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 actions; any sequence that meets the constraints of the partial order is a valid plan. Planning-graph approaches take a middle ground. Their output is a sequence of sets 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 sets 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 sets is less general than a partial order. It can be expressed immediately as a partial order, but the converse is false, e.g., a plan with three 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 sets unless an additional constraint is added.

We have seen that the main idea behind plan-space planning is the *least commitment principle*: i.e., to refine a partial plan, one flaw at a time, by adding only the ordering and binding constraints needed to solve that 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: *reachability analysis* and *disjunctive refinement.*

Reachability analysis addresses the issue of whether a state is reachable from some given state s_0. Disjunctive refinement consists of addressing one or several flaws

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.

through a disjunction of resolvers. Because 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 and the planning-graph structure (Section 6.2) before getting into planning-graph algorithms (Section 6.3).

6.2 Planning Graphs

The planning-graph structure provides an efficient way to estimate which set of propositions[2] is possibly reachable from s_0 with which actions. We first discuss here the general notion of state reachability, which cannot be computed in a tractable way. We then introduce the relaxed reachability estimate provided by a planning graph. Then we detail the issues of independence between actions and mutual exclusion for actions and propositions in a planning graph.

6.2.1 Reachability Trees

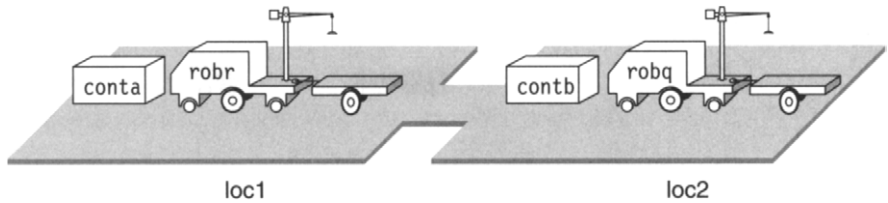
Given a set A of actions, a state s is *reachable* from some initial state s_0 if there is a sequence of actions in A that defines a path from s_0 to s. Reachability analysis consists of analyzing which states can be reached from s_0 in some number of steps and how to reach them. Reachability can be computed *exactly* through a *reachability tree* that gives $\hat{\Gamma}(s_0)$, or it can be *approximated* through a planning graph developed as shown 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 20 actions corresponding to the instances of the three operators in Figure 6.1.

To simplify the forthcoming figures, let us denote ground atoms by propositional symbols.

---

2. In the context of a first-order language classical planning, we will use the terms *ground atom* and *proposition* interchangeably.



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'), ¬ 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), ¬ in(c, l), ¬ 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), ¬ loaded(r, c)

Figure 6.1 A simplified DWR problem.

- r_1 and r_2 stand for at(robr,loc1) and at(robr,loc2), respectively.
- q_1 and q_2 stand for at(robq,loc1) and at(robq,loc2), respectively.
- a_1, a_2, a_r, and a_q stand for container conta in location loc1, in location loc2, loaded on robr, and loaded on robq, respectively.
- 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), respectively.

Let us also denote the 20 actions in A as follows.

- Mr12 is the action move(robr,loc1,loc2), Mr21 is the opposite move, and Mq12 and Mq21 are the similar 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.
- Uar1, Uar2, Uaq1, Uaq2, Ubr1, Ubr2, Ubq1, and Ubq2 are the unload actions.

The reachability tree for this domain, partially developed down to level 2 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 Σ and whose edges corresponds to actions of Σ. The root of T is the state s_0. The children of a node s are

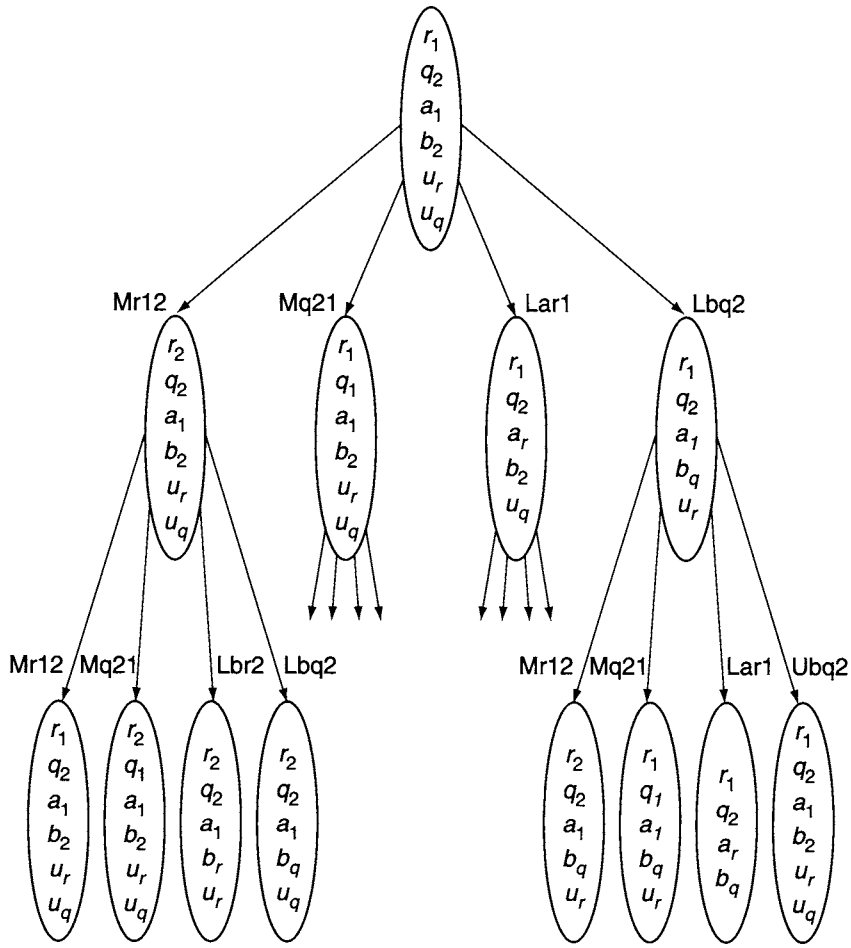


Figure 6.2 Reachability tree.

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 iff 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 Example 6.1 (omitting for clarity most of the back arcs from a node to its parents). However, even this reachability graph would be a very large, impractical size, as large as the number of states in the domain.

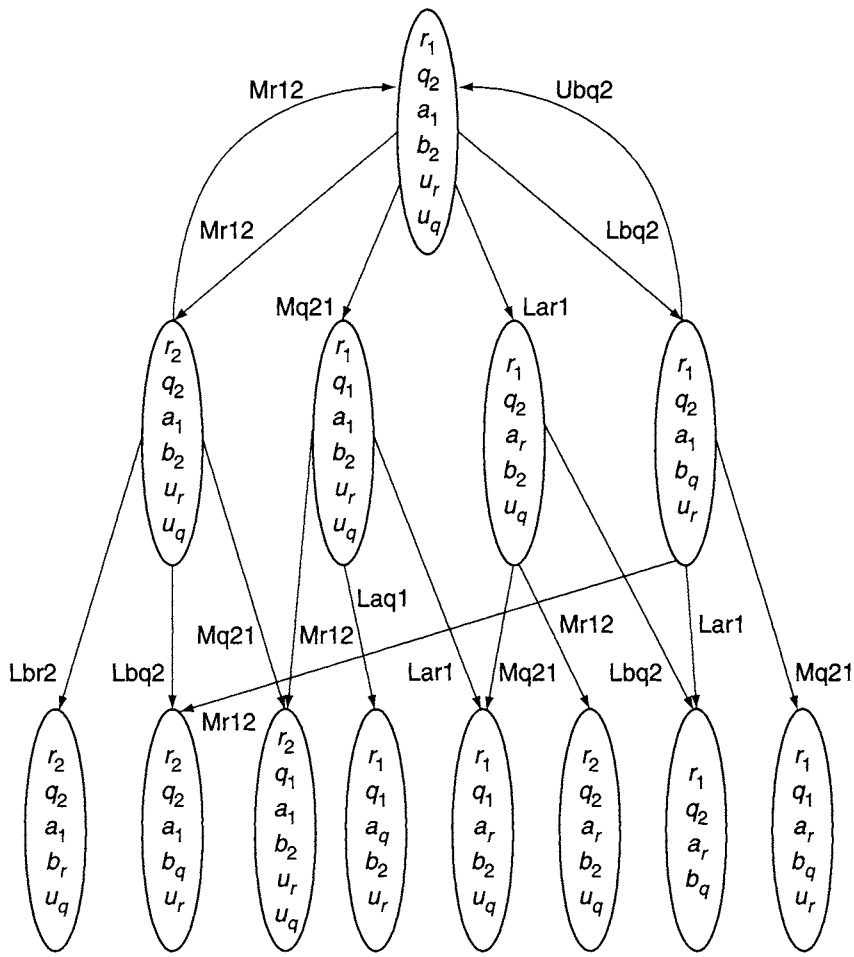


Figure 6.3 Reachability graph.

6.2.2 Reachability with Planning Graphs

A major contribution of the Graphplan planner is a relaxation of the reachability analysis. The approach provides an incomplete condition of reachability through 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, the planning graph considers an *inclusive disjunction* of actions from one node to the next that 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 Example 6.1 (see page 114) 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 is to keep track of incompatible propositions for each set of propositions and incompatible actions for each disjunction of actions. Let us explain informally how this is performed.

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: 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 *negative*. Note that negative effects are not deleted from P_1, thus $P_0 \subseteq P_1$.[3] This process is pursued from one level to the next. This is illustrated in Figure 6.4 for Example 6.1 down to level 3. (Dashed lines correspond to negative effects, and 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 no longer a sequence of actions corresponding directly to a path in Σ, as defined in Chapter 2. Here, a plan Π 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 π_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 that is a subset of P_1. From this state, actions in $\pi_2 \subseteq A_2$ would proceed and so on until a level π_k, whose actions lead to a state meeting the goal. Let us define these notions more precisely.

---

3. 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 $P_0 \subseteq P_1$.

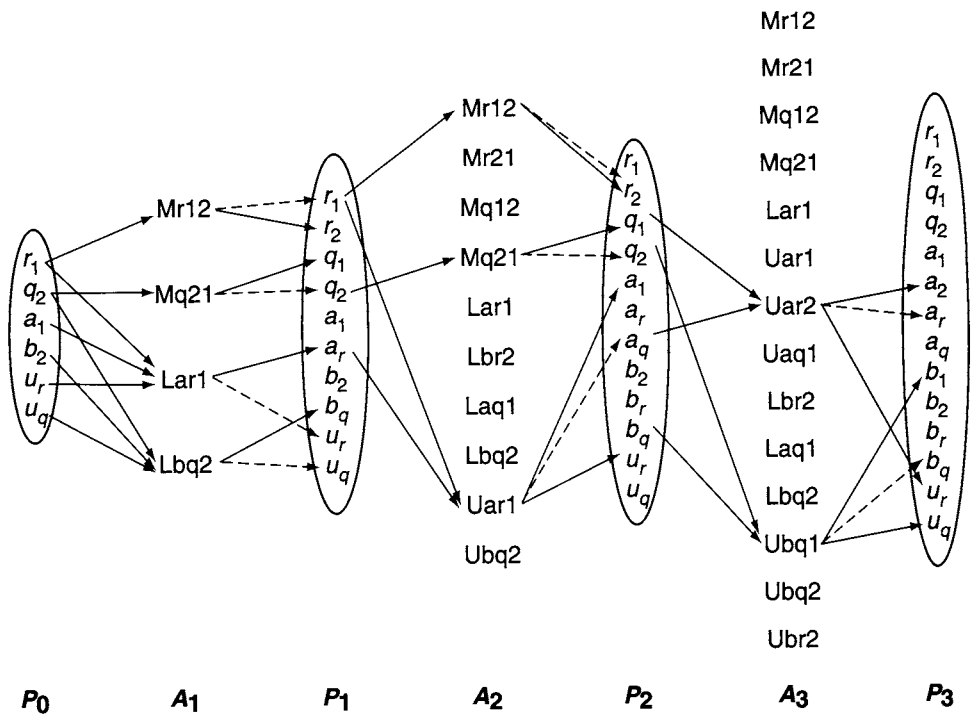


Figure 6.4 Planning graph.

6.2.3 Independent Actions and Layered Plans

In Example 6.1 (see page 114), the two actions Mr12 (i.e., move(robr, 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 ⟨Mr12, Mq21⟩ and ⟨Mq21, Mr12⟩ 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 robr is in loc2, hence Lar1 is not applicable.

Definition 6.1 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 π is independent when every pair of π is independent. ■

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

- a deletes a precondition of b: the ordering $a \prec b$ will not be permitted; or

- a deletes a positive effect of b: the resulting state will depend on their order; or
- symmetrically for negative effects of b with respect to a: b deletes a precondition on a positive effect of 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 beforehand for all problems of that domain.

Definition 6.2 A set π of independent actions is applicable to a state s iff $\text{precond}(\pi) \subseteq s$. The *result* of applying the set π 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 6.1 *If a set π of independent actions is applicable to s then, for any permutation $\langle a_1, \ldots, a_k\rangle$ of the elements of π, the sequence $\langle a_1, \ldots, a_k\rangle$ is applicable to s, and the state resulting from the application of π 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 Exercise 6.6) 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 6.3 A *layered plan* is a sequence of sets of 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 π_1 is applicable to s_0, π_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 6.2 *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 π_1, followed by any permutation of π_2 ..., followed by any permutation of π_n is a path from s_0 to a goal state.*

This proposition follows directly from Proposition 6.1.

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 π_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 π_1, and 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 pair (r_1, r_2), (q_1, q_2), (a_r, u_r), (b_q, u_q) in level P_1 of Figure 6.4. In order to deal uniformly with this second type of incompatibility between propositions, it is convenient to introduce for each proposition p a neutral *no-op* action, noted α_p, whose precondition and sole effect is p.[4] If an action a has p as a negative effect, then according to our definition, a and α_p are dependent actions; their positive effects are incompatible.

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 Example 6.1, (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 actions may be used in level π_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 shown in Definition 6.4.

Definition 6.4 Two actions a and b in 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 nonmutex in some latter level A_j of a planning graph.

Example 6.2 Mutex relations for Example 6.1 (see page 114) 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. ■

4. 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.

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	$\{a_2\} \times \{a_r, a_1, r_1, a_q, b_r\}$
	$\{b_1\} \times \{b_q, b_2, q_2, a_q, b_r\}$
	$\{a_r\} \times \{u_r, a_1, a_q, b_r\}$
	$\{b_q\} \times \{u_q, b_2, a_q, b_r\}$
	$\{a_q\} \times \{a_1, u_q\}$
	$\{b_r\} \times \{b_2, u_r\}$
	$\{r_1\} \times \{r_2\}$
	$\{q_1\} \times \{q_2\}$

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

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

Proposition 6.3 *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 $(a, b) \notin \mu A_i$.*

Proof Every proposition p in a level P_i is supported by at least its no-op action α_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 nonmutex pair of propositions remains nonmutex 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 6.4 *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 μP_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 pairs of them are mutex because it does not make sense to start searching a graph that does not meet the necessary condition of Proposition 6.4.

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 α_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 to generating the sets $A_i, P_i, \mu A_i$, and μP_i, respectively, from the elements in the previous level $i-1$.

Let us analyze some properties of a planning graph.

Proposition 6.5 *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), $|\mu A_i| \leq (m+n)^2$, 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 of the problem (O, s_0, g). This is the case because, according to classical planning assumptions, operators cannot

```
Expand(⟨P0, A1, μA1, P1, μP1, ..., Ai−1, μAi−1, Pi−1, μPi−1⟩)
    Ai ← {a ∈ A | precond(a) ⊆ Pi−1 and precond²(a) ∩ μPi−1 = ∅}
    Pi ← {p | ∃a ∈ Ai : p ∈ effects⁺(a)}
    μAi ← {(a, b) ∈ Ai², a ≠ b | effects⁻(a) ∩ [precond(b) ∪ effects⁺(b)] ≠ ∅
              or effects⁻(b) ∩ [precond(a) ∪ effects⁺(a)] ≠ ∅
              or ∃(p, q) ∈ μPi−1 : p ∈ precond(a), q ∈ precond(b)}
    μPi ← {(p, q) ∈ Pi², p ≠ q | ∀a, b ∈ Ai, a ≠ b :
              p ∈ effects⁺(a), q ∈ effects⁺(b) ⇒ (a, b) ∈ μAi}
    for each a ∈ Ai do: link a with precondition arcs to precond(a) in Pi−1
        positive arcs to effects⁺(a) and negative arcs to effects⁻(a) in Pi
    return(⟨P0, A1, μA1, ..., Pi−1, μPi−1, Ai, μAi, Pi, μPi⟩)
end
```

Figure 6.5 Expansion of a planning graph.

create new constant symbols.[5] Hence, if c is the number of constant symbols given in the problem, $e = \max_{o \in O}\{|\text{effects}^+(o)|\}$, and α 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 next.

Definition 6.5 A *fixed-point level* in a planning graph G is a level κ such that for $\forall i, i > \kappa$, level i of G is identical to level κ, i.e., $P_i = P_\kappa$, $\mu P_i = \mu P_\kappa$, $A_i = A_\kappa$, and $\mu A_i = \mu A_\kappa$. ■

Proposition 6.6 *Every planning graph G has a fixed-point level κ, 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 (1) there is a finite number of propositions in a planning problem, (2) $\forall i$, $P_{i-1} \subseteq P_i$, and (3) 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 as many propositions and, in that case, it has an equal number or fewer mutex pairs. Because these monotonic differences are bounded, at some point $(P_{i-1} = P_i)$ and $(\mu P_{i-1} = \mu P_i)$. 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.

- Because $(|P_{k-1}| = |P_k|)$ and $\forall i$, $P_{i-1} \subseteq P_i$, it follows that $(P_{k-1} = P_k)$.
- Because $(P_{k-1} = P_k)$ and $(|\mu P_{k-1}| = |\mu P_k|)$, then $(\mu P_{k-1} = \mu P_k)$. This is the case because a nonmutex pair of propositions at $k-1$ remains nonmutex at level k (Proposition 6.3).
- A_{k+1} depends only on P_k and μP_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 (because $\mu P_{k-1} = \mu P_k$), 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 is mutex, i.e., $g \in P_i$ and $g^2 \cap \mu P_i = \emptyset$. The search procedure looks for a set $\pi_i \in A_i$ of nonmutex actions

5. This is due to assumption A0 about a finite Σ (see Chapter 1).

that achieve these propositions. Preconditions of elements of π_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.

Example 6.3 The goal $g = \{a_2, b_1\}$ of Example 6.1 (see page 114) 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 nonmutex, hence $\pi_3 = \{\text{Uar2, Ubq1}\}$.

At level 2, the preconditions of the actions in π_3 become the new goal: $\{r_2, a_r, q_1, b_q\}$. r_2 is achieved by α_{r_2} or by Mr12 in A_2; a_r by α_{a_r} or by Lar1. Out of the four combinations of these actions, three are mutex pairs: (Mr21,Lar1), (α_{r_2},Lar1), and (α_{r_2},α_{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 two couples of actions achieving q_1 and b_q: (Mq21,Lbq2), (α_{q_1},Lbq2), and (α_{q_1},α_{b_q}); are mutex pairs. Hence the only possibility in A_2 for achieving this subgoal is the subset $\pi_2 = \{\text{Mr12}, \alpha_{a_r}, \text{Mq21}, \alpha_{b_q}\}$.

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

Level 0 is successfully reached.

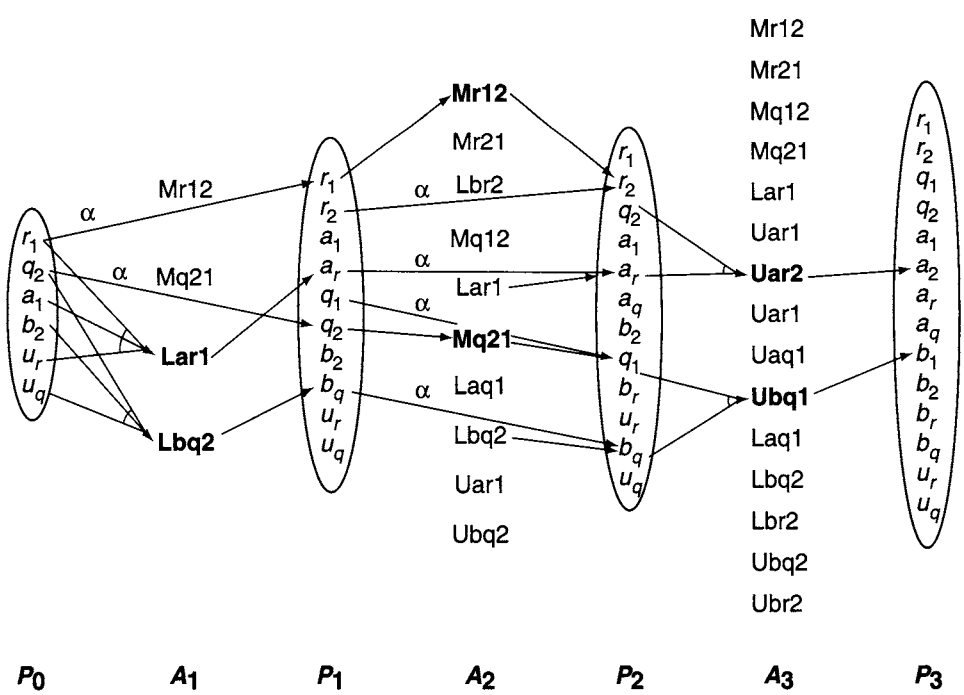


Figure 6.6 A solution plan.

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

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 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 Figure 6.6 as connected arcs).

The mutex relation between propositions provides 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 time in future searches. This recording is performed by procedure Extract (Figure 6.7) into a *nogood* hash-table denoted ∇. This hash table is indexed by the level of the failed goal because 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 (Figure 6.8). If it succeeds in reaching level 0, then it returns an empty sequence, from which pending recursions successfully return a solution plan. It records failed tuples into the ∇ 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 to establish g at this level from mutex and other nogoods found or established at the previous level.

```
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.

```
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.

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 actions already selected for that level, it nondeterministically chooses one action a that tentatively extends the current subset π_i through a recursive call at the same level. This is performed on a subset of goals minus p and minus all positive effect of a in g. As usual, a failure for this nondeterministic choice is a backtrack point over other alternatives for achieving p, if any, or a backtracking further up if all *resolvers* of p have been tried. When g is empty, then π_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.[6] Here CSP *variables* are goal propositions, and their *values* are possible actions achieving them. The procedure chooses a value for a variable compatible with previous choices (nonmutex) 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 (Figure 6.9) with the graph expansion, search steps, and termination condition. Graphplan performs an initial graph expansion until either it reaches a level containing all goal propositions without mutex or it arrives at a fixed-point level in G. If the latter happens first, then the goal is not achievable. Otherwise, a search for a solution is performed. If no

6. This view will be further detailed in Chapter 8.

```
Graphplan(A, s0, g)
    i ← 0,    ∇ ← ∅ ,    P0 ← s0
    G ← ⟨P0⟩
    until [g ⊆ Pi and g^2 ∩ μPi = ∅] or Fixedpoint(G) do
        i ← i + 1
        G ← Expand(G)
    if g ⊈ Pi or g^2 ∩ μPi ≠ ∅ 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.

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 level has been reached, until success or until the termination condition is satisfied. This termination condition requires that the number of nogood tuples in $\nabla(\kappa)$ at the fixed-point level κ, 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 κ to the fixed-point level of the planning graph when the fixed-point level 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 fewer steps must make at least one of the goal tuples in $\nabla_j(i)$ true at a level i, and that none of these tuples is achievable in i levels.

Proposition 6.7 *$\forall i, j$ 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 to achieve g with the actions in A_i and it fails: each set of actions in A_i that provides g is either mutex or 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 6.8 *The* Graphplan *algorithm is sound, and 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 Π 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 π_n achieves the set of problem goals, π_{n-1} achieves precond(π_n), etc. Finally, when GP-Search calls Extract on the recursion $i = 1$, we are sure that all precond(π_1) are in P_0. Hence the layered plan Π meets Definition 6.3 (see page 120) 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, i.e., either the fixed point κ is reached before attaining a level i that contains all goal propositions (no pair of which is 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 6.4 (see page 123), hence the problem is unsolvable.

In the latter case:

- $\nabla_{j-1}(\kappa) = \nabla_j(\kappa)$ because of Proposition 6.7.
- $\nabla_{j-1}(\kappa) = \nabla_j(\kappa + 1)$ because the last $i - \kappa$ levels are identical, i.e., $\nabla_{j-1}(\kappa)$ is to stage $j - 1$ what $\nabla_j(\kappa + 1)$ is to stage j.

These two equations entail $\nabla_j(\kappa) = \nabla_j(\kappa + 1)$: 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 6.7, 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.

1. The mutex relation on incompatible pairs of actions and propositions, and the weak reachability condition of Proposition 6.4 (see page 123), offer a very good insight about the interaction between the goals of a problem and about which goals are possibly achievable at some level.
2. 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 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 6.5 (see page 124)), 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

The planning-graph techniques can be extended and improved along different directions. Several extensions to the planning language will be presented. Various improvements to planning-graph algorithms will then be discussed. Finally, the principle of independent actions in a graph layer will be relaxed to a less demanding relation.

6.4.1 Extending the Language

The planning algorithm described in Section 6.3.3 takes as input a problem (O, s_0, g), 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. A more expressive language is often 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 also has to extend s_0 with respect to the newly introduced *not-p* predicate in order to maintain a consistent and *closed*[7] initial world.

Example 6.4 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), ¬ occupied(m)
effects: at(r, m), occupied(m), ¬ occupied(l), ¬ 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), ¬ occupied(l), ¬ at(r, l),
not-occupied(l), ¬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) and 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 (precond$_i$, effects$_i$). Hence it is easy to rewrite it as several operators, one for each such 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 directly handling an extended language.

Generalizing Graphplan for directly handling operators with disjunctive preconditions can be 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*, 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, now has to take into account not the actions but their AND-connectors (see Exercise 6.11).

Directly handling 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 6.1 (see page 120), i.e., that an independent set of actions defines the same state transitions for any permutation of the set. One also has 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,

7. That is, any proposition that is not explicitly stated is false.

over to its antecedent condition either in a positive or in a negative way. One also has to come up with ways to compute and propagate 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 on the planner's performance.

The monotonic properties of the planning graph are essential to this purpose. Because $P_{i-1} \subseteq P_i$ 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 6.3 (see page 123), 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 its fixed-point level κ. One just has to maintain the only changes that can appear after this level, i.e., in the nogood table of nonachievable tuples. Here also the monotonic property of Proposition 6.7 (see page 130), i.e., $\nabla_j(i) \subseteq \nabla_{j+1}(i)$, allows incremental management.

Finally, several general programming techniques can also be useful for memory management. For example, the bitvector data structure allows one 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, one for each of 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, e.g., the predicates adjacent, attached, and belong are rigid: there is no operator that changes their truth values. 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 one to infer nonobvious 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 relations, 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 $g' \in \nabla_i$ 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 ∇_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 of selecting first a proposition p that 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 actions 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 one to choose for achieving p the action that appears earliest in the graph.[8]

Finally, a number of algorithmic techniques allow one to improve the efficiency of the search. For example, one is 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 apply to the search in a planning graph, which is a particular CSP problem.[9]

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 π_i. In practice, we do not necessarily need to have *every* permutation of each set be a valid sequence of actions. We only need to ensure that there exists *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.

---

8. This topic will be further developed in Chapter 9, which is devoted to heuristics.
9. This point is developed in Section 8.6.2 of the CSP chapter.

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 6.4 (see page 121) we can say that two actions a and b are mutex either

- 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 relations 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.

Example 6.5 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 the 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 graph (i) with the independence relation, preconditions of c are mutex in P_1; because of the no-op α_q they become nonmutex in P_2. Action c appears in A_3 giving the goal in P_3. In graph (ii) with the allowance relation, q and r are nonmutex in P_1, and the goal is reached one level earlier. ■

The benefit of the allowance relation, i.e., fewer mutex pairs and a smaller fixed-point level, has a cost. Since the allowance relation is not symmetrical, a set of pairwise nonmutex 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 nonmutex (pending

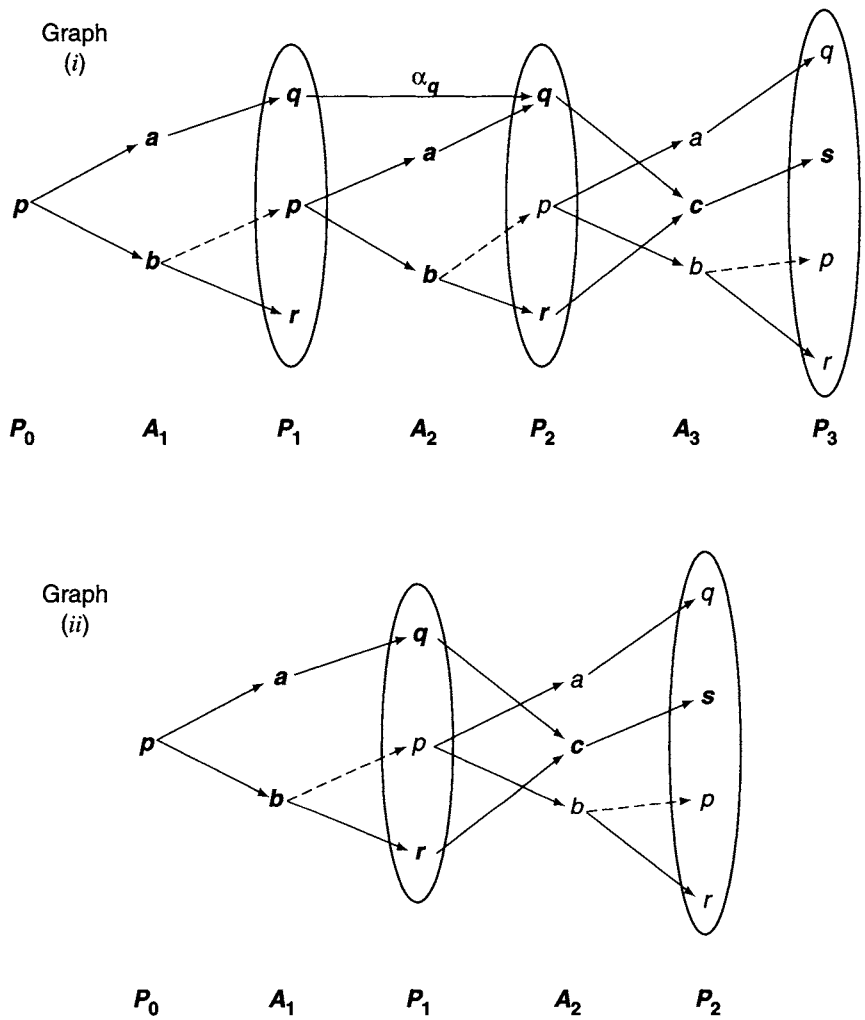


Figure 6.10 Planning graphs for independence (*i*) and allowance (*ii*) relations.

nonmutex 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 nonmutex 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 π_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 Section 6.3: $\gamma(s, \pi_i) = (s - \text{effects}^-(\pi_i)) \cup \text{effects}^+(\pi_i)$. All propositions of Section 6.2.3 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 needs 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 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, π_i, i)
    if g = ∅ then do
        if π_i is not allowed then return(failure)
        Π ← Extract(G, ⋃ {precond(a) | ∀a ∈ π_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 μA_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 [74, 75] 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. For several years, a significant fraction of the papers in every planning conference was concerned with planning-graph techniques. An extensive survey 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 Kambhampati *et al.* [305], who introduced three approximations of the reachability

tree (the "unioned plangraph," where every level is the union of 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). They also proposed a backward construction of the planning graph starting from the goal. This paper relied on previous work (e.g., [306]) to analyze Graphplan as a disjunctive refinement planner and to propose CSP techniques for improving it [158]. Some CSP techniques, such as forward-checking, were already in the initial Graphplan article [75]. But other contributions (e.g., [299, 300]) 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 into the restricted language of Graphplan a classical planning problem with some of the extensions of Section 6.4.1 is due to Gazen and Knoblock [213]. Several contributions for handling directly and efficiently extended constructs, like conditional effects, have also been developed [20, 332]. Some of the improvements to the encoding, memory management, and algorithms of the planning-graph techniques (see Section 6.4.2) appeared in other sources [192, 489]. A significant part of the work for extending the language or improving the algorithms took part along with the development of two Graphplan successors, the IPP [331] and STAN [367] planners. Several domain analysis techniques [218, 420, 565] for removing nonrelevant propositions and action instances and for detecting symmetries and invariance properties were proposed in order to focus the graph and the search. Domain analysis techniques have been extensively developed in a system called the Type Inference Module [192] and integrated to STAN [193].

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 because there is no semantics of concurrency in layered plans.[10] This is clearly illustrated in the extension from the independence to the allowance relations, i.e., the requirement to have all permutations of actions in a layer π_i equivalent, and the weaker requirement that there is at least one permutation that achieves the effects of π_i from its preconditions. This extension from independence to allowance is due to Cayrol *et al.* [112] for a planner called LCGP.

The work on LCGP led also to contributions on level-based heuristics for Graphplan [111]. Similar heuristics were independently proposed by Kambhampati and Nigenda [304]. More elaborated heuristics relying on local search techniques were proposed in the LPG planner [220] and led to significant performances, as illustrated in AIPS'02 planning competition results [195]. A related issue that arose at that time (and to which we will return in Chapter 9) is the use of Graphplan not as a planner but as a technique to derive heuristics for state-based planners [271, 423, 425].

The relationship between two techniques that were developed in parallel, the planning graph and the SAT-based techniques (see Chapter 7) have been analyzed by several authors (e.g., [42, 316]).

---

10. See Chapter 14, and particularly Section 14.2.5, which is devoted to concurrent actions with interfering effects.

Many other extensions to the planning-graph techniques have been studied, e.g., handling resources [330], dealing with uncertainty [73, 554], and managing a partial specification of the domain [488].

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 4.3)?

6.4 How much time (give big-O figures) do Expand and Extract procedures take? Give answers for both (a) the amount of time needed during a single iteration of Graphplan's while loop and (b) the cumulative time needed over all iterations of Graphplan's while loop.

6.5 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 $G \subseteq G'$?

6.6 Prove Proposition 6.1 (see page 120) about the result of a set of independent actions.

6.7 Prove Proposition 6.4 (see page 123) about the necessary condition for reaching a goal in a planning graph.

6.8 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 for the transition function, i.e., $\gamma(a, s) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$.

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

6.10 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:

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

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

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?

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

6.11 Detail the modifications required for handling operators with disjunctive preconditions in the modification of mutex relations and in the planning procedures.

6.12 Apply the Graphplan algorithm to a modified version of the problem in Example 6.1 (see page 114) 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.

6.13 Apply Graphplan with the allowance relation to the same planning problem mentioned in Exercise 6.12. Compare the two planning graphs and the obtained solutions.

6.14 In Propositions 6.1 and 6.2 (see page 120), 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.

6.16 Download one of the public-domain implementations of Graphplan, e.g., the original Graphplan,[11] IPP,[12] STAN,[13] or SGP.[14] Test the full DWR example on several domains and problems with an increasing number of locations, robots, and containers. Discuss the practical range of applicability of this planner for the DWR application.

11. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/Planning/Graphplan/
12. http://www.informatik.uni-freiburg.de/ koehler/ipp/IPPcode4.1.tar.gz
13. http://planning.cis.strath.ac.uk/STAN
14. ftp://ftp.cs.washington.edu/pub/ai/sgp.tgz