if its preconditions are true in the state. Negative preconditions cannot be handled[1]. The action may have both negative and positive effects. Negative effects delete propositions from the state, and positive effects add propositions.

Figure 10.1 shows the basic difference between the state space for a planning problem and the corresponding planning graph. State space search applies an action to a given state and generates a successor state, as shown on the left. For each action that is applicable, it generates a separate candidate state, which will be a starting point for further exploration. In the figure, we assume two actions a_1 and $a_2$that are applicable, with the preconditions linked by edges. The planning graph is a structure, as shown on the right, which *merges* the states produced by the different actions that are applicable. The resulting set of propositions forms a layer, as does the set of actions that resulted in this layer. The planning graph is a layered graph made up of such alternating sets of action layers and proposition layers. This is in contrast to the search tree of states that state space search generates.

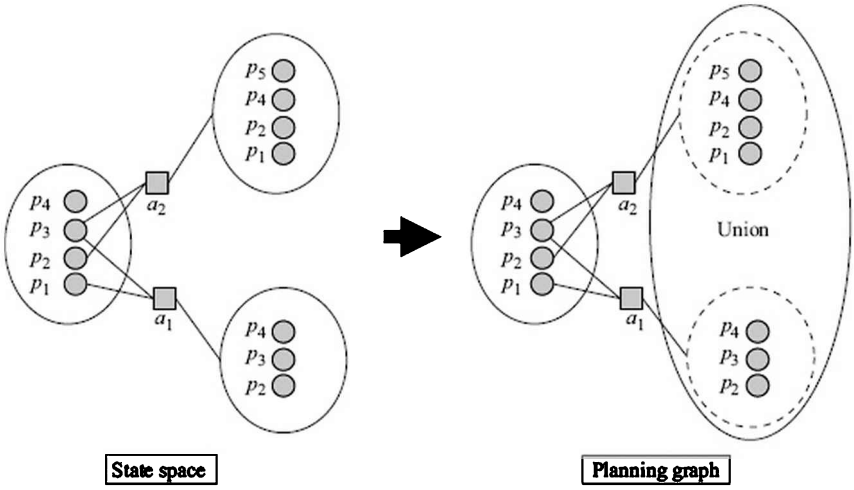

FIGURE 10.1 The planning graph merges the resultant states of all actions that are individually applicable in a given state.

How do we interpret the set of propositions in the proposition layer? Is the union of two (or more) states a state? And what is the semantics of the action layer? Does it mean that the actions in each layer can be executed together? Let us look at a concrete example from the blocks world domain (see Chapter 7). The figure below depicts the action layer and the fact layer for a state containing three blocks. One can see that there are propositions in the new layer that cannot hold together. For example, *On*(*A*, *B*) and *Hold*(*A*)[2] cannot be true at the same time, and therefore cannot be part of a state. Likewise, *On*(*A*, *B*) and *Clear*(*B*), *OnT*(*C*) and *Hold*(*C*) cannot be a part of a state as well. We can also see

that the two actions in the action layer, *PkUp*(*C*) and *UnSt*(*A*, *B*), cannot happen at the same time, given the blocks world assumption of a one armed robot.

Propositions in one layer that cannot be true simultaneously and actions in a layer that cannot happen simultaneously define a binary relation called *mutex* in each layer. The *mutex* relation, which stands for *MUTual EXclusion*, can be shown in the planning graph as edges connecting nodes in a layer. Another question one might ask is about the delete effects of actions. In the above example, the *PkUp*(*C*) has a delete effect *OnT*(*C*). Should the proposition be allowed in the next layer? The answer is *yes*, because if the first action in the plan were to be *UnSt*(*A*, *B*), the resulting layer would have *OnT*(*C*). This is the Frame Problem (McCarthy & Hayes, 1969) turning up again, and the way it is solved is by introducing a special action called *No-op*(*P*) for every proposition *P*. The *No-op*(*P*) action has preconditions {*P*} and effects {*P*}. It basically serves to preserve every proposition.

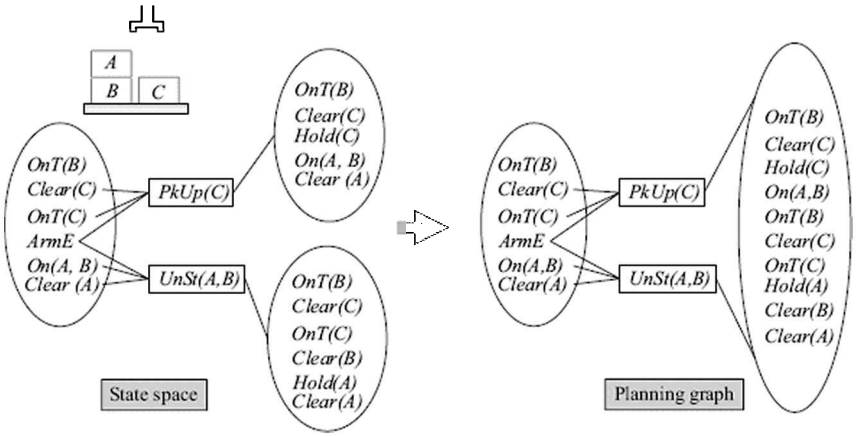

FIGURE 10.2 The proposition layer in the planning graph may contain inconsistent propositions, and the action layer may contain two actions that cannot be done in parallel.

To account for the relation between actions and their effects, we introduce edges between the action in a given layer and each of its effects in the next (proposition) layer. These edges are of two kinds, one for the add effects and the other for the delete effects. A proposition node may have both a positive edge and a negative edge coming in from different actions. Only one, though, can be part of a valid plan.

10.1.1 The Planning Graph

A planning graph, therefore, is a layered graph consisting of alternating layers of proposition and action nodes (P_0, A_1, P_1, A_2, P_2, A_3, P_3, …, A_n, P_n). For every proposition $p \in P_i$, there is a special action called *No-*

$op(p) \in A_{i+1}$ that carries forward p to P_{i+1}. There are four kinds of edges between nodes.

- A precondition edge $<p_i, a_{i+1}>$, linking the precondition p_i to an action a_{i+1}. These are depicted by solid lines in our figures.
- A positive edge $<a_i, p_i>$, linking an action a_i to an add effect p_i of a_i. These are also depicted by solid lines in our figures.
- A negative edge $<a_i, p_i>$, linking an action a_i to a delete effect p_i of a_i. These are depicted by solid lines with rounded heads in our figures.
- A mutex edge linking two propositions in the same layer, or two actions in the same layer. These are depicted by thick dotted lines in our figures.

The planning graph is made up of the following sets associated with each index i.

- The set of actions A_i in the i^{th} layer.
- The set of propositions P_i in the i^{th} layer.
- The set of positive effect links $PostP_i$ of actions in the i^{th} layer.
- The set of negative effect links $PostN_i$ of actions in the i^{th} layer.
- The set of preconditions links $PreP_i$ of actions in the i^{th} layer from P_{i-1}.
- The set of action *mutexes* in the i^{th} layer.
- The set of proposition *mutexes* in the i^{th} layer.

Figure 10.3 illustrates the structure of a planning graph. The initial state of the planning problem defines the layer P_0. Since P_0 represents a *given state*, there can be no *mutex* edges in this layer. In the graph shown below, there are four actions—a_1, a_2, a_3 and a_4. An action appears in a layer, only if all its preconditions are *nonmutex* in the previous proposition layer. The first two actions a_1 and a_2 appear in the first action layer A_1, and the other two appear in layers A_2 and A_3 one by one. For every proposition, there is a *No-op* action shown in dashed edges in the figure. Only some of the *mutex* edges have been shown to avoid cluttering up the figure.

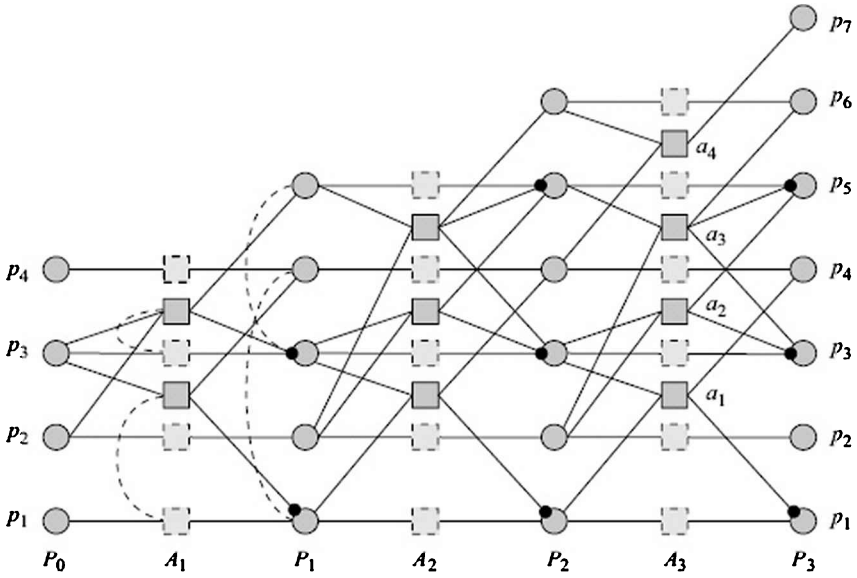

FIGURE 10.3 A planning graph is a layered graph with alternate layers of propositions and actions. The actions with dashed edges are *No-op* actions.

Very often we do not show the *No-op* actions explicitly in the planning graph figures. Instead, we draw edges directly from the proposition in one layer to the proposition in succeeding layers. These are depicted as thick, dashed, grey lines in the figure below. Figure 10.4 depicts a few layers of a planning graph for a simple blocks world problem. Observe that the planning graph does not depend upon the goals to be solved in a specific planning problem.

One of the first observations one can make is that the nodes in the graph grow monotonically with every new layer. This is because every proposition has an associated *No-op* action. Once a proposition appears in a layer, it will always occur in subsequent layers. It is a little less obvious, but when an action appears in an action layer, it occurs in every subsequent layer as well. This is because its preconditions that appeared nonmutex in a layer continue to show up *nonmutex* in subsequent layers. This will become clear when we define the mutex relation formally as follows.

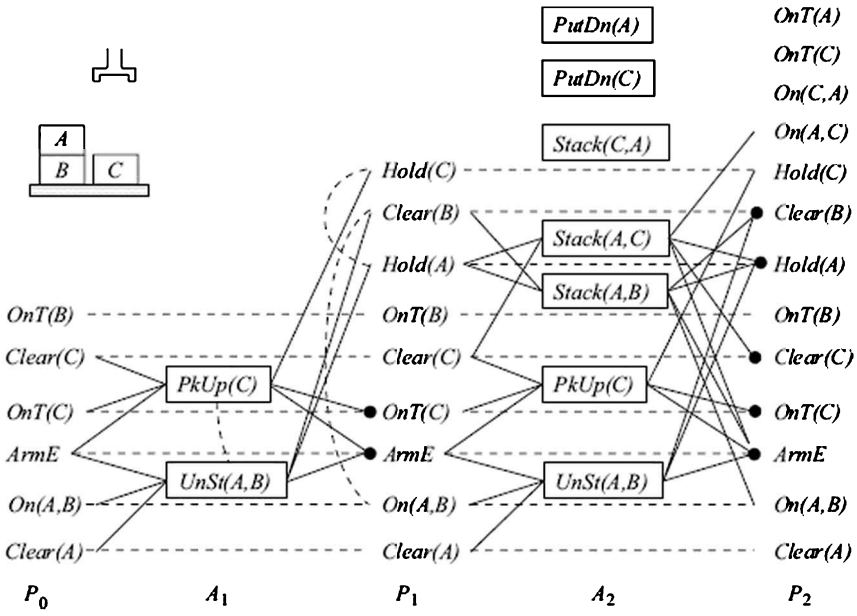

FIGURE 10.4 The planning graph for a simple blocks world problem. Only a few mutex edges are shown. Also, the edges linking some of the new actions in A_2 are not shown.

More interestingly, the proposition layers and action layers in a planning graph stabilize or *level off* after a while. Once the graph has levelled off, no new propositions or actions will appear. However, the number of *mutexes* may still change. In the starting layer, there are no mutexes. But as the graph is built, mutexes appear. However, these mutexes could disappear in future layers. Once they disappear, they can never appear again. Consider the blocks world problem in which two blocks, say *A* and *B*, that have to be unstacked from two blocks, say *C* and *D* respectively, and are to be put on the table. In the first action layer, there will be two actions, *UnSt*(*A*, *C*) and *UnSt*(*B*, *D*), with effects *Hold*(*A*) and *Hold*(*B*) in P_1. These actions and their effects will naturally be mutex since only one block can be picked up by the one armed robot. In the next level A_2, there are two actions *PtDn*(*A*) and *PtDn*(*B*), resulting in propositions *OnT*(*A*) and *OnT*(*B*) in layer P_2. Again, it is easy to see that these propositions are mutex in P_2. However, with two more layers of actions and propositions, the two propositions *OnT*(*A*) and *OnT*(*B*) are *not mutex* in the layer P_4.

10.1.2 Mutex Relations

The plans produced by *Graphplan* may have actions that occur in the same layer. This means that they could be done in parallel without affecting the outcome. Moreover, if one wants linear plans then these actions could be linearized in any order. The mutex relations identify a

pair of actions or propositions that cannot be present in the same layer in a plan.

Two actions $a \in A_i$ and $b \in A_i$ are mutex if one of the following holds,

1. Competing Needs There exist propositions $p_a \in$ *preconditions*(a) and $p_b \in$ *preconditions*(b) such that p_a and p_b are mutex (in the preceding layer).

2. Inconsistent Effects There exists a proposition p such that $p \in effects^+(a)$ and $p \in effects^-(b)$ or *vice versa*. The semantics of these actions in parallel are not defined. And if they are linearized then the outcome will depend upon the order.

3. Interference There exists a proposition p such that $p \in$ *preconditions*(a) and $p \in effects^-(b)$ or *vice versa*. Then only one linear ordering of the two actions would be feasible.

4. There exists a proposition p such that $p \in$ *preconditions*(a) and $p \in effects^-(a)$ *and also* $p \in$ *preconditions*(b) and $p \in effects^-(b)$. That is, the proposition is consumed by each action, and hence only one of them can be executed. This condition can be seen as a special case of the condition 3 in which neither linearization is allowed.

Two propositions $p \in P_i$ and $q \in P_i$ are mutex if *all* combinations of actions $<a, b>$ such that $a \in A_i$ and $b \in A_i$ and $p \in effects^+(a)$ and $q \in effects^+(b)$ are mutex. Observe that if two propositions are not mutex in a layer, they will be not mutex in all subsequent layers because of the *No-op* actions.

10.1.3 Building the Planning Graph

The process of building the planning graph continues till either the goal propositions have appeared nonmutex in the planning graph or the planning graph has levelled off.

1. The procedure to extend the planning graph by one level is described in Figure 10.5. The algorithm *ExtendGraph* takes as input all sets of data for each layer constructed so far, the index i of the layer to be constructed, and the set propositional actions A. The set A may be generated from the start state S and the set of operators O in a pre-processing phase. The algorithm begins by copying the sets for the $(i-1)^{th}$ layer, except the *mutexes*. This is because the set of propositions and actions are always carried forward. The algorithm then checks if any new actions are applicable (steps 8, 9). If there are, it adds elements to the different sets (steps 10–17). Then in steps 18 to 23, it computes all the mutex relations that hold at the i^{th} layer in the graph, and

returns the planning graph augmented by one layer in the last step. The applicability of actions in the planning graph needs to check that its preconditions are present *and* that none of the preconditions are mutex with each other. This is done by the procedure *ApplicablePG*(a_k, P_{i-1}, MuP_{i-1}) that takes the action being considered, the previous proposition layer, and the previous proposition mutex layer as input. Note that the test for mutex has both $< p_m, p_k > \in MuP_{i-1}$ or $< p_k, p_m > \in MuP_{i-1}$ for readability, though strictly speaking, one is enough.

The functions *MutexA* and *MutexP* are left as an exercise for the reader. The first task that algorithm *GraphPlan* takes up is the construction of the planning graph. The algorithm *PlanningGraph* described in Figure 10.6 starts off by initializing the layer zero of the planning graph. This contains only one non-empty set and that is P_0, which is initialized to the given start state *S*. The others are initialized to empty lists and the only purpose they serve is as input to the call to *ExtendGraph*. We assume that the procedure is called only if the initial state is not the goal state. The process of extending the graph (lines 11 and 12) continues till any one of the following two conditions is achieved.

1. The newest proposition layer contains all the goal propositions, and there is no mutex relation between any of the goal propositions. This is tested by the procedure *GoalPropExist* in the figure below. Like the *ApplicablePG* procedure, here also the test for mutex has both $< p_m, p_k > \in MuP_i$ or $< p_k, p_m > \in MuP_i$ for readability, though strictly speaking one is enough.
2. The planning graph has levelled off. This means that for two consecutive levels,

   $(P_{i-1} = P_i$ and $MuP_{i-1} = MuP_i)$

   If two consecutive levels have the same set of propositions with the same set of mutex relations between them, it means that no new action can make an appearance. Hence, if the goal propositions are not present in a levelled planning graph, they can never appear, and the problem has no solution.

```
ExtendGraph (i, A, < A_0, MuA_0, PreE_0, PostP_0, PostN_0, P_0, MuP_0, …,
          A_i-1, MuA_i-1, PreE_i-1, PostP_i-1, PostN_i-1, P_i-1, MuP_i-1>)
1  A_i ← A_i-1
2  P_i ← P_i-1
3  MuA_i ← ( )
4  MuP_i ← ( )
5  PreE_i ← PreE_i-1
6  PostP_i ← PostP_i-1
7  PostN_i ← PostN_i-1
8  for each a_k ∈ (A \ A_i)
9    if ApplicablePG(a_k, P_i-1, MuP_i-1)
10      then  A_i ← cons(a_k, A_i)
11            P_i ← append(effects+(a_k), P_i)
12            for each p∈precond(a_k)
13                PreE_i ← cons(<p, a_k>, PreE_i)
14            for each p∈effects+(a_k)
15                PostP_i ← cons(<a_k, p>, PostP_i)
16            for each p∈effects-(a_k)
17                PostN_i ← cons(<a_k, p>, PostN_i)
18 for each a_k ∈ A_i and a_m≠a_k ∈ A_i
19   if MutexA(a_m, a_k, MuP_i-1, PreE_i, PostP_i, PostN_i)
20        then MuA_i ← cons(<a_m, a_k>, MuA_i)
21 for each p_k∈ P_i and p_m≠p_k ∈ P_i
22   if MutexP(p_m, p_k, , PostP_i, MuA_i)
23        then  MuP_i ← cons(<p_m, p_k>, MuP_i)
24 return <A_0, MuA_0, PreE_0, PostP_0, PostN_0, P_0 MuP_0, …,
                A_i, MuA_i, PreE_i, PostP_i, PostN_i, P_i, MuP_i>

ApplicablePG(a_k, P_i-1, MuP_i-1)
1  Preconds ← precond(a_k)
2  if Preconds ⊆ P_i-1
3     then for each p_m ∈ Preconds
4             for each p_k ∈ Preconds
5                if < p_m, p_k> ∈ MuP_i-1 or < p_k, p_m> ∈ MuP_i-1
6                   then return false
7     else return false
8  return true
```

FIGURE 10.5 The procedure *ExtendGraph* creates the i^{th} action layer and the i^{th} proposition layer. It begins by copying the two preceding layers, and then adds any new applicable actions, and effects, and links to preconditions and effects. After adding all the actions and their effects, it computes afresh the mutex relation for the new action layer and the proposition layer.

The algorithm returns the planning graph along with the level *i* and the flag *levelled*. If *levelled* = *false* then the planning graph has been extended to the first layer in which the goal propositions exist without any mutex relations between them. The (calling) algorithm now needs to investigate whether a valid plan can be extracted from the planning graph. If *levelled* = *true* then the problem has no solution.

```
PlanningGraph (S, G, A)
1   P_0 ← S
2   A_0 ← ( )
3   MuA_0 ← ( )
4   MuP_0 ← ( )
5   PreE_0 ← ( )
6   PostP_0 ← ( )
7   PostN_0 ← ( )
8   i ← 0
9   leveled ← false
10  while not(GoalPropExist(G, P_i, MuP_i)) or leveled)
11      i ← i + 1
12      PlanGraph ← ExtendGraph (i, A,
                    <A_0, MuA_0, PreE_0, PostP_0, PostN_0, P_0, MuP_0, …,
                    A_i-1, MuA_i-1, PreE_i-1, PostP_i-1, PostN_i-1, P_i-1, MuP_i-1>)
13      if (P_i-1 = P_i and MuP_i-1 = MuP_i) then leveled ← true
14  return <PlanGraph, i, leveled>

GoalPropExist(G, P_k, MuP_k))
1  if G ⊆ P_k
2     then for each p_m ∈ G
3             for each p_k ∈ G
4                if < p_m, p_k> ∈ MuP_k  or < p_k, p_m> ∈ MuP_k
5                   then return false
6     else return false
7  return true
```

FIGURE 10.6 The algorithm *PlanningGraph* takes as input the start state *S*, the goal propositions *G*, and the set of ground actions *A*. It begins by extending the graph to layer one. After that, it keeps calling *ExtendGraph* until one of the following conditions become true. One, the goal propositions have appeared nonmutex in the latest layer. Or two, the planning graph has levelled off. The empty sets at level zero have been created only to allow a uniform call to *ExtendGraph*.

Note that the two cases are distinct and there is no possibility of both the tests connected by *or* in step 10 becoming true simultaneously.

The algorithm *Graphplan* is described below. It begins by checking if *G* is a subset of *S*. In that case, a plan need not be found. Otherwise, *Graphplan* calls *Planning Graph*. If the procedure *PlanningGraph* returns a planning graph with all goal propositions nonmutex, it is possible, but not necessary, that a valid plan might exist in the graph. We assume a procedure *ExtractPlan*(*G*, *i*, *PlanGraph*) that either extracts a plan *p* if there exists one, or returns *"nix"*.

If the procedure *ExtractPlan* returns *"nix"* at layer *i*, there are two possibilities.

The first is that the shortest plan, which may have parallel actions, has more stages than the layers in the planning graph. To investigate this possibility, *Graphplan* extends the planning graph one step at a time, checking whether a plan exists at each stage. Observe that this is a kind of iterative deepening behaviour, and always results in the shortest plans being found. As the planning graph is iteratively extended, it levels off at some stage. Let us call this level *n*. That is $P_{n-1} = P_n$ and $MuP_{n-1} = MuP_n$.

The second possibility is that a plan does not exist at all, even though

the goal propositions occur without mutex relations amongst them. The question then is: *when should the algorithm stop extending the graph and terminate with failure to find a plan?* Observe that a layer in the planning graph supplies the preconditions for the succeeding layer. Conversely, the layer can be seen as containing the subgoals that need to be solved for the goals in the succeeding layer, and layers beyond that as well. The basic idea in formulating the termination criteria is to identify a layer in the planning graph in which the sets of subgoals are not solvable *and* the sets have stabilized as well. The most obvious[3] candidate for this layer is layer n, in which the planning graph has levelled off. If one knew that *all possible* subsets that are candidates for being the subgoal sets at that layer have been considered *and* found to be unsolvable then one can conclude that the planning problem has no solution.

Let us say that the planning graph has gone beyond the level n, and has reached level i at which it has called the *ExtractPlan* procedure. Let S_n^i be the set of sets of subgoal propositions that *ExtractPlan* tried at level n (and failed). Then, if there are two consecutive levels $(t-1)$ and t such that the size[4]of S_n^{t-1} is equal to the size of S_n^t then the algorithm can terminate with the output that no plan exists. Observe that the number of sets (at level n) associated with layer $(i+1)$ would always be greater than sets associated with layer i, because the latter is contained in the former. And the number of such sets that is possible is finite because each set is a subset of P_n. Hence, at some point the number of sets will stop growing. The reader is referred to (Blum and Furst, 1997) for a proof of correctness of this termination criterion.

In the algorithm in Figure 10.7, we have assumed a function *SizeSubgoalSets* that returns the number of different subgoal sets that are possible in the level n when *Graphplan* is attempting to find a plan of i stages. This procedure is dependent upon the *ExtractPlan* procedure. We will describe it with the backward search phase used by Blum and Furst a little later. The algorithm *Graphplan* can be seen to operate in three stages. In the first stage, the initial planning graph is built and if the goal propositions have appeared then it attempts to extract a plan (lines 2–7). If the *ExtractPlan* procedure returns "*nix*", the algorithm incrementally extends the graph looking for a solution till it levels off (lines 8–15). When it does level off, it marks that level in the variable n, and computes the size of the set of sets of subgoals (lines 12–15). At this stage, this value will be 1 because the subgoal set is the original goal set G here. Finally, in the third phase (lines 16–21), *Graphplan* continues extending the planning graph looking for a plan. This happens till the termination condition is reached in line 21 where it reports failure. If at any time in phases 2 and 3 *ExtractPlan* returns a plan, the algorithm skips both these phases and terminates with the plan.

The plan that algorithm *Graphplan* returns has the following structure,

$$\boldsymbol{\pi} = (\{a_{11}, \ldots, a_{1p}\}, \{a_{21}, \ldots, a_{2q}\}, \{a_{31}, \ldots, a_{3r}\}, \ldots, \{a_{m1}, \ldots, a_{ms}\})$$

That is, it contains m ordered sets of actions. The first set $\{a_{11}, \ldots, a_{1p}\}$ contains actions that can be executed in parallel in stage one, the second set $\{a_{21}, \ldots, a_{2q}\}$ in stage two, and so on. If one desired a linear plan then the set for each stage can be linearized in any order.

10.1.4 Extracting the Plan

When the planning graph reaches a level that contains all the goal propositions G in the newest layer with no mutex relations amongst them, it is time to check whether a plan can be extracted from the planning graph. At this stage, we have the goal propositions and the start propositions nonmutex. It remains to be checked whether there exists a plan $\boldsymbol{\pi} = (\{a_{11}, \ldots, a_{1p}\}, \{a_{21}, \ldots, a_{2q}\}, \{a_{31}, \ldots, a_{3r}\}, \ldots, \{a_{m1}, \ldots, a_{ms}\})$, such that the set of actions at every layer $\{a_{k1}, \ldots, a_{kz}\}$ are applicable and non mutex, and that the goal propositions have support from the last set of actions, including the *No-op* actions.

```
Graphplan (start = S, goal = G, actions = A)
1  if G ⊆ S then return ( )
2  PG ← PlanningGraph (S, G, A)
3  PlanGraph ← First(PG)
4  i ← Second(PG)
5  leveled ← Third(PG)
6  if leveled  =  true then return "no plan exists"
7  π ← ExtractPlan(G, i, PlanGraph)
8  while π  =  "nix" and not leveled
9      i ← i + 1
10     PlanGraph ← ExtendGraph (i, A, PlanGraph)
11     π ← ExtractPlan(G, i, PlanGraph)
12     if (P_{i-1} =  P_i and MuP_{i-1} =  MuP_i)
13                  then leveled ← true
14                       n ← i
15                       S_i ← SizeSubgoalSets(i, n, Plangraph)
16  while π  =  "nix"
17      i ← i + 1
18      PlanGraph ← ExtendGraph (i, A, PlanGraph)
19      π ← ExtractPlan(G, i, PlanGraph)
20      S_i ← SizeSubgoalSets(i, n, Plangraph)
21      if S_i  =  S_{i-1} then return "no plan exists"
22  return π
```

FIGURE 10.7 Algorithm *Graphplan* begins by calling procedure *PlanningGraph* to construct the initial planning graph. *PlanningGraph* returns a triple from the three components which are extracted using the (assumed) functions *First*, *Second* and *Third*. From here on, *Graphplan* calls *ExtractPlan* and extends the planning graph, till either the termination criterion is reached or a plan is found. The function *SizeSubgoalSets* can be computed by inspecting the memory of failed goal sets maintained by *ExtractPlan*.

One way of looking at the plan existence question is to ask whether the goal propositions have support from a nonmutex set of actions. If yes then the combined preconditions of these actions can be viewed as a

regressed subgoal set, which can be solved recursively. In fact, this is the approach taken by the algorithm *Graphplan*. It is possible that there is more than one combination of actions supporting the goal set, leading to different possible subgoal sets that the procedure must explore. Figure 10.8 shows two possible ways in which the given goal set $\{p_4, p_5, p_7\}$ can be regressed to a set of subgoals at the preceding layer.

Let G_k be the goal set at the k^{th} layer. Let G^1_{k-1}, G^2_{k-1}, G^3_{k-1}, ... be the different possible goal sets that G_k can regress to. Then the *ExtractPlan* procedure will have to investigate all possible subgoals sets via which a plan may exist. In the above figure, $G_k = \{p_4, p_5, p_7\}$ and we have two subgoal sets $G^1_{k-1} = \{p_1, p_2, p_4, p_6\}$ and $G^2_{k-1} = \{p_4, p_5, p_6\}$. The reader is encouraged to find more subgoal sets.

The algorithm *GraphPlan* does a depth-first search in which it regresses the goal sets to subgoal sets. The procedure is like the *Backward State Space Search* of Figure 7.8, except that only those relevant actions are considered that are in the planning graph. Another difference is that the *ExtractPlan* procedure used by *Graphplan* allows sets of nonmutex actions to be considered simultaneously in a layer. The space explored by the backward search *ExtractPlan* is depicted in Figure 10.9.

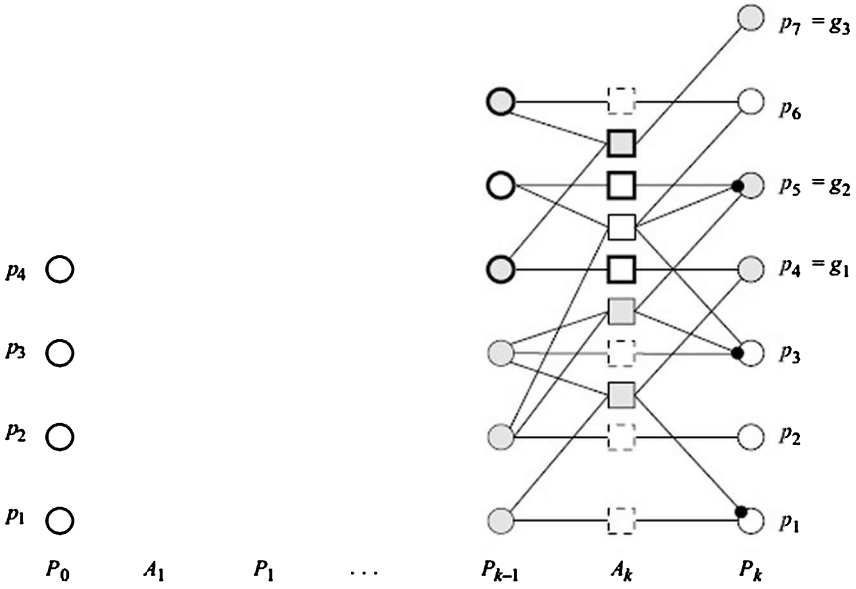

FIGURE 10.8 Given the goal propositions $\{p_4, p_5, p_7\}$ at level k, two possible subgoal sets at level (k − 1) are $\{p_1, p_2, p_3, p_4, p_6\}$ shown in shaded nodes, and $\{p_4, p_5, p_6\}$ shown with thick edges.

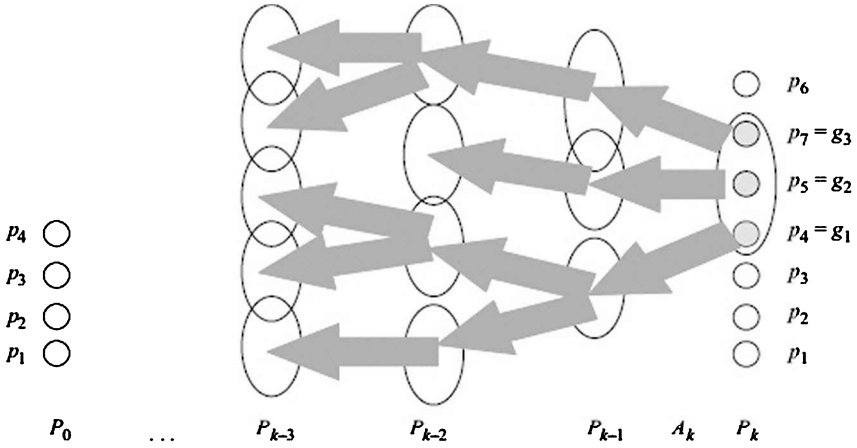

FIGURE 10.9 *Graphplan* does a depth first search, starting from the goal set on the right. A move in the backward search is a regression of the entire goal set to a subgoal set at the preceding layer.

The backward search call (at any recursive level) can terminate either by reaching the level P_0 with success or result in failure. If it is the former then a plan has been found and algorithm *Graphplan* can terminate. If it is the latter, the *ExtractPlan memoizes* the failed goal set. This is a kind of *nogood learning* that can be exploited in future (backward) searches. If a failed goal set is regressed to, in the future the algorithm can immediately backtrack from there without trying to solve it.

The algorithm *ExtractPlan* is described in Figure10.10. The procedure *RegressGoalSet*, called in line 1, returns *SetOfSubGoalSets* the set of possible subgoal sets along with the actions that regress to them as illustrated in Figure 10.8. Each element of the returned set is a pair <$Actions_i$, $Subgoals_{i-1}$>, where $Actions_i$ is the set of nonmutex actions that regress to the nonmutex set of propositions $Subgoals_{i-1}$ in the previous layer. The procedure, which is left as an exercise for the reader, tries all combinations of *relevant* actions (see Chapter 7) that can be chosen in the goal-set layer, such that they achieve the goal set in a nonmutex manner. This set of actions may include *No-op* actions as well. We also assume two functions, *Memoize* and *Memoized*, that manage a memoization memory called *mem*. This memory could be implemented in various ways, for example as list structures or as hash tables. *Memoize*(*goalSet*, *i*, *mem*) adds a set of goals *goalSet* that cannot be solved together at layer *i* to the memory, and *Memoized* checks the memory *mem* to find out whether a goal set is marked as failed at level *i*. Lines 6–16 implement a backtracking based search algorithm that tries all options in *setOfSubgoalSets* one by one. Line 11 checks for the base case in recursion, in which the goal-set layer is P_1. The actions, which have been tested to be nonmutex in the call to *RegressGoalSet* are returned as the plan. Note that these happen in parallel, or can be linearized in any order. Otherwise, in line 12, a recursive call is made to

ExtractPlan. If the recursive call finds a subplan, it is appended to the actions that generated the subgoal set and returned as a plan. If the subplan was *"nix"* then *while loop* looks at the next subgoal set. If all goal sets are exhausted, the (calling) goal set is memoized and *Extractplan* returns "nix" (steps 15, 16).

```
ExtractPlan (G, i, PlanGraph)
1  setOfSubgoalSets ← RegressGoalSet(G, i, PlanGraph)
2  setOfSubgoalSets  ← FilterMemoized(setOfSubgoalSets, i-1, mem)
3  if empty(setOfSubgoalSets)
4       then mem ← Memoize(G, i, mem)
5            return "nix"
6  while not empty(setOfSubgoalSets)
7       sGoal ← First(setOfSubgoalSets)
8       setOfSubgoalSets ← Rest(setOfSubgoalSets)
9       actions ← First(sGoal)
10      subGoals ← Second(sGoal)
11      if i = 1 then return actions
12      subPlan ← ExtractPlan(subGoals, i-1, PlanGraph)
13      if subPlan ≠ "nix"
14            then return append(subPlan, actions)
15  mem ← Memoize(G, i, mem)
16  return "nix"

FilterMemoized (setOfSubgoalSets, i, mem)
1  for each subGoalSet ∈ setOfSubgoalSets
2       subGoals ← Second(subgoalSet)
3       if Memoized(subGoals, i, mem)
4            then remove subGoalSet from setOfSubgoalSets
5  return setOfSubgoalSets
```

FIGURE 10.10 The procedure *ExtractPlan* does a backward depth first search from a given layer. We assume the function *RegressGoalSets* that regresses to the previous layer. If a call to *ExtractPlan* succeeds, it returns a plan. Otherwise it returns *"nix"* and also marks the goal set it was called with as 'failed' in that layer. We also assume functions *Memoize* which adds to the memory of failed goalsets *mem*, and *Memoized* that checks whether a goalset exists in it.

The algorithm *Graphplan* is guaranteed to find a plan that is shortest in the number of stages in the plan, where a stage consists of one or more *independent* actions.

It has been argued that the planning graph is polynomial in size, in terms of the number of objects, the number of operators with a fixed number of preconditions, the number of add effects and the layer number. And the planning graph can be constructed in polynomial time (Blum and Furst, 1997). Given that the planning problem is known to be hard (Gupta and Nau, 1992), we can infer that most of the work done by *Graphplan* is in the plan extraction phase. Two lines of exploration now present themselves. One is to try and make the plan extraction phase as efficient as possible. Some of the ideas from constraint solvers like forward checking can be applied here. In fact, in the following section, we will explore viewing the plan extraction task as a constraint satisfaction problem. The second is to find other ways to exploit the planning graph in other ways. We will also look at methods to use the planning graph to

generate heuristics to guide state space search.

10.1.5 STAN

One approach to make *Graphplan* more efficient was implemented in the program STAN (*st*ate *an*alysis) reported by Derek Long and Maria Fox (1999). The reader can observe that the first few lines of the *ExtendGraph* algorithm of Figure 10.5 are devoted to copying information from the previous layer. This incurs a cost both in time and space. Given that both propositions and actions are carried over to succeeding layers, STAN simply keeps one copy of each and keeps track of when the proposition and action was first introduced. Given an initial state and the set of planning operators, the overall set of (ground) actions and propositions is well defined. STAN represents the set of possible ground actions and propositions as bit vectors called the *action spike* and the *fact spike* respectively. The *rank* of an entry in the spike marks its first appearance. A *fact rank* is a consecutive sequence of fact headers of the same rank. Facts are represented using fact headers arranged according to rank, and the headers contain,

1. a name which is the predicate and arguments that comprise the fact itself,
2. an index *i*, giving the position of the fact in the fact array,
3. a bit mask, which is a fact spike vector in which the i^{th} bit is set and all other bits are unset,
4. a reference identifying its achieving *No-op*,
5. an action spike vector of consumers with bits set for all the actions which use this fact as a precondition, and
6. a fact-level package storing the layer dependent information about that fact.

An action header is likewise made up of,

1. the name of the action,
2. an index *i* giving the position of the action in the action array,
3. a bit mask which is an action spike vector in which the i^{th} bit is set and all other bits are unset,
4. a flag indicating whether the action is a *No-op*,
5. three fact spike vectors, called *precs*, *adds* and *dels*, and
6. an action level package storing the layer dependent information about that action.

For every action, the *precs* spike vector marks the preconditions of the action. The spike is a fact spike with the preconditions of the action set as 1 and the rest as 0. Thus, testing for applicability of the action is done by a bit-level logical operation. Similarly, the *adds* spike marks the add effects, and *dels* marks the delete effects of the action.

The above information associated with facts and actions is layer independent information. The rank associated with every fact or action

only indicates the first layer in which they appear. The layer-dependent information is stored in a fact-level package and an action-level package.

The fact-level package is an array of pairs, one for each rank in the spike. The pair is made up of,

1. a Fact Mutex Vector (FMV), *mvecs*, which is a fact-spike vector indicating which facts the given fact is mutex with, and
2. an Achievement Vector (AV) which is an action spike indicating which actions add that fact.

The action-level package are made up of,

1. an Action Mutex Vector (AMV), which is an action spike indicating which actions are mutex with the given actions,
2. a list (MA) of actions that are mutex with the given action, and
3. a bit vector *changedActs* to keep track of temporary mutex relations.

Mutexes can also be checked by bit-level logical operations. Actions mutexes may be permanent or temporary. Two actions are permanently mutex if their preconditions, add effects and delete effects interact. Two actions a and b are (permanently) mutex if $((precs(a) \vee adds(a)) \wedge dels(b)) \vee ((precs(b) \vee adds(b)) \wedge dels(a))$ is nonzero. Two actions may be temporarily mutex if some of their preconditions are mutex. Let $\{p_{a1}, \ldots, p_{ak}\}$ be the preconditions of action a. Then actions a and b are mutex if $(mvecs(p_{a1}) \vee \ldots \vee mvecs(p_{a1})) \wedge precs(a))$ is nonzero. Given that mutexes between actions can only disappear, STAN keeps track of actions becoming nonmutex in the *changedActs* vector to avoid repeated checks for temporary mutexes.

STAN first constructs the spike graph till the “opening layer” where all the goals are present nonmutex. This is analogous to the *PlanningGraph* algorithm of Figure 10.6. After that it alternates between testing for a plan and extending the graph till it has levelled off, or reached the fixed point. If a plan is still not found, STAN leaves the spike graph as it is from here on, since no changes can be made. It does further exploration using a *wavefront*. The idea is that one needs to consider all possible subgoal sets at the fixed point to be able to terminate with failure. STAN maintains a *queue* of goal sets to be considered at the fixed point layer, and a *buffer* in which new goal sets are considered. The new goal sets are the ones that would have been generated if the planning graph had been extended beyond the fixed point. When the graph is (notionally) extended, STAN adds new subgoal sets to the *queue*. When it cannot solve a (new) subgoal set in this *queue*, it propagates forward the goal set into the *buffer*, leading to new goal sets to be added to the *queue*, which it now recedes back to try solving them. This forward and backward movement from the fixed point is characterized by the name *wavefront*. For more details on STAN and on the proofs of soundness, completeness and termination of the wavefront mechanism, the reader is referred to (Long

and Fox, 1999).

10.1.6 Conditional Effects

Another direction of development is to extend the planning graph methods to richer domains. One of the first extensions reported was on ADL set. The Action Description Language (ADL) was a richer alternative to STRIPS domains and was later subsumed in PDDL2.1. The ADL language can be described as (Fox and Long, 2003),

```
adl = :strips + :typing
    + :negative-preconditions
    + :disjunctive-preconditions
    + :equality
    + :quantified-preconditions
    + :conditional-effects
```

Negative preconditions can be handled by introducing a separate proposition to represent a negated one. Consider the proposition *HaveBook*(*Wren&Martin*). Consider an action that says that if *HaveBook*(*Wren&Martin*) is true then one must buy the book. Instead of expressing the negated fact, one can introduce the new proposition *NotHaveBook*(*Wren&Martin*) and maintain it permanently mutex with *HaveBook*(*Wren&Martin*). Then instead of just deleting a proposition in an action, one can *also* add its negation. Observe that this approach requires us to abandon the negation by failure closed world assumption and takes us into an open world formulation. Consider the *Unstack*(*A*, *B*) action in the STRIPS domain, which has the proposition *On*(*A*, *B*) in the delete list. The closed world assumption says that if *On*(*A*, *B*) is not present in the state representation, it is false. That requires the algorithm to scan the representation, if testing for the negation was required. Having explicit negated formulas requires us to maintain one of the two, a proposition or its negation, explicitly. Having both would be inconsistent. Having a disjunction of both could represent uncertainty, as we will see below. One can also think of the add effect *Clear*(*B*) of the action *Unstack*(*A*, *B*) as a kind of expression of the fact ¬*On*(*A*, *B*). And given that only *A* was on *B*, one can in fact assert that nothing is on *B*. This is the way STRIPS handled the frame problem. Thus, *Clear*(*B*) ≡ ¬*xOn*(*x*, *B*) ≡ ∀*x* ¬ ∃*On*(*x*, *B*). When we have *Clear*(*B*) as a precondition for the action, say *Stack*(*C*, *B*), we are really testing the quantified precondition it is equivalent to, which also has a negation in it.

Conditional effects do extend the language beyond STRIPS (see (Gazen and Knoblock, 1997), (Koehler et al., 1997), (Anderson et al., 1998)). A conditional effect of an action is an effect that comes into being, only when a specified condition is true. Consider the action of driving a school bus from point *A* to point *B*, *Drive*(*bus*21, *a*, *b*). Also consider the

action *Board*(*Person*, *Bus*) with effect *In*(*Person*, *Bus*), and the predicate *At*(*Object*, *Location*). What should be the effects of the action *Drive*(*bus*21, *a*, *b*)? Conditional effects allow us to say that when the bus is driven from point *A* to point *B* then anyone who had boarded the bus will also be at point *B*. The *Drive* operator could be represented as (Koehler et al., 1997),

```
Action name:      DriveBus
Parameters:       Bus (type bus); A, B (type Location); X (type Object)
Preconditions:    At(Bus, A)
Effects:          Del(At(Bus, A)), Add(At(Bus, B)),
                  ∀x(In(x, Bus) ⊃ (Del(At(x, A)), Add(At(x, B)))
```

The above action could be represented in PDDL as,

```
(:action driveBus
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (at ?bus ?from)
      :effect        (and (not (at ?bus ?from)) (at ?bus ?to)
                          (forall (?x - object)
                                  (when (in ? x ?bus)
                                  (and (not (at ? x ?from)) (at ? x ?to)))))
```

It has been observed that an action with conditional effects can be viewed as a set of actions with different preconditions. To see this, let us first translate the above operator into a quantifier free one for a domain in which there are only two persons who could be in the school bus, Aditi and Jeena. The operator would then look like,

```
(:action driveBus
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (at ?bus ?from)
      :effect        (and  (not (at ?bus ?from)) (at ?bus ?to)
                           (when (in aditi ?bus)
                                 (and (not (at aditi ?from))
                                                      (at aditi ?to)))
                           (when (in jeena ?bus)
                                 (and (not (at jeena ?from))
                                                    (at jeena ?to)))))
```

Now this action with conditional effects can be translated into four STRIPS actions, called *aspects* in (Gazen and Knoblock, 1997). The four actions/aspects are,

```
(:action driveBus1
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (at ?bus ?from)
      :effect        (and (not (at ?bus ?from)) (at ?bus ?to)))
(:action driveBus2
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (and (at ?bus ?from) (in aditi ?bus))
      :effect        (and  (not (at ?bus ?from)) (at ?bus ?to)
                           (not (at aditi ?from)) (at aditi ?to))))
(:action driveBus3
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (and (at ?bus ?from) (in jeena ?bus))
      :effect        (and  (not (at ?bus ?from)) (at ?bus ?to)
                           (not (at jeena ?from)) (at jeena ?to))))
(:action driveBus4
      :parameters    (?bus -bus ?from ?to - location)
      :precondition  (and (at ?bus ?from) (in aditi ?bus) (in jeena ?bus))
      :effect        (and (not (at ?bus ?from)) (at ?bus ?to)
                           (not (at aditi ?from)) (at aditi ?to))
                           (not (at jeena ?from)) (at jeena ?to))))
```

However, converting actions into aspects can result in a blow-up in the number of actions the planner has to deal with, and an approach to handle them directly would be desirable. One such approach to extend *Graphplan* to conditional effects was implemented in a planner called IP^2 (Koehler et al., 1997).

The planning graph construction process for IP^2 in the way the new proposition layer is constructed. The operator o with conditional effects can be characterized as,

$prec_o$: the preconditions
add_o : the unconditional add effects
del_o : the unconditional delete effects
$prec_i$ … add_i, del_i : the i^{th} conditional effect

Like in *Graphplan*, an instance of the operator is added to the j^{th} layer A_j, if the preconditions $prec_o$ are present in the $(j - 1)^{th}$ layer P_{j-1} nonmutex amongst themselves. The unconditional add effects add_o are then added to the j^{th} proposition layer P_j, and if any proposition $p \in del_o$ is present in P_j then a delete link is added between the action and p. After that, for every conditional effect $(prec_i \supset add_i, del_i)$, a proposition $p \epsilon\ add_i$ is added to the j^{th} layer P if the following conditions hold.

1. $prec_i \subseteq P_{j-1}$
2. All propositions in $prec_i$ are nonmutex with each other in P_{j-1}
3. All propositions in $prec_i$ are nonmutex with all propositions in $prec_o$ in P_{j-1}

When all the goal propositions show up nonmutex in k^{th} proposition layer, *IP*^{2} algorithm embarks upon a plan finding search. Here too, it differs a little from *Graphplan*, in that as it searches backward it also maintains a set of negative goal sets C_j in layers $j<k$ preceding the last layer, along with subgoal sets G_j. These negative goal propositions may be added to prevent undesirable conditional effects. We illustrate their

utility with an example.

Consider the example of the two students, Aditi and Jeena, mentioned above. Let one day the goal be *{(at aditi home), (at jeena school)}*, perhaps because Aditi was not well. Given that (*at aditi home*) is mutex with (*at aditi school*), it is imperative that the proposition (*at aditi school*) does not appear in the final layer. Otherwise, the goal set will not be nonmutex. Now we need the action *busDrive*(*bus*21, *home*, *school*) because we need to achieve (*at jeena school*) which is a conditional effect in that action. For this conditional effect to be achieved, the condition (*in jeena bus*) must be true in the $(k - 1)^{st}$ layer. There is an action *board*(*jeena*, *bus*) that would have achieved (*in jeena bus*) when both the bus and Jeena were "at home". Thus, the plan would have two actions—Jeena boards the bus, and the bus is driven to the school. But how does one ensure that Aditi has not landed up in school as a conditional effect of the same *busDrive*(*bus*21, *home*, *school*) action as well? IP^2 can observe that the effect (*at aditi school*) interferes with the goal proposition (*at aditi home*) but it needs the *busDrive*(*bus*21, *home*, *school*). To ensure that Aditi does not land up in school, it adds a *negative goal* to the $(k–1)^{st}$ layer which says that (*in aditi bus*) is part of the negative goal set. The way such a negative goal is handled is that if there is any action that has it as an add effect then that action is not allowed in the backward search phase. Thus, while IP^2 does select the *busDrive*(*bus*21, *home*, *school*) action, it constructs a plan which does not have the *board*(*aditi*, *bus*) action.

The detailed algorithm is left as an (advanced) exercise for the reader. For hints and a statement on soundness *of* IP^2, the reader is referred to (Koehler et al., 1997)

10.1.7 Conformant Graphplan

Very often a planning agent has to deal with uncertain information. This uncertainty can be of different forms. The agent may not know the world completely, or the actions of an agent may not have well-defined effects. We humans face such problems all the time, and have evolved many approaches to address these problems. These include exploiting experience and knowledge based methods, and also probabilistic reasoning. We will explore some of these approaches in later chapters. Here, we look at the problem in the framework of automatic planning.

One approach to planning in the face of uncertain world knowledge is called *contingent planning* (Weld et al., 1998), (Majercik and Littman, 2003), (Albore et al., 2009). The idea here is to do more than find a simple plan. Instead, the aim is to synthesize a plan which can cater to more than one situation. The plans produced in contingent planning incorporate sensing steps that help decide between different courses of actions, all encapsulated in the plan. Bridge players are used to planning the play of their hands in a contingent fashion. For example, a bridge

player might play a few cards in a suit to test how the cards in that suit are divided amongst the opponents. Depending upon the outcome, she chooses one of competing courses of further action. Contingent planning happens in many real world situations. A group of friends planning an evening out may go and try for some movie tickets, and if not available may fall back to visiting a park.

Some well known puzzles are problems of contingent planning. For example, the problem of identifying the defective (heavier or lighter) ball from a set of twelve balls, using a pair of weighing scales. Observe that a move here has different possible effects. For example, you might keep balls 3 and 7 on the left pan, and balls 8 and 9 on the right pan. The balance could go to one of the three states—left heavier, right heavier, or balanced. In contingent planning, one can sense this state. Like most planning problems, one may demand a solution optimal in the number of comparisons one is allowed to make. An interesting problem that has both a contingent solution as well as a conformant one is described in the exercises.

Figure 10.11 illustrates common planning problem faced by residents of the IIT Madras campus. We consider two options available to people if they want to go the Central station to catch a train. They can walk to the MRTS station and take a train. Or they can look around for an autorickshaw and investigate whether the driver is willing[5] to go to the station, and if one is found hire it. There is another possible obstacle though, in the traffic, for the hiring-auto plan to succeed. A contingent planner may try for an auto before deciding to walk, but a conformant planner would have to commit to the MRTS plan because that succeeds in all *possible worlds*.

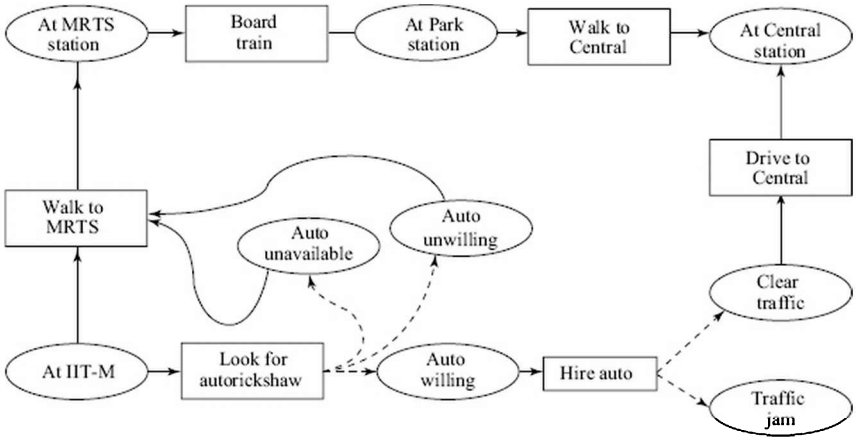

FIGURE 10.11 Reaching a train station. We assume two options are available. The dashed lines represent possible worlds a planner might encounter. A contingent planner could try both options, but a conformant planner would have to commit to a set of actions.

Conformant planning aims at building plans that can cater to different

situations but in which sensing of the state is not allowed. The uncertainty may be in knowing the facts (state) or in the effects of actions (resulting state). David Smith and Daniel Weld show how *Graphplan* can be extended to perform conformant planning in a system call *Conformant Graphplan* (*CGP*) (Smith and Weld, 1998). *CGP* represents uncertain knowledge by a disjunction of propositions, or using the exclusive-or. For example, a technician may not be sure whether an instrument has been calibrated or not before using it. Then a conformant plan could be to calibrate it anyway before using it. Smith and Weld approach such uncertainty by constructing planning graphs for each of the possible worlds in which a component of the disjunction (or XOR) might be true, and then finding solutions in each of them.

We illustrate how *CGP* tackles the bomb disposal problem, first posed by McDermott (McDermott, 1987) as described in (Smith and Weld, 1998). Consider a planning problem in which you have two packages and exactly one of them contains a bomb. This fact is represented by (*In*(*bomb*, *package*l) ⊕ *In*(*bomb*, *package*2)). The exclusive-or entails that there are two possible worlds, one w_1 in which the bomb is in $package_1$ and the other w_2 in which it is in $package_2$. Let there be exactly one toilet that you have access to and an action called *Dunk* defined as follows, which can diffuse the bomb.

```
(:action dunk
       :parameters     (?p - package ?t -toilet)
       :precondition
       :effect         (when (in bomb ?p) (diffused bomb)))
```

The action says that if you dunk a package in the toilet then if the package contains the bomb, the bomb will be diffused. This action has only one aspect with non-empty effects,

```
(:action dunk*
        :parameters    (?p - package ?t -toilet)
        :precondition  (in bomb ?p)
        :effect        (diffused bomb))))
```

The action *Dunk** will have two instances—*Dunk**(*p*1, *t*) and *Dunk**(*p*2, *t*). Each instance will be applicable in the corresponding possible world. *CGP* constructs a separate planning graph for each possible world, and looks for a solution when the goal set appears nonmutex in all the planning graphs. Figure 10.12 shows the two planning graphs constructed for the possible worlds w_1 and w_2. It then starts a plan finding exercise moving backwards, level by level, in parallel in each planning graph. If it can find a set of actions in each possible world then it returns the union of the plans found.

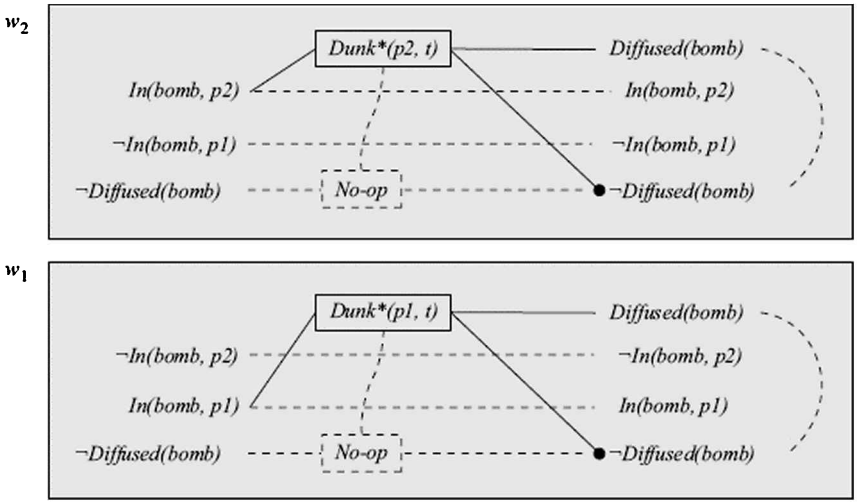

FIGURE 10.12 *Conformant Graphplan* constructs a planning graph for each possible world.

It finds that the goal *Diffused*(*bomb*) appears in both the planning graphs at level 1, and further that, the two actions achieve the goal. Thus, it returns a plan of dunking both packages.

The above example does not have interference between the actions across the different planning graphs. To illustrate that case, we augment our dunking action to have an additional effect that it clogs the toilet. The action, *Dunk*2, is given below.

```
(:action dunk2
       :parameters   (?p - package ?t -toilet)
       :precondition (not (clogged ?t))
       :effect       (and (clogged ?t)
                          (when (in bomb ?p) (diffused bomb))))
```

The modified dunk action requires that the toilet be unclogged to be applicable, and has an effect that it clogs the toilet. Now the two actions of dunking the two packages are individually applicable in the two planning graphs, which are constructed for the two possible worlds. The real world, on the other hand, is one of the two worlds, w_1 and w_2, but we do not know which. When we try and extract a plan keeping track of interference across worlds, we are unable to select the two dunk actions together, because they are now mutex as they delete the precondition ~*Clogged*(*t*) required by both, and removed by both. Consequently, we can only choose one of them.

However, both actions appear in both planning graphs, because irrespective of whether the package contains the bomb or not, it has the effect of clogging the toilet. The two aspects of the dunk action are as follows:

```
      (:action dunk2+
             :parameters   (?p - package ?t -toilet)
             :precondition (and (not (clogged ?t)) (in bomb ?p))
             :effect (and  (clogged ?t) (diffused bomb)))
and,
      (:action dunk2-
             :parameters   (?p - package ?t -toilet)
             :precondition (not (clogged ?t))
             :effect            (clogged ?t))
```

Further, since two possible worlds are in fact complete information variations of a single real world with incomplete information, the two aspects may induce each other in different planning graphs, if the preconditions hold. That is, if we choose the *Dunk*2+(*p*1, *t*) action in w_1, then we have to choose the *Dunk*2-(*p*1, *t*) action in w_2, because we can only choose the actions and not the possible world we are in. Once dunking p_1 is chosen as the action then its relevant aspects have to be chosen in both possible worlds. The situation is depicted in Figure 10.13 below.

In the planning graph for each possible world, we have two dunk actions, one for each package. The specific aspects of these actions that appear depend upon the possible world. But irrespective of which possible world we are in, the aspects of the two actions are mutex, as discussed above. Thus, in each possible world, we can only select one dunk action, and having chosen one, we are then compelled to choose the corresponding induced action in the other world.

We adopt the notation used by Smith and Weld in which *A*:*w* stands for action *A* in world *w*, and *P*:*w* stands for proposition *P* in world *w*. Let us say that we choose *Dunk*2+(*p1*, *t*):*w1*. This action results the goal proposition *Diffused*(*bomb*):*w1*. But it induces *Dunk2*-(*p1*, *t*):*w2*, which inhibits *Dunk2*+(*p2*, *t*):*w2* being mutex with it. The *No-op*:*w2 for* ~*Diffused*(*bomb*):*w2* results in the proposition ~*Diffused*(*bomb*):*w2*. The goal *Diffused*(*bomb*) appears only in one planning graph and the *CPG* will continue to extend the planning graphs till it reaches its termination criteria, which in this case is the two graphs levelling off.

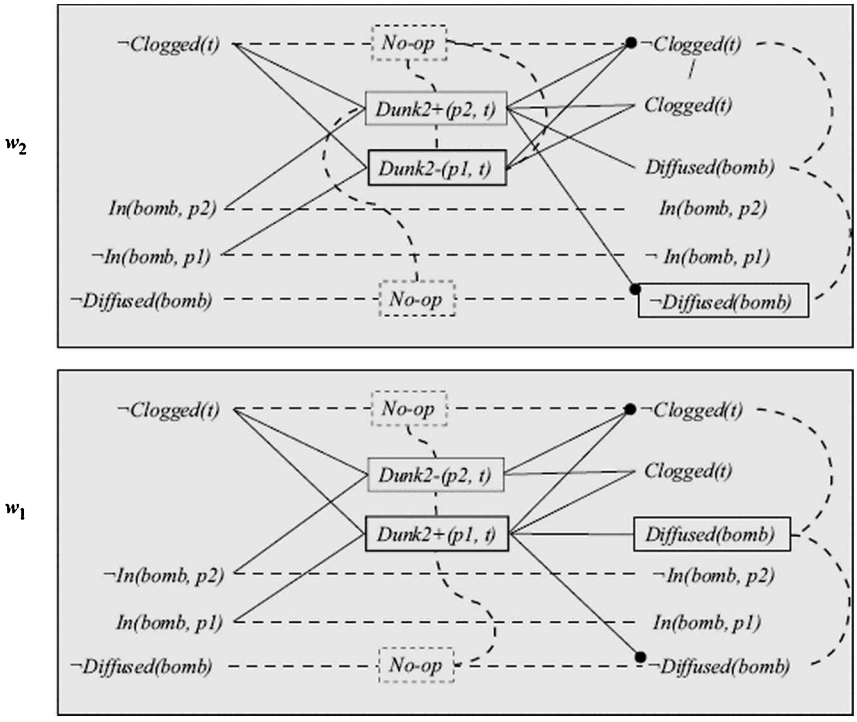

FIGURE 10.13 An action in one planning graph can induce another action in another planning graph. For example *Dunk2+*(*p1*, *t*) in *w*1 induces *Dunk2-*(*p1*, *t*) in *w*2, and vice versa, shown here with thick boxes. The corresponding state of the bomb in the two worlds is indicated by the boxed propositions.

In fact, the two actions *Dunk2+*(*p1*, *t*):*w1* and *Dunk2+*(*p2*, *t*):*w2* can be labelled mutex which will avoid the fruitless search that discovers that they cannot happen together. The notion of induced mutex is defined as follows.

Induced Mutex If an action *A*:*w* necessarily induces an aspect *A′*:*v*, and if *A′*:*v* is mutex with an action *B*:*u* then *A*:*w* is mutex with *B*:*u*. Note that the possible worlds *u* and *v* need not be distinct and the result also applies for *B*:*v*.

It there were two toilets, t_1 and t_2, then the *CPG* would have found the solution of dunking each package in a different toilet. The reader is encouraged to construct the corresponding planning graphs on paper, and verify that there are two distinct conformant plans for diffusing the bomb.

Given an action *A* that is desirable in some world, one can sometimes prevent an undesirable induced aspect *A′* in a different possible world. This is done by *confronting* the undesirable aspect by making some precondition of *A′* false. This is similar to the process in which only desirable aspects are included in IP^2 as described in the previous section.

Conformant GraphPlan can also include uncertainty in the effects of actions. Consider the case when the dunking action may or may not clog the toilet. This could be represented as different possible outcomes in *PDDL*-like[6] fashion as follows.

```
(:action dunk3
      :parameters     (?p - package ?t -toilet)
      :precondition   (not  (clogged ?t))
      :effect         (or   (when(in bomb ?p) (diffused bomb))
                            (and (clogged ?t)
                            (when(in bomb ?p) (diffused bomb))))
```

This action will have two aspects as before.

```
(:action dunk3+
      :parameters    (?p - package ?t -toilet)
      :precondition (and  (in bomb ?p) (not (clogged ?t)))
      :effect        (or   (diffused bomb))
                     (and  (clogged ?t) (diffused bomb))))
(:action dunk3-
      :parameters    (?p - package ?t -toilet)
      :precondition (not  (clogged ?t))
      :effect        (or   ( )
                           (clogged ?t)))
```

Inclusion of such an aspect in the planning graph in world w_i will result in w_i splitting into two new worlds w_{i1} and w_{i2} in the next layer, containing the two different effects. Observe that action *dunk*3- may not have any effect in some resulting world. If we were to expand the first layer of the planning graph in w_1 then we would include two aspects *Dunk3+*(*p1, t*):*w1* and *Dunk3-*(*p2, t*):*w1*. These two actions would split the world w_1 into four worlds—w_{11}, w_{12}, w_{13} and w_{14}. In w_{11}, the package p_1 would be dunked, the bomb would be diffused and the toilet clogged; in w_{12} the package p_1 would be dunked, the bomb would be diffused and the toilet would not be clogged; in w_{13}, the package p_2 would be dunked, the bomb would be diffused and the toilet clogged; and in w_{14}, the package p_2 would be dunked, the bomb would not be diffused and the toilet would not be clogged. As one can see, there is going to be an explosion in the number of worlds the planner has to deal with. The solution extraction process, however, remains the same. For more details and for suggestions to keep the number of possible worlds in check, the interested reader is referred to (Smith and Weld, 1998).

More recently, there have been approaches extending and exploiting the planning graph for planning with durative actions. We will explore durative actions later in the chapter.

10.2 Planning as Constraint Satisfaction