**UNIT -IV**

**Planning**

**Classical Planning:** Definition of Classical Planning, Algorithms for Planning with State-Space Search, Planning Graphs, other Classical Planning Approaches, Analysis of Planning approaches.

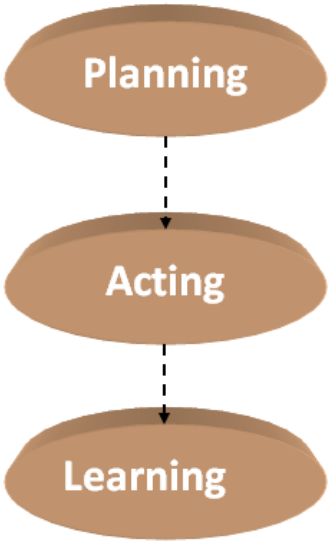
**Planning and Acting in the Real World:** Time, Schedules, and Resources, Hierarchical Planning, Planning and Acting in Nondeterministic Domains, Multi agent Planning.

---

## CLASSICAL PLANNING

Classical Planning is the planning where an agent takes advantage of the problem structure to construct complex plans of an action. The agent performs three tasks in classical planning:

- Planning: The agent plans after knowing what the problem is.
- Acting: It decides what action it has to take.
- Learning: The actions taken by the agent make him learn new things.




A language known as PDDL(Planning Domain Definition Language) which is used to represent all actions into one action schema.

PDLL describes the four basic things needed in a search problem:

- Initial state: It is the representation of each state as the conjunction of the ground and functionless atoms.
- Actions: It is defined by a set of action schemas which implicitly define the ACTION() and RESULT() functions.
- Result: It is obtained by the set of actions used by the agent.
- Goal: It is the same as a precondition, which is a conjunction of literals (whose value is either positive or negative).

There are various examples which will make PDLL understandable:

- Air cargo transport
- The spare tire problem
- The blocks world and many more.

Let's discuss one of them

- Air cargo transport

This problem can be illustrated with the help of the following actions:

- Load: This action is taken to load cargo.
- Unload: This action is taken to unload the cargo when it reaches its destination.
- Fly: This action is taken to fly from one place to another.

Therefore, the Air cargo transport problem is based on loading and unloading the cargo and flying it from one place to another.

Below is the PDLL description for Air cargo transport:

Init (On(C1, SFO) ? On(C2, JFK) ? On(P1, SFO) ? On(P2, JFK)? Cargo(C1) ? Cargo(C2) ? Plane(P1) ?
Plane(P2)
? Airport (JFK) ? Airport (SFO))
Goal (On(C1, JFK) ? On(C2, SFO))
Action(Load (c, p, a),
PRECOND: On(c, a) ? On(p, a) ? Cargo(c) ? Plane(p) ? Airport (a)
EFFECT: ? On(c, a) ? In(c, p))
Action(Unload(c, p, a),
PRECOND: In(c, p) ? On(p, a) ? Cargo(c) ? Plane(p) ? Airport (a)
EFFECT: On(c, a) ? ?In(c, p))
Action(Fly(p, from, to),
PRECOND: On(p, from) ? Plane(p) ? Airport (from) ? Airport (to)
EFFECT: ? On(p, from) ? On(p, to))

The above described actions, (i.e., load, unload, and fly) affects the following two predicates:

- (c,p): In this, the cargo is inside the plane p.
- (x,a): In this, the object x is at the airport a. Here, objects can be the cargo or plane.

It is to be noted that when the plane flies from one place to another, it should carry all cargo inside it. It becomes difficult with the PDLL to give solutions for such a problem. Because PDLL does not have a universal quantifier. Thus, the following approach is used:

- a piece of cargo ceases to beOn anywhere when it is In a plane.
- The cargo only becomesOn the new airport when it is unloaded.

Therefore, the planning for the solution is:

Load (C1, P1, SFO), Fly(P1, SFO, JFK),Unload(C1, P1, JFK),
Load (C2, P2, JFK), Fly(P2, JFK, SFO),Unload(C2, P2, SFO)] .

Note: Some problems can be ignored because they do not cause any problem in planning.

- The spare tire problem

The problem is that the agent needs to change the flat tire. The aim is to place a good spare tire over the car's axle. There are four actions used to define the spare tire problem:

1. Remove the spare from the trunk.
2. Remove the flat spare from the axle.
3. Putting the spare on the axle.
4. Leave the car unattended overnight. Assuming that the car is parked in an unsafe neighborhood.

The PDLL description for the spare tire problem is:

Init(Tire1(Flat ) ? Tire1(Spare) ? At(Flat , Axle) ? At(Spare, Trunk ))

Goal (At(Spare, Axle))
Action(Remove(obj , loc),
PRECOND: At(obj , loc)
EFFECT: ? At(obj , loc) ? At(obj , Ground))
Action(PutOn(t , Axle),
PRECOND: Tire1(t) ? At(t , Ground) ?¬At(Flat , Axle)
EFFECT: ? At(t , Ground) ? At(t , Axle))
Action(LeaveOvernight ,
PRECOND:
EFFECT: ? At(Spare, Ground) ?¬At(Spare, Axle) ?¬At(Spare, Trunk)
?¬At(Flat, Ground) ?¬At(Flat , Axle) ?¬At(Flat, Trunk))
The solution to the problem is:
[Remove(Flat,Axle),Remove(Spare,Trunk), PutOn(Spare, Axle)].

Similarly, we can design PDLL for various problems.

**Complexity of the classical planning**

In classical planning, there occur following two decision problems:

1. PlanSAT: It is the question asking if there exists any plan that solves a planning problem.
2. Bounded PlanSAT: It is the question asking if there is a solution of length k or less than it.

We found that:

- PlanSAT and Bounded PlanSAT are decidable for classical planning.
- Both decision problems lie in the complexity class PSPACE, which is larger than NP.

Note: PSPACE is the class which refers to those problems that can be solved via a deterministic Turing machine under a polynomial time space.

From the above, it can be concluded that:

1. PlanSAT is P whereas Bounded PlanSAT is NP-complete.
2. Optimal planning is hard with respect to sub-optimal planning.

**Advantages of Classical Planning**

There are following advantages of Classical planning:

- It has provided the facility to develop accurate domain-independent heuristics.
- The systems are easy to understand and work efficiently.

**ALGORITHMS FOR PLANNING WITH STATE-SPACE SEARCH**

In artificial intelligence, a process known as state space search is used to explore all potential configurations or states of an instance until one with the necessary feature is found. A state is a time snapshot representing some aspect of the problem.

The following points highlight the two main planning methods used to solve AI problems. The methods are:

1. Planning with State-Space Search
2. Goal Stack Planning.

**Method # 1. Planning with State-Space Search:**

The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: forward from the initial state or backward from the goal, as shown in Fig. 8.5. We can also use the explicit action and goal representations to derive effective heuristics automatically.

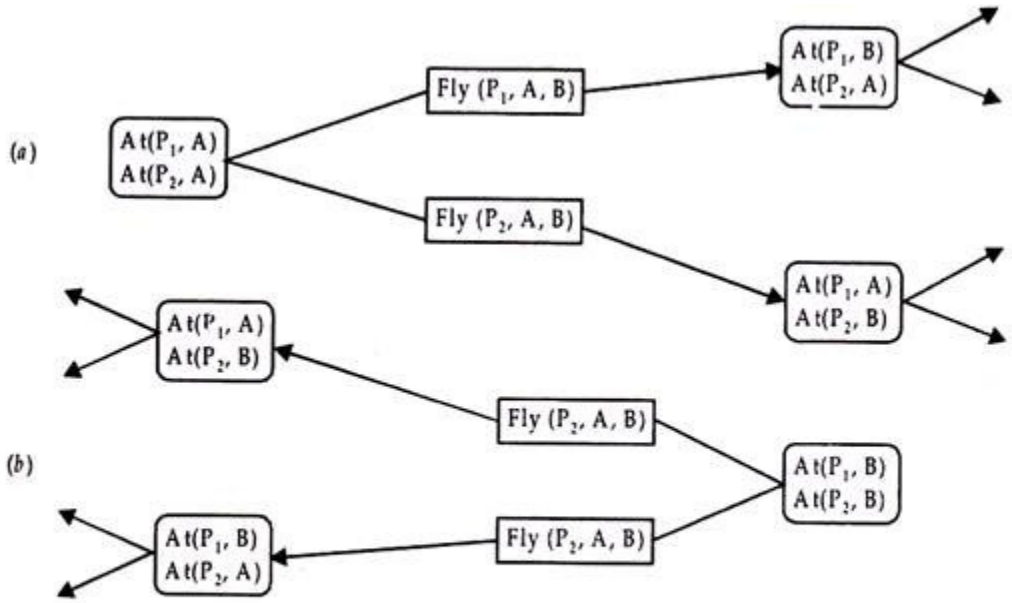



Fig. 8.5. *Two approaches to searching for a plan, (a) Forward (Progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state, (b) Backward (regression) state-space search: a belief-state search starting at the goal statc(s) and using the inverse of the actions to search backward for the initial state.*

**i. Forward State-Space Search:**

Planning with forward state-space search is similar to the problem-solving approach. It is sometimes called progression planning, because it moves in the forward direction.

We start with the problem's initial state, considering sequences of actions until we reach a goal state.

The formulation of planning problem as state-space search problems is as follows:

i. The initial state of the search is the initial state from the planning problem. In general each state will be a set of positive ground literals; literals not appearing are false.

ii. The actions which are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals.

iii. The goal test checks whether the state satisfies the goal of the planning problem.

iv. The step cost of each action is typically 1. Although it would be easy to allow different costs for different actions, this was seldom done by STRIPS planners.

Since function symbols are not present, the state space of a planning problem is finite and therefore, any graph search algorithm such as A * will be a complete planning algorithm.

From the early days of planning research it is known that forward state-space search is too inefficient to be practical. Mainly, this is because of a big branching factor since forward search does not address only relevant actions, (all applicable actions are considered). Consider for example, an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo.

The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded).

On average, let's say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about 1000 nodes. It is thus clear that a very accurate heuristic will be needed to make this kind of search efficient.

**ii. Backward State-Space Search:**

Backward search can be difficult to implement when the goal states are described by a set of constraints which are not listed explicitly. In particular, it is not always obvious how to generate a description of the possible predecessors of the set of goal states. The STRIPS representation makes this quite easy because sets of states can be described by the literals which must be true in those states.
The main advantage of backward search is that it allows us to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely.

$$\text{At}(C_1\, B) \wedge \text{At}(C_2\, B) \ldots\ldots\ldots\ldots\ldots\ldots \text{At}(C_{20}, B)$$

Now consider the conjunct At ($C_1$, B). Working backwards, we can seek those actions which have this as an effect,
There is only one:

$$\text{Unload}(C_1 p, B),$$

where plane p is unspecified.

We may note that there are many irrelevant actions which can also lead to a goal state. For example, we can fly an empty plane from Mumbai to Chennai; this action reaches a goal state from a predecessor state in which the plane is at Mumbai and all the goal conjuncts are satisfied. A backward search which allows irrelevant actions will still be complete, but it will be much less efficient. If a solution exists, it should be found by a backward search which allows only relevant action.

This restriction to relevant actions only means that backward search often has a much lower branching factor than forward search. For example, our air cargo problem has about 1000 actions leading forward from the initial state, but only 20 actions working backward from the goal. Hence backward search is more efficient than forward searching.

Searching backwards is also called regression planning. The principal question in regression planning is: what are the states from which applying a given action leads to the goal? Computing the description of these states is called regressing the goal through the action. To see how it works, once again consider the air cargo example.

We have the goal

$$\text{At}(C_1\, B) \wedge \text{At}(C_2\, B) \wedge \ldots \wedge \text{At}(C_{20}, B)$$

The relevant action UNLOAD ($C_1$ ‘ p, B) achieves the first conjunct. The action will work only if its preconditions are satisfied. Therefore, any predecessor state must include these preconditions: In ($C_1$, p) ∧ At (p, B) as sub-goals. Moreover, the sub-goal At ($C_1$, B) should not be true in the predecessor state which will no doubt be a goal but not relevant one (justify).

Thus, the predecessor description is:

$$In(C_1, p) \wedge At(p, B) \wedge At(C_2, B) \wedge \ldots\ldots\ldots\ldots\ldots\ldots \wedge At(C_{20}, B)$$

In addition to insisting that actions achieve some desired literal, we must insist that the actions do not undo any desired literals. An action which satisfies this restriction is called consistent. For example, the action load ($C_2$, p) would not be consistent with the current goal, because it would negate the literal At ($C_2$, B) (verify).

Given definitions of relevance and consistency, we can now describe the general process of constructing predecessors for backward search. Given a goal description G, let A be an action which is relevant and consistent.

The corresponding predecessor is constructed as follows:
I. Any positive effects of A which appear in G are deleted.
II. Each precondition literal of A is added, unless it already appears.

Any of the standard search algorithms can be used to carry out the search. Termination occurs when a predecessor description is generated which is satisfied by the initial state of the planning problem. In first-order logic, satisfaction might require a substitution for variables in the predecessor description. For example, the predecessor description in the preceding paragraph is satisfied by the initial state.

$$In(C_1, P_{12}) \wedge At(P_{12}, B) \wedge At(C_2, B) \wedge \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge At(C_{20}, B)$$

with substitution ($P/P_{12}$). The substitution must be applied to the action leading from the state to the goal, producing the solution

$$[Unload(C_1, P_{12}, B)]$$

iii. Heuristics for State-Space Search:
It turns out that neither forward nor backward search is efficient without a good heuristic function. Let us recall that in a search technique a heuristic function estimates the distance from a state to the goal. In STRIPS planning, the cost of each action is 1, so the distance is the number of actions.

The basic idea is to look at the effects of the actions and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates, most of the time without too much computation. We might also be able to derive an admissible heuristic- (one which does not overestimate). This could be achieved with A * search.

There are two approaches which can be tried. The first is to derive a relaxed problem from the given problem specification. The optimal solution cost for the relaxed problem — which is very easy to solve-gives an admissible heuristic for the admissible original problem. A problem with fewer restrictions on the actions is called a relaxed problem.

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem. The heuristic is admissible because the optimal solution in the original problem, is by definition, also a solution in

the relaxed problem and therefore must be at least as expensive as the optimal solution in the relaxed problem. The second approach is to pretend that a pure divide-and- conquer algorithm will work.

This is called the sub-goal independence assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each sub-goal independently. The sub-goal independence assumption can be optimistic or pessimistic.

It is optimistic when there are negative interactions between the sub-plans for each sub-goal-for example, when an action in one sub-plan deletes a goal achieved by another sub-plan. It is pessimistic, and therefore inadmissible, when sub-plans contain redundant actions-for instance, two actions that could be replaced by a single action in the merged plan.

So let us consider how to derive relaxed planning problems. Since explicit representations of preconditions and effects are available (compared with ordinary search space where the successor states are not known) the process will work by modifying those representations.

The simplest idea is to relax the problem by removing all preconditions from the actions. Then every action will always be applicable, and any literal can be achieved in one step, if there is an applicable action (goal is impossible if action is not applicable).

It implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals – almost but not quite, because:
(1) There may be two actions each which of deletes the goal literal achieved by the other, and
(2) Some actions may achieve multiple goals.

If we combine our relaxed problem with the sub-goal independence assumption, both of these issues are assumed and the resulting heuristic is exactly the number of unsatisfied goals.
In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions which achieve multiple goals. First, we relax the problem further by removing negative effects. Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal.
For example, consider
Goal ($A \wedge B \wedge C$)
Action (X Effect: A A P)
Action (Y, Effect: B A C A Q)
Action (Z, Effect: B A P A Q)

The minimal set cover of goal {A, B, C} is given by actions {X, Y}, so the set cover heuristic returns a cost of 2. This improves on the sub-goal independence assumption, which gives a heuristic value of 3. There is one minor irritation: the set cover problem is NP hard. A simple greedy-set cover algorithm is guaranteed to return a value which is within a factor of log n of the true minimum value, where n is the number of literals in the goal, and usually works much better than this. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect A ∧ ¬ B in the original problem, it will have the effect A in the relaxed problem. This means that we need not worry about negative interactions between sub-plans, because no action can delete the literals achieved by another action.

The solution cost of the resulting relaxed problem gives what is called the empty-delete-list heuristic. This heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm. In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.

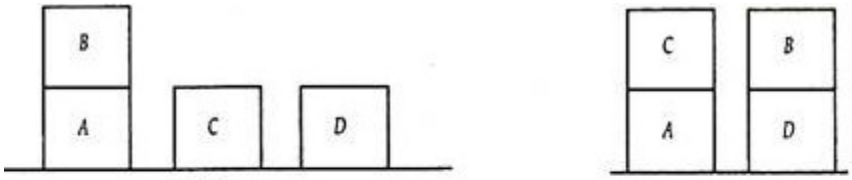
The heuristics described here can be used in either the progression or the regression direction. At present, progression planners using the empty-delete-list heuristic hold the lead. But it is likely to change as new heuristics and new search techniques are being explored. Since planning is exponentially hard, no algorithm will be efficient for all problems, but mostly practical problems can be solved with the heuristic method.

Method # 2. Goal Stack Planning:

This was perhaps the first method used to solve the problems in which the goal interacted and was the approach used by STRIPS. The planner used a single stack which contains both goals and operators which are proposed to satisfy those goals.

It also depends on a database which describes the current situation and a set of operators described as PRECONDITION, ADD, and DELETE lists. Let us illustrate the working of this method with the help of an example of a block world, shown in Fig. 8.6. At the start of the solution, the goal stack is simple.

ON(C, A)
ON(B, D)
ONTABLE(A)
ONTABLE(D)
ONTABLE(C)




ON(B, A) ∧
ON TABLE (A) ∧
ON TABLE (C) ∧
ARMEMPTY

ON(C, A) ∧
ON (B, D) ∧
ON TABLE (A) ∧
ON TABLE (D)

**Fig. 8.6.** *A simple blocks world example.*

But we want to separate this problem into four sub-problems, one for each component of the original goal. Two of the sub-problems ON TABLE (A) and ON TABLE (D) are already true in the initial state. So we need to work on the remaining two. There are two goal stacks depending on the order of tackling the sub problems.

| | |
|---|---|
| *ON (C, A)* | *ON (B, D)* |
| *ON(B, D)* | *ON(C, A)* |
| *[ON(C, A) ^ ON(B, D) ^ OTAD]* | *[ON (C, A) ^ ON (B, D) ^ OTAD]* |

where OTDA is the abbreviation for ONTABLE (A) ∧ ONTABLE (D). Single line below the operator represents the goal.

Let us recapitulate the process of finding the goal in STRIPS. In each succeeding step of the planner, the top goal on the stack is pursued. When a sequence of operators which satisfied the goal is found, that sequence is applied to the state description, yielding a new description. Next the goal which is then at the top of the stack is explored and an attempt is made to satisfy it, starting from the situation which was produced as a result of satisfying the first goal.

This process continues until the goal stack is empty. Towards the last check, the original goal is compared to the final state derived from the application of the chosen operators. If any components of the goal are not satisfied, maybe they were satisfied at one time but went unsatisfying during the course of a subsequent step, then those unresolved parts of the goal are reinserted onto the stack and the process resumes.

Let us continue with the example and explore the first alternative. Since ON(C, A) does not hold in the current (initial) stale, we check for the operators which could cause it to be true — only one operator STACK(C, A) out of the four defined in the block-world example (Fig. 8.3), could cause this.

*STACK(x, y)*

| | | |
|---|---|---|
| *P* | : | CLEAR(*y*) ^ HOLDING(*x*) |
| *D* | : | CLEAR(*y*) ^ HOLDING(*x*) |
| *A* | : | ARMEMPTY ^ ON(*x*, *y*) |

*UNSTACK (x, y) :*

| | | |
|---|---|---|
| *P* | : | ON(*x*, *y*) ^ CLEAR(*x*) ^ ARMEMPTY |
| *D* | : | ON(*x*, *y*) ^ ARMEMPTY |
| *A* | : | HOLDING(*x*) ^ CLEAR(*y*) |

*PICKUP(x)*

| | | |
|---|---|---|
| *P* | : | CLEAR(*x*) ^ ONTABLE(*x*) ^ ARMEMPTY |
| *D* | : | ONTABLE(*x*) ARMEMPTY |
| *A* | : | HOLDING(*x*) |

*PUTDOWN(x)*

| | | |
|---|---|---|
| *P* | : | HOLDING(*x*) |
| *D* | : | HOLDING(*x*) |
| *A* | : | ONTABLE(*x*) ^ ARMEMPTY |

**Fig. 8.3.** *STRIPS.Style operators for the block world.*

Hence ON(C, A) is replaced by STACK(C, A); yielding:

STACK(C, A)
ON (B, D)
ON (C, A) ON (C, A) ∧ ON (B, D) ∧ OTAD

But in order to apply STACK, (C, A) its preconditions must hold, which now become sub-goals. The new compound sub-goal CLEAR (A) ∧ HOLDING(C) must be broken into components and decide on the order. Help is sought from some heuristic.

HOLDING(x) is very easy to achieve: put down something else and then pickup the desired object. In order to do anything else, the robot will need to use the arm. So if we achieve HOLDING first and then try to do something else the robot will have to use its arm. So if we achieve HOLDING first and then try to do something else, will imply that HOLDING is no longer true towards the end.

So the heuristic used is:
If HOLDING is one of several goals to be achieved at once, it should be tackled last, the other sub goal, CLEAR (A) should be tackled first.
So the new goal stack becomes:

CLEAR (A)
HOLDING(C)
CLEAR (A) ⋀ HOLDING(C)
STACK(C, A)
ON (B, D) ∧ ON (C, A) ∧ ON (B, D) ∧ OTAD.

This kind of heuristic information could be contained in the precondition list itself by stating the predicates in the order in which they should be achieved.

Next, whether CLEAR (A) true or not, is checked. It is not. The only operator UNSTACK (B, A) makes it true. Its preconditions form, the sub-goals, so the new goal stack becomes:

ON (B, A)
CLEAR (B)
ARMEMPTY
ON (B, A) ∧ CLEAR (B) ∧ ARMEMPTY
UNSTACK (B, A)
HOLDING(C)
CLEAR (A) ∧ HOLDING(C)
STACK (C, A)
ON (B, D)
ON (C, A) ∧ ON (B, D) ∧ OTAD.

Now on comparing the top element of the goal stack ON (B, A) to the block world problem initial state it is found is satisfied. So it is popped off and the next goal CLEAR (B) considered. It is also already true (How, it is left as an exercise). So this goal can also be popped from the stack. The third pre-condition for UNSTACK (B, A) – ARMEMPTY also holds good; hence can be popped off the stack.

The next element of the stack is the combined goal representing all of the preconditions for the UNSTACK (B, A). It is also satisfied, so it can also be popped off the stack. Now the top element of the stack is the operator UNSTACK (B, A).

Since its preconditions are satisfied, it can be applied to produce a new world model from which the rest of the problem solving process can continue. This is done by using ADD and DELETE lists specified for UNSTACK. We also note that UNSTACK (B, A) is the first operator of the proposed solution sequence.
Now the database corresponding to blocks world model is:

ONTABLE (A) ∧ ONTABLE(C) ∧ ONTABLE (D) ∧ HOLDING (B) ∧ CLEAR (A).

The goal stack now is:

HOLDING(C)

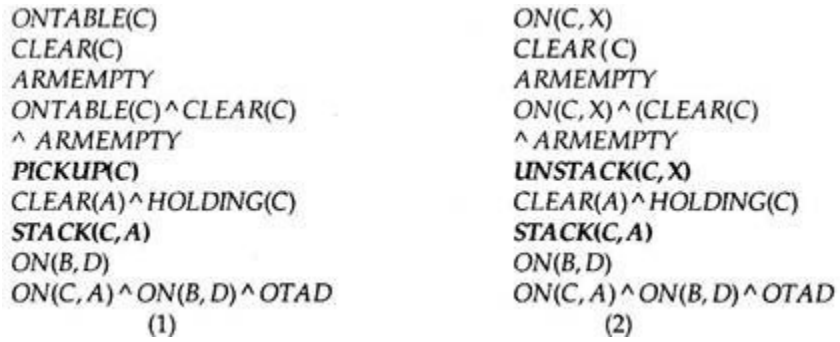
CLEAR (A) ∧ HOLDING(C)

STACK(C, A)

ON (B, D)

ON(C, A) ∧ ON (B, D) ∧ OTAD

The next goal to be satisfied is HOLDING(C) and this is made true by two operators PICK UP(C) and UNSTACK(c, x), where x could be any block from which the block c could be unstacked.
Using those two operators the two branches of the search tree are:




(1) (2) In stack (2) a variable x, appears at three places. Though, any block could be substituted for x but it is important that the same block be matched to x at all these places. So it is important that each time a variable is introduced into the goal stack, it be given a name distinct from any other variables already in the stack. For this purpose, whenever a candidate object is chosen to match a variable the bindings must be recorded so that any future bindings of the same variable be done to the same object.

Now the question is which of the two alternatives will be selected?
Alternative 1 is better than the alternative 2, because block C is not on anything so pickup(c) is better than un-stacking it; because in order to un-stack it has first to be stacked with some block. Of course this could be done but this would be a fruitless effort. But how could a program know that? By chance if alternative 2 is tried then in order to satisfy ON(c, x) (pre condition of operator UNSTACK(X, Y) we would have to STACK C onto some block x.
The goal stack would then become:

CLEAR (x)

HOLDING(c)

CLEAR(x) ∧ HOLDING(c)

STACK(c, x)

CLEAR(c)

ARMEMPTY

ON(c, x) ∧ CLEAR(c) ∧ ARMEMPTY

UNSTACK(c, x)

CLEAR (A) ∧ HOLDING(c)

STACK(C, A)

ON (B, D)

ON(C, A) ∧ ON(B, D) ∧ OTAD.

One of the pre-conditions of STACK is HOLDING(c). This is what we were trying to achieve by applying UNSTACK, which required us to apply STACK so that the condition ON(c, x) would be satisfied, that is we are back to our original goal. In addition, there are other goals (such as ARMEMPTY). So with type of planning this alternative is unproductive, so need be terminated. If block C had been on another block in the current state, ON(c, x) would have been satisfied immediately with no need to do a STACK and this move would have led to a good solution.

So, let us now consider the alternative 1. This uses PICKUP to get the arm holding C. The top element on the goal stack is ONTABLE(C), which is already satisfied so it is popped off. The next element is CLEAR(C), which is also satisfied, so also popped off. The third precondition of PICKUP(C), ARMEMPTY is not satisfied because the arm is picking up B, and HOLDING (B) holds good.

This condition becomes true through the two operators STACK (B, x) and PUTDOWN (B) , that is B can be either put on a table or on another block. Which alternative should be selected? Since the goal state contains ON (B, D), B can be put on D, so we choose to apply STACK (B, D); by binding D to x.
Now the goal stack becomes:

CLEAR (D)
HOLDING (B)
CLEAR (D) ∧ HOLDING (B)
STACK (B, D)
ONTABLE(C) ∧ CLEAR(C) ∧ ARMEMPTY PICKUP(C)
CLEAR (A) ∧ HOLDING(C)
STACK (C, A)
ON (B, D)
ON(C, A) ∧ ON (B, D) ∧ OTAD.

CLEAR (D) and HOLDING (B) are both true; the operation STACK (B, D) can be performed, producing the model.

ONTABLE (A) ∧ ONTABLE(C) ∧ ONTABLE (D) ∧ ON (B, D) ∧ ARMEMPTY.

All of the pre-conditions for PICKUP(C) are now satisfied, so it too can be executed. Since all the pre-conditions for STACK(C, A) are true, it also can be executed.
Now, consider the second part of the original goal, ON (B, D). But this has already been satisfied by the operations which were used to satisfy the first sub-goal. The reader should ascertain for himself; that ON (B, D) can be popped off the goal stack.

We then check the combined goal, the last step towards finding the solution:

ON(C, A) ∧ ON(B, D) ∧ ONTABLE(A) ∧ ONTABLE(D)

to make sure that all the four parts hold good, so the problem is solved.

The answer by the planner can be (the order of the operators will be):
i. UNSTACK (B, A)
ii. STACK (B, D)
iii. PICKUP(C)
iv. STACK(C, A)

**PLANNING GRAPHS**

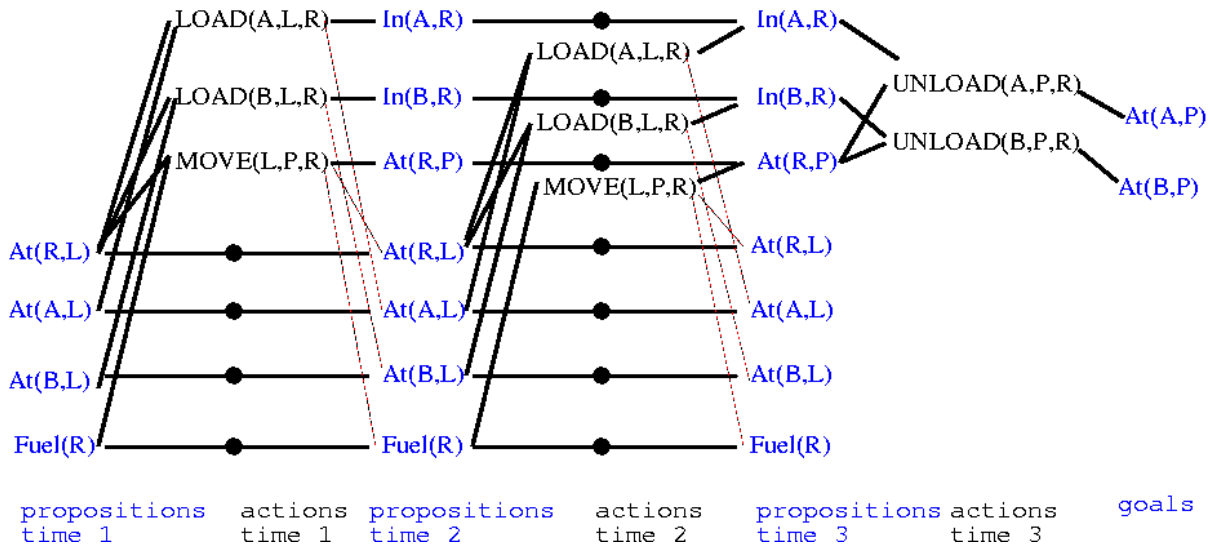
A planning graph is similar to a valid plan, but without the requirement that the actions at a given time step not interfere (i.e., corresponds roughly to conflicts not resolved).

- Directed, leveled graph: that is, the nodes can be partitioned into disjoint sets $L_1, L_2, \ldots, L_n$ such that the edges only connect nodes in adjacent levels.
- Two kinds of nodes: alternate between proposition nodes (labeled by, e.g., On(A,B)) and action nodes (labeled by, e.g., UNSTACK(A,B)):
  1st level: propositions true at time 1 (initial state)
  2nd level: possible actions at time 1
  3rd level: propositions possibly true at time 2
  4th level: possible actions at time 2
  5th level: propositions possibly true at time 3 (and so forth)

Three kinds of edges:

- precondition-edges connect actions in action-level i to their preconditions in proposition level i.
- add-edges connect action nodes in action-level i to their add-effects in proposition level i+1.
- delete-edges connect action nodes in action-level i to their delete-effects in proposition level i+1.

Example - Rocket Domain
One rocket R, two pieces of cargo A and B, start location L, destination P. Delete edges as dashed lines. Explicit no-op actions (with dot). Not all nodes at 2nd and 3rd action levels displayed.




Valid Plans
A plan is called valid:

- Actions at time 1, at time 2, at time 3 etc
- Several actions may occur at the same time so long as they do not interfere with each other, i.e., if one does not delete a precondition or an add-effect of the other. (In a linear plan these independent parallel actions could be arranged in any order with exactly the same outcome.
- A valid plan may perform an action at time t=1 if all its preconditions are true in the initial state.
- A valid plan may perform an action at time t>1 if the plan makes all its preconditions true at time t.

- Because of no-op actions we may define a proposition to be true at time t>1 iff it is an add-effect of some action taken at time t-1.
- A valid plan must make all goals true at the final time step.

Mutual Exclusion

- Two actions at a given action level are mutually exclusive if no valid plan could possibly contain both.
- Two propositions at a given proposition level are mutually exclusive if no valid plan could possibly make both true.

Identify mutual exclusion relationships to reduce the search for a subgraph of a Planning Graph that might correspond to a valid plan.

- Graphplan notices and records mutual exclusion relationships by propagating them through the Planning Graph using a few simple rules.
- The rules do not guarantee to find all mutual exclusion relationships (just typically a large number of them).

Two ways to determine mutual exclusions of actions a and b:

- Interference: If either of the actions deletes a precondition or an add-effect of the other. (standard notion of non-independence; depends only on the operator definitions)
- Competing needs: If there is a precondition of action a and a precondition of action b that are marked as mutually exclusive of each other in the previous proposition level.
- Two propositions p and q in a proposition level are marked as exclusive if all ways of creating proposition p are exclusive of all ways of creating proposition q. Specifically: if each action having an add-edge to proposition p is marked as exclusive of each action b having an add-edge to proposition q.

Example

Rocket domain:

- Initial state: At(Rocket1,London), Fuel(Rocket1)
- The actions
  MOVE(Rocket1,London,Paris) and LOAD(Alex,Rocket1,London)
  are exclusive at time 1 because the first deletes the proposition At(Rocket1,London) which is a precondition of the second
- The propositions At(Rocket1,London) and At(Rocket1,Paris) are exclusive at time 2 because all ways of generating the first (only no-op does) are exclusive of all ways of generating the second (only MOVE does).

Exclusion Relations

- The Competing needs relation and the exclusivity between propositions are not just logical properties of the operators, they also depend on the interplay between operators and the initial state.
- Example: Initially one object is not at two different places. The operators guarantee that this relation remains invariant during the plan construction. [If, however, it were not true initially, it wouldn't necessarily hold later on.]

Description of the Algorithm

Planning_Graph:= Initial_state,
i:= 1, No_Plan_found:= TRUE, Plan_exists:= TRUE
WHILE No_Plan_found and Plan_exists DO
1. i:= i+1

2. Take the Planning_Graph from stage i-1, extend it one time step, that is, add the next action level and the following proposition level.
3. Search for a valid plan of length i in Planning_Graph.

If plan found No_Plan_found:= FALSE, store plan in Plan.

If detected that no plan can exist Plan_exists:= FALSE

END WHILE

IF No_Plan_found=FALSE RETURN Plan ENDIF

IF Plan_exists=FALSE RETURN "No Plan Exists" ENDIF

Properties

- Graphplan's algorithm is sound and complete: only legal plans are generated. If there is a legal plan then Graphplan finds one.
- In each loop i of the algorithm the algorithm either discovers a plan or proves that no plan having i or fewer steps exists.

Extending planning graphs works as follows:

- For each operator and each way of instantiating its preconditions to propositions in the previous level, insert an action node if no two of its preconditions are labeled as mutually exclusive.
- Insert all the no-op actions & insert the precondition edges.
- Check the action nodes for exclusivity and create an "actions-that-I-am-exclusive-of" list for each action.
- Generate proposition level by inserting add-effects.

Searching for a Plan

Given a Planning Graph, search for a valid plan using a backward-chaining strategy.

- Level-by-level approach (to make best use of the mutual exclusion constraints).
- Given a set of goals at time t, it attempts to find a set of actions (no-ops included) at time t-1 having these goals as add effects.
- The preconditions of these actions form a set of subgoals at time t-1 (if these goals can be achieved in t-1, then the original goals can be achieved in t steps).
- If the goal set at time t-1 turns out not to be solvable, try to find a different set of actions (backtracking), continuing until it either succeeds or proves that the original set of goals is not solvable at time t.

Memoization

- When a set of (sub)goals at some time t is determined to be not solvable, then before backtracking Graphplan memoises this fact (goal set and the time t) in a hash table.
- Before searching for plans of a set of subproblems look up the hash table whether it is proved to be unsolvable.

Two aspects:

- Speed up
- Termination check

Minimal Action sets - Goal Orderings

- Let G be a set of goals at time t. A non-exclusive set of actions A at time t-1 is a minimal set of actions achieving G if:
    1. every goal in G is an add-effect of some action in A, and

2. no action can be removed from A so that the add effects of the actions remaining still contain G.

- It suffices to look at minimal action sets.
- Graphplan's strategy is breadth-first like, hence fairly insensitive to goal orderings, in particular when considering only minimal action sets.

Termination on Unsolvable Problems

- If a proposition appears in some proposition level then (because of no-ops) it also appears in all future proposition levels.
- Only a finite set of propositions can be created by Strips-style operators. Hence there is a minimal n such that propositions $P_n$ are equal to $P_{n+i}$ for all i>= 0. The graph is leveled off.
- Let $S^t_i$ the collection of all sets memoised as unsolvable at level i at stage t:
    1. Any plan considered must make one of the goal sets in $S^t_i$ true at time i, and
    2. None of the goal sets in $S^t_i$ is achievable in i steps.
- If a graph has leveled off at some level n and a stage t has passed in which $abs(S^{t-1}_n)=abs(S^t_n)$ then "No Plan Exists."
- Graphplan outputs "No Plan Exists" if and only if the problem is unsolvable.

**OTHER CLASSICAL PLANNING APPROACHES**

Currently the most popular and effective approaches to fully automated planning are:
• Translating to a Boolean satisfiability (SAT) problem
• Forward state-space search with carefully crafted heuristics
• Search using a planning graph

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

**Classical planning as Boolean satisfiability**

In Section 7.7.4 we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:
• Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
• Define the initial state: assert F 0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.
• Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A on another block, $On(A,x) \wedge Block(x)$ in a world with objects A,B and C, would be replaced by the goal $(On(A,A) \wedge Block(A)) \vee (On(A,B) \wedge Block(B)) \vee (On(A,C) \wedge Block(C))$.

• Add successor-state axioms: For each fluent F, add an axiom of the form $F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$ , where ActionCausesF is a disjunction of all the ground actions that have F in their add list, and ActionCausesNotF is a disjunction of all the ground actions that have F in their delete list.
• Add precondition axioms: For each ground action A, add the axiom $A^t \Rightarrow PRE(A)^t$ , that is, if an action is taken at time t, then the preconditions must have been true. • Add action exclusion axioms: say that every action is distinct from every other action. The resulting translation is in the form that we can hand to SATPLAN to find a solution.

**Planning as first-order logical deduction: Situation calculus**

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from A to B regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.
The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, South2 means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching situations, using a representation called situation calculus that works like this:

• The initial state is called a situation. If s is a situation and a is an action, then RESULT(s,a) is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same: $(RESULT(s,a) = RESULT(s',a')) \Leftrightarrow (s = s' \wedge a = a')$. Some examples of actions and situations are shown in Figure 10.12.
• A function or relation that can vary from one situation to the next is fluent. By convention, the situation s is always the last argument to the fluent, for example At(x,l,s) is a relational fluent that is true when object x is at location l in situation s, and Location is a functional fluent such that Location(x,s) = l holds in the same situations as At(x,l,s).
• Each action's preconditions are described with a possibility axiom that says when the action can be taken. It has the form $\Phi(s) \Rightarrow Poss(a,s)$ where $\Phi(s)$ is some formula involving s that describes the preconditions. An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:
$Alive(Agent,s) \wedge Have(Agent, Arrow,s) \Rightarrow Poss(Shoot,s)$

• Each fluent is described with a successor-state axiom that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form
Action is possible ⇒
(Fluent is true in result state ⇔ Action's effect made it true ∨ It was true before and action left it alone)

For example, the axiom for the relational fluent Holding says that the agent is holding some gold g after executing a possible action if and only if the action was a Grab of g or if the agent was already holding g and the action was not releasing it:

Poss(a,s) ⇒ (Holding(Agent,g, Result(a,s)) ⇔ a = Grab(g) ∨ (Holding(Agent,g,s) ∧ a )= Release(g))) .

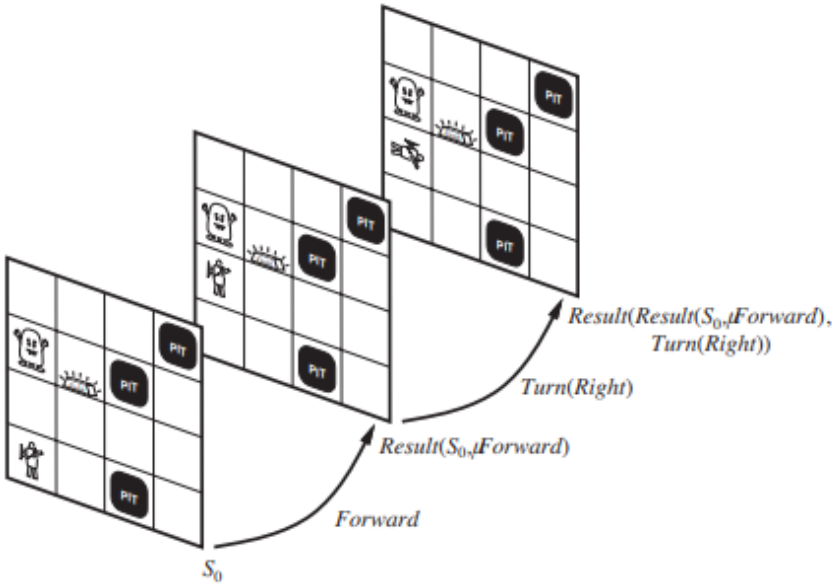



**Figure 10.12** Situations as the results of actions in the wumpus world.

• We need unique action axioms so that the agent can deduce that, for example, a )= Release(g). For each distinct pair of action names Ai and Aj we have an axiom that says the actions are different: Ai(x,...) )= Aj (y,...) and for each action name Ai we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:
Ai(x1,... ,xn)= Ai(y1,... ,yn) ⇔ x1 = y1 ∧ ... ∧ xn = yn .
• A solution is a situation (and hence a sequence of actions) that satisfies the goal.
Work in situation calculus has done a lot to define the formal semantics of planning and to open up new areas of investigation. But so far there have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus.

**Planning as constraint satisfaction**

We have seen that constraint satisfaction has a lot in common with Boolean satisfiability, and we have seen that CSP techniques are effective for scheduling problems, so it is not surprising that it is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem (Section 10.4.1), with one important simplification: at each time step we need only a single variable, Actiont , whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms. It is also possible to encode a planning graph into a CSP. This is the approach taken by GP-CSP (Do and Kambhampati, 2003).

**Planning as refinement of partially ordered plans**

All the approaches we have seen so far construct totally ordered plans consisting of strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently.

An alternative is to represent plans as partially ordered structures: a plan is a set of actions and a set of constraints of the form Before(ai ,aj ) saying that one action occurs before another. In the bottom of Figure 10.13, we see a partially ordered plan that is a solution to the spare tire problem. Actions are boxes and ordering constraints are arrows. Note that Remove(Spare, Trunk) and Remove(Flat, Axle) can be done in either order as long as they are both completed before the PutOn(Spare, Axle) action.

Partially ordered plans are created by a search through the space of plans rather than through the state space. We start with the empty plan consisting of just the initial state and the goal, with no actions in between, as in the top of Figure 10.13. The search procedure then looks for a flaw in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else). A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves At(Spare, Axle). One way to correct the flaw is to insert into the plan

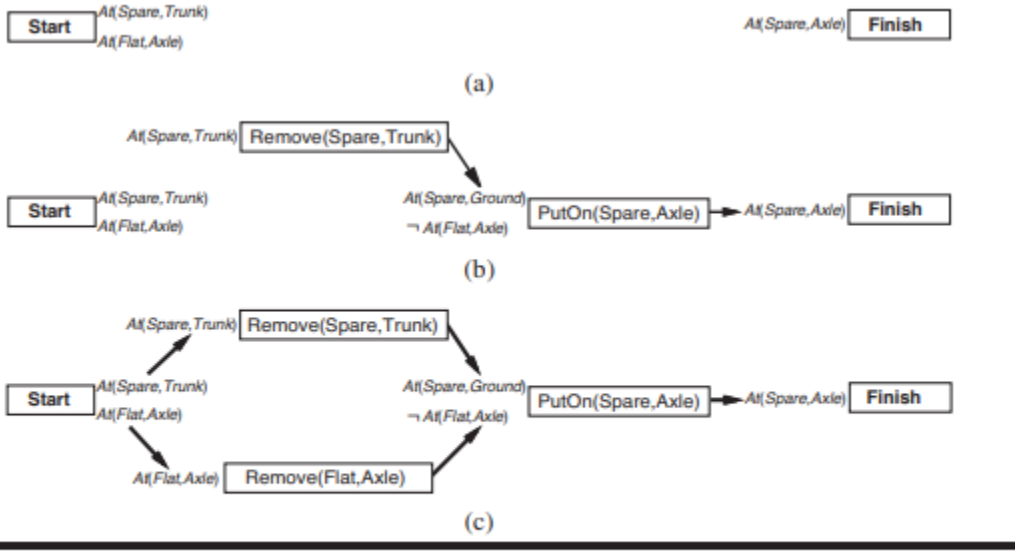



**Figure 10.13** (a) the tire problem expressed as an empty plan. (b) an incomplete partially ordered plan for the tire problem. Boxes represent actions and arrows indicate that one action must occur before another. (c) a complete partially-ordered solution.

the action PutOn(Spare, Axle). Of course that introduces some new flaws: the preconditions of the new action are not achieved. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13. At every step, we make the least commitment possible to fix the flaw. For example, in adding the action Remove(Spare, Trunk) we need to commit to having it occur before PutOn(Spare, Axle), but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of

a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partialorder planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems.

However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

## ANALYSIS OF PLANNING APPROACHES

Planning combines the two major areas of AI we have covered so far: search and logic. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are n propositions in a domain, then there are 2 n states. As we have seen, planning is PSPACEhard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has serializable subgoals if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B, which in turn is on C, which in turn is on the Table, as in Figure 10.4 on page 371), then the subgoals are serializable bottom to top: if we first achieve C on Table, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner that commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is designed by its engineers to be as easy as possible to control

(subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

## PLANNING AND ACTING IN THE REAL WORLD: TIME, SCHEDULES, AND RESOURCES

Planning is considered the logical side of acting. Everything we humans do is with a definite goal in mind, and all our actions are oriented towards achieving our goal. Similarly, Planning is also done for Artificial Intelligence. For example, Planning is required to reach a particular destination.

The classical planning representation talks about what to do, and in what order, but the representation cannot talk about time: how long an action takes and when it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of scheduling. The real world also imposes many resource constraints; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is "plan first, schedule later": that is, we divide the overall problem into a planning phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later scheduling phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

$$
\begin{aligned}
&Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},\\
&\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\})\\
&Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))\\
&Action(AddEngine1, \text{DURATION:}30,\\
&\quad \text{USE:}EngineHoists(1))\\
&Action(AddEngine2, \text{DURATION:}60,\\
&\quad \text{USE:}EngineHoists(1))\\
&Action(AddWheels1, \text{DURATION:}30,\\
&\quad \text{CONSUME:}LugNuts(20), \text{USE:}WheelStations(1))\\
&Action(AddWheels2, \text{DURATION:}15,\\
&\quad \text{CONSUME:}LugNuts(20), \text{USE:}WheelStations(1))\\
&Action(Inspect_i, \text{DURATION:}10,\\
&\quad \text{USE:}Inspectors(1))
\end{aligned}
$$

**Figure 11.1** A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action $A$ must precede action $B$.

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

**Representing temporal and resource constraints**

A typical job-shop scheduling problem, as first introduced in Section 6.1.2, consists of a set of jobs, each of which JOB consists of a collection of actions with ordering constraints among them. Each action has a duration and a set of resource constraints required by the action. Each constraint specifies a type of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is consumable (e.g., the bolts are no longer available for use) or reusable (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be produced by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the makespan.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form [AddEngine,AddWheels, Inspect ]. Then the Resources statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are consumed as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

The representation of resources as numerical quantities, such as Inspectors (2), rather than as named entities, such as Inspector (I1) and Inspector (I2), is an example of a very general technique called aggregation. AGGREGATION The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter which inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent Inspect actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

**Solving scheduling problems**

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the critical path method (CPM) to this graph to determine

the possible start and end times of each action. A path through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with Start and ending with Finish. (For example, there are two paths in the partial-order plan in Figure 11.2.)

The critical path is that path whose total duration is longest; the path is "critical" because it determines the duration of the entire plan—shortening other paths doesn't shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity LS − ES is known as the slack of an action. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the ES and LS times for all the actions constitute a schedule for the problem.

The following formulas serve as a definition for ES and LS and also as the outline of a dynamic-programming algorithm to compute them. A and B are actions, and $A \lessdot B$ means that A comes before B:

$ES(Start) = 0$
$ES(B) = \max_{A \lessdot B} ES(A) + Duration(A)$
$LS(Finish) = ES(Finish)$
$LS(A) = \min_{B} \quad A\ LS(B) - Duration(A)$ .

The idea is that we start by assigning ES(Start ) to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set ES(B) to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the Finish action.

The complexity of the critical path algorithm is just O(Nb), where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a conjunction of linear inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the AddEngine actions, which begin at the same time in Figure 11.2, require the same EngineHoist and so cannot overlap. The "cannot overlap" constraint is a disjunction of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

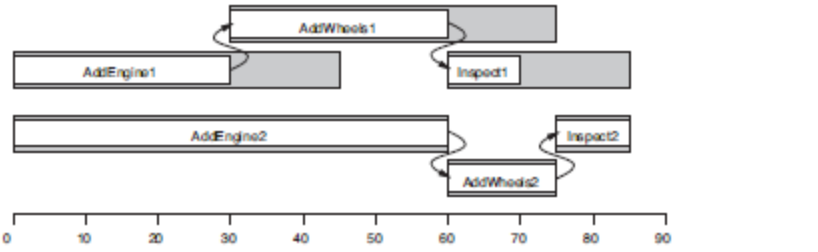




**Figure 11.2** Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

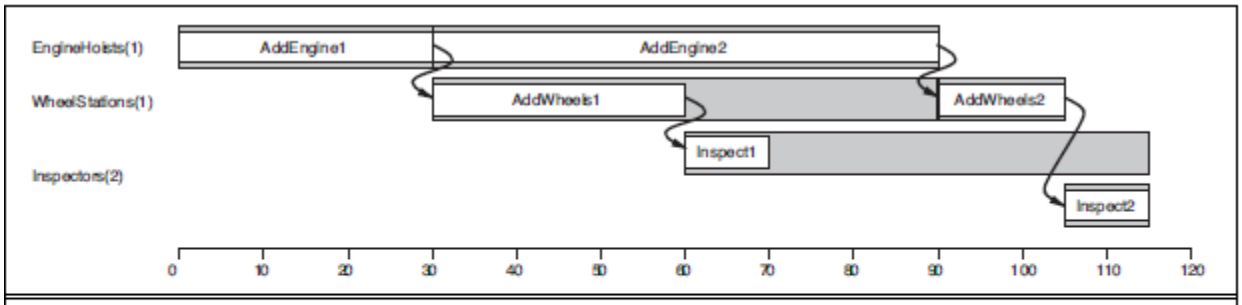



**Figure 11.3** A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler et al., 1993). Many approaches have been tried, including branch-and bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Chapters 3 and 4. One simple MINIMUM SLACK but popular heuristic is the minimum slack algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the ES and LS times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130–minute solution, not the 115–minute solution of Figure 11.3.

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a non overlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to integrate planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.

**HIERARCHICAL PLANNING**

Hierarchical Planning is an Artificial Intelligence (AI) problem solving approach for a certain kind of planning problems -- the kind focusing on problem decomposition, where problems are step-wise refined into smaller and smaller ones until the problem is finally solved.

Hierarchical Planning - describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions. For plans executed by the human brain, atomic actions are muscle activations.

In very round numbers, we have about $10^3$ muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about $10^9$ seconds in all. Thus, a human life contains about $10^{13}$ actions, give or take one or two orders of magnitude.

Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around $10^{10}$ actions. This is a lot more than 1000. To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be ―Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.” Given such a plan, the action ―Go to San Francisco airport‖ can be viewed as a planning task in itself, with a solution such as ―Drive to the long-term parking lot; park; take the shuttle to the terminal.”

Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the

shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices.

The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a large number of individual actions; for large-scale problems, this is completely impractical.

- In order to solve hard problems, a problem solver may have to generate long plans.
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main issues is found.
- Then an attempt can be made to fill in the appropriate details.
- Early attempts to do this involved the use of macro operators, in which larger operators were built from smaller ones.
- In this approach, no details are eliminated from actual descriptions of the operators.

ABSTRIPS

A better approach developed in ABSTRIPS systems which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction are ignored. ABSTRIPS approach is as follows:

- First solve the problem completely, considering only preconditions whose criticality value is the highest possible.
- These values reflect the expected difficulty of satisfying the precondition.
- To do this, do exactly what STRIPS did, but simply ignore the preconditions of lower than peak criticality. Once this is done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level.
- Augment the plan with operators that satisfy those preconditions.
- Because this approach explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it is called length-first approach.

The assignment of appropriate criticality value is crucial to the success of this hierarchical planning method. Those preconditions that no operator can satisfy are clearly the most critical. Example, solving a problem of moving the robot, for applying an operator, PUSH-THROUGH DOOR, the precondition that there exists a door big enough for the robot to get through is of high criticality since there is nothing we can do about it if it is not true.

**PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS**

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. The methods here are: sensorless planning (also known as conformant planning) for environments with no

observations; contingency planning for partially observable and nondeterministic environments; and online planning and replanning for unknown environments.

Indeterminacy in the World

Bounded indeterminacy: actions can have unpredictable effects, but the possible effects can be listed in the action description axioms

Unbounded indeterminacy: set of possible preconditions or effects either is unknown or is too large to be completely enumerated closely related to qualification problem

Solutions

Conformant or sensorless planning - Devise a plan that works regardless of state or outcome such plans may not exist

Conditional planning

Plan to obtain information (observation actions) Subplan for each contingency, e.g., [Check(Tire1), if Intact(Tire1) then Inflate(Tire1) else CallAAA Expensive because it plans for many unlikely cases

Monitoring/Replanning

Assume normal states, outcomes Check progress during execution, replan if necessary Unanticipated outcomes may lead to failure (e.g., no AAA card) (Really need a combination; plan for likely/serious eventualities, deal with others when they arise, as they must eventually)

Conformant planning

Search in space of belief states (sets of possible actual states)

Conditional planning

If the world is nondeterministic or partially observable then percepts usually provide information, i.e., split up the belief state Conditional plans check (any consequence of KB +) percept [... , if C then Plan A else Plan B, ..=]

Execution: check C against current KB, execute ―then‖ or ―else‖

Need to handle nondeterminism by building into the plan conditional steps that check the state of the environment at run time, and then decide what to do.

Augment STRIPS to allow for nondeterminism:

Add Disjunctive effects (e.g., to model when action sometimes fails):
*Action*(*Left*, PRECOND:*AtR*, EFFECT: *AtL* ∨ *AtR*)
Add Conditional effects (i.e., depends on state in which it's executed):
Form: **when** <*condition*> : <*effect*>
*Action*(*Suck*, PRECOND:,
EFFECT: (**when** *AtL: CleanL*) ∧ (**when** *AtR: CleanR*))

Create Conditional steps:
**if** <*test*> **then** *plan-A* **else** *plan-B*

Need some plan for every possible percept and action outcome

(Cf. game playing: some response for every opponent move)
(Cf. backward chaining: some rule such that every premise satisfied

Use: AND–OR tree search (very similar to backward chaining algorithm)
Similar to game tree in minimax search

Differences: Max and Min nodes become OR and AND nodes

- Robot takes action in ―state‖ nodes.
- Nature decides outcome at ―chance‖ nodes.
- Plan needs to take some action at every state it reaches (i.e., Or nodes)
- Plan must handle every outcome for the action it takes (i.e., And nodes)

Solution is a subtree with (1) goal node at every leaf, (2) one action specified at each state node, and (3) includes every outcome branch at chance nodes.

Assume: You have a chair, a table, and some cans of paint; all colors are unknown.

Goal: chair and table have the same color.
How would each of the following handle this problem?

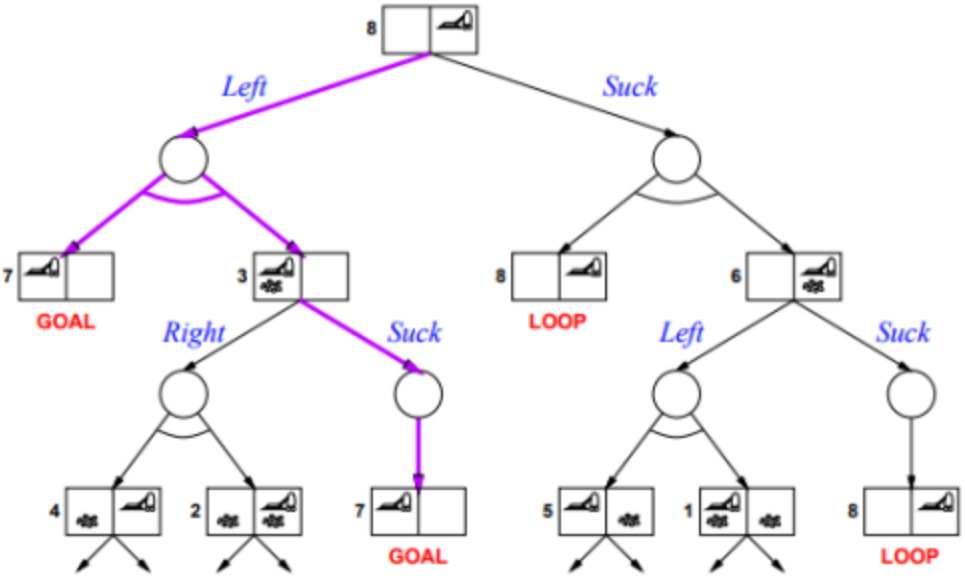
Classical planning: Can't handle it, because the initial state isn't fully specified.

Sensorless/Conformant planning: Open can of paint and apply it to both chair and table.

Conditional planning: Sense the color of the table and chair. If it's the same, then we're done. If not, sense labels on the paint cans; if there is a can that is the same color as one piece of furniture, then apply the paint to the other piece. Otherwise, paint both pieces with any color.

## Example: "Game Tree", Fully Observable World

Double Murphy: sucking or arriving may dirty a clean square




Plan: [*Left*, **if** $AtL \wedge CleanL \wedge CleanR$ **then** [] **else** *Suck* ]

Monitoring/replanning: Similar to conditional planner, but perhaps with fewer branches at first, which are filled in as needed at runtime. Also, would check for unexpected outcomes (e.g., missed a spot in painting, so repaint)

Incomplete info: use conditional plans; conformant planning (can use belief states)

Incorrect info: use execution monitoring and replanning

**MULTIAGENT PLANNING**

We have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a multiagent planning problem in which it tries to achieve its own goals with the help or hindrance of others.

Between the purely single-agent and truly multiagent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do multi effector planning to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory— multi effector planning becomes multibody planning. A multibody problem is still a "standard" single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body.

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multi agent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multi- agent doubles team, on the other hand, each agent decides what to do; without some method for coordination, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

Planning with multiple simultaneous actions

For the time being, we will treat the multi effector, multibody, and multi agent settings in the same way, labeling them generically as multi actor settings, using the generic term actor to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multi actor setting.

A correct plan is one that, if executed by the actors, achieves the goal. (In the true multi-agent setting, of course, the agents may not agree to execute any particular plan, but at least they will know what plans would work if they did agree to execute them.) For simplicity, we assume perfect synchronization: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

Having put the actors together into a multi actor system with a huge branching factor, the principal focus of research on multi actor planning has been to decouple the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are loosely coupled, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI.

$Actors(A, B)$
$Init(At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$
$\quad Approaching(Ball, RightBaseline)) \wedge Partner(A, B) \wedge Partner(B, A)$
$Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet))$
$Action(Hit(actor, Ball),$
$\quad$ PRECOND: $Approaching(Ball, loc) \wedge At(actor, loc)$
$\quad$ EFFECT: $Returned(Ball))$
$Action(Go(actor, to),$
$\quad$ PRECOND: $At(actor, loc) \wedge to \neq loc,$
$\quad$ EFFECT: $At(actor, to) \wedge \neg At(actor, loc))$

**Figure 11.10** The doubles tennis problem. Two actors $A$ and $B$ are playing together and can be in one of four locations: $LeftBaseline$, $RightBaseline$, $LeftNet$, and $RightNet$. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let's see how this works for the doubles tennis problem. Let's suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

Planning with multiple agents Cooperation and coordination:

Now let us consider the true multi agent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the "the" in "the joint solution" is misleading. For our doubles team, more than one joint solution exists:

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if A chooses plan 2 and B chooses plan 1, then nobody will return the ball. Conversely, if A chooses 1 and B chooses 2, then they will both try to hit the ball.

One option is to adopt a convention before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention "stick to your side of the court" would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention "stay on the right side of the road" in most countries; the alternative, "stay on the left side," works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called social laws.

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen's job is to re- produce, not to do centralized planning—and with very limited computation,

Communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role ac- cording to the local conditions it observes. One final example of cooperative multi agent behavior appears in the flocking behavior of birds.

We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a boid) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors


**Figure 11.11** (a) A simulated flock of birds, using Reynold's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

If all the boids execute this policy, the flock exhibits the emergent behavior of flying as a pseudo rigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents. The most difficult multi agent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control.