LECTURE NOTES–UNIT IV

PLANNING

ECS302

ARTIFICIAL INTELLIGENCE

Module IV Lecture Notes [*Planning*]

Syllabus

The Planning Problem, Planning with State-Space Search, Partial-Order Planning

1. The Planning Problem:

Planning is the task of coming up with a sequence of actions that will achieve Goal. Consider what can happen when an ordinary problem-solving agent using standard search algorithms like depth-first, A*, and so on-comes up against large, real-world problems. So, better planning agents are required.

1) Consider an internet buying Agent whose task is to buy a textbook (search based)
   So planner should works back from goal to BUY(x) we need to have a plan Have(x).
2) Finding Good Heuristic function
   Finally the problem solver is inefficient because it can't take advantage of problem decomposition. i.e.,
   For example: problem of delivering a set of overnight packages to their respective destinations, which are scattered across a country. So planning system will do;
   - *Open up action & goal representation to allow selection.*
   - *Divide & Conquer by sub goals*
   - *Relax Requirement for sequential construction of solutions*

1.1 The language of planning problems:

Consider the basic representation language of classical planners, known as ***STRIPS*** language.

Representation of states:

- Decompose the world in to logical conditions and represent a state as conjunction of positive literals
- Must be grounded & function free.

  For example: At (Plane1, Melbourne) ∧ At (Plane2, Sydney)

- Literals such as At (x, y) or At (Father (Fred), Sydney) are not allowed.

Representation of goals:

- **A** goal is a partially specified state, represented as a conjunction of positive ground literals, such as Rich ∧ Famous or At (P2, LAS).
- A propositional state *s* satisfies a goal g if *s* contains all the atoms in g.
- For example, the state *Rich ∧ Famous ∧ Miserable* satisfies the goal *Rich ∧ Famous*.

Representation of actions:

- An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed.
- For example, an action for flying a plane from one location to another is:

  Action (Fly (p, from, to),

  PRECOND: At (p, from) ∧ Plane (p) ∧ Airport (from) ∧ Airport (to)

  EFFECT: ¬ At (p, from) ∧ At (p, to))

This is called as "Action Schema", which consists of three parts:

a) The ***action name*** and ***parameter list***.
   For example, *Fly (p, from, to)*-serves to identify the action.
b) The ***precondition*** is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
c) The ***effect*** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal ¬*P* is asserted to be false. Variables in the effect must also appear in the action's parameter list.

→To improve readability, some planning systems divide the effect into the ***add list*** for positive literals and the ***delete list*** for negative literals.

→An action is ***applicable*** in any state that satisfies the precondition; otherwise, the action has no effect.

For a first-order action schema, establishing applicability will involve a substitution θ for the variables in the precondition. For example, suppose the current state is described by

$$At(Pl, JFK) \wedge At(P2, SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport\ (JFK) \wedge Airport\ (SFO).$$

This state satisfies the precondition;

$$At\ (p, from) \wedge Plane\ (p) \wedge Airport\ (from) \wedge Airport\ (to)$$

With substitution $\{p/Pl, from/JFK, to/SFO)$

So, the action *Fly (Pl, JFK, SFO)* is applicable.

Starting in state's', the result of executing an applicable action '*a*' is a state's' that is the same as *s* except that any positive literal *P* in the effect of a is added to *s* and any negative literal ¬*P* is removed from s'. Thus, after *Fly (Pl, JFK, SFO),* the current state becomes

$$At(Pl, SFO) \wedge At(P2, SFO) \wedge Plane(Pl) \wedge Plane(P2) \wedge Airport\ (JFK) \wedge Airport\ (SFO)\ .$$

STRIPS assumption: Stands for [Standard Research Institute Problem Solver]

Here every literal not mentioned in the effect remains unchanged. It is a restricted language for planning and the representation is chosen to make planning Algorithms simpler and efficient.

In recent years, it has become clear that STRIPS is insufficiently expressive for some real domains. As a result, many language variants have been developed. Figure 1.1 briefly describes one important one, the Action Description Language or ADL, by comparing it with the basic **STRIPS** language. In **ADL**, the *Fly* action could be written as;

Action(***Fly***(*p* : *Plane, from* : *Airport, to* : *Airport*),
PRECOND: $At(p, from) \wedge (from \neq to)$
EFFECT: $\neg At(p, from) \wedge At(p, to)$).

STRIPS **Language**	ADL **Language**
Only positive literals in states: *Poor* A *Unknown*	Positive and negative literals in states: ¬*Rich* A ¬*Famous*
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect **P** A ¬*Q* means add **P** and delete Q.	Effect **P** A ¬*Q* means add **P** and ¬*Q* and delete ¬*P* and Q.
Only ground literals in goals: *Rich* A *Famous*	Quantified variables in goals: $\exists x At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: *Rich* A *Famous*	Goals allow conjunction and disjunction: ¬*Poor* A *(Famous V Smart)*
Effects are conjunctions.	Conditional effects allowed: **when P**: E means E is an effect only if *P* is satisfied.
No support for equality.	Equality predicate (x = y) is built in.
No support for types.	Variables can have types, as in (*p* : *Plane*).

Figure 1.1 Comparison of STRIPS and ADL languages for representing planning problems.

STRIPS Problems:

1) Transportation of "Air Cargo between Airports"

Init(At(Cl, SFO) ^ At(C2, JFK) ^ At(P1, SFO) ^ At(P2, JFK)
^ Cargo(Cl) ^ Cargo(C2) ^ Plane(P1) ^ Plane(P2)
^ Airport(JFK) ^ Airport(SF0))
Goal (At (CI , *JFK) A At (C2, SFO))*
Action(Load(c, p, a),
PRECOND: *At(c, a) ^ At(p, a) ^ Cargo(c) ^ Plane(p) ^ Airport(a)*
EFFECT: ¬ *At(c, a) ^ In(c, p))*
Action(Unload(c, p, a),
PRECOND: *In(c, p) ^ At(p, a) ^ Cargo(c) ^ Plane(p) ^ Airport(a)*
EFFECT: *At(c, a) ^ ¬ In(c, p))*
Action(Fly(p, from, to),
PRECOND: *At(p, from) ^ Plane(p) ^ Airport(from) ^ Airport(to)*
EFFECT: ¬*At(p, from) ^ At(p, to))*

Figure 1.2 A STRIPS problem involving transportation of air cargo between airports.

The following plan is a solution to the problem:

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P2, SFO)].$$

1) The Spare Tire Problem:

Problem of changing a flat tire. The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. There are just *four actions:* Removing the spare from the trunk,

Removing the flat tire from the axle,

Putting the spare on the axle, and

Leaving the car unattended overnight.

The ADL description of the problem is shown below;

Init(At (Flat, Axle) ^ At(Spare, Trunk))
Goal (At (Spare, Axle))
Action(Remove(Spare, Trunk),
PRECOND: *At(Spare, Trunk)*
EFFECT: ¬ *At (Spare, Trunk) ^ At (Spare, Ground))*
Action(Remove(Flat, .Axle),
PRECOND: *At(Flat, Axle)*
EFFECT: 1 *At(Flat, Axle) ^ At(Flat, Ground))*
Action(Put On (Spare, Axle),
PRECOND: *At(Spare, Ground) ^ ¬ At (Flat, Axle)*
EFFECT: ¬ *At(Spare, Ground) ^ At(Spare, Axle))*
Action(Leave Overnight,
PRECOND:
EFFECT: ¬ *At(Spare, Ground) ^ ¬ At(Spam, Axle) ^ ¬ At(Spare, Trunk)*
^ ¬ At(Flat, Ground) ^ ¬ At(Flat, Axle))

Figure 1.3 The simple spare tire problem.

2) The Blocks world Problem:

This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block *C* on D.

We will use ***On (b, x)*** to indicate that block *b* is on *x,* where *x* is either another block or the table. The action for moving block *b* from the top of *x* to the top of y will be ***Move (b, x, y).*** The action *Move* moves a block *b* from *x* to y if both *b* and *y* are clear. After the move is made, *x* is clear but *y* is not. *A* formal description of *Move* in STRIPS is

Action (Move (b, z*, y)* ,
PRECOND: *On(b, x)* ^ *Clear(b)* ^ *Clear(y),*
EFFECT: *On (b, y)* ^ *Clear(x)* ^ ¬*On (b, x)* ^ ¬*Clear(y))*

Unfortunately, this action does not maintain *Clear* properly when *x* or y is the table. To fix this, we do two things. First, we introduce another action to move a block *b* from x to the table:

Action (Move To Table (b, x) ,
PRECOND: *On(b, x)* ^ *Clear(b),*
EFFECT: *On (b, Table)* ^ *Clear(x) A* ¬*On (b, x))*

Second, we take the interpretation of *Clear (b)* to be "there is a clear space on *b* to hold a block." Under this interpretation, *Clear (Table)* will always be true.

Init(On(A, Table) ^ *On(B, Table)* ^ *On(C, Table)*
A Block(A) ^ *Block(B)* ^ *Block(C)*
Clear(A) ^ *Clear(B)* ^ *Clear(C))*
Goal(On(A, B) ^ *On(B, C))*
Action(Move(b, x, y),
PRECOND: *On(b, x)* ^ *Clear(b)* ^ *Clear(y)* ^ *Block(b)* ^ *(b* # x*)* ^ *(b* # *y)* ^ *(x* # *y),*
EFFECT: *On(b,* y*)* ^ *Clear(x)* ^ ¬*On(b, x)* ^ ¬ *Clear(y))*

Action(Move To Table(b, x) , PRECOND: *On(b, x) ^ Clear(b) ^ Block(b) ^ (b # x),* EFFECT: *On(b, Table) ^ Clear(x) ^ ¬ On(b, x))*
***Figure 1.4** A planning problem in the blocks world: building a three-block tower. One solution is the sequence [Move (B, T able, C), Move (A, Table, B)].*

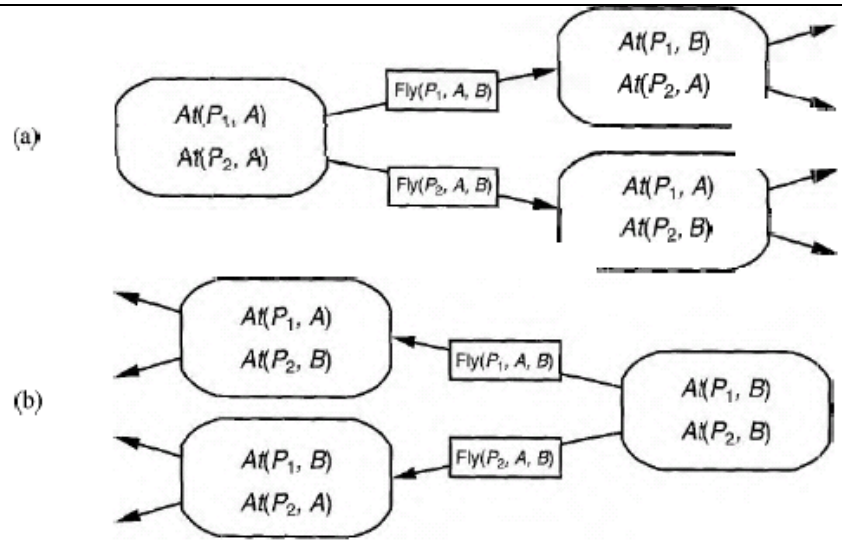

***Figure 1.5** Two approaches to searching for a plan. (a) Forward (progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state. (b) Backward (regression) state-space search: a belief-state search starting at the goal state(s) and using the inverse of the actions to search backward for th e initial state.*

2. Planning With State Space Search:

2.1 Forward state-space search:

It is a problem solving approach also called as "Progression Planning". Because it moves in forward direction.

Procedure:

1) 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.
2) The **actions** that 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.

3) The **goal test** checks whether the state satisfies the goal of the planning problem.
4) The **step cost** of each action is typically 1. Although it would be easy to allow different costs for different actions, this is seldom done by STRIPS planners.

First, forward search does not address the irrelevant action problem-all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic.

Consider "*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.

2.2 Backward state-space search:

It is how to generate a description of the possible **predecessors** of the set of goal states. We will see that the STRIPS representation makes this quite easy because sets of states can be described by the literals that must be true in those states.

Advantage: 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

$$At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B).$$

Now consider the conjunct *At (C1, B).*Working backwards, we can seek actions that have this as an effect. There is only one: Unload *(C1, p, B),* where plane *p* is unspecified.
Searching backwards is sometimes called ***regression*** planning. To see how to do it, consider the air cargo example. We have the goal

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

The relevant action *Unload (C1, p, B),* which achieves the first conjunct. i.e.,

$$In\ (C1, p) \wedge At\ (p, B)$$

Moreover, the sub goal *At (C1, B)* should not be true in the predecessor state. Thus, the predecessor description is;

$$In\ (C1, p) \wedge\ At\ (p, B) \wedge\ At\ (C2, B) \wedge\ \ldots \wedge At\ (C20, B)$$

If we insist that the actions *not* undo any desired literals. An action that satisfies this restriction is called **consistent.** For example, the action *Load (C2, p)* would not be consistent with the current goal, because it would negate the literal *At (C2, B)*.

Heuristics for state-space search:

We might also be able to derive an *admissible heuristic-one* that does not overestimate. This could be used with A* search to find optimal solutions. There are two approaches that can be tried.

a) The first is to derive a *relaxed problem* from the given problem specification.
b) 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 sub goals is approximated by the sum of the costs of solving each sub goal *independently.*

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. 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.

3. Partial Order Planning

Forward and backward state-space search are particular forms of *totally ordered* plan search. They explore only strictly linear sequences of actions directly connected to the start or goal. So, Rather than work on each sub problem separately, they must always make decisions about how to sequence actions from all the sub problems. We would prefer an approach that works on several sub goals independently, solves them with several sub plans, and then combines the sub plans. This is called *"Partial – Order Planning".*

Advantage: Flexibility in order that which it constructs.

Consider an example of "*Putting on a pair of Shoes*".

Goal (Right Shoe On ^ Left Shoe On)
Init ()
Action(Right Shoe, PRECOND: Right Sock On, EFFECT: Right Shoe On)
Action(Right Sock, EFFECT: Right Sock On)
Action(Left Shoe; PRECOND: Left Sock on, EFFECT: Left Shoe on)
Action (Left Sock, EFFECT: Left sock ON)

→A planner should be able to come up with the two-action sequence;
Right sock followed by Right shoe to achieve the first conjunct of the goal and the sequence
Left sock followed by Left Shoe for the second conjunct.
Then the two sequences can be combined to yield the *final plan*.

→Any planning algorithm that can place two actions into a plan without specifying which PLANNER comes first is called a *partial-order planner [POP].*
Figure 3.1 shows the partial-order plan that is the solution to the shoes and socks problem. Note that the solution is represented as a graph of actions, not a sequence. Note also the "dummy" actions called *Start* and *Finish*, which mark the beginning and end of the plan. Calling them actions simplifies things, because now every step of a plan is an action. The partial-order solution corresponds to six possible total-order plans; each of these is called a *linearization* of the partial-order plan.

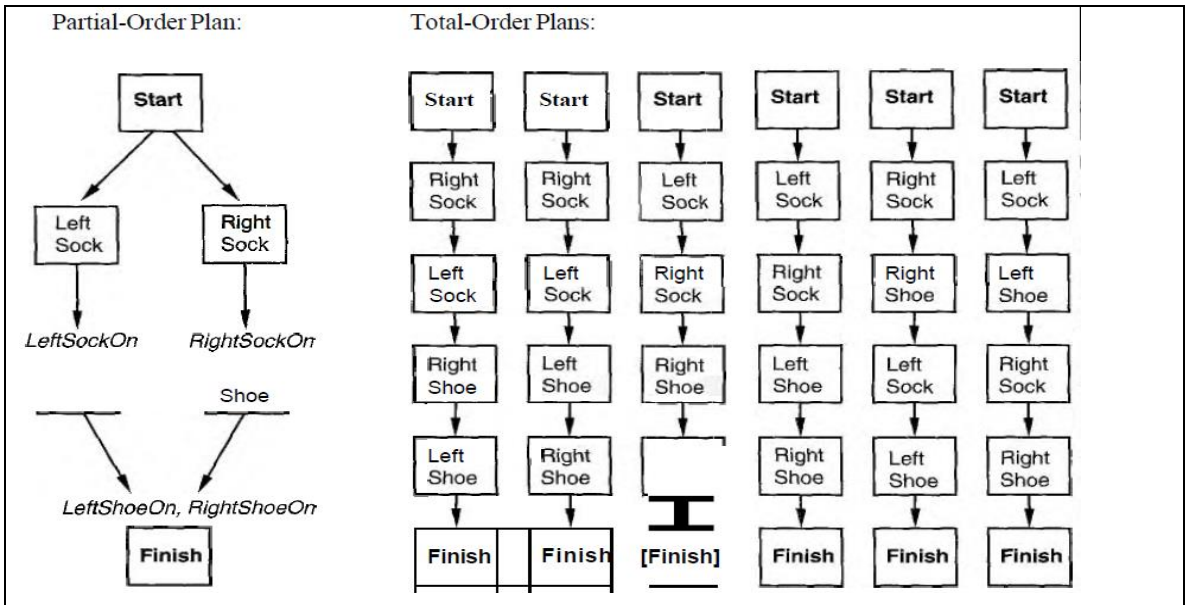

Figure 3.1 *A partial-order plan for putting on shoes and socks, and the six corresponding linearization's into total-order plans.*

Each plan has the following four components;

1) A set of **actions** that make up the steps of the plan. These are taken from the set of actions in the planning problem. The "empty" plan contains just the Start and Finish actions.
   Start has no preconditions and has as its effect
   Finish has no effects and has as its preconditions
2) A set of **ordering constraints.** Each ordering constraint is of the form A $\prec$ B, which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle-such as A $\prec$ *B* and B $\prec$ A-represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.
3) A set of **causal links.** A causal link between two actions A and B in the plan is written as A $\prec$ B and is read as "A **achieves** *p* for **B."** For example, the causal link

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

Asserts that *Right Sock On* is an effect of the *Right Sock* action and a precondition of *Right Shoe.*

A plan may not be extended by adding a new action *C* that **conflicts** with the causal link. An action C conflicts with A $\xrightarrow{p}$ B if C has the effect $\neg p$ and if *C* could come after A and before B.

Some authors call causal links **protection intervals,** because the link A $\xrightarrow{p}$ B protects p from being negated over the interval from A to B.

4) A set of **open preconditions.** A precondition is open if it is not achieved by some action in the plan.

Consider the following components resemble the above plan;

Actions:{*RightSock*, RightShoe, *LeftSock*, *LeftShoe*, Start, Finish)
Orderings:{*RightSock* $\prec$ RightShoe, *LeftSock* $\prec$ *LeftShoe*}
Links:{*RightSock* $\xrightarrow{RightSockOn}$ RightShoe, *LeftSock* $\xrightarrow{LeftSockOn}$ LeftShoe,
RightShoe $\xrightarrow{RightShoeOn}$ *Finish*, *LeftShoe* $\xrightarrow{LeftShoeOn}$ *Finish*}
Open Preconditions:{}.

***Consistent plan*:** It is a plan in which there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a ***solution*.** It should convince the reader with the following fact: "*every linearization of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state*".

Now formulate the search problem that POP solves. We will begin with a formulation suitable for propositional planning problems, leaving the first-order complications for later. As usual, the definition includes the initial state, actions, and goal test.

The initial plan contains Start and Finish, the ordering constraint Start $\prec$ Finish, and no causal links and has all the preconditions in Finish as open preconditions.

The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p. Consistency is enforced as follows:

1. The causal link A $\xrightarrow{p}$ B and the ordering constraint A $\prec$ B are added to the plan. Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add Start $\prec$ A and A $\prec$ Finish.

2. We resolve conflicts between the new causal link and all existing actions and between the action A (if it is new) and all existing causal links. A conflict between A $\xrightarrow{p}$ B and *C* is resolved by making *C* occur at some time outside the protection interval, either by adding B $\prec$ C or *C* $\prec$ A. We add successor states for either or both if they result in consistent plans.

The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

A partial-order planning example:

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
PRECOND: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
PRECOND: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Figure 3.2 The simple flat tire problem description.

The search for a solution begins with the initial plan, containing; *Start* action with the effect *At (Spare, Trunk)* ^ *At (Flat, Axle)* and *Finish* action with the sole precondition *At (Spare, Axle).* Then we generate successors by picking an open precondition to work on (irrevocably) and choosing among the possible actions to achieve it.

The sequence of events is as follows:

1. Pick the only open precondition, *At (Spare, Axle)* of *Finish.* Choose the only applicable action, *Put On (Spare, Axle).*

2. Pick the *At (Spare, Ground)* precondition of *PutOn (Spare, Axle).* Choose the only applicable action, *Remove (Spare, Trunk)* to achieve it. The resulting plan is shown in Figure 3.3.

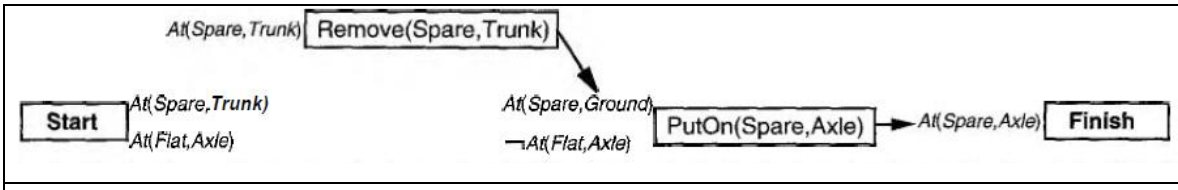

Figure 3.3 *The incomplete partial-order plan for the tire problem, after choosing actions for the first two open preconditions. Boxes represent actions, with preconditions on the left and effects on the right. Dark arrows represent causal links protecting the proposition at the head of the arrow.*

3. Pick the ¬*At (Flat, Axle)* precondition of *PutOn (Spare, Axle).* Just to be contrary, choose the *Leave Overnight* action rather than the *Remove (Flat, Axle)* action. Notice that *Leave Overnight* also has the effect ¬*At (Spare, Ground),* which means it conflicts with the causal link

$$Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$$

To resolve the conflict we add an ordering constraint putting *Leave Overnight* before *Remove (Spare, Trunk).*

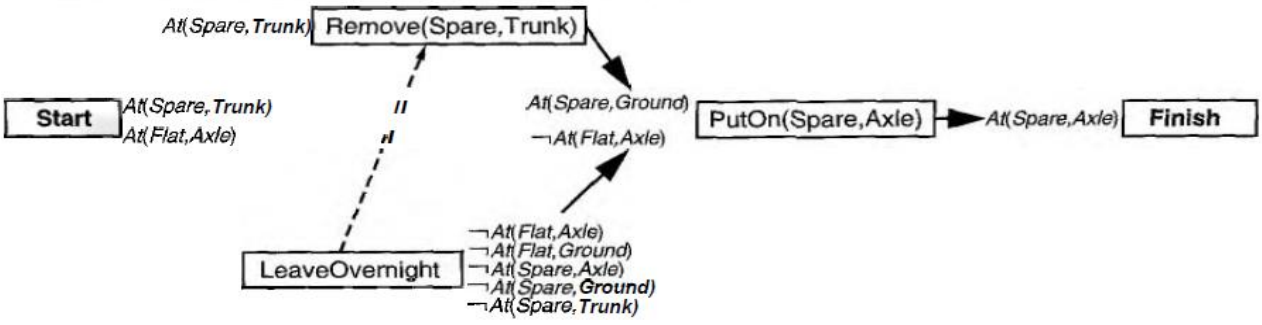

Figure 3.4 *The plan after choosing Leave Overnight as the action for achieving ~At (Flat, Axle). To avoid a conflict with the causal link from Remove (Spare, Trunk) that protects At (Spare, Ground), Leave Overnight is constrained to occur before Remove (Spare, Trunk), as shown by the dashed arrow.*

4. The only remaining open precondition at this point is the *At (Spare, Punk)* precondition of the action *Remove (Spare, Trunk)*. The only action that can achieve it is the existing *Start* action, but the causal link from *Start* to *Remove (Spare, Trunk)* is in conflict with the ~*At (Spare, Trunk)* effect of *Leave Overnight*. This time there is no way to resolve the conflict with *Leave Overnight:* we cannot order it before *Start* (because nothing can come before *Start)*, and we cannot order it after *Remove (Spare, Trunk)* (because there is already a constraint ordering it before *Remove (Spare, Trunk))*. So we are forced to back up, remove the *Leave Overnight* action and the last two causal links, and return to the state in Figure 3.3. In essence, the planner has proved that *Leave Overnight* doesn't work as a way to change a tire.

5. Consider again the ¬*At (Flat, Axle)* precondition of *PutOn (Spare, Axle)*. This time, we choose *Remove (Flat, Axle)*.

6. Once again, pick the *At (Spare, Trunk)* precondition of *Remove (Spare, Trunk)* and choose *Start* to achieve it. This time there are no conflicts.

7. Pick the *At (Flat, Axle)* precondition of *Remove (Flat, Axle),* and choose *Start* to achieve it. This gives us a complete, consistent. Plan-in other words a solution-as shown in Figure 3.5.

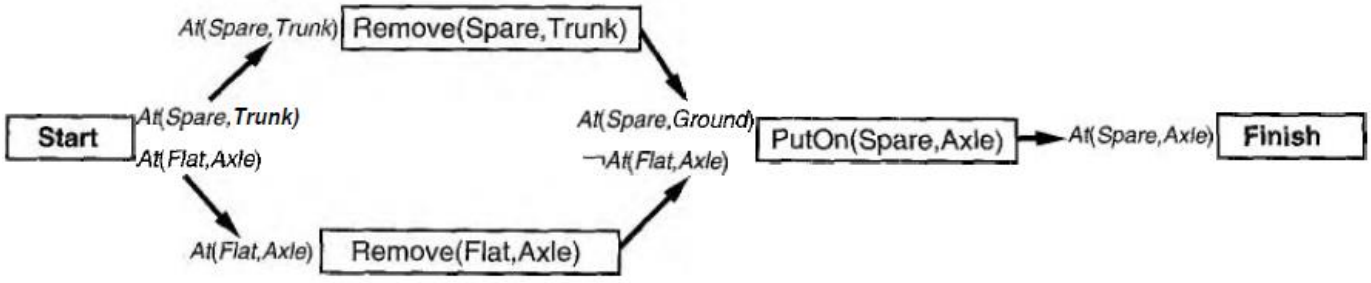

Figure 3.5 *The final solution to the tire problem. Note that Remove (Spare, Trunk) and Remove (Flat, Axle) can be done in either order, as long as they are completed before the Put On (Spare, Axle) action.*

Partial-order planning with unbound variables:

Here we consider the complications that can arise when POP is used with first order action representations that include variables. Suppose we have a blocks world problem with the open precondition *On (A,* B) and the action

Action(Move(b, x, y),
PRECOND: *On(b,* x) ^ *Clear(b)* ^ *Clear(y),*
EFFECT: On (b, y) ^ *Clear(x)* ^ ¬*On (b, x)* ^ ¬*Clear(y))*

This action achieves *On (A, B)* because the effect O*n* (b, y) unifies with *On (A, B)* with the substitution *{b/A, y/B)*. We then apply this substitution to the action, yielding;

Action(Move(A, x, B),
PRECOND: On(A, x) ^ *Clear(A)* ^ *Clear(B),*
EFFECT: On(A, B) ^ *Clear(x)* ^¬ *On(A, x)* ^ ¬*Clear(B))*

This leaves the variable x unbound. That is, the action says to move block *A* from somewhere, without yet saying whence. This is another example of the least commitment principle: we can delay making the choice until some other step in the plan makes it for us. For example, suppose we have *On (A,* D) in the initial state. Then the *Start* action can be used to achieve *On (A, x),* binding *x* to D. This strategy of waiting for more information before choosing *x* is often more efficient than trying every possible value of *x* and backtracking for each one that fails.

The presence of variables in preconditions and actions complicates the process of detecting and resolving conflicts. For example, when *Move (A, x, B)* is added to the plan, we will need a causal link

$$Move(A, x, B) \xrightarrow{On(A,B)} Finish$$

If there is another action *M2* with effect ¬*On (A, z),* then *M2* conflicts only if *z* is *B*. To accommodate this possibility, we extend the representation of plans to include a set of ***inequality constraints*** of the form *z* # X where *z* is a variable and X is either another variable or a constant symbol. In this case, we would resolve the conflict by adding *z* # *B,* which means that future extensions to the plan can instantiate *z* to any value except *B*. Anytime we apply a substitution to a plan, we must check that the inequalities do not contradict the substitution.

For example, a substitution that includes x/y conflicts with the inequality constraint *x* # y. Such conflicts cannot be resolved, so the planner must backtrack.

Heuristics for partial-order planning:

Compared with total-order planning, partial-order planning has a clear advantage in being able to decompose problems into sub problems. It also has a disadvantage in that it does not represent states directly, so it is harder to estimate how far a partial-order plan is from achieving a goal.

- The most obvious "*heuristic is to count the number of distinct open preconditions*". This can be improved by subtracting the number of open preconditions that match literals in the *Start* state. As in the total-order case, this overestimates the cost when there are actions that achieve multiple goals and underestimates the cost when there are negative interactions between plan steps.
- The heuristic function is used to choose which plan to refine. Given this choice, the algorithm generates successors based on the selection of a single open precondition to work on. As in the case of variable selection on constraint satisfaction algorithms, this selection has a large impact on efficiency.
- The **most-constrained-variable** heuristic from CSPs can be adapted for planning algorithms and seems to work well. The idea is to select the open condition that can be satisfied in the fewest number of ways.

There are two special cases of this heuristic.

1) First, if an open condition cannot be achieved by any action, the heuristic will select it; this is a good idea because early detection of impossibility can save a great deal of work.
2) Second, if an open condition can be achieved in only one way, then it should be selected because the decision is unavoidable and could provide additional constraints on other choices still to be made.

----------The End----------