## 7.5 Plan Space Planning

The planning approaches described above reason with states. The planner is basically looking at a state and a goal. If the state satisfies the goal then it terminates. Otherwise, it makes a search move *over the state space* looking for actions to add to the plan.

An alternative view is to consider the *space of all possible plans*, and search in this space for a plan. We will call such approaches as *plan space planning*. Most algorithms in this category represent a plan as actions arranged in a partial order, and hence we also use the term *partial order planning*. Unlike in the state space methods described above, there is no restriction on the order in which actions are *added* to the plan. Since state space methods focus on the state, or on the goal which is a partial state description, the search methods look at ways to go to neighbouring states that are one move away. Thus, states are explored in a linear fashion, and the plans, which are themselves linear structures, grow linearly, being extended at one of the two ends. Plan space planning approaches work with plan structures, and have the ability, in principle, to modify or extend any part of a plan. That is why these methods are also known as *nonlinear planning* methods. Because of the fact that they can modify any part of a plan, the plan space planning approaches are not constrained to focus on any one subgoal continuously. They can shift attention midway, and in the process often solve problems, like the Sussman anomaly correctly as shown in Figure 7.13.

The start node for search in plan space planning is the empty plan $\Pi_0$. It is represented by two actions that we will call $A_0$ and $A_\infty$. These two special actions will be part of every plan. The first, $A_0$, has no preconditions, and its effects are the predicates describing the start state. The second, $A_\infty$, has as its preconditions the goal predicates, and has no effects, as shown in Figure 7.14 (for the tiny planning problem from Figure 7.18). Thus, every planning problem will have a distinct start node that will capture both the start state and the goal description.

```
RecursiveGSP(givenState, givenGoal, actions)
1  state ← givenState
2  goal ← givenGoal
3  plan ← ()
4  while TRUE
5    do if Satisfies(state, goal)
6         then  return plan
7         else  Let R be set of relevant actions for goal
8             for each subgoal g ∈ goal      /* AND node */
9               do
10                 CHOOSE an action a in R   /* OR node */
11                 if no such action exists
12                   then return FAIL
13                 subPlan ← RecursiveGSP(state, Preconditions(a), actions)
14                         /* searches in the backward fashion */
15                 if subPlan = FAIL
16                   then return FAIL
17                   else    /* a deterministic version will backtrack */
18                     state ← PlanProgress(subPlan, state)
19                       /* progresses State in the forward direction */
20                     state ← Progress(a, state)
21                         /* assembles plan in the forward direction */
22                     plan ← (plan • subPlan • a)
```

**Figure 7.11** Recursive *GSP* illustrates the dual nature of *GSP*. It considers actions by their relevance, but selects them only on applicability. For simplicity, we write a nondeterministic version with a CHOOSE operator that makes the correct choice. The deterministic version may make a wrong choice, but backtrack to try again.

```
PlanProgress(plan, state)
1  if Empty(plan)
2    then  return state
3    else  state ← Progress(Head(plan), state)
4 return PlanProgress(Tail(plan), state)
```

**Figure 7.12** Algorithm *PlanProgress* iteratively progresses over the actions in a plan.

As planning proceeds, more actions are added to the plan. Interestingly, there is no constraint on where actions should be added. State space planning algorithms grow linear, partial plans at one end. Plan space methods separate the tasks of selection of an action and its placement in the plan.

In addition to the set of actions, the plan representation contains links between actions. The links are of two types. The first, called *ordering links*, are used to capture ordering information where it is known. The initial plan for example, has a default link $(A_0 \prec A_\infty)_\infty$ to assert that the start action $A_0$ happens before the end action $A_\infty$. The second kind of link, first introduced in a system called NONLIN (Tate, 1977), is called a *causal link*. A causal link $(A_i, P, A_j)$ between two actions $A_i$ and $A_j$ can be established when an affect $P$ of action $A_i$ is a precondition for action $A_j$. Action $A_i$ is the *producer* of predicate $P$, and action $A_j$ is the *consumer* of $P$. Figure 7.15 illustrates a causal link between the two actions *Pickup*(*A*) and *Stack*(*A, B*).

When the link is *established*, it represents a commitment on the part of action $A_i$ to support the precondition $P$ for action $A_j$. Once established, the algorithm will have to ensure that the link is not disrupted or *clobbered*. If it is

clobbered during the planning process then it will have to be *declobbered*. A causal link ($A_i$, $P$, $A_j$) is said to have a *threat* if there exists another action $A_t$ in the plan that potentially deletes $P$, that is, has (*not* $P$) in its effects. One may *even* treat an action as a threat if it produces $P$, because it threatens to make action $A_i$ redundant (McAllester and Rosenblitt, 1991; Kambhampati, 1993).

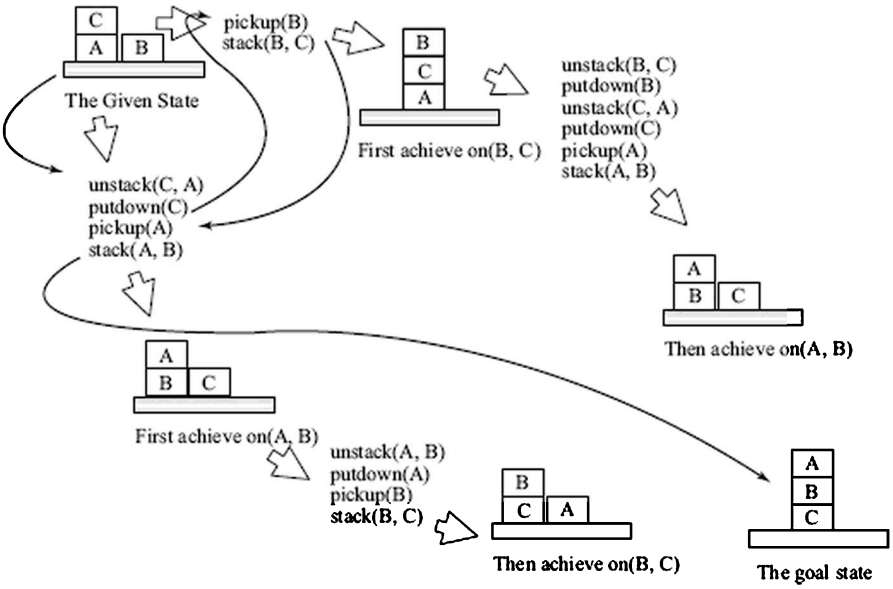
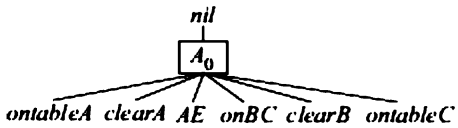



**Figure 7.13** To find an optimal plan for the Sussman anomaly, the algorithm should start off with on(*A*, *B*), switch to on(*B*, *C*) on the way, and return to on(*A*, *B*) again, instead of attempting them linearly.



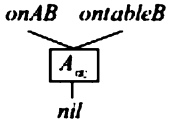



**Figure 7.14** The empty plan $\Pi_0$ has two actions. Action $A_0$ produces the start state, while action $A_\infty$ accepts the goal state.

If an action *A* has a precondition *P* that is not causally linked then we say that *P* is an *open precondition*[7]. A solution plan cannot have any open preconditions or threats. Together, the two are also called *flaws*. It has been shown (Penberthy, 1992) that if a partial plan does not have any flaws then it is a solution to the planning problem. The starting plan containing the two actions ($A_0$, $A_\infty$) will be a solution plan only if the goal predicates are already true in the start state. That is, for each precondition of $A_\infty$, the action $A_0$ is the producer, and the corresponding causal links are established. If the algorithm can establish these links then it can terminate.

Plan space planning often follows what is known as a *least commitment strategy* (Weld, 1994). This implies that the algorithm commits to a particular feature in a plan, only when it has to. For example, in the blocks world, if the planner has to achieve *armempty* when it is holding a block say *B*. It might want to choose the *stack* action, but it may not know where to stack the block. It then makes sense to use only partially instantiated operators, and insert *stack*(*B, ?X*)[8] into the plan instead of choosing to guess and instantiate it to say *stack*(*B, Q*). We will use partially instantiated operators. Then for each variable, one has to keep track of the binding. So in addition to the two kinds of links, the plan representation will also contain a set of *binding constraints* that contain information of what specific variables can be bound to or cannot be bound to.

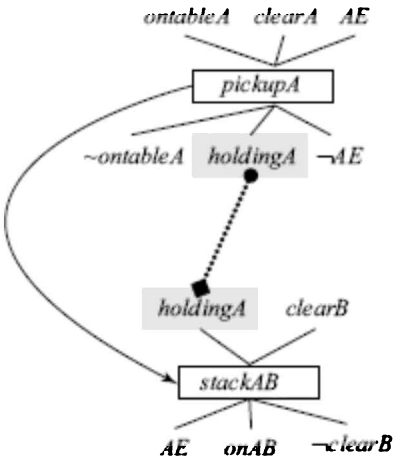



**Figure 7.15** The causal link (*Pickup(A), holding(A), stack(A, B)*) is shown with the dotted link. The curved arrow is the ordering link between the two actions.

Thus, a node in plan space search will represent a partial plan, and will contain a set of partially instantiated operators, causal and ordering links, and binding constraints.

## *Definition*

A partial plan P is a 4-tuple

$\Pi = (A, O, L, B)$ where

— $A$ is the set of partially instantiated operators in the plan,

— $O$ is the set of ordering relations of the form $(A_i \prec A_j)$,

— $L$ is the set of causal links of the form $(A_i, P, A_j)$,

— $B$ is the set of binding constraints of the form $(?X = ?Y)$, $(?X \neq ?Y)$, or $(?X \in D_X)$ where $D_X$ is a subset of the domain of $?X$.

The initial node in plan space search is defined by the partial plan,

$$\Pi_0 = (\{A_0, A_\infty\}, \{(A_0 \prec A_\infty)\}, \{\}, \{\})$$

The search space is the *implicit* directed graph whose vertices are partial plans and whose edges correspond to refinement operations. Each refinement operator transforms a partial plan $\Pi$ into a successor partial plan $\Pi'$ by augmenting one of the sets. The different refinement operations are,

— adding an action or partially instantiated operator to $A$,

— adding an ordering constraint to $O$,

— adding a causal link to $L$, or

— adding a binding constraint to $B$.

The choice in the refinement process is itself driven by the need to remove *flaws* from a partial plan. As described above, flaws can be of two types; open subgoals or threats. We look at ways to remove each of them.

An open goal refers to a precondition $P$ of some action $A_P$ in the plan that is not causally linked to another action. A causal link for $P$ can be found in two ways.

1. If an existing action $A_e$ produces $P$ and it is consistent to add $(A_e \prec A_P)$ then one can establish a causal link $(A_e, P, A_P)$ to the partial plan.
2. If no such existing action can be found then one has to insert a new action $A_{new}$ to the partial plan, add the corresponding causal link $(A_{new}, P, A_P)$, and add the ordering link $(A_e \prec A_P)$ to the partial plan.

An action $A_{threat}$ that can possibly disrupt an existing causal link $(A, P, A_j)$ is a threat to the link. Disruption will occur if all three of the following happen:

1. $A_{threat}$ has an effect $\neg Q$ such the $P$ can be unified[9] with $Q$.
2. It is possible for $A_{threat}$ to happen after $A_t$.
3. It is possible for $A_{threat}$ to happen before $A_j$.

If all the three happen then we say that the threat has materialized. To eliminate the threat, one needs to ensure that at least one of the three conditions for the threat is not met. This can be done by the following:

**1. Separation** Ensure that $P$ and $Q$ cannot unify. This can be done by adding an appropriate binding constraint to the set $B$ in the partial plan.

**2. Promotion** Promote the action $A_{threat}$ to happen before it can disrupt the causal link. That is, insert an ordering link $(A_{threat} \prec A_j)$ into the partial plan (add it to set $O$).

**3. Demotion** Demote the action to happen after both the causal link actions. That is, add an ordering link $(A \prec A_{threat})$ to the set $O$ in the partial plan.

Figure 7.16 shows an example of a threat. Assume that at some stage there is a partial plan with two actions *pickup(A)* and *stack(B, ?Y)*. Of the many open conditions, let the algorithm choose *clear(A)* and insert a new action *unstack(?X, A)* in the plan establishing the causal link (*unstack(?X, A), clear(A), pickup(A)*).

At this point, one can notice that the existing action *stack(B, ?Y)* is a threat to the newly established causal link, because produces ¬*clear(?Y)*, and if ? *Y* is bound to *A* then it could *possibly* disrupt the link. To resolve this threat, one the of three methods described above can be chosen. They are given below:

**1. Separation** Ensure that *clear(?Y)* and *clear(A)* cannot unify. This can be done by adding the binding constraint (? $Y \neq A$) to the set *B* in the partial plan.

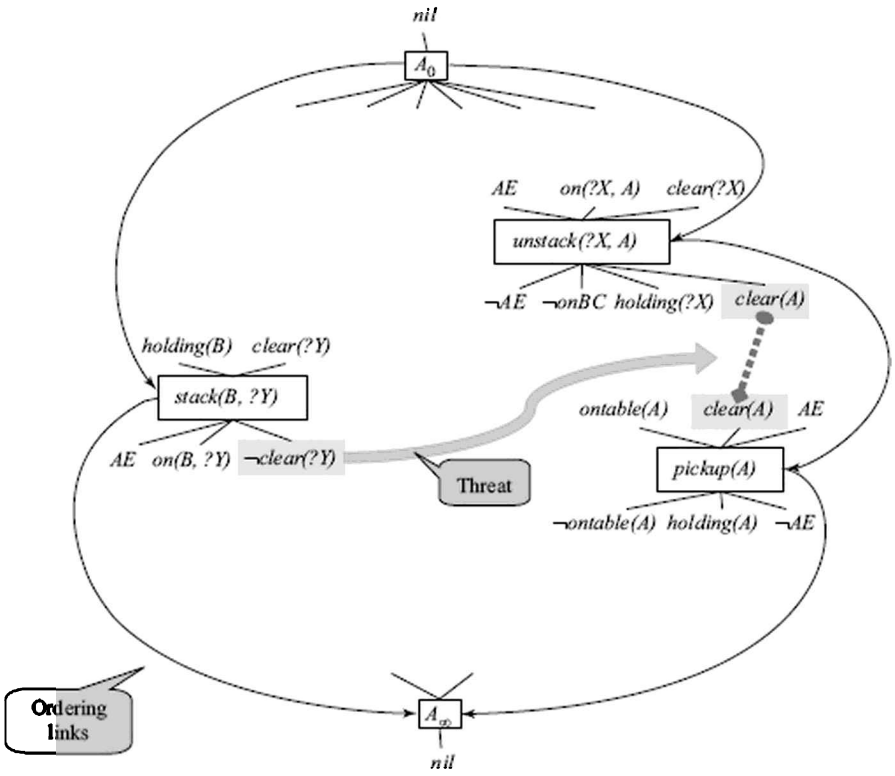



**Figure 7.16** The action stack(*B*, ?*Y*) is a threat to the causal link for the proposition clear(*A*) produced by *unstack*(*?X, A*) and consumed by *pickup*(*A*).

**2. Promotion** Promote the action *stack*(*B, ?Y*). Insert an ordering link (*stack*(*B, ?Y*) ≺ *unstack*(*?X, A*) into the partial plan.

**3. Demotion** Demote the action *stack*(*B, ?Y*) to happen after the causal link actions. That is, add an ordering link ($pickup(A)_j \prec stack(B, ?Y)$ to the set *O* in the partial plan.

At any stage of planning, a refinement step can be applied to the partial plan, provided it does not introduce an inconsistency. For the ordering links to be consistent, there must be no cycles in the directed graph representing the ordering relations. In other words, the action must begin with $A_0$ and move forward without turning back, ending with $A_\infty$. Ordering links are added (a) along with causal links and (b) during promotion or demotion of actions. Before each of these operations, a check must be made for consistency. Likewise, for the binding constraints to be consistent, all assignments to variables should be consistent with all the binding constraints. Whenever a new binding constraint is added, a check for consistency for the variables involved must be made[10]. The causal links too must not have cycles, but a check is not required because whenever a causal link is added to the partial plan, a corresponding ordering link is added as well, and it suffices to check that the ordering links are consistent.

The basic plan space algorithm described below (Figure 7.17) is adapted from (Ghallab et al., 2004). It is initially invoked with the empty plan $P_0$ as the argument.

```
PSP(π)
  1   flaws ← OpenGoals(π) ∪ Threats(π)
  2   if Empty(flaws)
  3     then return Π
  4   CHOOSE f ∈ flaws
  5   resolvers ← Resolve(f, π)   /* the set of resolvers for flaw f */
  6   if Empty(resolvers)
  7     then return FAIL
  8   CHOOSE r ∈ resolvers
  9   π' ← Refine(r, π)
  10  return PSP(π')
```

**Figure 7.17** The plan space planning (PSP) procedure selects a flaw in a given plan, and looks for a resolver that can remove the flaw. The Refine procedure applies the chosen resolver, and the algorithm PSP is called recursively.

In Line 1, the set of flaws is identified as the union of the set of open conditions and the set of threats. The procedures that return the two constituent sets need to maintain the two sets and update them incrementally in each cycle during the call to *Refine* in Line 9. The set of open conditions is usually maintained as an agenda of goals, to which new ones are added every time open conditions are created. Likewise, every time an ordering condition is added, a check is made for the possibility of a threat. Observe that whenever actions are added, one or more causal links as well as ordering links are added as well. In Line 4, one of the flaws is chosen for resolving. In theory, any flaw could be chosen because all the flaws have to be resolved for the algorithm to terminate. In practice, while implementing a deterministic algorithm choosing a flaw that has a smaller number of resolvers could lead to less backtracking. In Line 5, the set of resolvers for the chosen flaw that can be consistently applied are identified. The algorithm has to check that no ordering constraints and binding constraints are violated by the addition on new ones. This ensures that in Line 9, the resolver, chosen nondeterministically[11]in Line 8, can be applied without any problem, and procedure *Refine* has only to update the relevant structures being

maintained. The *PSP* algorithm treats both kinds of flaws equally.

A variation of the above algorithm called *Partial Order Planner* (POP) works only with open goals in an agenda. Every time it finds a way of satisfying the open goal, it looks for any threats created and resolves them before moving onto to the next goal on the agenda. Thus, plan space planning or partial order planning can be seen to go through a cycle of "R" steps. Remove a flaw (or an open goal) from the agenda. Resolve the flaw. Refine the partial plan, and in the process, Revise the agenda.

Let us look at a small but complete example. Let the initial state of the problem be {*ontable*(*A*), *clear*(*A*), *ontable*(*C*), *on*(*B, C*), *clear*(*B*), *AE*} where *AE* stands for *armempty*. Let the goal predicates be {*on*(*A, B*), *ontable*(*B*)}. The problem is depicted in Figure 7.18.

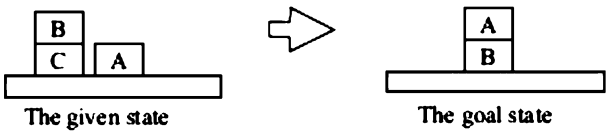



**Figure 7.18** A small planning problem. Note that nothing is said about block C in the goal state. It could be on the table or on block A.

Figure 7.19 shows a partial plan after four actions have been added as described below.

The empty plan with actions $A_0$ and $A_\infty$ captures the initial state and the goal clauses. The two goal clauses are the open conditions. The planner chooses one of them, say *on*(*A, B*), and inserts an action *stack*(*A, B*). It inserts an ordering link (*stack*(*A, B*) $\prec A_\infty$), and a causal link (*stack*(*A, B*), *on*(*A, B*), $A_\infty$). The preconditions of the action *stack*(*A, B*) now appear as open conditions, *holding*(*A*) and *clear*(*B*).

Let us say that the planner then looks at *clear*(*B*) and finds that it has a producer in the action $A_0$. It establishes a causal link (shown in the figure) and an ordering link (not shown) for *clear*(*B*). It could then take up the open condition *ontable*(*B*) and add the action *putdown*(*B*), and the corresponding links. Next comes the open condition *holding*(*A*), and for that, it inserts the action *pickup*(*A*) and its links.

The only open condition left at this point is *holding*(*B*), and for the planner, insert the action *unstack* (*B, C*) into the plan. At this point, three threats appear, marked as $t_1$, $t_2$ and $t_3$ in the figure.

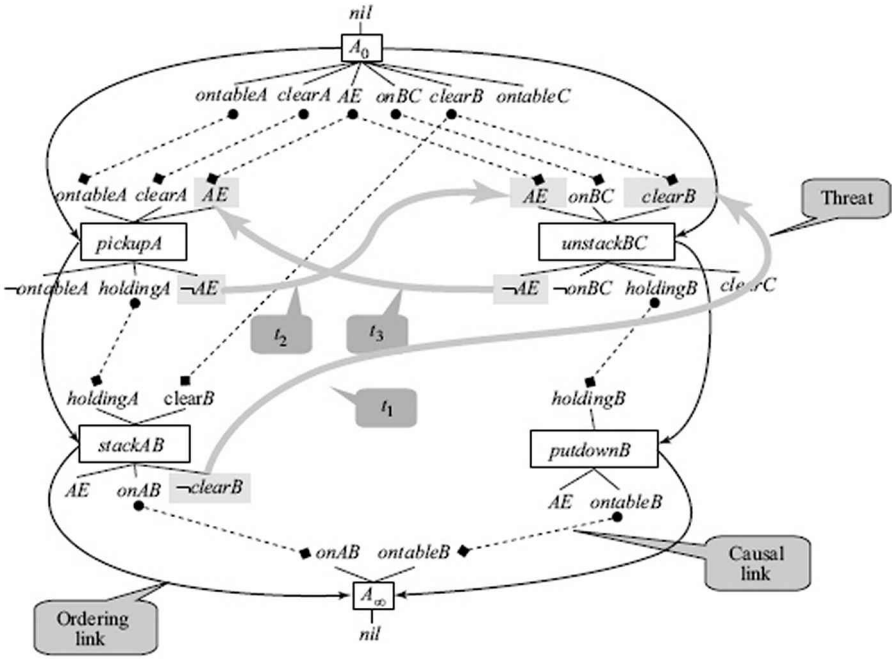



**Figure 7.19** A partial plan with four actions added in the order *stackAB*, *putdownB*, *pickupA*, *unstackBC*. After the last action, there are no open conditions, but three threats $t_1$, $t_1$, and $t_3$ appear.

Threat $t_1$ is that if *stack*(*A, B*) is done before *unstack*(*B, C*) it will disrupt (or *clobber*) the precondition *clear*(*B*) of the latter. This can be *resolved* by *demoting* action *stack*(*A, B*) to happen after *unstack*(*B, C*). This is done by adding an ordering link (*unstack*(*B*, *C*) $\prec$ *stack*(*A*, *B*)) into the plan.

Threat $t_2$ and threat $t_3$ are symmetrical in nature. Let us assume that the planner correctly resolves $t_2$ by likewise demoting action *pickup*(*A*) by adding the link (*unstack*(*B, C*) $\prec$ *pickup*(*A*)). Notice now that the threat $t_3$ has become definite, in the sense that we know that the *AE* condition for *pickup*(*A*) is going to be clobbered by *unstack*(*B, C*). The corresponding causal link is broken, and we have the open condition *AE* for *pickup*(*A*). We cannot demote *unstack*(*B, C*) without making the ordering links inconsistent. Instead, the planner looks for an action to achieve *AE* again. It finds it within the plan itself. Action *putdown*(*B*) can produce *AE* for *pickup*(*A*) to consume. Thus, *declobbering*, as it was first described in a system called TWEAK (Chapman, 1987), is done by inserting ordering link (*putdown*(*B*) $\prec$ *pickup*(*A*)) and causal link (*putdown*(*B*), *AE*, *pickup*(*A*)) between the two actions. The resulting plan with no threats or open conditions is shown in Figure 7.20. Only the necessary subset of ordering links is shown for clarity, like in a Hasse diagram.

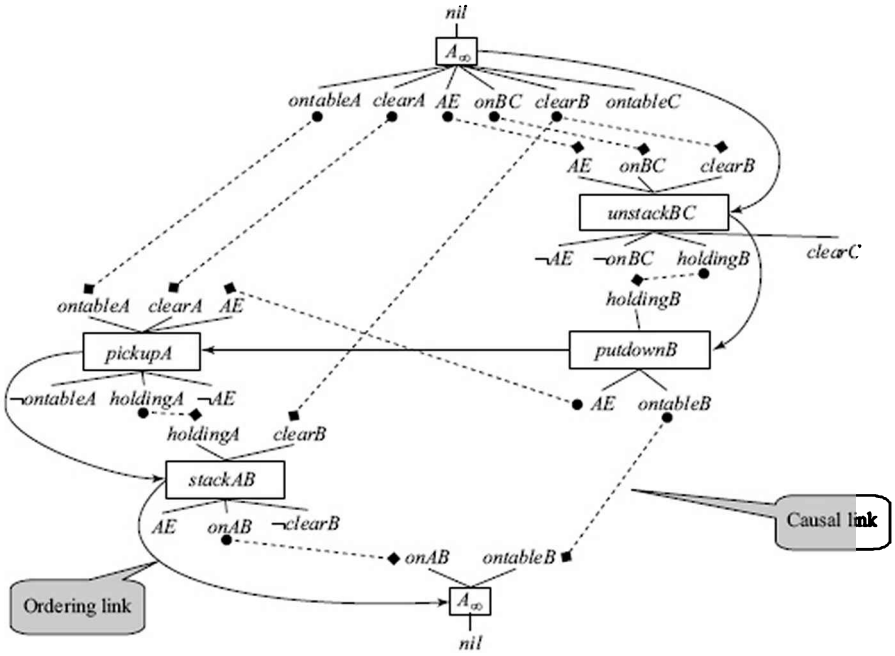



**Figure 7.20** The final plan after threats $t_1$, $t_2$ and $t_3$ are resolved. There are no open conditions or threats, and hence this is a solution plan.

The reader would have observed that the resulting plan is a linear plan. This is inevitable in the blocks world domain, because the one-armed robot can hold only one block, and therefore only one action can be done at a time. If the robot could hold more than one object then one could have partial plans that are not linear orders. For example, a parallel step could say *pickup*(*A*) and *pickup*(*B*) without specifying the order, that is there would be no ordering constraint between the two actions, and the plan could be correctly linearized in any order.

An example of such parallel actions could be the “dressing up for school” problem, for which the plan is shown in Figure 7.21. The subplan has four actions—*wear*(*left, sock*), *wear*(*right, sock*), *wear*(*right, shoe*) and *wear*(*left, shoe*)—that follow a *comb-hair* action with only constraints that each sock must be worn before the respective shoe. Otherwise, the actions could be done in any order. This gives us a compact representation of a plan, which stands for all the linear plans that are consistent with the ordering constraints.

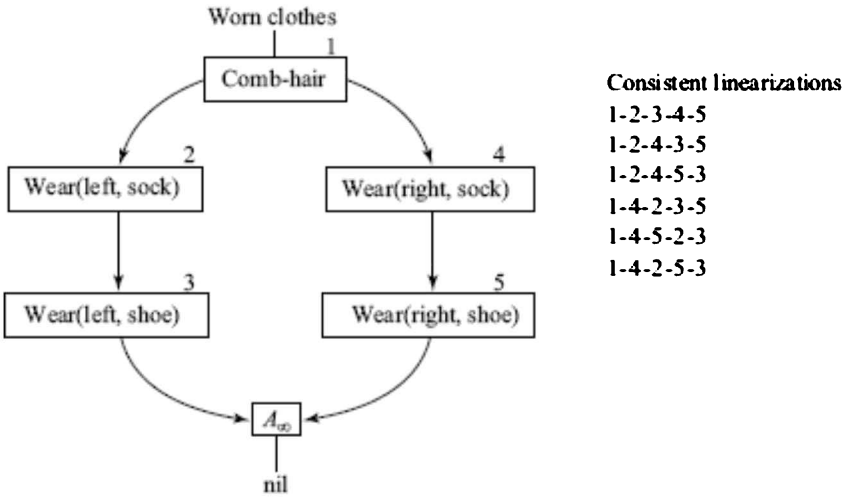



**Figure 7.21** A partial plan may stand for many linear plans.

The algorithms above have been written in a nondeterministic manner. While this results in a succinct description, it hides the fact that the algorithm still has to search through a combination of choices. In practice, one will have to choose a combination of heuristics to guide the search, and introduce backtracking to make it complete. The reader might have noticed that the search space for PSP is infinite, even when the state space is finite. Given a two block problem in the blocks world domain, there are only four distinct states possible. But the number of candidate plans is infinite, because one can always insert the two actions (*pickup*(*X*), *putdown*(*X*)) where applicable *any number of times.* Since the representation does not involve states at all, one cannot check that the same state is being visited again and again. One can, in principle, explore plans of arbitrary length without moving towards a solution. Thus, there is a need to control the search strategy efficiently. One way to do so is to explore plans in an iterative deepening manner, keeping an iteratively increasing bound on the number of actions in the partial plan. This would ensure that plans are considered in the order of increasing length, and the solution plan would be found in finite time.

The reader would also have noticed that the PSP algorithm has a backward reasoning flavour, since it looks for ways to resolve open conditions, though the representation is very different from BSSP. While BSSP can run into spurious states, PSP avoids that pitfall having done away with states altogether in its representation. The sequence in which both consider actions can still be similar. The partial plan representation on the other hand is powerful enough to accommodate other search strategies, with some modifications as discussed below. The approaches described below have a flavour of top down or goal-directed problem solving.

## 7.5.1 Means Ends Analysis

*Means Ends Analysis* (*MEA*) is a strategy first introduced in a system called *General Problem Solver* (*GPS*) (Newell and Simon, 1963). It was one of the first general purpose approach to problem solving, or planning. The basic idea in *MEA* is to,

- Compare the given state and the desired state and arrive at a set of differences
- Choose the most significant difference and look for a plan to reduce that difference recursively
- Apply the operators chosen and look at the problem again for any more differences

We can think of the *GSP* algorithm in *STRIPS* as an example of the *MEA* strategy. As described in Figure 7.11, the *recursiveGSP* algorithm looks at the goal propositions, and attempts to solve one of them before considering the next one. The *STRIPS* planner focuses on the goal propositions. Each goal proposition is a difference to be tackled. It chooses a goal to tackle from the goal set in a predefined manner. *MEA*, on the other hand, has a more top-down view of problem solving, and it requires the following set of procedures,

- A MATCH procedure that compares two states and returns a set of differences, if any, between them. There exists an efficient algorithm for doing this, the *Rete* algorithm, described in Chapter 6 in a different context.
- A procedure for ordering the differences to arrive at a set of goals (Ends) to be achieved
- A set of operators (Means) to reduce the differences.

The versatility of the algorithm depends upon the availability of knowledge for the above three procedures. In *STRIPS*, the set of differences is simply the set of open goals or subgoals, and as observed above, it does not order the differences. The set of operators can be provided in a modular form as in PDDL. Consequently, the algorithm can be written in a domain independent form. On the flip side, as we have seen, *GSP* does not always find the best plans. In more general cases, some of the above knowledge would require more domain related procedures and may not be general enough. We illustrate the *MEA* strategy with a few examples below. The reader is encouraged to think of the strategy as working on a partially ordered plan structure, in which actions are inserted because they address the most significant differences between the desired state and the *effects* in the current partial plan. In particular we may insert actions based on criteria other than open condition satisfaction and threat removal, and work with a more general notion of flaws which may vary from domain to domain. We can think of the flaws (in the more general sense) as the differences in *GPS*. Furthermore, we order the flaws in some order of importance or difficulty and tackle them in that order.

Consider the task of planning a trip from IIT Madras (IITM) in Chennai to the Technical University in Munich (TUM). The problem is to overcome the difference (distance) in our current state (at IITM) and the desired state (at TUM). An FSSP may start by planning with applicable moves from IITM, and a *BSSP* or *GSP* may start looking at the relevant moves from the goal (at TUM). They would try and construct the plans in linear fashion at the level of

detail of actions. A *GPS* like solver may first address the biggest difference, in terms of distance, and observe that we can reduce the distance from Chennai to Frankfurt (the most significant difference) because we have the means to do so. That is, there is a flight from Chennai to Frankfurt. As one can notice, this information comes from the domain. We often think of such an operator first because we visualize the differences as distances and operators as means of transport. We could be working with an operator difference table that looks as follows.

**Table 7.1** A possible operator difference table for travelling, lists the modes of transport that can be used to cover different distance ranges.

| | *Modes of transport* | | | | | |
|---|---|---|---|---|---|---|
| *Distances* | *Aeroplane* | *Train* | *Car* | *Taxi* | *Bus* | *Walk* |
| More than 5000 km | yes | | | | | |
| 100 km to 5000 km | yes | yes | yes | | | |
| 1 km to 100 km | | yes | yes | yes | yes | |
| Less than 2 km | | | yes | | yes | yes |

Observe that it is still not straightforward to use this table, because looking at the original problem, one has to choose from the set of available options and operator that would reduce the largest difference. Selecting a Chennai–Frankfurt flight[12], for example, implies the knowledge that there are no Chennai– Munich flights. Once we manage to decide upon the operator, we are left with two new problems to solve. They are,

1. Given: (at IITM), Desired: (at Chennai airport)
2. Given: (at Frankfurt airport), Desired: (at TUM)

One can solve these independently, consulting the operator difference table at each stage. A more concrete example that is easier to implement is the Towers of Hanoi[13] problem. This problem

also illustrates the utility of being able to order the differences. The domain constitutes of three pegs, or locations, and a tower of *N* disks on one peg to start with. The disks are all of different diameters, and any disk can be placed only on top of a larger disk or on an empty peg. A move constitutes of transferring a disk from the top of one peg, to another peg where possible. Figure 7.22 depicts a problem with three disks.

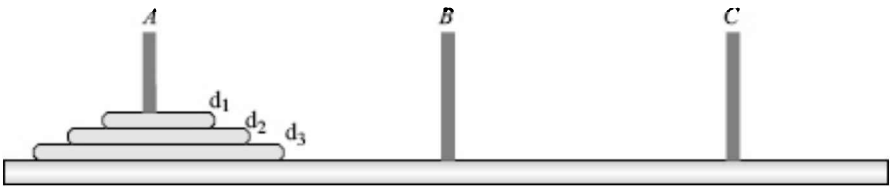



**Figure 7.22** The Towers of Hanoi. The task is to move the tower of *N* (3 in the figure) disks from location peg *A* to peg *C*. Only the topmost disk on a peg may be moved, and cannot be placed on a smaller disk. The problem is known to have solutions (plans) that are exponentially long: $(2^N - 1)$ moves for towers with *N* disks.

The Towers of Hanoi problem is very structured and often used to illustrate recursive algorithms (to move a complete tower move all disks, but

one recursively to a temporary peg, move the remaining disk to the destination, and move the other disks recursively to the destination). However, if the problem is treated as a planning problem, it poses a challenge because the optimal solution is known to be exponential in length. For a problem with $N$ disks, the number of moves in the solution is $(2^N - 1)$. Combined with the fact that in each intermediate state, either two or three moves are possible, one can see that the search space is very large.

The *MEA* strategy works well with the ordering information that the largest disks are hardest to move since they may have more disks on top of them. Then, if the differences measured are in terms of differences in location of each disk, the differences for the largest disks must be reduced first. In Figure 7.2, three disks have to be moved from tower *A* to tower *C*. The differences in order of importance are,

- $D_3$: disk $d_3$ is on peg *A* and not on peg *C*
- $D_2$: disk $d_2$ is on peg *A* and not on peg *C*
- $D_1$: disk $d_1$ is on peg *A* and not on peg *C*

Having chosen the difference $D_3$ to reduce, *GPS* creates a goal $G_3$ of reducing that difference. This is because the operator to reduce the difference may not be applicable in the given state. For example, the operator to reduce $D_3$ is to move disk $d_3$ from *A* to *C*, but that can be only done if disk $d_3$ is clear, and peg *C* is empty (because $d_3$ is the largest disk; otherwise the condition would be that the existing disk on peg *C* is larger). These conditions would be true if disks $d_1$ and $d_2$ were not on peg *A*, and also not on peg *C*. Thus, *GPS* would recursively create a new set of differences as follows,

- $D_{32}$: disk $d_2$ is on peg *A* and not on peg *B*
- $D_{31}$: disk $d_1$ is on peg *A* and not on peg *B*

And work towards reducing $D_{32}$. Continuing in the same manner, it would create a new goal to move $d_2$ from *A* to *B*, and then another to move $d_1$ from *A* to *C*. This is a move it can make, and it will do so going to a new state $S_1$ in which $d_1$ is on *C*. It can now move $d_2$ from peg *A* to peg *B*, thus achieving the goal of reducing $D_{32}$. Now it looks at the difference $D_{31}$ and revises the difference to

- $D'_{13}$: disk $d_1$ is on peg *C* and not on peg *B*

which can be reduced by moving $d_1$ from peg *C* to peg *B*. Having done all this, *GPS* is now in a position to solve goal $G_3$ by moving disk $d_3$ to peg *C*. After this, it goes on to reduce the revised differences $D_2$ and $D_1$, revised because the world may have changed in the interim period. The reader would recognize the similarity of the "trace" with the one for *GSP*, and in fact as observed earlier, *GSP* is a special case of *GPS*. If one were to use the *GSP* planner to solve the Towers of Hanoi and give it additional information on ordering of goal propositions (lowest blocks/disks in goal set first), we would have in effect the algorithm described above.

Problem decomposition by the *MEA* strategy can also be seen to generate an AND/OR tree (Nilsson, 1971). At the *AND* level, the strategy breaks up the problem into two parts, which are ordered (see Figure 7.23). In the first part, it poses a goal to reduce the largest difference, and in the remaining part, it addresses the remaining differences. Reducing the largest

difference itself could be done in several ways and each of them becomes a choice at the *OR* level. Having selected an operator, the *MEA* strategy recursively poses the problem of achieving the preconditions for that operator and the task of applying that operator when its preconditions are achieved.

The reader would again have noticed the likeness in structure of the *recursiveGSP* algorithm in Figure 7.11 and Figure 7.23 showing the AND/OR tree above. One of the many contributions of the pioneering work on *GPS* is the recursive approach to problem solving.

The key to performance in *GPS* is an ordering of the differences to be addressed, and the availability of operators to reduce the differences. One desirable property of the operators is that applying an operator to reduce a difference should not undo the work of a previous operator. We have seen that such a problem can occur even in the blocks world with the example on Sussman's anomaly. While it may not be possible to find primitive operators that satisfy this property, Korf has shown that one could construct operator difference tables in which the operators are not primitive operators, but chunked together into macros. In other words, one can even tackle nonserializable subgoals by devising macro moves that can be applied in a serial order (see also the section on Peak-to-Peak Heuristics in Chapter 3). Table 7.2 below shows the macro table for the Eight-puzzle (taken from (Korf, 85)) for the goal state shown in Figure 7.24.

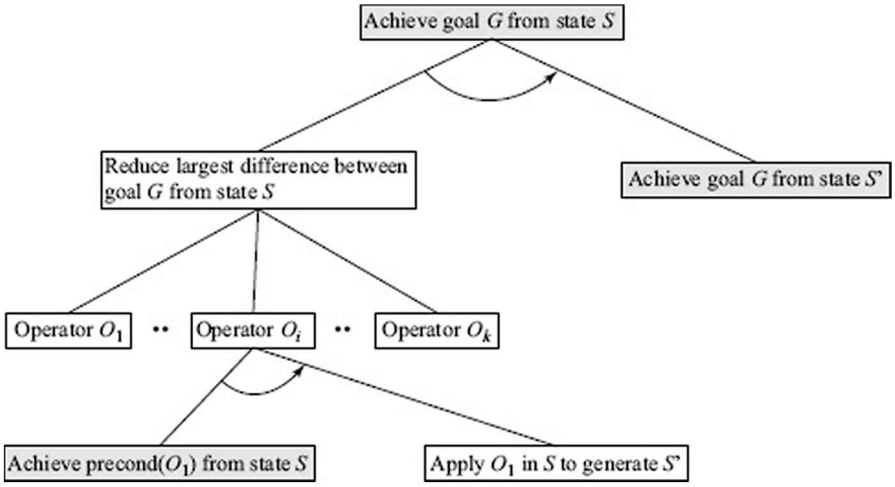



**Figure 7.23** The *MEA* strategy generates an *AND*/*OR* tree by replicating the above structure below the shaded nodes when a recursive call is made.

**Table 7.2** A macro operator table for the Eight-puzzle

| | | TILES | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| POSITIONS 0 | — | | | | | | |
| 1 | UL | — | | | | | |
| 2 | U | RDLU | — | | | | |
| 3 | UR | DLURRDLU | DLUR | — | | | |
| 4 | R | LDRURDLU | LDRU | RDLLURDRUL | — | | |
| 5 | DR | ULDRURDLDRUL | LURDLDRU | LDRULURDDLUR | LURD | — | |
| 6 | D | URDLDRUL | ULDDRU | URDDLULDRRUL | ULDR | RDLLUURDLDRRUL | — |
| 7 | DL | RULDDRUL | DRUULDRDLU | RULDRDLULDRRUL | URDLULDR | ULDRURDLLURD | URDL |
| 8 | L | DRUL | RULLDDRU | RDLULDRRUL | RULLDR | ULDRRULDLURD | RULD |

An entry in row *I* and column *J* is a macro move for reducing the difference in the position of tile *J* if it is currently in the final position of tile *I*. Tile 0 stands for the empty tile. The macro moves are composed from four primitive moves *R*, *D*, *L*, and *U* which stand for moving an appropriate tile right, down, left or up. Note that the notation is unambiguous, because only one tile can make each of the primitive moves at any time. For example, in the position in Figure 7.24, move *L* means tile 4 is moved into the empty place, and *U* means tile 6 is moved.

The shaded squares can be interpreted as follows. If square at row *I* and column *J* is shaded, it means that the move to reduce the difference for tile *J* does not interfere with the location *I* (position of tile *I* in the goal state). In other words, the difference for tile *I* is invariant to the reduction of difference for tile *J*. In general, if we have operators such that we can arrange the invariance between differences in a triangular form then one can reduce the differences in a serial order (see note by G Ernst (1987) for another example of the triangle property for the Fool's Disk puzzle).

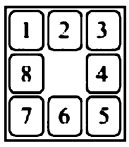



**Figure 7.24** The Goal State for which the macro Table 7.2 is constructed.

In the above table, the differences are arranged in a triangular invariance order. This means that if we reduce the differences in the given order then the problem can be solved in a linear fashion. Thus, the differences to be reduced are in the order from tile 0 to tile 6, by the end of which the solution is reached. In other words, given any starting position, one can find a solution for the above goal in the Eight-puzzle by bringing tile 0 into place, bringing tile 1 into place, and so on using at each point the appropriate macro from the table. The reader would have observed that these are not the only macro moves that are possible. For example, the above table does not

tell us what to do if we want to reduce the difference for tile 4, if it is in the position of tile 2. The point is that such a macro is not required for completeness, because by the time the turn of tile 4 comes, tile 2 would already be in place, and hence tile 4 could not have been in the location of tile 2. Also the macros are not unique. For example, the macro UL at location (1, 0) could also been LU. It has been shown that if we can find operators that allow us to arrange the differences so that the invariance relations can be arranged in a triangle then the solver is complete (Banerji, 1977). Using the macro table, one can solve the problem from any starting position when a solution exists. The solution found may not be optimal though. Finally, one could choose a different order of solving the differences, and then find the appropriate macro moves. In fact, the work by Korf demonstrates how this can be done.

### 7.5.2 *NOAH*

An important characteristic of top-down problem solving is that one should start by looking at the problem in totality at the abstract level, and then work on the detail. The search based methods we have seen earlier operate at the level of primitive moves or operators, and try and synthesize plans or solutions from them. In the *GPS* formulation, one does look at a problem and address the largest difference, but it needs an appropriate set of operators. Korf showed that one can construct appropriate macros from primitive operators and use them in planning with the *MEA* strategy. But in doing so, all the hard work has been pushed into the task of finding or learning those macro operators. Given the appropriate operator difference table, there is no search required any longer, and the plan or solution can be constructed by looking up the table. There is a need for search algorithms to be able to operate at different levels of abstraction. *STRIPS* like systems also use the *MEA* strategy, but are compelled to work with the primitive operators from the word go.

One of the first systems to adopt a hierarchical approach to planning was *ABSTRIPS* (Sacerdoti, 1974). As the name suggests, *ABSTRIPS* is an extension or abstraction of *STRIPS* and works on the same problem representations. It, however, uses some additional information to search in a different manner. The additional information it uses is a partial ordering of the predicates in the domain, determined by a *criticality value* assigned to each predicate. This criticality value is very similar to the ordering of differences in *GPS*. *ABSTRIPS* start off by ignoring all but the most critical predicates and searches for a solution, with only the most critical predicates in the preconditions. During this process, it ignores all other preconditions of the operators. Having found a plan, it moves to the next level of criticality. This is done by looking at the predicate with the next level of criticality, and posing subproblems to achieve them wherever they become visible in the top level plan. The solutions for these subproblems are inserted into the top level plan, and the algorithm moves on to the next level.

The hierarchy of abstraction in *ABSTRIPS* is generated by ignoring preconditions in the decreasing order of criticality. Another way a hierarchy of

actions can be generated is by considering the effects of groups of actions, somewhat like the macros described earlier, but treated as operators. This would enable the problem solver to operate with higher level tasks. Thus, planning a vacation trip might have a high level solution like (book-tickets, go-to-railway-station, travel-to-destination, look-for-hotel, check-in-to-hotel, hike-for-three-days, return) at different levels of abstraction[14]. This solution will in turn have to be expanded into sequence of actions at the primitive action level. It is the set of primitive actions that can be carried out in the domain. Thus, high level operators are like instructions in high level programming languages that have to be interpreted by sequences of the primitive instructions provided by the machine hardware. One of the first systems to look at planning in such a hierarchical space was called *NOAH* (Networks of Action Hierarchies), also developed by Sacerdoti (Sacerdoti, 1977). *NOAH* represents plans as *task networks*. The task networks are partial orders, but are not partial plans. Instead, they represent complete plans at a high level of abstraction. *NOAH* solves a problem by successively refining parts of the task network, till the network has only primitive tasks that can be done by domain operators. After every refinement step, *NOAH* passes control to a set of routines called *critics*. Each critic inspects the partial plan for a particular defect and, if found, suggests a remedial action.

We look at how *NOAH* would solve the Sussman's Anomaly. *NOAH* starts off like *GPS* by addressing the problem at a high level and proceeds to work in the details, by refining a part of the task network. The initial task network contains a single node. The given task is to achieve the goal "*on*(*A*, *B*) $\wedge$ *on* (*B*, *C*)". Like *STRIPS*, *NOAH* too decomposes this task into two subtasks for each of the subgoals, but unlike *STRIPS*, it does not impose an order on the two subtasks. Instead, as shown in Figure 7.25, it creates a network by adding two special action nodes called *split* and *join*, to form a partial order. This is the *least commitment strategy* in which the planner delays commitment to decisions, until it can make an informed choice. The plan at this level is to split the task into two subtasks, then solve each of them in some order, and finally join the results of the two subplans. In the next two steps, *NOAH* refines the two tasks, as shown in Figure 7.26 below. It replaces, or expands, each of the nodes into networks of lower level actions. The task "*Achieve on*(*A*, *B*)" can be accomplished by clearing blocks *A* and *B*, in some order, and then "putting" *A* on top of *B*. The higher level action "Put" can be expanded like a macro into *pickup* or *unstack*, as the case may be, followed by the *stack* operator.

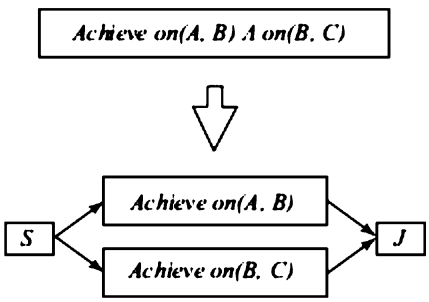


**Figure 7.25** *NOAH* starts off by decomposing the higher level task into two tasks without committing to an order. The two tasks are linked by a Split node *S* and a Join node *J*.

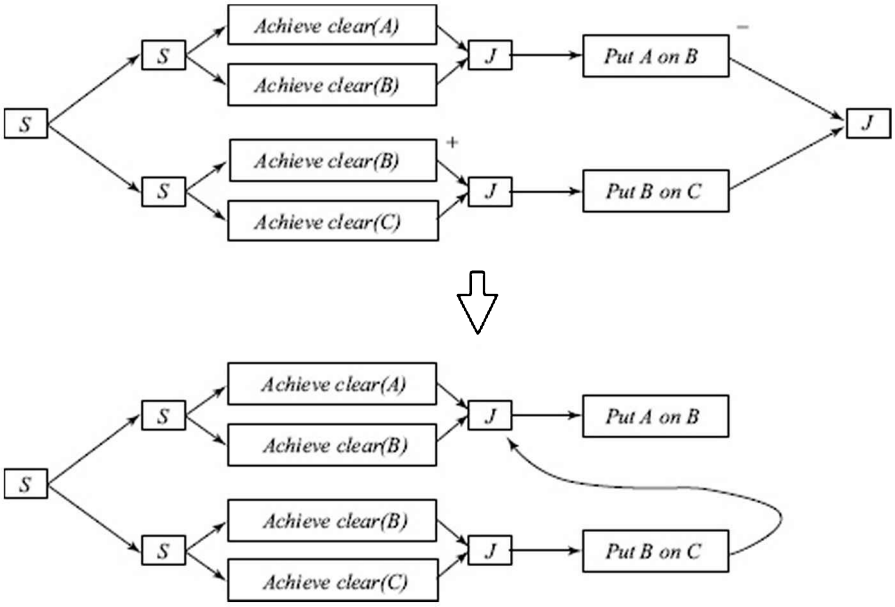



**Figure 7.26** *NOAH* refines each of the two tasks and hands over control to the Critics. Critic: Resolve-Conflicts notices the conflict marked "+" and "−;" on condition clear(B). Suggests that the action labelled "+" be promoted before action labelled "−" to resolve the conflict (threat).

Having expanded both the tasks, it calls in the critics. Critic: *Resolve-Conflicts* notices the conflict marked "+" and "−" on condition *clear*(*B*). It suggests that the action labelled "+" be promoted before action labelled "−" to resolve the conflict (threat), as shown in Figure 7.26. At this point the Critic: *Elminate-Redundand-Preconditions* observes that the condition marked "+" in Figure 7.27 is redundant, and suggests removal of one of the two instances of the task. The figure shows this transformation.

Once the critics have done their job, *NOAH* goes back to refinement. It sees that it can clear block *A* by removing block *C* from it and expands that node. Again, the critic *Resolve-Conflicts* notices the conflict (threat) marked "+" and "−" on condition *clear*(*C*). It suggests that the action labelled "+" be promoted, and the network is now as shown in Figure 7.28.

Now the critic *Eliminate-Redundant-Preconditions* observes that *clear*(*C*) marked "+" is being achieved twice, and suggests that one of the redundant nodes be removed. The resulting network is displayed in Figure 7.29.

The reader would have noticed that by now, *NOAH* has computed the best ordering of its subtasks, and all that remains to be done is to expand the "Put" actions into the lower level domain actions.

## 7.5.3 Hierarchical Planning

In hierarchical planning, high level planning operators are refined into low level ones. The systems described above illustrate various features one

would like in an hierarchical planner. The planner should be able to reason about the problem with high-level operators and then drill down into the details. Like *ABSTRIPS*, it should be able to first focus of critical predicates and ignore the others. Like *NOAH*, it should be able to refine the high level task into networks of low level operators, with more detailed preconditions. In general, the networks should be partial orders containing only the necessary ordering information, to enable the planner to deploy the least commitment ordering strategy demonstrated by plan space planners and by *NOAH*. And like Korf's macro operators, the final plans must be composed by putting together primitive actions from the domain.

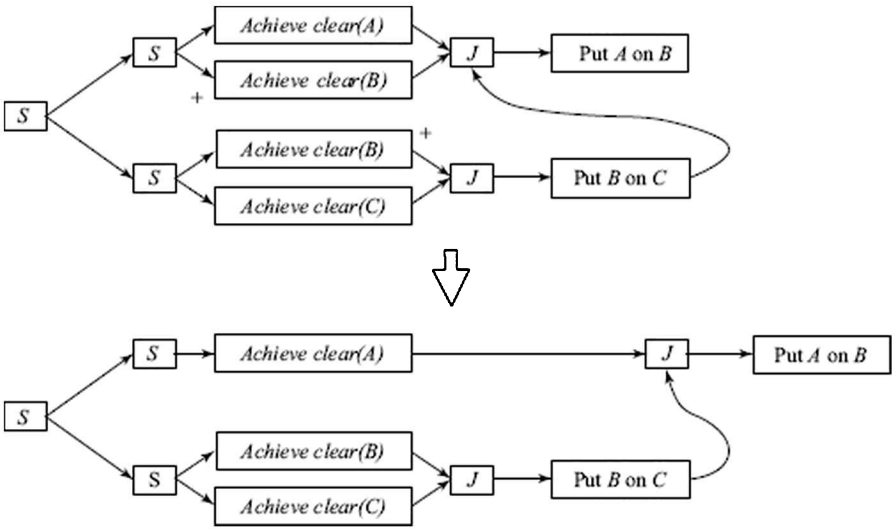



**Figure 7.27** Critic: Elminate-Redundant-Preconditions observes that condition marked "+" is redundant. Suggests removal of one task.

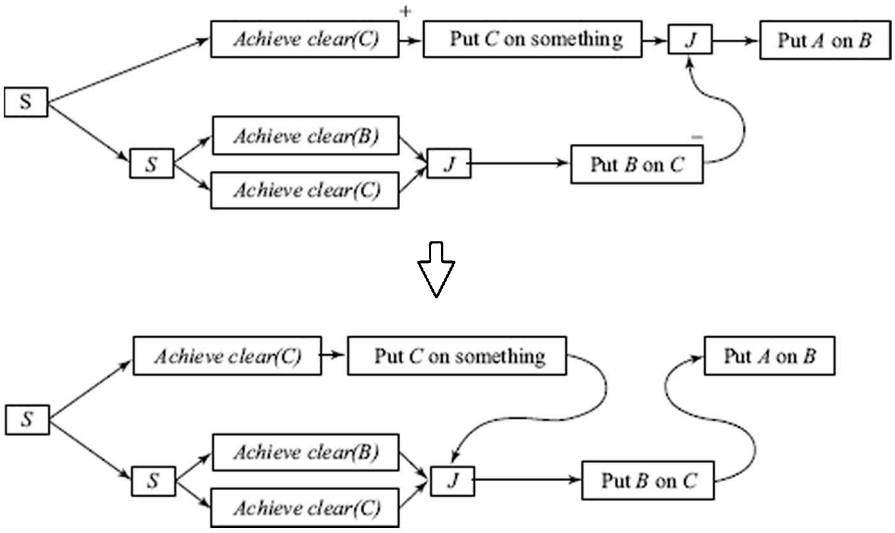


**Figure 7.28** *Achieve clear*(A) is refined to moving *C* onto something after achieving *clear(C)*. Critic: Resolve-Conflicts notices the conflict marked "+" and "–" on condition *clear(C)*. Suggests that the action labelled + be promoted.

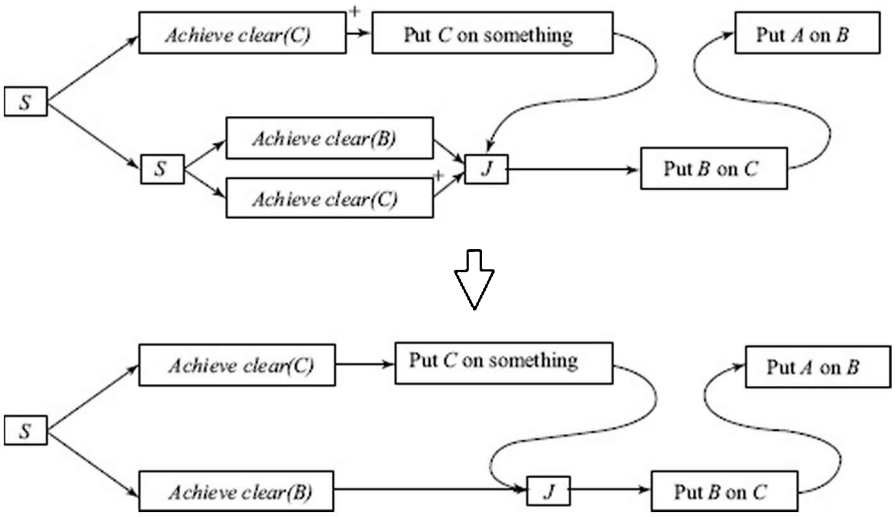



**Figure 7.29** Critic: Elminate-Redundant-Preconditions observes that condition marked + is redundant. Suggests removal of one task. The high level operator "Put *X* on *Y*" can be further decomposed into *STRIPS* actions. Blocks *B* and *C* need no actions to clear them. But by now, *NOAH* has found the correct order for the optimal plan.

The schematic diagram of the expansion of a high level operator is shown in Figure 7.30. The operator has its *own* set of preconditions and effects, shown with thick dashed arrows. The effects may be thought of as desired effects, because after refinement, the low level operators may also have other effects, shown with thin dashed arrows, which we can call 'side effects'. Like *ABSTRIPS*, searching and putting together a high level plan based on the preconditions and effects of the high level operator may be considered. But after expansion, other preconditions and effects may come into play. The preconditions may have to be satisfied, and the post-conditions may have to be inspected for threats and redundant conditions.

Similar approaches to hierarchical task network (*HTN*) planning have been reported in the literature (Wilkins, 1988; Erol, 1994; Tate, 1994). In a system called *SHOP2* (Simple Hierarchical Ordered Planner version 2) (Nau, 2003), the high level actions are called *tasks* and the decomposition of operators is done by *methods.* A *SHOP2* method constitutes of a name, the name of the task it decomposes, a set of subtasks that the method generates, and constraints between the subtasks. Planning involves a combination of method selection and task decomposition, until one is left only with primitive tasks and the task network generated is consistent.

The advantages of using hierarchical planning methods are that in principle, they allow the planner to start at a high level and work out the details selectively. Secondly, they make it possible for the users to feed in problem solving knowledge in the form of the high level operators, and thus exploit human generated knowledge, in addition to the learning of macros proposed by Korf. The last point, however, is a double edged feature,

because to exploit the features of *HTN* planning, one has to be able to devise appropriate high level operators, and algorithms to efficiently interleave the tasks of search, decomposition and constraint reconciliation, specially in domains where the lower level operators may have to be interleaved. But given the fact that planning by search at the level of domain level operators is hard, hierarchical planning is an approach where the combinatorial explosion can be contained. However, to do so, one has to be able to represent the problem and the operators at different levels of abstraction, and also to establish the relations between them.

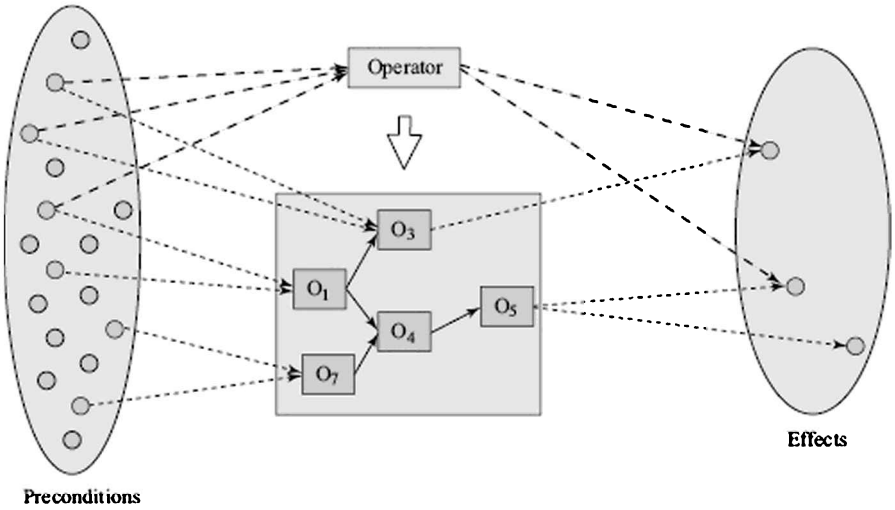



**Figure 7.30** Schematic of a high level operator. The operator has its own preconditions and effects, and is refined to a partially ordered network of lower level operators.

This is essentially a problem of knowledge representation, and therefore *HTN* planners have a strong knowledge based flavour. Such knowledge can effectively compress the search space, as we saw in the Eight-puzzle example using macro operators. But in doing so, they may lose the generality and completeness of search. One of the challenges in the domain of planning is to combine the domain compression achievable through the deployment of knowledge with the flexibility and completeness of search. Ideally, a system should be able to exploit knowledge wherever it can, and fall back on search where it cannot. One can then think of complete planning systems that can become more and more efficient, as they have access to more and more knowledge. This knowledge could be generated though approaches to machine learning, but could also be acquired from external sources, like a teacher, or an experienced elder when the planner is in a social setup.

## 7.6 A Unified Framework for Planning

The planners described in this chapter represent themes that were explored up to the mid-nineties in the twentieth century. Most researchers tended to