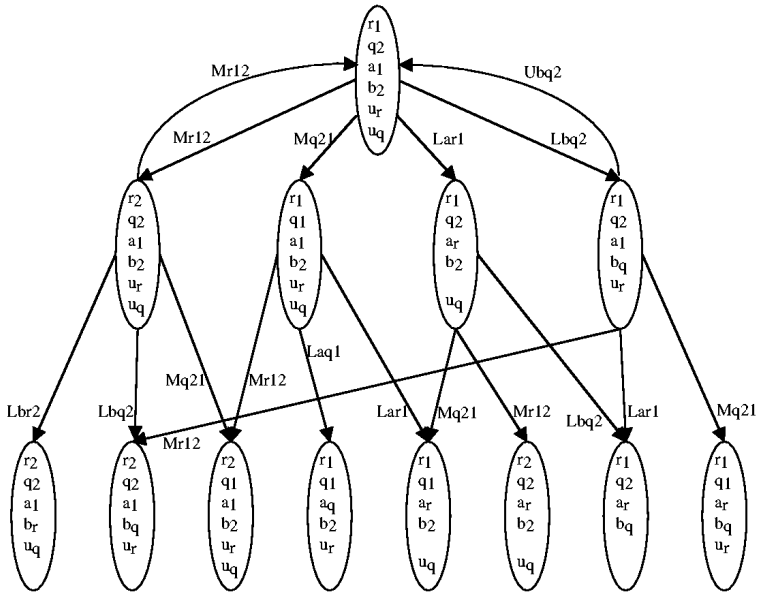



Figure 6.3: Reachability graph.

bility through a structure called a *planning graph*. A goal is reachable from $s_0$ *only if* it appears in some node of the planning graph. However, this is not a sufficient condition anymore. This weak reachability condition is compensated for by a low complexity: the planning graph is of polynomial size and can be built in polynomial time in the size of the input.

The basic idea in a planning graph is to consider at every level of this structure not individual states but, to a first approximation, the *union* of sets of propositions in several states. Instead of mutually exclusive actions branching out from a node, it considers an *inclusive disjunction* of actions from one node to the next which contains all the effects of these actions. In a reachability graph a node is associated with the propositions that *necessarily* hold for that node. In a planning graph, a node contains propositions that *possibly* hold at some point. However, while a state is a consistent set of propositions, the union of sets of propositions for several states does not preserve consistency. In the previous example we would have propositions showing robots in two places, containers in several locations, etc. Similarly, not all actions within a disjunction are compatible, they may interfere. A solution to that is to keep track of incompatible propositions for each set of propositions, and of incompatible actions for each disjunction of actions. Let us explain informally how this is performed.

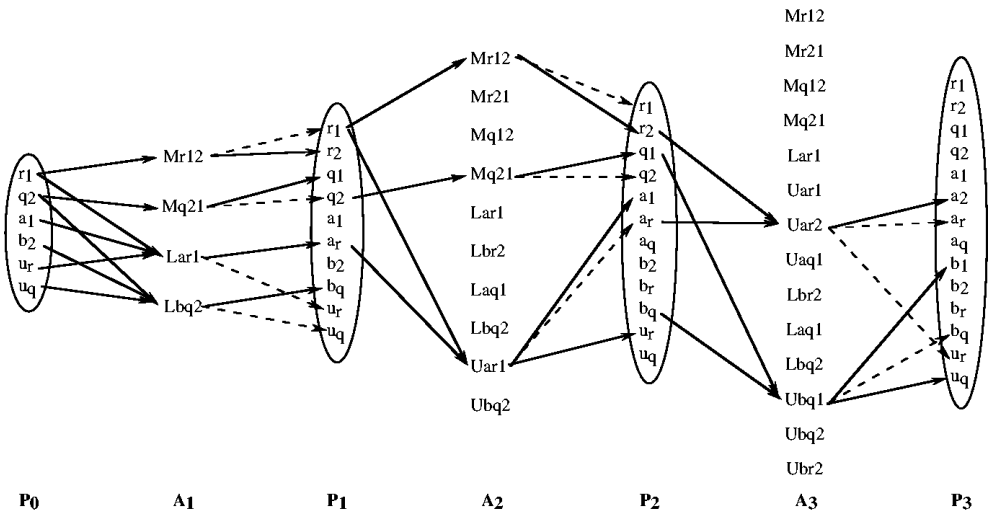



Figure 6.4: Planning graph.

A planning graph is a directed *layered* graph: arcs are permitted only from one layer to the next. Nodes in level 0 of the graph correspond to the set $P_0$ of propositions denoting the initial state $s_0$ of a planning problem. Level 1 contains two layers, called an action level $A_1$ and a proposition level $P_1$:

- $A_1$ is the set of actions (ground instances of operators) whose preconditions are nodes in $P_0$,
- $P_1$ is defined as the union of $P_0$ and the sets of positive effects of actions in $A_1$.

An action node in $A_1$ is connected with incoming *precondition arcs* from its preconditions in $P_0$, with outgoing arcs to its positive effects and to its negative effects in $P_1$. Outgoing arcs are labeled as *positive* or as *negative*. Note that negative effects are not deleted from $P_1$, thus $P_0 \subseteq P_1$.[2] This process is pursued from one level to the next. This is illustrated in Figure 6.4 for the above example down to level 3 (dashed lines correspond to negative effects, not all arcs are shown).

In accordance with the idea of inclusive disjunction in $A_i$ and of union of propositions in $P_i$, a plan associated to a planning graph is not any more

[2]The persistence principle or "frame axiom," which states that unless it is explicitly modified, a proposition persists from one state to the next, is modeled here through this definition that makes $P_0 \subseteq P_1$.

a sequence of actions corresponding directly to a path in $\Sigma$, as defined in Chapter 2. Here, a plan $\Pi$ is *sequence of sets of actions* $\Pi = \langle \pi_1, \pi_2, \ldots, \pi_k \rangle$. It will be qualified as a *layered plan* since it is organized into levels corresponding to those of the planning graph, with $\pi_i \subseteq A_i$. The first level $\pi_1$ is a subset of "*independent*" actions in $A_1$ that can be applied in *any* order to the initial state and can lead to a state which is a *subset* of $P_1$. From this state, actions in $\pi_2 \subseteq A_2$ would proceed, and so on until a level $\pi_k$, whose actions lead to a state meeting the goal. Let us define these notions more precisely.

### 6.2.3 Independent actions and Layered Plans

In our example, the two actions Mr12 (that is, $move(r, loc1, loc2)$) and Mq21 in $A_1$ are *independent*: they can appear at the beginning of a plan in any order, and the two sequences $\langle Mr12, Mq21 \rangle$ and $\langle Mq21, Mr12 \rangle$ when applied to $s_0$ lead to the same state. Similarly for the pair Mr12 and Lbq2. But the two actions Mr12 and Lar1 are not independent: a plan starting with Mr12 will be in a state where Lar1 is not applicable. More formally:

**Definition 6.2** Two actions $(a, b)$ are independent iff:
$\text{effects}^-(a) \cap [\text{precond}(b) \cup \text{effects}^+(b)] = \emptyset$ and
$\text{effects}^-(b) \cap [\text{precond}(a) \cup \text{effects}^+(a)] = \emptyset$.
A set of actions $\pi$ is independent when every pair of $\pi$ is independent. □

Conversely, two actions $a$ and $b$ are *dependent* if either:

- $a$ deletes a precondition of $b$: the ordering $a \prec b$ will not be permitted;
- $a$ deletes a positive effect of $b$: the resulting state will dependent on their order;
- symmetrically for negative effects of $b$ with respect to $a$.

Note that the independence of actions is not specific to a particular planning problem: it is an intrinsic property of the actions of a domain that can be computed before hand for all problems of that domain.

**Definition 6.3** A set $\pi$ of independent actions is applicable to a state $s$ iff $\text{precond}(\pi) \subseteq s$. The result of applying the set $\pi$ to $s$ is defined as:
$\gamma(s, \pi) = (s - \text{effects}^-(\pi)) \cup \text{effects}^+(\pi)$, where:
$\text{precond}(\pi) = \bigcup \{\text{precond}(a) \mid \forall a \in \pi\}$,
$\text{effects}^+(\pi) = \bigcup \{\text{effects}^+(a) \mid \forall a \in \pi\}$, and
$\text{effects}^-(\pi) = \bigcup \{\text{effects}^-(a) \mid \forall a \in \pi\}$. □

**Proposition 6.4** *If a set $\pi$ of independent actions is applicable to $s$ then, for any permutation $\langle a_1, \ldots, a_k \rangle$ of the elements of $\pi$, the sequence $\langle a_1, \ldots, a_k \rangle$ is applicable to $s$, and the state resulting from the application of $\pi$ to $s$ is such that: $\gamma(s, \pi) = \gamma(\ldots \gamma(\gamma(s, a_1), a_2) \ldots a_k)$.*

This proposition (whose proof is left as the Exercise 6.5) allows us to go back to the standard semantics of a plan as a path in a state transition system from the initial state to a goal.

**Definition 6.5** A layered plan is a sequence of sets actions. The layered plan $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ is a solution to a problem $(O, s_0, g)$ iff each set $\pi_i \in \Pi$ is independent, and the set $\pi_1$ is applicable to $s_0$, $\pi_2$ is applicable to $\gamma(s_0, \pi_1)$, ..., etc, and $g \subseteq \gamma(\ldots \gamma(\gamma(s_0, \pi_1), \pi_2) \ldots \pi_n)$. □

**Proposition 6.6** *If $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ is a solution plan to a problem $(O, s_0, g)$ then a sequence of actions corresponding to any permutation of the elements of $\pi_1$, followed by any permutation $\pi_2$ ..., followed by any permutation of $\pi_n$ is a path from the $s_0$ to a goal state.*

This proposition follows directly from Proposition 6.4.

### 6.2.4 Mutual Exclusion Relations

Two dependent actions in the action level $A_1$ of the planning graph cannot appear simultaneously in the first level $\pi_1$ of a plan. Hence, the positive effects of two dependent actions in $A_1$ are incompatible propositions in $P_1$, unless these propositions are also positive effects of some other independent actions. In our example, $r_2$ and $a_r$ are the positive effects respectively of Mr12 and Lar1, and only of these dependent actions. These two propositions are incompatible in $P_1$ in the following sense: they cannot be reached through a single level of actions $\pi_1$. Similarly for $q_1$ and $b_q$.

Furthermore, negative and positive effects of an action are also incompatible propositions. This is the case for the couple $(r_1, r_2)$, $(q_1, q_2)$, $(a_r, u_r)$, $(b_q, u_q)$ in level $P_1$ of the previous figure. In order to deal uniformly with these second type of incompatibility between propositions, it is convenient to introduce for each proposition $p$ a neutral *no-op* action, noted $\alpha_p$, whose precondition and sole effect is $p$.[3] If an action $a$ has $p$ as a negative effect, then according to our definition, $a$ and $\alpha_p$ are dependent actions; their positive effects are incompatible.

[3]Hence, the result of no-op actions is to copy all the propositions of $P_{i-1}$ into $P_i$: no-ops are also a way of modeling the persistence principle.

Dependency between actions in an action level $A_i$ of the planning graph leads to incompatible propositions in the proposition level $P_i$. Conversely, incompatible propositions in a level $P_i$ lead to additional incompatible actions in the following level $A_{i+1}$. These are the actions whose preconditions are incompatible. In our example, $(r_1, r_2)$ are incompatible in $P_1$. Consequently Lar1 and Mr21 are incompatible in $A_2$. Note that an action whose preconditions are incompatible is simply removed from $A_{i+1}$. This is the case for Uar2 ($r_2$ and $a_r$ incompatible) and for Ubq2 in $A_2$. Indeed, while an incompatible pair in $A_i$ is useful because one of its action may used in a level $\pi_i$ of a plan, there is no sense in keeping an impossible action.

The incompatibility relations between actions and between propositions in a planning graph, also called mutual exclusion or *mutex* relations, are formally defined as follows:

**Definition 6.7** Two actions $a$ and $b$ in a level $A_i$ are mutex if either $a$ and $b$ are dependent, or if a precondition of $a$ is mutex with a precondition of $b$. Two propositions $p$ and $q$ in $P_i$ are mutex if every action in $A_i$ that has $p$ as a positive effect (including no-op actions) is mutex with every action that produces $q$, and there is no action in $A_i$ that produces both $p$ and $q$. □

Note that dependent actions are necessarily mutex. Dependency is an intrinsic property of the actions in a domain, while the mutex relation takes into account additional constraints of the problem at hand. For the same problem, a pair of actions may be mutex in some action level $A_i$ and become non mutex in some latter level $A_j$ of a planning graph.

**Example 6.8** Mutex relations for the above example are listed in table 6.1, giving for each proposition or action at every level the list of elements that are mutually exclusive with it, omitting for simplicity the no-op actions (a star "*" denotes mutex actions that are independent but have mutex preconditions). □

In the rest of the chapter, we will denote the set of mutex pairs in $A_i$ as $\mu A_i$, and the set of mutex pairs in $P_i$ as $\mu P_i$. Let us remark that:

- dependency between actions as well as mutex between actions or proposition are *symmetrical* relations;
- for $\forall i : P_{i-1} \subseteq P_i$, and $A_{i-1} \subseteq A_i$

**Proposition 6.9** *If two propositions $p$ and $q$ are in $P_{i-1}$ and $(p, q) \notin \mu P_{i-1}$ then $(p, q) \notin \mu P_i$. If two actions $a$ and $b$ are in $A_{i-1}$ and $(a, b) \notin \mu A_{i-1}$ then $(a, b) \notin \mu A_i$.*

Table 6.1: Mutex actions and propositions.

| **Level** | **Mutex elements** |
|---|---|
| $A_1$ | {Mr12} ×{ Lar1} |
| | {Mq21} ×{ Lbq2} |
| $P_1$ | $\{r_2\} \times \{ r_1, a_r\}$ |
| | $\{q_1\} \times \{ q_2, b_q\}$ |
| | $\{a_r\} \times \{ a_1, u_r\}$ |
| | $\{b_q\} \times \{ b_2, u_q\}$ |
| $A_2$ | {Mr12} ×{ Mr21, Lar1, Uar1} |
| | {Mr21} ×{ Lbr2, Lar1*, Uar1*} |
| | {Mq12} ×{ Mq21, Laq1, Lbq2*, Ubq2*} |
| | {Mq21} ×{ Lbq2, Ubq2} |
| | {Lar1} ×{ Uar1, Laq1, Lbr2} |
| | {Lbr2} ×{ Ubq2, Lbq2, Uar1, Mr12*} |
| | {Laq1} ×{ Uar1, Ubq2, Lbq2, Mq21*} |
| | {Lbq2} ×{ Ubq2} |
| $P_2$ | $\{b_r\} \times \{ r_1, b_2, u_r, b_q, a_r\}$ |
| | $\{a_q\} \times \{ q_2, a_1, u_q, b_q, a_r\}$ |
| | $\{r_1\} \times \{ r_2\}$ |
| | $\{q_1\} \times \{ q_2\}$ |
| | $\{a_r\} \times \{ a_1, u_r\}$ |
| | $\{b_q\} \times \{ b_2, u_q\}$ |
| $A_3$ | {Mr12} ×{ Mr21, Lar1, Uar1, Lbr2*, Uar2*} |
| | {Mr21} ×{ Lbr2, Uar2, Ubr2} |
| | {Mq12} ×{ Mq21, Laq1, Uaq1, Ubq1, Ubq2*} |
| | {Mq21} ×{ Lbq2, Ubq2, Laq1*, Ubq1*} |
| | {Lar1} ×{ Uar1, Uaq1, Laq1, Uar2, Ubr2, Lbr2, Mr21*} |
| | {Lbr2} ×{ Ubr2, Ubq2, Lbq2, Uar1, Uar2, Ubq1*} |
| | {Laq1} ×{ Uar1, Uaq1, Ubq1, Ubq2, Lbq2, Uar2*} |
| | {Lbq2} ×{ Ubr2, Ubq2, Uaq1, Ubq1, Mq12*} |
| | {Uaq1} ×{ Uar1, Uar2, Ubq1, Ubq2, Mq21}* |
| | {Ubr2} ×{ Uar1, Uar2, Ubq1, Ubq2, Mr12}* |
| | {Uar1} ×{ Uar2, Mr21*} |
| | {Ubq1} ×{ Ubq2} |
| $P_3$ | $\{a_2\} \times \{ a_r, a_1, r_1, a_q, b_r\}$ |
| | $\{b_1\} \times \{ b_q, b_2, q_2, a_q, b_r\}$ |
| | $\{a_r\} \times \{ u_r, a_1, a_q, b_r\}$ |
| | $\{b_q\} \times \{ u_q, b_2, a_q, b_r\}$ |
| | $\{a_q\} \times \{ a_1, u_q\}$ |
| | $\{b_r\} \times \{ b_2, u_r\}$ |
| | $\{r_1\} \times \{ r_2\}$ |
| | $\{q_1\} \times \{ q_2\}$ |

**Proof** Every proposition $p$ in a level $P_i$ is supported by at least its no-op action $\alpha_p$. Two no-op actions are necessarily independent. If $p$ and $q$ in $P_{i-1}$ are such that $(p, q) \notin \mu P_{i-1}$ then $(\alpha_p, \alpha_q) \notin \mu A_i$. Hence, a non-mutex pair of propositions remains non-mutex in the following level. Similarly, if $(a, b) \notin \mu A_{i-1}$ then $a$ and $b$ are independent and their preconditions in $P_{i-1}$ are not mutex; both properties remain valid at the following level. □

According to this result, propositions and actions in a planning graph monotonically increase from one level to the next, while mutex pairs monotonically decrease. These monotonicity properties are essential to the complexity and termination of the planning graph techniques.

**Proposition 6.10** *A set $g$ of propositions is reachable from $s_0$ only if there is in the corresponding planning graph a proposition layer $P_i$ such that $g \in P_i$ and no pair of propositions in $g$ are in $\mu P_i$.*

## 6.3 The Graphplan Planner

The Graphplan algorithm performs a procedure close to *iterative deepening*, discovering a new part of the search space at each iteration. It iteratively expands the planning graph by one level, then it searches backward from the last level of this graph for a solution. The first expansion, however, proceeds to a level $P_i$ in which all of the goal propositions are included and no pair of them are mutex, since it does not make sense to start searching a graph that does not meet the necessary condition of Proposition 6.10.

The iterative loop of graph expansion and search is pursued until either a plan is found or a failure termination condition is met. Let us detail the algorithm and its properties.

### 6.3.1 Expanding the Planning Graph

Let $(O, s_0, g)$ be a planning problem in the classical representation such that $s_0$ and $g$ are sets of propositions and operators in $O$ have no negated literals in their preconditions. Let $A$ be the union of all ground instances of operators in $O$ and of all no-op actions $\alpha_p$ for every proposition $p$ of that problem; the no-op action for $p$ is defined as $\text{precond}(\alpha_p) = \text{effects}^+(\alpha_p) = \{p\}$, and $\text{effects}^- = \emptyset$. A planning graph for that problem expanded up to level $i$ is a sequence of layers of nodes and of mutex pairs

$$G = \langle P_0, A_1, \mu A_1, P_1, \mu P_1, \ldots, A_i, \mu A_i, P_i, \mu P_i \rangle.$$

This planning graph does not depend on $g$; it can be used for different planning problems that have the same set of planning operators $O$ and initial state $s_0$.

Starting initially from $P_0 \leftarrow s_0$, the expansion of $G$ from level $i-1$ to level $i$ is given by the Expand procedure (figure 6.5). The steps of this procedure correspond respectively to generating the sets $A_i, P_i, \mu A_i$, and $\mu P_i$ from the elements in the previous level $i-1$.

Expand($\langle P_0, A_1, \mu A_1, P_1, \mu P_1, \ldots, A_{i-1}, \mu A_{i-1}, P_{i-1}, \mu P_{i-1}\rangle$)
  $A_i \leftarrow \{a \in A \mid \text{precond}(a) \subseteq P_{i-1}$ and $\text{precond}^2(a) \cap \mu P_{i-1} = \emptyset\}$
  $P_i \leftarrow \{p \mid \exists a \in A_i : p \in \text{effects}^+(a)\}$
  $\mu A_i \leftarrow \{(a,b) \in A_i^2, a \neq b \mid \text{effects}^-(a) \cap [\text{precond}(b) \cup \text{effects}^+(b)] \neq \emptyset$
    or $\text{effects}^-(b) \cap [\text{precond}(a) \cup \text{effects}^+(a)] \neq \emptyset$
    or $\exists (p,q) \in \mu P_{i-1} : p \in \text{precond}(a), q \in \text{precond}(b)\}$
  $\mu P_i \leftarrow \{(p,q) \in P_i^2, p \neq q \mid \forall a, b \in A_i, a \neq b :$
    $p \in \text{effects}^+(a), q \in \text{effects}^+(b) \Rightarrow (a,b) \in \mu A_i\}$
  for each $a \in A_i$ do: link $a$ with precondition arcs to $\text{precond}(a)$ in $P_{i-1}$
    positive arcs to $\text{effects}^+(a)$ and negative arcs to $\text{effects}^-(a)$ in $P_i$
  return($\langle P_0, A_1, \mu A_1, \ldots, P_{i-1}, \mu P_{i-1}, A_i, \mu A_i, P_i, \mu P_i\rangle$)
end

Figure 6.5: Expansion of a Planning graph

Let us analyze some properties of a planning graph.

**Proposition 6.11** *The size of a planning graph down to level $k$ and the time required to expand it to that level are polynomial in the size of the planning problem.*

**Proof** If the planning problem $(O, s_0, g)$ has a total of $n$ propositions and $m$ actions, then $\forall i : |P_i| \leq n$, and $|A_i| \leq m + n$ (including no-op actions), and $|\mu A_i| \leq (m+n)^2$, and $|\mu P_i| \leq n^2$. The steps involved in the generation of these sets are of polynomial complexity in the size of the sets.

Furthermore, $n$ and $m$ are polynomial in the size the problem $(O, s_0, g)$. This is the case since, according to our assumption A0, operators cannot create new constant symbols. Hence, if $c$ is the number of constant symbols given in the problem, $e$=$\max_{o \in O}\{|\text{effects}^+(o)|\}$, and $\alpha$ is an upper bound on the number of parameters of any operator, then $m \leq |O| \times c^\alpha$, and $n \leq |s_0| + e \times |O| \times c^\alpha$. □

Moreover, the number of distinct levels in a planning graph is bounded: at some stage, the graph reaches a *fixed-point* level, as defined bellow.

**Definition 6.12** A fixed-point level in a planning graph $G$ is a level $\kappa$ such that for $\forall i, i > \kappa$, level $i$ of $G$ is identical to level $\kappa$, that is $P_i = P_\kappa$, $\mu P_i = \mu P_\kappa$, $A_i = A_\kappa$ and $\mu A_i = \mu A_\kappa$. □

**Proposition 6.13** *Every planning graph $G$ has a fixed-point level $\kappa$ which is the smallest $k$ such that $|P_{k-1}| = |P_k|$ and $|\mu P_{k-1}| = |\mu P_k|$.*

**Proof** To show that the planning graph has a fixed-point level, notice that (i) there is a finite number of propositions in a planning problem, (ii) $\forall i$, $P_{i-1} \subseteq P_i$, and (iii) if a pair $(p, q) \notin \mu P_{i-1}$ then $(p, q) \notin \mu P_i$. Hence, a proposition level $P_i$ either has more propositions than $P_{i-1}$ or it has fewer mutex pairs. Since these monotonic differences are bounded, at some point $P_{i-1} = P_i$, $\mu P_{i-1} = \mu P_i$. Hence $A_{i+1} = A_i$ and $\mu A_{i+1} = \mu A_i$.

Now, suppose that $|P_{k-1}| = |P_k|$ and $|\mu P_{k-1}| = |\mu P_k|$; let us show that all levels starting at $k$ are identical:

- Since $(|P_{k-1}| = |P_k|)$ and $\forall i$, $P_{i-1} \subseteq P_i$ it follows that $(P_{k-1} = P_k)$.
- Since $(P_{k-1} = P_k)$ and $(|\mu P_{k-1}| = |\mu P_k|)$, $\mu P_{k-1} = \mu P_k$. This is the case since a non-mutex pair of propositions at $k-1$ remains non-mutex at level $k$ (Proposition 6.9).
- $A_{k+1}$ depends only on $P_k$ and $\mu P_k$. Thus $(P_{k-1} = P_k)$ and $(\mu P_{k-1} = \mu P_k)$ implies $A_{k+1} = A_k$, and consequently $P_{k+1} = P_k$. The two sets $A_{k+1}$ and $A_k$ have the same dependency constraints (that are intrinsic to actions) and the same mutex between their preconditions (since $\mu P_{k-1} = \mu P_k$), thus $\mu A_{k+1} = \mu A_k$. Consequently $\mu P_{k+1} = \mu P_k$.

Level $k+1$ being identical to level $k$, the same level will repeat for all $i$ such that $i > k$. □

### 6.3.2 Searching the Planning Graph

The search for a solution plan in a planning graph proceeds back from a level $P_i$ that includes all goal propositions, no pair of which being mutex, that is, $g \in P_i$ and $g^2 \cap \mu P_i = \emptyset$. The search procedure looks for a set $\pi_i \in A_i$ of non-mutex actions that achieve these propositions. Preconditions of elements of $\pi_i$ become the new goal for level $i-1$ and so on. A failure to meet the goal of some level $j$ leads to a backtrack over other subsets of $A_{j+1}$. If level 0 is successfully reached, then the corresponding sequence $\langle \pi_1, \ldots, \pi_i \rangle$ is a solution plan.

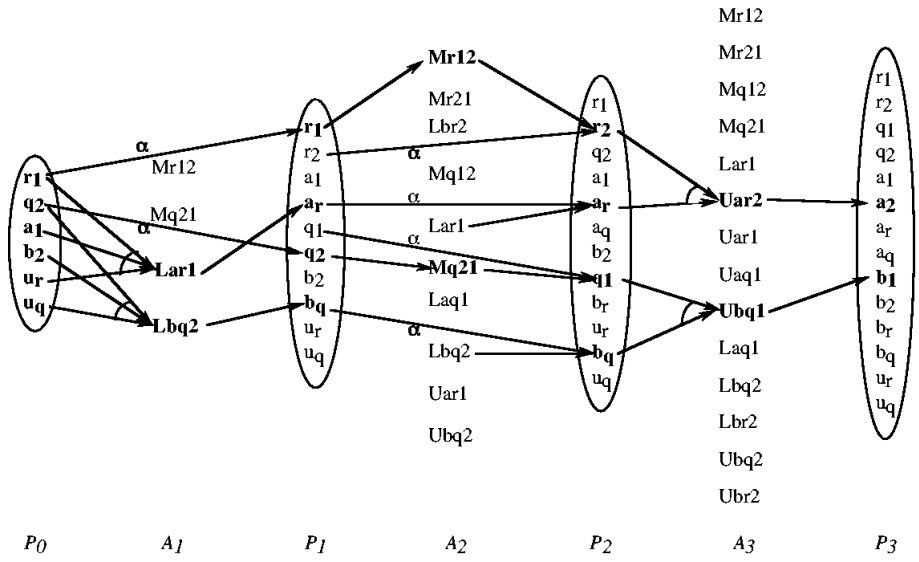



Figure 6.6: A solution plan

**Example 6.14** The goal $g = \{a_2, b_1\}$ of the previous example is in $P_3$ without mutex (see figure 6.6 where goal propositions and selected actions are shown in bold). The only actions in $A_3$ achieving $g$ are respectively Uar2 and Uqb1. They are non-mutex, hence $\pi_3 = \{Uar2, Ubq1\}$.

At level 2, the preconditions of the actions in $\pi_3$ become the new goal: $\{r_2, a_r, q_1, b_q\}$. $r_2$ is achieved by $\alpha_{r_2}$ or by Mr12 in $A_2$; $a_r$ by $\alpha_{a_r}$ or by Lar1. Out of the 4 combinations of these actions, 3 are mutex pairs: (Mr21, Lar1), ($\alpha_{r_2}$, Lar1) and ($\alpha_{r_2}$, $\alpha_{a_r}$), the last two are mutex because they require mutex preconditions $(r_1, r_2)$ and $(r_2, a_r)$ in $P_1$. Similarly for the 2 couples of actions achieving $q_1$ and $b_q$: (Mq21, Lbq2), ($\alpha_{q_1}$, Lbq2) and ($\alpha_{q_1}$, $\alpha_{b_q}$) are mutex pairs. Hence the only possibility in $A_2$ for achieving this subgoal is the subset $\pi_2$={Mr12, $\alpha_{a_r}$, Mq21, $\alpha_{b_q}$}.

At level 1, the new goal is $\{r_1, a_r, q_2, b_q\}$. Its propositions are achieved respectively by $\alpha_{r_1}$, Lar1, $\alpha_{q_2}$, Lbq2.

Level 0 is successfully reached.

The solution plan is thus the sequence of subsets, without no-op actions: $\Pi = \langle\{Lar1, Lbq2\}, \{Mr12, Mq21\}, \{Uar2, Ubq1\}\rangle$. □

The extraction of a plan from a planning graph corresponds to a search in an *And/Or* subgraph of the planning graph:

- from a proposition in goal $g$, *Or-branches* are arcs from all actions in the

```
Extract(G, g, i)
    if i = 0 then return (⟨⟩)
    if g ∈ ∇(i) then return(failure)
    π_i ← GP-Search(G, g, ∅, i)
    if π_i ≠ failure then return(π_i)
    ∇(i) ← ∇(i) ∪ {g}
    return(failure)
end
```

Figure 6.7: Extraction of a plan for a goal $g$

preceding action level that support this proposition, i.e., positive arcs to that proposition;

- from an action node, *And-branches* are its precondition arcs (shown in fig.6.6 as connected arcs).

The mutex relation between propositions provide only forbidden pairs, not tuples. But the search may show that a tuple of more than two propositions corresponding to an intermediate subgoal fails. Because of the backtracking and iterative deepening, the search may have to analyze that same tuple more than once. Recording the tuples that failed may save future search. This recording is performed by procedure Extract (fig. 6.7) into a *nogood* hash-table denoted $\nabla$. This hash table is indexed by the level of the failed goal, since a goal $g$ may fail at level $i$ and succeed at $j > i$.

Extract takes as input a planning graph $G$, a current set of goal propositions $g$ and a level $i$. It extracts a set of actions $\pi_i \subseteq A_i$ that achieves propositions of $g$ by recursively calling the GP-Search procedure (fig. 6.8). If it succeeds in reaching level 0 then it returns an empty sequence, from which pending recursions returns successfully a solution plan. It records failed tuples into the $\nabla$ table and it checks each current goal with respect to recorded tuples. Note that a tuple $g$ is added to the nogood table at a level $i$ only if the call to GP-search fails at establishing $g$ at this level from mutex and other nogoods found or established at the previous level.

The GP-Search procedure selects each goal proposition $p$ at a time, in some heuristic order. Among the *resolvers* of $p$, i.e., actions that achieve $p$ and that are not mutex with already selected actions for that level, it non-deterministically chooses one action $a$ which tentatively extends the current subset $\pi_i$ through a recursive call at the same level. This is performed on a subset of goals reduced by $p$ and by any other positive effect of $a$ in $g$. As usual, a failure for this non-deterministic choice is a backtrack point over

```
GP-Search(G, g, π_i, i)
    if g = ∅ then do
        Π ← Extract(G, ⋃{precond(a) | ∀a ∈ π_i}, i − 1)
        if Π = failure then return(failure)
        return(Π.⟨π_i⟩)
    else do
        select any p ∈ g
        resolvers ← {a ∈ A_i | p ∈ effects⁺(a) and ∀b ∈ π_i : (a, b) ∉ μA_i}
        if resolvers = ∅ then return(failure)
        nondeterministically choose a ∈ resolvers
        return(GP-Search(G, g − effects⁺(a), π_i ∪ {a}, i))
end
```

Figure 6.8: Search for actions $\pi_i \in A_i$ that achieve goal $g$

other alternatives for achieving $p$, if any, or a backtracking further up if all *resolvers* of $p$ have been tried out. When $g$ is empty then $\pi_i$ is complete; the search recursively tries to extract a solution for the following level $i-1$.

One may view the GP-Search procedure as a kind of CSP solver.[4] Here CSP *variables* are goal propositions, their *values* are possible actions achieving them. The procedure chooses a value for a variable compatible with previous choices (non-mutex), and recursively tries to solve other pending variables. This view can be very beneficial if one applies to procedure GP-Search the CSP heuristics, e.g., for the ordering of variables and for the choice of values, and techniques such as intelligent backtracking or forward propagation. The latter is easily added to the procedure: before recursion, a potential value $a$ for achieving $p$ is propagated forward on resolvers of pending variables in $g$; $a$ is removed from consideration if it leads to an empty resolver for some pending goal proposition.

We are now ready to specify the Graphplan algorithm (fig. 6.9) with the graph expansion and search steps, and the termination condition. Graphplan performs an initial graph expansion until either it reaches a level containing all goal propositions without mutex, or until it arrives at a fixed-point in $G$. If the latter happens first, then the goal is not achievable. Otherwise a search for a solution is performed. If no solution is found at this stage, the algorithm iteratively expands, then searches the graph $G$.

This iterative deepening is pursued even *after* a fixed-point has been reached, until success or until the termination condition is satisfied. This

[4] This view will be further detailed in Chapter 8

```
Graphplan(A, s_0, g)
    i ← 0,    ∇ ← ∅ ,    P_0 ← s_0
    G ← ⟨P_0⟩
    until [g ⊆ P_i and g^2 ∩ μP_i = ∅] or Fixedpoint(G) do
        i ← i + 1
        G ← Expand(G)
    if g ⊈ P_i or g^2 ∩ μP_i ≠ ∅ then return(failure)
    Π ← Extract(G, g, i)
    if Fixedpoint(G) then η ← |∇(κ)|
    else η ← 0
    while Π = failure do
        i ← i + 1
        G ← Expand(G)
        Π ← Extract(G, g, i)
        if Π= failure and Fixedpoint(G) then
            if η = |∇(κ)| then return(failure)
            η ← |∇(κ)|
    return(Π)
end
```

Figure 6.9: The Graphplan Algorithm

termination condition requires that the number of nogood tuples in $\nabla(\kappa)$ at the fixed-point level $\kappa$, stabilizes after two successive failures.

In addition to Expand and Extract, the Graphplan algorithm calls the procedure Fixedpoint($G$) that checks the fixed-point condition; this procedure sets $\kappa$ to the fixed-point level of the planning graph when the fixed-point is reached.

### 6.3.3 Analysis of Graphplan

In order to prove the soundness, completeness and termination of Graphplan, let us first analyze how the nogood table evolves along successive deepening stages of $G$. Let $\nabla_j(i)$ be the set of nogood tuples found at level $i$ after the unsuccessful completion of a deepening stage down to a level $j > i$. The failure of stage $j$ means that any plan of $j$ or less steps must make at least one the goal tuples in $\nabla_j(i)$ true at a level $i$, and that none of these tuples is achievable in $i$ levels.

**Proposition 6.15** *$\forall i, j$ such that $j > i$, $\nabla_j(i) \subseteq \nabla_{j+1}(i)$*

**Proof** A tuple of goal propositions $g$ is added as a nogood in $\nabla_j(i)$ only when Graphplan has performed an exhaustive search for all ways of achieving $g$ with the actions in $A_i$ and it fails: each subset of actions in $A_i$ providing $g$ is either mutex or it involves a tuple of preconditions $g'$ that was shown to be a nogood at the previous level $\nabla_k(i-1)$, for $i < k \leq j$. In other words, only the levels from 0 to $i$ in $G$ are responsible for the failure of the tuple $g$ at level $i$. By iterative deepening, the algorithm may find that $g$ is solvable at some later level $i' > i$, but regardless of how many iterative deepening stages are performed, once $g$ is in $\nabla_j(i)$ it remains in $\nabla_{j+1}(i)$ and in the nogood table at the level $i$ in all subsequent deepening stages. □

**Proposition 6.16** *The Graphplan algorithm is sound, complete, and it terminates: it returns failure iff the planning problem $(O, s_0, g)$ has no solution, otherwise it returns a sequence of sets of actions $\Pi$ that is a solution plan to the problem.*

**Proof** To show the soundness of the algorithm, assume that Graphplan returns the sequence $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$. The set *resolvers*, as defined in GP-Search, is such that every set of actions $\pi_i \in \Pi$ is independent. Furthermore, the set of actions $\pi_n$ achieves the set of problem goals, $\pi_{n-1}$ achieves precond$(\pi_n)$, etc. Finally, when GP-Search calls Extract on the recursion $i = 1$, we are sure that all precond$(\pi_1)$ are in $P_0$. Hence the layered plan $\Pi$ meets Definition 6.5 of a solution plan to the problem.

Suppose that instead of finding a solution, the algorithm stops on one of the two failure termination conditions, that is either

- the fixed point $\kappa$ is reached before attaining a level $i$ that contains all goal propositions, no pair of which being mutex, or
- there are two successive deepening stages such that $|\nabla_{j-1}(\kappa)| = |\nabla_j(\kappa)|$.

In the former case $G$ does not have a level that meets the necessary condition of Proposition 6.10, hence the problem is unsolvable.
In the latter case:

- $\nabla_{j-1}(\kappa) = \nabla_j(\kappa)$; this is because of Proposition 6.15, and
- $\nabla_{j-1}(\kappa) = \nabla_j(\kappa + 1)$; this is because the last $i - \kappa$ levels are identical, that is, $\nabla_{j-1}(\kappa)$ is to stage $j - 1$ what $\nabla_j(\kappa + 1)$ is to stage $j$.

These two equations entails $\nabla_j(\kappa) = \nabla_j(\kappa + 1)$: all unsolvable goal tuples at the fixed point level (including the original goals of the problem) are also unsolvable at the next level $\kappa + 1$. Hence the problem is unsolvable.

Finally, we have to show that Graphplan necessarily stops when the planning problem is unsolvable. Because of Proposition 6.15, the number of nogood goal tuples at any level grows monotonically, and there is a finite maximum number of goal tuples. Hence, there is necessarily a point where the second failure termination condition is reached, if the first failure condition did not apply before. □

To end this section, let us underline two main features of Graphplan:

- The mutex relation on incompatible pairs of actions and propositions, and the weak reachability condition of Proposition 6.10, offer a very good insight about the interaction between the goals of a problem, and about which goals are possibly achievable at some level.
- Because of the monotonic properties of the planning graph, the algorithm is guaranteed to terminate; the fixed-point feature together with the reachability condition provide an efficient failure termination condition, in particular when the goal propositions without mutex are not reachable no search at all is performed.

Because of these features and of its backward constraint directed search, Graphplan brought a significant speed-up and contributed to the scalability of planning. Evidently, Graphplan does not change the intrinsic complexity of planning, which is PSPACE-complete in the set theoretic representation. Since we showed that the expansion of the planning graph is performed in polynomial time (Proposition 6.11), this means that the costly part of the algorithm is in the search of the planning graph. Furthermore, the memory requirement of the planning graph data structure can be a significant limiting factor. Several techniques and heuristics have been devised to speed-up the search and to improve the memory management of its data structure. They will be introduced in the next section.

## 6.4 Extensions and Improvements of Graphplan

### 6.4.1 Extending the Language

The planning algorithm described in the previous section takes as input a problem $(O, s_0, g)$ which is stated in a restricted classical representation where $s_0$ and $g$ are sets of propositions and operators in $O$ have no negated literals in their preconditions. For a realistic problem, a more expressive language is desirable. Let us illustrate here how some of the extensions of the classical representation described in Section 2.4 can be taken into account in Graphplan.

Handling negation in the preconditions of operators and in goals is easily performed by introducing a new predicate *not-p* to replace the negation of a predicate $p$ in preconditions or goals (see Section 2.6). This replacement requires

- adding *not-p* in effects$^-$ when $p$ is in effects$^+$ of an operator $o$, and
- adding *not-p* in effects$^+$ when $p$ is in effects$^-$ of $o$

One has also to extend $s_0$ with respect to the newly introduced *not-p* predicates is order to maintain a consistent and *closed*[5] initial world.

**Example 6.17** The DWR domain has the following operator:

move$(r, l, m)$ ;; robot $r$ moves from location $l$ to location $m$
precond: adjacent$(l, m)$, at$(r, l)$, $\neg$ occupied$(m)$
effects: at$(r, m)$, occupied$(m)$, $\neg$ occupied$(l)$, $\neg$ at$(r, l)$

The negation in the precondition is handled by introducing the predicate not-occupied in the following way:

move$(r, l, m)$ ;; robot $r$ moves from location $l$ to location $m$
precond: adjacent$(l, m)$, at$(r, l)$, not-occupied$(m)$
effects: at$(r, m)$, occupied$(m)$, $\neg$ occupied$(l)$, $\neg$ at$(r, l)$,
not-occupied$(l)$, $\neg$not-occupied$(m)$

Furthermore, if a problem has three locations l1, l2, l3, such that only l1 is initially occupied, we need to add to the initial state the propositions: not-occupied(l2), not-occupied(l3). □

This approach, which rewrites a planning problem into the restricted representation required by Graphplan, can also be used for handling the other extensions discussed in Section 2.4. For example, recall that an operator with a conditional effect can be expanded into an equivalent set of pairs $(\text{precond}_i, \text{effects}_i)$. Hence it is easy to rewrite it as several operators, one for each such a pair. Quantified conditional effects are similarly expanded. However, such an expansion may lead to an exponential number of operators. It is preferable to generalize the algorithm for handling directly an extended language.

Generalizing Graphplan for handling directly operators with disjunctive preconditions is easily done by considering the edges from an action in $A_i$ to its preconditions in $P_{i-1}$ as being a disjunctive set of *And-connectors*,

[5] that is, any proposition that is not explicitly stated is false

as in And-Or graphs. The definition of mutex between actions needs to be generalized with respect to these connectors. The set of *resolvers* in GP-search among which a nondeterministic choice is made for achieving a goal, has now to take into account not the actions, but their And-connectors (see Exercise 6.12).

Handling directly operators with conditional effects requires more significant modifications. One has to start with a generalized definition of dependency between actions taking into account their conditional effects. This is needed in order to keep the desirable result of Proposition 6.4, i.e., that an independent set of actions defines the same state transitions for any permutation of the set. One has also to define a new structure of the planning graph for handling the conditional effects, e.g., for propagating a desired goal at level $P_i$ which is a conditional effect, over to its antecedent condition, either in a positive or in a negative way. One has also to come up with ways for computing and propagating mutex relations, and with a generalization of the search procedure in this new planning graph. For example, the planner called IPP labels an edge from an action to a proposition by the conditions under which this proposition is an effect of the action. These labels are taken into account for the graph expansion and search. However, they are not exploited for finding all possible mutex, hence leaving a heavier load on the search.

### 6.4.2 Improving the Planner

**Memory management.** The planning graph data structure makes explicit all the ground atoms and instantiated actions of a problem. It has to be implemented carefully in order to maintain a reasonable memory demand that is not a limiting factor to the planner's performance.

The monotonic properties of the planning graph are essential to this purpose. Since $P_{i-1} \subseteq P_i$, and $A_{i-1} \subseteq A_i$, one does not need to keep these sets explicitly, but only to record for each proposition $p$ the level $i$ at which $p$ appeared for the first time in the graph, and similarly for each action.

Because of Proposition 6.9, a symmetrical technique can be used for the mutex relations, that is, to record the level at which a mutex disappeared for the first time. Furthermore, there is no need to record the planning graph after it's fixed point level $\kappa$. One has just to maintain the only changes that can appear after this level, i.e., in the nogood table of non-achievable tuples. Here also the monotonic property of Proposition 6.15, i.e., $\nabla_j(i) \subseteq \nabla_{j+1}(i)$, allows an incremental management.

Finally, several general programming techniques can also be useful for

the memory management. For example, the bitvector data structure allows to encode a state and a proposition level $P_i$ as a vector of $n$ bits, where $n$ is the number of propositions in the problem; an action is encoded as four such vectors for its positive and negative preconditions and effects.

**Focusing and improving the search.** The description of a domain involves rigid predicates that do not vary from state to state. In the DWR domain for example, the predicates adjacent, attached and belong are rigid: there is no operator that changes their truth value. Once operators are instantiated into ground actions for a given problem, one may remove the rigid predicates from preconditions and effects because they play no further role in the planning process. This simplification reduces the number of actions. For example, there will be no action load(crane3,loc1,cont2,rob1) if belong(crane3,loc1) is false, i.e., if crane3 is not in location loc1. Because of this removal, one may also have flexible predicates that become invariant for a given problem, triggering more removals. There can be a great benefit in preprocessing a planning problem in order to focus the processing and the search on the sole relevant facts and actions. This preprocessing can be quite sophisticated and may allow to infer non obvious object types, symmetries and invariant properties, such as permanent mutex relations, hence simplifying the mutex computation. It may even find mutex propositions that cannot be detected by Graphplan because of the binary propagation.

Nogood tuples, as well as mutex, play an essential role in pruning the search. However, if we are searching to achieve a set of goals $g$ in a level $i$, and if there $g' \in \nabla_i$ such that $g' \subset g$, we will not detect that $g$ is not achievable and prune the search. The Extract procedure can be extended to test this type of set inclusion, but this may involve a significant overhead. It turns out however that the termination condition of the algorithm, i.e., $|\nabla_{j-1}(\kappa)| = |\nabla_j(\kappa)|$, holds even if the procedure records and keeps in $\nabla_i$ only nogood tuples $g$ such that no subset of $g$ has been proven to be a nogood. With this modification the set inclusion test can be efficiently implemented.

In addition to pruning, the GP-search procedure has to be focused with heuristics for selecting the next proposition $p$ in the current set $g$ and for nondeterministically choosing the action in *resolvers*. A general heuristics consists in selecting first a proposition $p$ which leads to the smallest set of *resolvers*, i.e., the proposition $p$ achieved by the smallest number of actions. For example, if $p$ is achieved by just one action, then $p$ does not involve a backtrack point and is better processed as early as possible in the search tree. A symmetrical heuristics for the choice of an action supporting $p$ is to prefer

no-op first. Other heuristics that are more specific to the planning graph structure and more informed take into account the level at which actions and propositions appear for the first time in the graph. The later a proposition appears in the planning graph, the most constrained it is. Hence, one would select the latest propositions first. A symmetrical reasoning leads to choose for achieving $p$ the action that appears the earliest in the graph.[6]

There are finally a number of algorithmic techniques that allow to improve the efficiency of the search. One of them is for example the *forward-checking* technique: before choosing an action $a$ in *resolvers* for handling $p$, one checks that this choice will not leave another pending proposition in $g$ with an empty set of resolvers. Forward-checking is a general algorithm for solving constraint satisfaction problems. It turns out that several other CSP techniques are applicable to the search in a planning graph, which is a particular CSP problem.[7]

### 6.4.3 Extending the Independence Relation

We introduced the concept of layered plans with a very strong requirement of *independent* actions in each set $\pi_i$. In practice, we do not necessarily need to have *every* permutation of each set as a valid sequence of actions. We only need to ensure that there exist *at least one* such permutation. This is the purpose of the relation between actions, called the *allowance* relation, which is less constrained than the independence relation while keeping the advantages of the planning graph.

An action $a$ *allows* an action $b$ when $b$ can be applied after $a$ and the resulting state contains the union of the positive effects of $a$ and $b$. This is the case when $a$ does not delete a precondition of $b$ and $b$ does not delete a positive effect of $a$:

$$a \text{ allows } b \text{ iff } \text{effects}^-(a) \cap \text{precond}(b) = \emptyset \text{ and } \text{effects}^-(b) \cap \text{effects}^+(a) = \emptyset$$

Allowance is weaker than independence. Independence implies allowance: if $a$ and $b$ are independent then $a$ allows $b$ and $b$ allows $a$. Note that when $a$ allows $b$ but $b$ does not allow $a$ then $a$ has to be ordered before $b$. Note also that allowance is not a symmetrical relation.

If we replace the independence relation with the allowance relation in Definition 6.7 we can say that two actions $a$ and $b$ are mutex either:

[6] This topic will be further developed in Chapter 9 devoted to heuristics.

[7] This point is developed in Section 8.6.2 of the CSP chapter.

- when they have mutually exclusive preconditions, or
- when $a$ does not allow $b$ and $b$ does not allow $a$.

This definition leads to fewer mutex pairs between actions, and consequently to fewer mutex between propositions. On the same planning problem, the planning graph will have fewer or at most the same number of levels, before reaching a goal or a fixed point, than with the independence relation.

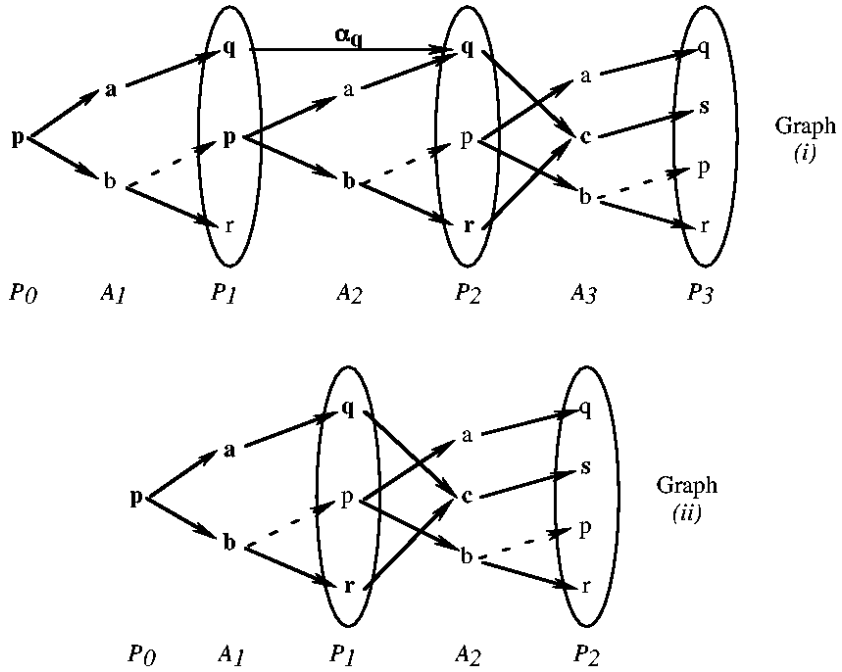



Figure 6.10: Planning graphs for independence $(i)$ and allowance $(ii)$ relations.

**Example 6.18** Let us illustrate the difference entailed by the two relations on a simple planning domain that has three actions $a$, $b$ and $c$ and four propositions $p, q, r$, and $s$:

- $\text{precond}(a) = \{p\}; \text{effects}^+(a) = \{q\}; \text{effects}^-(a) = \{\}$
- $\text{precond}(b) = \{p\}; \text{effects}^+(b) = \{r\}; \text{effects}^-(b) = \{p\}$
- $\text{precond}(c) = \{q, r\}; \text{effects}^+(c) = \{s\}; \text{effects}^-(c) = \{\}$

Actions $a$ and $b$ are not independent ($b$ deletes a precondition of $a$), hence they will be mutex in any level of the planning graph built with independence relation. However, $a$ allows $b$: these actions will not be mutex with

the allowance relation. The two graphs are illustrated in Figure 6.10 for a problem whose initial state is {p} and goal is {s} (solution plans are shown in bold). In the graph $(i)$ with the independence relation, preconditions of $c$ are mutex in $P_1$; because of the no-op $\alpha_q$ they become non-mutex in $P_2$; action $c$ appears in $A_2$ giving the goal in $P_3$. In the graph $(ii)$ with the allowance relation, $q$ and $r$ are non-mutex in $P_1$, the goal is reached one level earlier. □

The benefit of the allowance relation, that is fewer mutex pairs and a smaller fixed point level, has a cost. Since the allowance relation is not symmetrical, a set of pairwise non-mutex actions does not necessarily contain a "valid" permutation. For example, if action $a$ allows $b$, $b$ allows $c$ and $c$ allows $a$ but none of the opposite relations holds, then the three actions $a, b$ and $c$ can be non-mutex (pending non-mutex preconditions) but there is no permutation that gives an applicable sequence of actions and a resulting state corresponding to the union of their positive effects. While earlier a set of non-mutex actions was necessarily independent and could be selected in the search phase for a plan, here we have to add a further requirement for the allowance relation within a set.

A permutation $\langle a_1, \ldots, a_k \rangle$ of the elements of a set $\pi_i$ is *allowed* if every action allows all its followers in the permutation, i.e., $\forall j, k$ : if $j < k$ then $a_j$ allows $a_k$. A set is allowed if it has at least one allowed permutation.

The state resulting from the application of an allowed set can be defined as in the previous section: $\gamma(s, \pi_i) = (s - \text{effects}^-(\pi_i)) \cup \text{effects}^+(\pi_i)$. All propositions of section 6.3.3 can be rephrased for an allowed set and for a layered plan whose levels are allowed sets by just replacing "any permutation" with "any allowed permutation" (see Exercise 6.14).

In order to compute $\gamma(s, \pi_i)$ and to use such a set in the GP-Search procedure, one does not need to produce an allowed permutation and to commit the plan to it, one just need to check its existence. We already noticed that an ordering constraint "$a$ before $b$" would be required whenever $a$ allows $b$ but $b$ does not allow $a$. It is easy to prove that a set is allowed if and only if the relation consisting of all pairs $(a, b)$ such that "$b$ does not allow $a$" is cycle free. This can be checked with a topological sorting algorithm in complexity that is linear in the number of actions and allowance pairs. Such a test has to take place in the GP-Search procedure right before recursion on the following level:

GP-Search($G, g, \pi_i, i$)
  if $g = \emptyset$ then do
    if $\pi_i$ is not allowed then return(failure)
    $\Pi \leftarrow$ Extract($G, \bigcup\{\text{precond}(a) \mid \forall a \in \pi_i\}, i-1$)
  ... etc (as in figure 6.8)

It is easy to show that with the above modification and the modification in Expand for the definition of allowance in $\mu A_i$, the Graphplan algorithm keeps the same properties of soundness, completeness and termination. The allowance relation leads to fewer mutex pairs, hence to more actions in a level and to fewer levels in the planning graph. The reduced search space pays off in the performance of the algorithm. The benefit can be very significant for highly constrained problems where the search phase is very expensive.

## 6.5 Discussion and Historical Remarks

The Graphplan planner attracted considerable attention from the research community. The original papers on this planner [70, 71] are among the most cited references in AI planning. One reason for that was the spectacular improvement in planning performance introduced by Graphplan in comparison with the earlier plan-space planners. The other and probably more important reason is the richness of the planning-graph structure, which opened the way to a broad avenue of research and extensions. At some point, a significant ratio of the papers in every planning conference was concerned with the planning graph techniques. For several years, a significant fraction of the papers in every planning conference was concerned with planning-graph techniques. A complete discussion of these papers is beyond the scope of this section. Let us discuss a few illustrative contributions.

The analysis of the planning graph as a reachability structure is due to [296] who introduced three approximations of the reachability tree (the "unioned plangraph", where every level is the union states in the corresponding level of the reachability tree, the "naive plangraph" that does not take into account mutex in action levels, and the planning graph), and who also proposed a backward construction of the planning graph starting from the goal. This paper relied on previous work, e.g. [297], to analyze Graphplan as a disjunctive refinement planner, and to propose CSP techniques for improving it [151]. Some CSP techniques, such as forward checking, were already in the initial Graphplan article [71]. But other contributions, e.g., [290, 291] elaborated further by showing that a planning graph is a dynamic CSP, and by developing intelligent backtracking and efficient recording and

management of failure tuples.

The proposal for translating a planning problem from an extended classical representation into Graphplan restricted language is due to [204]. Several contributions for handling directly and efficiently extended constructs, like conditional effects, have been proposed, e.g., [322, 18]. A significant part of this work on the language extension took part along with the development of two graphplan successors, the IPP[321] and STAN [356] planners. Many improvements to the encoding, memory management and algorithms for the planning graph techniques are due to these two planners. Several domain analysis techniques to focus the graph and the search, such as [408, 557], have been extensively developed in a system called the Type Inference Module [183] and integrated to STAN [184].

Several articles on the planning graph techniques insisted on plans with *parallel* actions as an important contribution of Graphplan. We carefully avoided mentioning parallelism is this chapter, since there is no semantics of concurrency in layered plans.[8] This is clearly illustrated in the extension from the independence to the allowance relations, or from the requirement to have all permutations of actions in a layer $\pi_i$ equivalent to the weaker requirement that there is at least one permutation that achieves the effects of $\pi_i$ from its preconditions. This extension from independence to allowance is due to [108] for a planner called LCGP.

The work on LCGP led also to contributions on level-based heuristics for Graphplan [107]. Similar heuristics were independently proposed by [295]. More elaborated heuristics relying on local search techniques were proposed in the LPG planner [211] and led to significant performances, as illustrated in AIPS'02 planning competition results [186]. A related issue that arose at that time (and to which we will come back in the Chapter 9) is the use of Graphplan not as a planner but as a technique to derive heuristics for state-based planners [411, 413, 263].

The relationship between two techniques that were developed in parallel, the planning graph and the SAT-based techniques (see next chapter) have been analyzed by several authors, among which [307, 39]

Many other extensions to Graphplan have been studied such as, for example, the planning graph techniques for handling resources [320], and for dealing with uncertainty [535, 69] or partial specification of the domain [477].

[8] See Chapter 14, and particularly Section 14.2.5 that is devoted to concurrent actions with interfering effects

## 6.6 Exercises

**6.1** Suppose we run Graphplan on the painting problem described in Exercise 5.6.

(a) How many actions does it generate at level 1 of the planning graph? How many of these are maintenance actions?

(b) Expand the planning graph out to two levels, and draw the result.

(c) What is the first level at which Graphplan calls Extract?

(d) At what level will GraphPlan find a solution? What solution will it find?

(e) If we kept generating the graph out to infinity rather than stopping when Graphplan finds a solution, what is the first level of the graph at which the number of actions would reach its maximum?

**6.2** Redo Exercise 6.1 on the washing problem described in Exercise 5.7.

**6.3** How many times will Graphplan need to do graph expansion if we run it on the Sussman anomaly (see Example 4.5)?

**6.4** Let $P = (O, s_0, g)$ and $P' = (O, s_0, g')$ be the statements of two solvable planning problems such that $g \subseteq g'$. Suppose we run Graphplan on both problems, generating planning graphs $G$ and $G'$. Is $G \subseteq G'$?

**6.5** Prove Proposition 6.6 about the result of a set of independent actions.

**6.6** Prove Proposition 6.10 about the necessary condition for reaching a goal in a planning graph.

**6.7** Show that the definition of $P_i$ in the Expand procedure can be modified to be

$$P_i \leftarrow [P_{i-1} - \bigcap\{\text{effects}^-(a) \mid a \in A_i\}] \bigcup \{\text{effects}^+(a) \mid a \in A_i)\}.$$

Discuss how this relates to the usual formula, $\gamma(a, s) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$.

**6.8** Specify the Graphplan algorithm, including the procedures Expand, Extract and GP-Search, without the no-op actions. Discuss if this leads to a benefit in the presentation and/or in the implementation of the algorithm.

**6.9** Suppose we want to modify Graphplan so that it can use the following operators to increment and decrement a register $r$ that contains some amount $v$:

$\mathsf{add1}(r, v)$
  precond: $\mathsf{contains}(r, v)$
  effects: $\neg\mathsf{contains}(r, v)$, $\mathsf{contains}(r, v + 1)$
$\mathsf{sub1}(r, v)$
  precond: $\mathsf{contains}(r, v)$
  effects: $\neg\mathsf{contains}(r, v)$, $\mathsf{contains}(r, v - 1)$

We could modify Graphplan to instantiate these operators by having it instantiate $v$ and then compute the appropriate value for $v + 1$ or $v - 1$.

(a) What modifications will we need to make to Graphplan's graph-expansion subroutine, if any?

(b) Suppose we have the following initial state and goal:

$$s_0 = \{\mathsf{contains(r1,5)}, \mathsf{contains(r2,8)}\}$$
$$g = \{\mathsf{contains(r1,8)}, \mathsf{contains(r2,7)}\}$$

How many operator instances will we have at level 1 of the planning graph?

(c) What atoms will we have at level 2 of the planning graph?

(d) At what level of the planning graph will we start calling the solution-extraction subroutine?

(e) What modifications will we need to make to Graphplan's solution-extraction subroutine, if any?

(f) Why wouldn't it work to have the following operator to add an integer amount $w$ to a register $r$?

$\mathsf{addto}(r, v, w)$
  precond: $\mathsf{contains}(r, v)$
  effects: $\neg\mathsf{contains}(r, v)$, $\mathsf{contains}(v, v + w)$

**6.10** Apply the Graphplan algorithm to a modified version of the problem in Example 6.1 in which there is only one robot. Explain why the problem with two robots is simpler for Graphplan than the problem with just one robot.

**6.11** Test the full DWR domain with several locations, a few robots and a handful a containers on one the public-domain implementation of Graphplan (e.g. IPP or SGP). Discuss the practical range of applicability of this planner for the DWR domain.

**6.12** Detail the modifications required for handling with Graphplan operators with disjunctive preconditions, in the modification of mutex and in the planning procedures.

**6.13** Apply Graphplan with the allowance relation to the same planning problem. Compare the two planning graphs and the obtained solutions.

**6.14** In Propositions 6.4 and 6.6, replace the expression "any permutation" with "any allowed permutation" and prove these new propositions.

**6.15** Discuss the structure of plans as output by Graphplan with the allowance relation. Compare these plans to sequences of independent sets of actions, to plans that are simple sequences of actions, and to partially ordered sets of actions.