ARTIFICIAL INTELLIGENCE

UNIT-III

KNOWLEDGE INFERENCE

Knowledge Representation - Production based System, Frame based System. Inference - Backward Chaining, Forward Chaining, Rule value approach, Fuzzy Reasoning - Certainity factors, Bayesian Theory - Bayesian Network - Dempster Shafer Theory

3.0 Knowledge representation: -

- The task of coming up with a sequence of actions that will achieve a goal is called Planning.
- "*Deciding in ADVANCE what is to be done*"
- A problem solving methodology
- Generating a set of action that are likely to lead to achieving a goal
- Deciding on a course of actions before acting
- **Representation for states and Goals:-**
    - In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.
    - For example,
      At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^….
    - Goals are also described by conjunctions of literals.
    - For example,
      At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)
    - Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as
- **Representation for actions:-**
    - Our STRIPS operators consist of three components:
    - the ***action description*** is what an agent actually returns to the environment in order to do something.
    - the ***precondition*** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
    - the ***effect*** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.
    - Here's an example for the operator for going from one place to another:
        - **Op(Action:Go(there),**
        - **Precond:At(here)^Path(here, there),**
        - **Effect:At(there)^ ¬At(here))**
- **Representation of Plans:-**
    - Consider a simple problem:
    - Putting on a pair of shoes
    - Goal →RightShoeOn ^ LeftShoeOn
    - Four operators:

Op(Action:RightShoe,PreCond:RightSockOn,Effect:RightShoeON)
Op(Action:RightSock , Effect: RightSockOn)
Op(Action:LeftShoe, Precond:LeftSockOn, Effect:LeftShoeOn)
Op(Action:LeftSock,Effect:LeftSockOn)

Given:-

- A description of an initial state
- A set of actions
- A (partial) description of a goal state

Problem:-

- Find a sequence of actions (plan) which transforms the initial state into the goal state.

Application areas:-

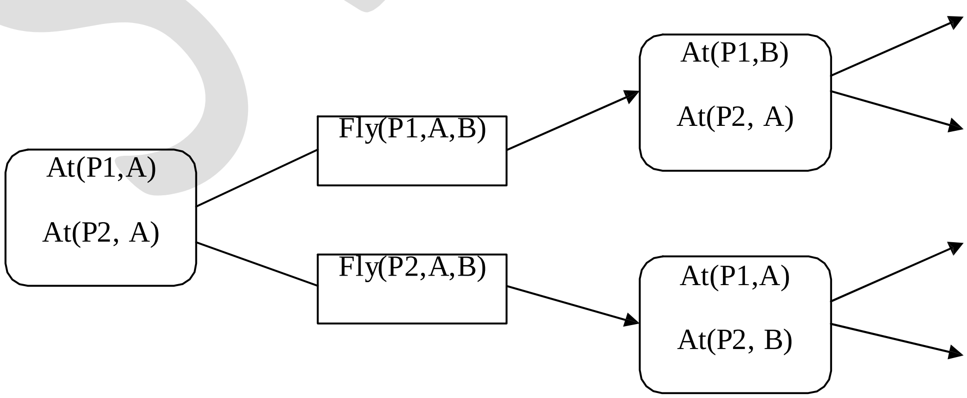
- Systems design
- Budgeting
- Manufacturing product
- Robot programming and control
- Military activities

Benefits of Planning:-

- Reducing search
- Resolving goal conflicts
- Providing basis for error recovery

3.1 Planning with State Space Search:

- Planning with state space search approach is used to construct a planning algorithm.
- This is most straightforward approach.
- The description of actions in a planning problem specifies both preconditions and effects.
- It is possible to search in either direction.
- Either from forward from the initial state or backward from the goal
- The following are the two types of state space search ,
  - Forward state-space search
  - Backward state-space search
- The following diagram shows the Forward state-space search



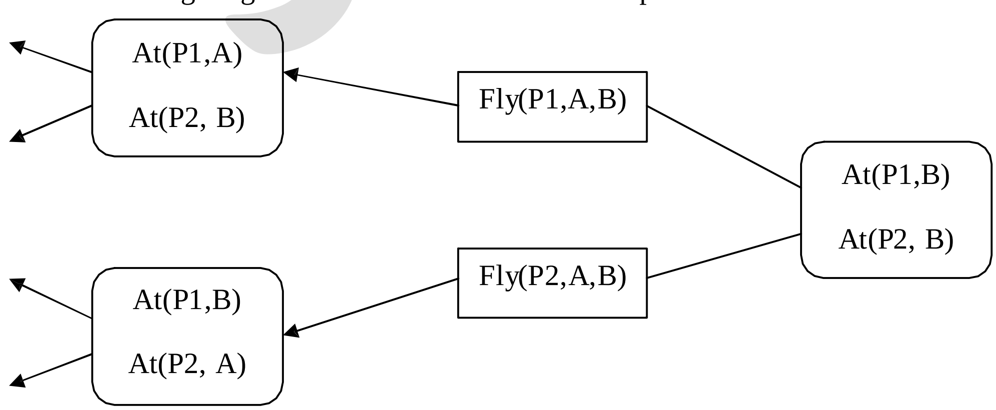
3.1.1 Forward state-space search:-

- Planning with forward state-space search is similar to the problem solving using Searching.

- It is sometimes called as progression Planning.
- It moves in the forward direction.
- we start in the problems initial state, considering sequence of actions until we find a sequence that reaches a goal state.
- The formulation of planning problems as state-space search problems is as follows,
    - 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.
    - 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.
    - The goal test checks whether the state satisfies the goal of the planning problem.
    - The step cost of each action is typically 1.
- This method was too inefficient.
- It does not address the irrelevant action problem, (i.e.) all applicable actions are considered from each state.
- This approach quickly bogs down without a good heuristics.
- For Example:-
    - Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo.
    - The Goal is to move the entire cargo form 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, then fly the plane to B, and unload the cargo.
    - But finding the solution can be difficult because the average branching factor is huge.

3.1.2 Backward state- space search:-

- Backward search is similar to bidirectional search.
- It can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly.
- It is not always obvious how to generate a description of the possible predecessors of the set of goal states.
- The main advantage of this 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.
- The following diagram shows the Backward state-space search



- For example:-
    - The goal in our 10-airport cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

      $$At(C1,B) \wedge At(C2,B) \wedge \ldots\ldots \wedge At(C20,B)$$
    - 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.
    - In this search restriction to relevant actions means that backward search often has a much lower branching factor than forward search.
- Searching backwards is sometimes called regression planning.
- The principal question 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.
- consider the air cargo example;- we have the goal as,

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

  and the relevant action Unload(C1,p,B), which achieves the first conjunct.
- The action will work only if its preconditions are satisfied.
- Therefore , any predecessor state must include these preconditions : In(C1,p) $\wedge$At(p,B), Moreover the subgoal At(C1,B) should not be true in the predecessor state.
- The predecessor description is

  $$In(C1,p) \wedge At(p,B) \wedge At(C2,B) \wedge \ldots\ldots \wedge At(C20,B)$$
- In addition to insisting that actions achieve some desired literal, we must insist that the actions not undo any desired literals.
- An action that satisfies this restriction is called consistent.
- From definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search.
- Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows
    - any positive effects of A that appear in G are deleted
    - Each precondition literal of A is added, unless it already appears
- Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem.

3.1.3 Heuristics for State-space search:-

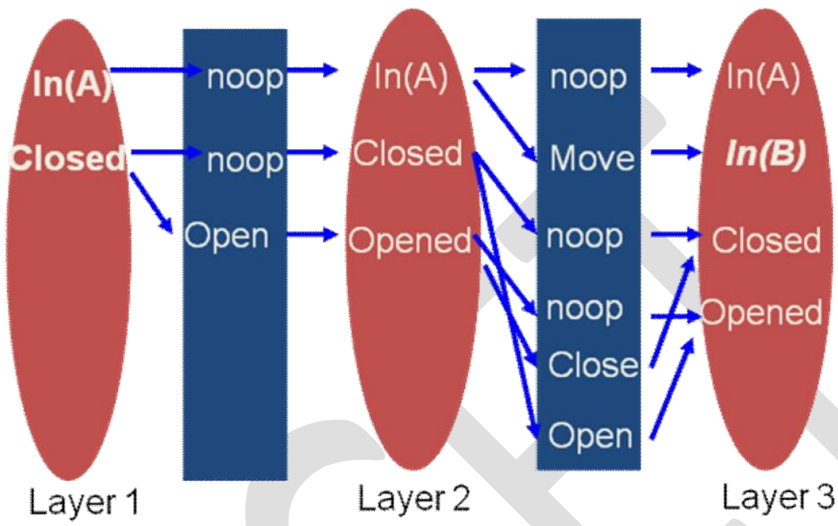
Heuristic Estimate:-

- The value of a state is a measure of how close it is to a goal state.
- This cannot be determined exactly (too hard), but can be approximated.
- One way of approximating is to use the relaxed problem.
    - Relaxation is achieved by ignoring the negative effects of the actions.
    - The relaxed action set, A’, is defined by:

      A’ = {<pre(a),add(a),0> | a in A}

Relaxed Distance Estimate

- Current: In(A), Closed Goal: In(B)



- Layers correspond to successive time points,
- # layers indicate minimum time to achieve goals.

Building the relaxed plan graph:-

- **Start** at the initial state
- **Repeatedly** apply all relaxed actions whose preconditions are satisfied.
  - Their (positive) effects are asserted at the next layer.
- **If** all actions applied and the goals are not all present in the final graph layer
  Then the problem is unsolvable.

Extracting Relaxed solution

- When a layer containing all of the goals is reached ,FF searches *backwards* for a plan.
- The earliest possible achiever is always used for any goal.
  - This maximizes the possibility for exploiting actions in the relaxed plan.
- The relaxed plan might contain many actions happening concurrently at a layer.
- The number of actions in the relaxed plan is an estimate of the true cost of achieving the goals.

How FF uses the Heuristics:-

- FF uses the heuristic to estimate how close each state is to a goal state
  - any state satisfying the goal propositions.

  - The actions in the relaxed plan are used as a guide to which actions to explore when extending the plan.
  - All actions in the relaxed plan at layer i that achieve at least one of the goals required at layer i+1 are considered helpful.
  - FF restricts attention to the helpful actions when searching forward from a state.

Properties of the Heuristics:-

- The relaxed plan that is extracted is not guaranteed to be the optimal relaxed plan.
- ➔ the heuristic is not admissible.
  - FF can produce non-optimal solutions.
  - Focusing only on helpful actions is not completeness preserving.
- ➔ Enforced hill-climbing is not completeness preserving.

3.2 Partial Order Planning:-

- Formally a planning algorithm has three inputs:
  - A description of the world in some formal language,
  - A description of the agent's goal in some formal language, and
  - A description of the possible actions that can be performed.
- The planner's o/p is a sequence of actions which when executed in any world satisfying the initial state description will achieve the goal.
- **Representation for states and Goals:-**
  - In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.
  - For example,
    At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^….
  - Goals are also described by conjunctions of literals.
  - For example,
    At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)
  - Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as
- **Representation for actions:-**
  - Our STRIPS operators consist of three components:
  - the ***action description*** is what an agent actually returns to the environment in order to do something.
  - the ***precondition*** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
  - the ***effect*** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.
  - Here's an example for the operator for going from one place to another:
    - **Op(Action:Go(there),**
    - **Precond:At(here)^Path(here, there),**
    - **Effect:At(there)^ ¬At(here))**
- **Representation of Plans:-**
  - Consider a simple problem:
    - Putting on a pair of shoes
    - Goal ➔RightShoeOn ^ LeftShoeOn
    - Four operators:

Op(Action:RightShoe,PreCond:RightSockOn,Effect:RightShoeON)
Op(Action:RightSock , Effect: RightSockOn)
Op(Action:LeftShoe, Precond:LeftSockOn, Effect:LeftShoeOn)
Op(Action:LeftSock,Effect:LeftSockOn)

- **Least Commitment:-** The general strategy of delaying a choice during search is called Least commitment.
- **Partial-order Planner:-** Any planning algorithm that can place two actions into a plan without specifying which come first is called a partial order planner.
- **Linearization:-** The partial-order solution corresponds to six possible total order plans ; each of these is called a linearization of the partial order plan.
- **Total order planner:-** Planner in which plans consist of a simple lists of steps.
- A plan is defined as a data structure
  - A set of plan steps
  - A set of step ordering
  - A set of variable binding constraints
  - A set of causal links : $s_i \xrightarrow{c} s_j$
    ”s_i achieves c for s_j”
- Initial plan before any refinements
  Start < Finish
  Refine and manipulate until a plan that is a solution

$$
\begin{aligned}
&\mathit{Plan}(\textsc{Steps}{:}\{\, S_1{:}\,\mathit{Op}(\textsc{Action}{:}\mathit{Start}),\\
&\qquad\qquad S_2{:}\,\mathit{Op}(\textsc{Action}{:}\mathit{Finish},\\
&\qquad\qquad\qquad \textsc{Precond}{:}\mathit{RightShoeOn} \land \mathit{LeftShoeOn})\},\\
&\quad \textsc{Orderings}{:}\,\{S_1 \prec S_2\},\\
&\quad \textsc{Bindings}{:}\,\{\},\\
&\quad \textsc{Links}{:}\,\{\})
\end{aligned}
$$

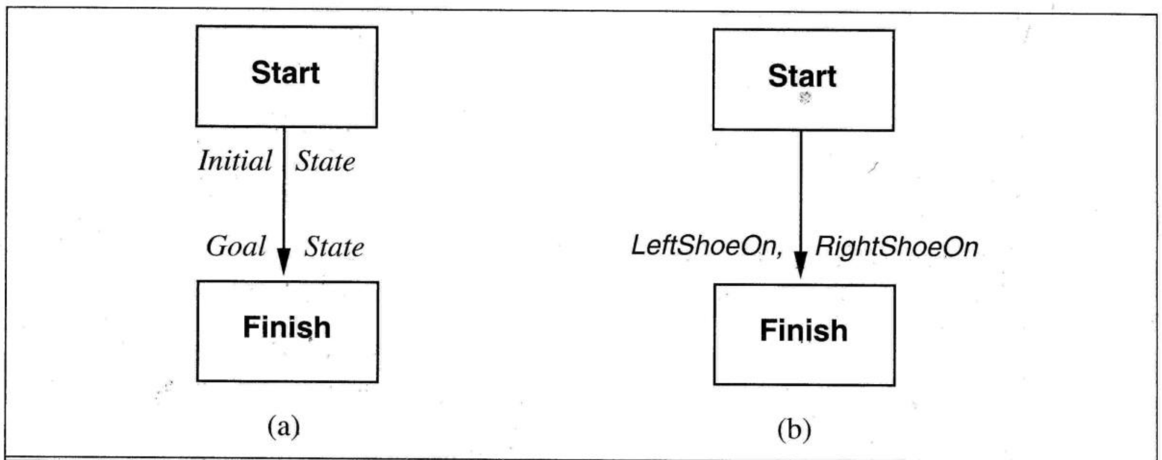
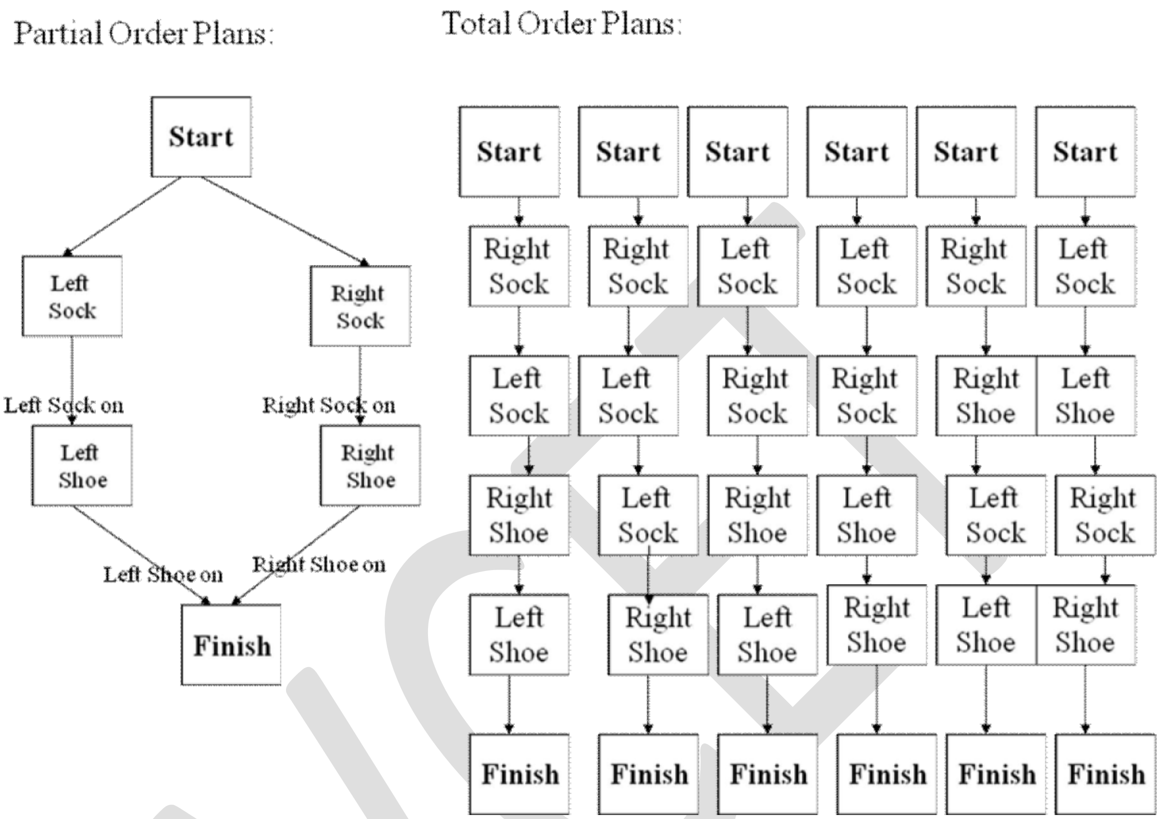


Figure 11.4 (a) Problems are defined by partial plans containing only *Start* and *Finish* steps. The initial state is entered as the effects of the *Start* step, and the goal state is the precondition of the *Finish* step. Ordering constraints are shown as arrows between boxes. (b) The initial plan for the shoes-and-socks problem.

- The following diagram shows the partial order plan for putting on shoes and socks, and the six corresponding linearization into total order plans.



- Solutions
    - solution : a plan that an agent guarantees achievement of the goal
    - a solution is a complete and consistent plan
    - a complete plan : every precondition of every step is achieved by some other step
    - a consistent plan : no contradictions in the ordering or binding constraints. When we meet a inconsistent plan we backtrack and try another branch

3.2.1 Partial order planning Algorithm:-

The following is the Partial order planning algorithm,

function pop(initial-state, conjunctive-goal, operators)
  // non-deterministic algorithm
  plan = make-initial-plan(initial-state, conjunctive-goal);
  loop:
    begin
      if solution?(plan) **then return** plan;
      (*S-need*, *c*) = select-subgoal(plan) ; // choose an unsolved goal
      choose-operator(plan, operators, *S-need*, *c*);
        // select an operator to solve that goal and revise plan
      resolve-threats(plan); // fix any threats created
    end

```
end

function solution?(plan)
  if causal-links-establishing-all-preconditions-of-all-steps(plan)
     and all-threats-resolved(plan)
     and all-temporal-ordering-constraints-consistent(plan)
     and all-variable-bindings-consistent(plan)
  then return true;
else return false;
end

function select-subgoal(plan)
  pick a plan step S-need from steps(plan) with a precondition c
     that has not been achieved;
  return (S-need, c);
end

procedure choose-operator(plan, operators, S-need, c)
  // solve "open precondition" of some step
  choose a step S-add by either
    Step Addition: adding a new step from operators that
      has c in its Add-list
    or Simple Establishment: picking an existing step in Steps(plan)
      that has c in its Add-list;
  if no such step then return fail;
  add causal link "S-add --->c S-need" to Links(plan);
  add temporal ordering constraint "S-add < S-need" to Orderings(plan);
  if S-add is a newly added step then
    begin
    add S-add to Steps(plan);
    add "Start < S-add" and "S-add < Finish" to Orderings(plan);
    end
end

procedure resolve-threats(plan)
  foreach S-threat that threatens link "Si --->c Sj" in Links(plan)
   begin    // "declobber" threat
     choose either
       Demotion: add "S-threat < Si" to Orderings(plan)
       or Promotion: add "Sj < S-threat" to Orderings(plan);
     if not(consistent(plan)) then return fail;
   end
end
```

- **Partial Order Planning Example:-**
    - Shopping problem: “get milk, banana, drill and bring them back home”
    - assumption
      1)Go action “can travel the two locations”
      2)no need money

- initial state : operator start
  Op(ACTION:Start,EFFECT:At(Home) ∧ Sells(HWS,Drill) ∧ Sells(SM,Milk), Sells(SM,Banana))
- goal state : Finish
  Op(ACTION:Finish, PRECOND:Have(Drill) ∧ Have(Milk) ∧ Have(Banana) ∧ At(Home))
- actions:
  Op(ACTION:Go(there),PRECOND:At(here),EFFECT:At(there) ∧ ¬At(here))
  Op(ACTION:Buy(x),PRECOND:At(store) ∧ Sells(store,x),EFFECT:Have(x))

- There are many possible ways in which the initial plan elaborated
  - one choice : three Buy actions for three preconditions of Finish action
  - second choice:sells precondition of Buy
    • Bold arrows:causal links, protection of precondition
    • Light arrows:ordering constraints

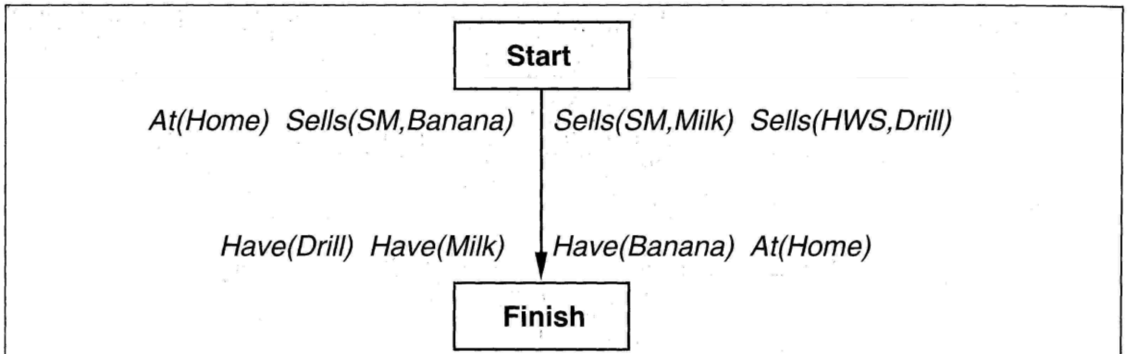
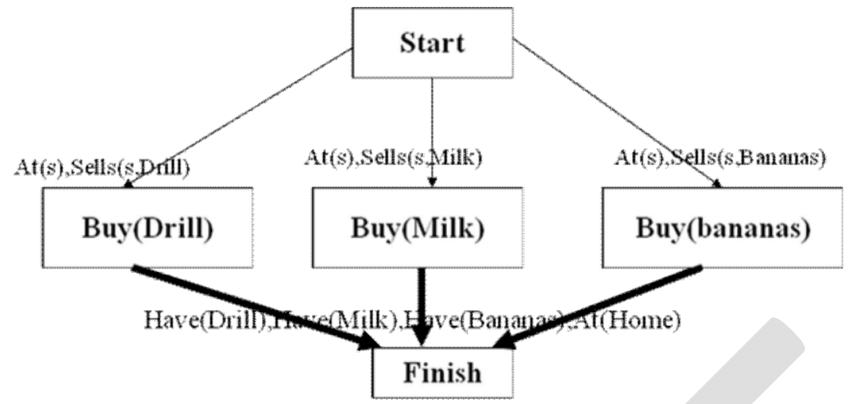
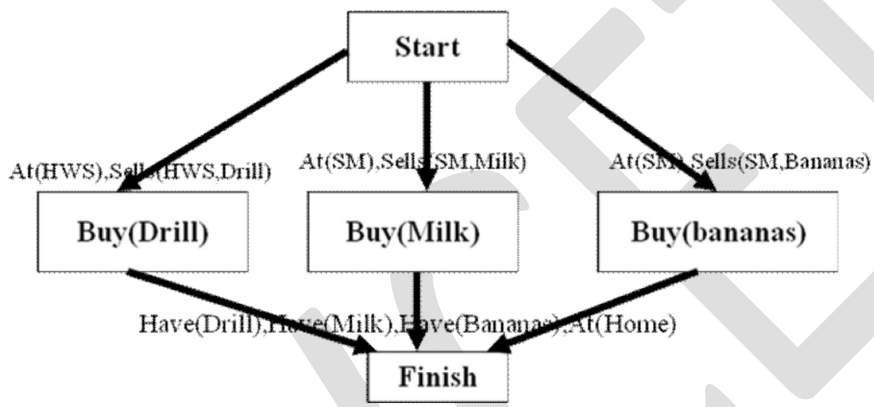


Figure 11.6 The initial plan for the shopping problem.

- The following diagram shows the,
  - partial plan that achieves three of four preconditions of finish
  - Refining the partial plan by adding casual links to achieve the sells preconditions of the buy steps





- causal links : protected links
  a causal link is protected by ensuring that threats are ordered to come before or after the protected link
- demotion : placed before
  promotion : placed after

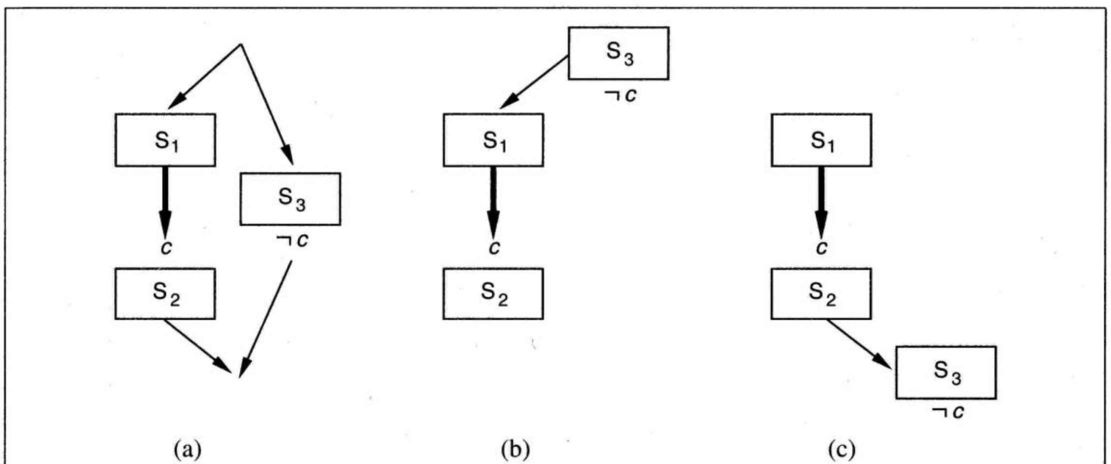
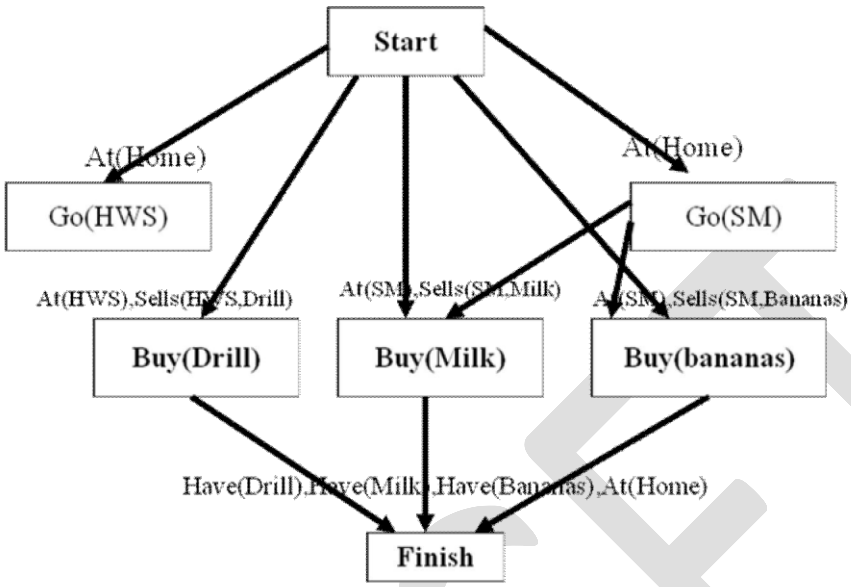
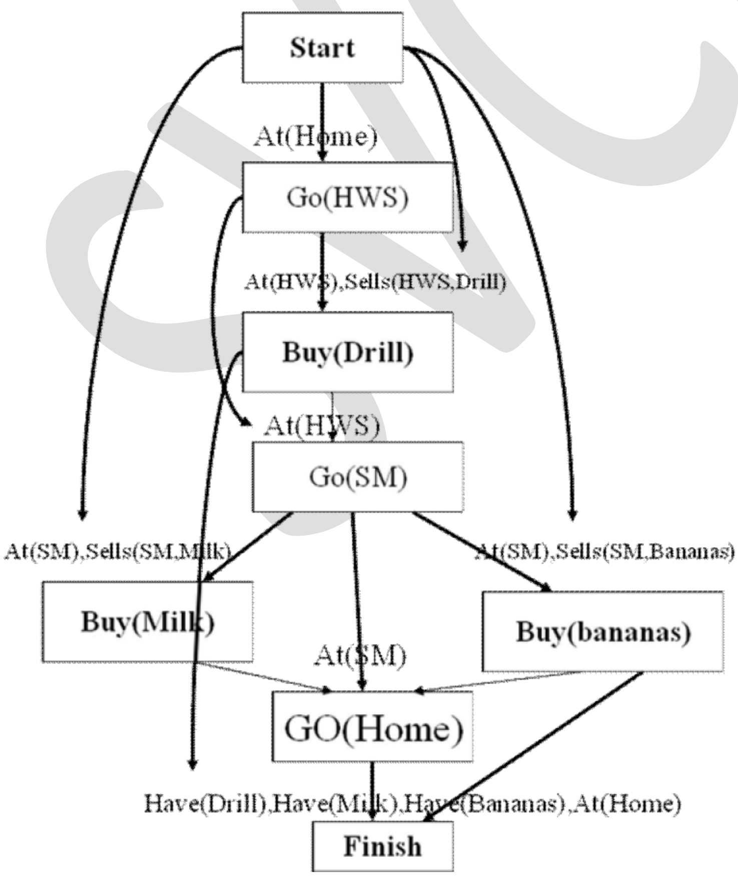


Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

- The following diagram shows the partial plan that achieves At Precondition of the three buy conditions



- The following diagram shows the solution of this problem,



- The following are the Knowledge engineering for plan,
- Methodology for solving problems with the planning approach
  (1) Decide what to talk about
  (2) Decide on a vocabulary of conditions, operators, and objects
  (3) Encode operators for the domain
  (4) Encode a description of the specific problem instance
  (5) pose problems to the planner and get back plans
- (ex) The blocks world
  - (1) what to talk about
    - cubic blocks sitting on a table
    - one block on top of another
    - A robot arm pick up a block and moves it to another position
  - (2) Vocabulary
    - objects:blocks and table
    - On(b,x) : b is on x
    - Move(b,x,y) : move b form x to y
    - ¬exist x On(x,b) or ∀x ¬On(x,b) : precondition
    - clear(x)
  - (3)Operators
    Op(ACTION:Move(b,x,y),
    PRECOND:On(b,x) ∧ Clear(b) ∧ Clear(y),
    EFFECT:On(b,y) ∧ Clear(x) ∧ ¬On(b,x) ∧ ¬Clear(y))
    Op(ACTION:MoveToTable(b,x),
    PRECOND:On(b,x) ∧ Clear(b),
    EFFECT:On(b,Table) ∧ Clear(x) ∧ ¬On(b,x))

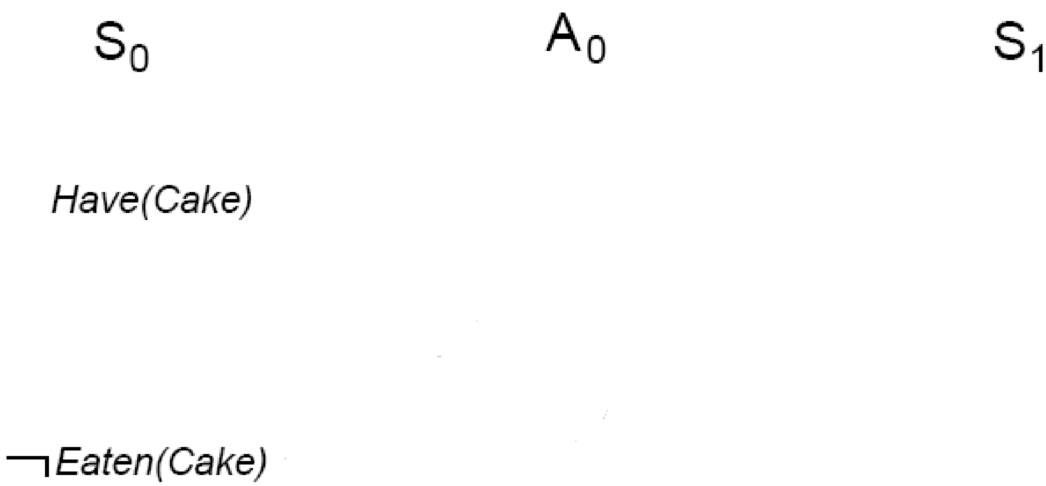
3.3 Planning Graph:-

- Planning graphs are an efficient way to create a representation of a planning problem that can be used to
  - Achieve better heuristic estimates
  - Directly construct plans
- Planning graphs only work for propositional problems.
- Planning graphs consists of a seq of levels that correspond to time steps in the plan.
  - Level 0 is the initial state.
  - Each level consists of a set of literals and a set of actions that represent what *might be* possible at that step in the plan
  - *Might be* is the key to efficiency
  - Records only a restricted subset of possible negative interactions among actions.
- Each level consists of
  - *Literals* = all those that *could* be true at that time step, depending upon the actions executed at preceding time steps.
  - *Actions* = all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.
- For Example:-

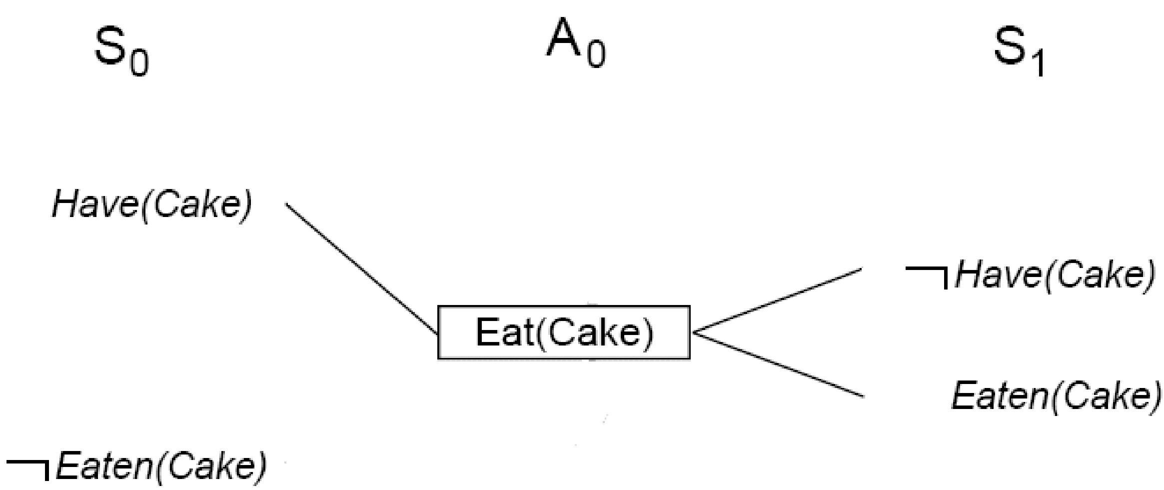
  Init(Have(Cake))
  Goal(Have(Cake) ∧ Eaten(Cake))

Action(Eat(Cake),
PRECOND: Have(Cake)
EFFECT: ¬Have(Cake) ∧ Eaten(Cake))
Action(Bake(Cake),
PRECOND: ¬ Have(Cake)
EFFECT: Have(Cake))

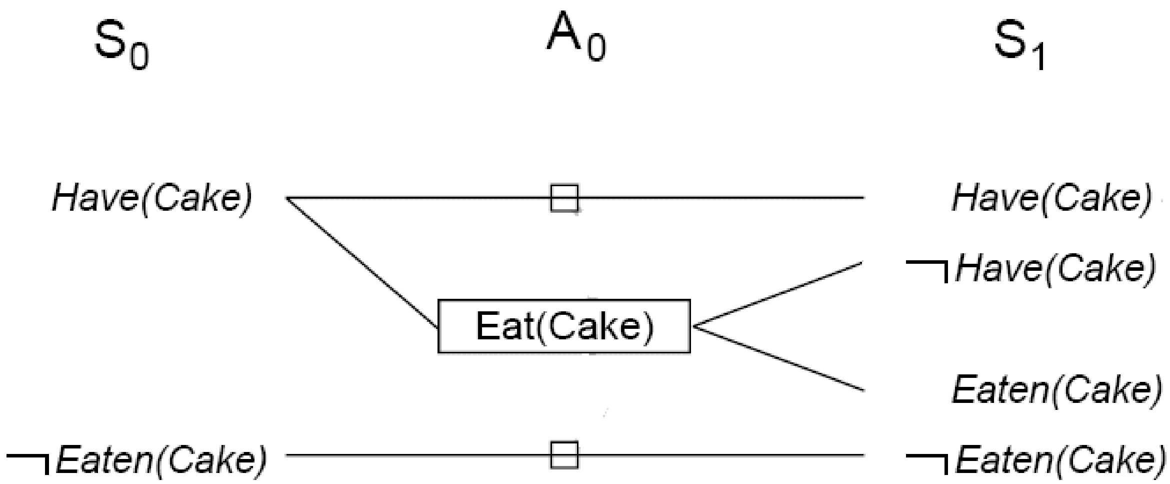
- Steps to create planning graph for the example,
  - Create level 0 from initial problem state.



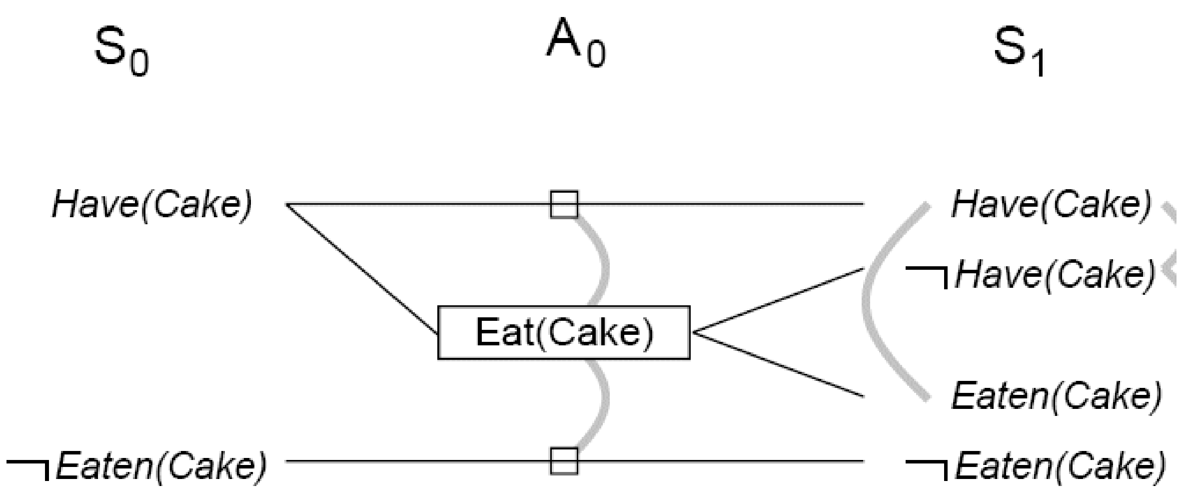
  - Add all applicable actions.
  - Add all effects to the next state.



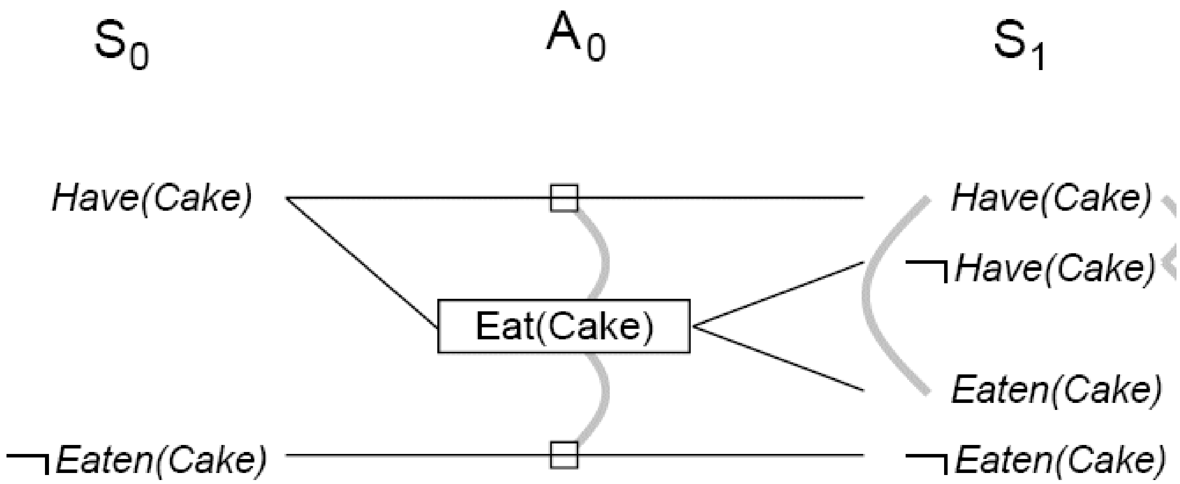
  - Add *persistence actions* (inaction = no-ops) to map all literals in state S_i to state S_{i+1}.



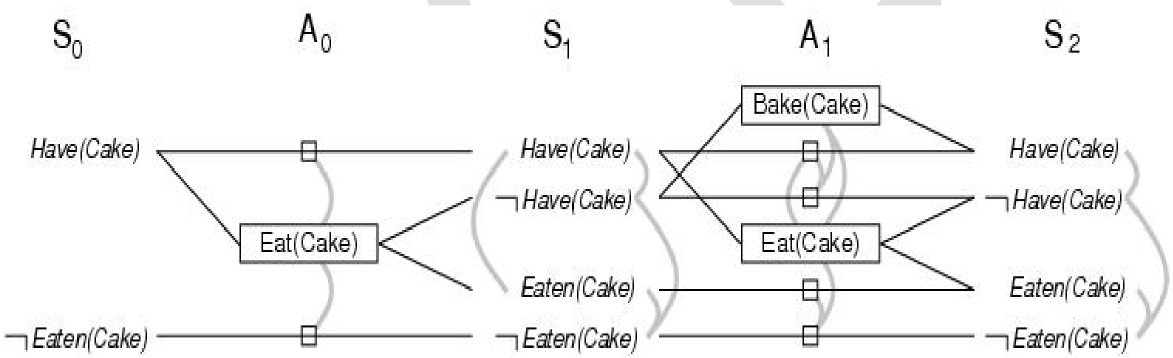
- Identify *mutual exclusions* between actions and literals based on potential conflicts.



- Mutual Exclusion:-
  - A mutex relation holds between **two actions** when:
    - *Inconsistent effects*: one action negates the effect of another.
    - *Interference*: one of the effects of one action is the negation of a precondition of the other.
    - *Competing needs*: one of the preconditions of one action is mutually exclusive with the precondition of the other.
  - A mutex relation holds between **two literals** when:
    - one is the negation of the other OR
    - each possible action pair that could achieve the literals is mutex (inconsistent support).
- Level S_1 contains all literals that could result from picking any subset of actions in A_0
  - Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  - S1 defines multiple states and the mutex links are the constraints that define this set of states.



- Repeat process until graph levels off:
  - two consecutive levels are identical, or
  - contain the same amount of literals (explanation follows later)



- In figure
  - rectangle denotes actions
  - small square denotes persistence actions
  - straight lines denotes preconditions and effects
  - curved lines denotes mutex links

3.3.1 Planning Graphs for Heuristic Estimation:-

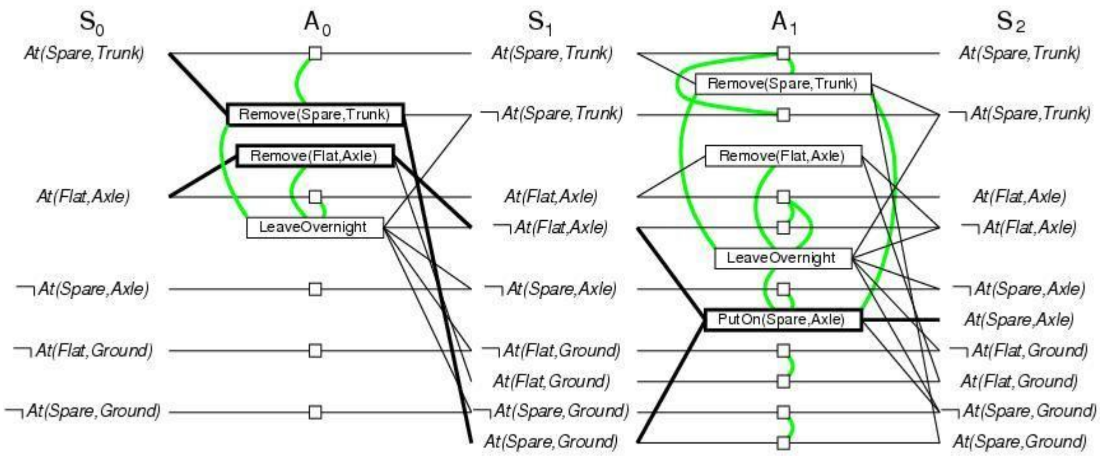
- PG's provide information about the problem
  - PG is a relaxed problem.
  - A literal that does not appear in the final level of the graph cannot be achieved by any plan.
    - H(n) = ∞
  - Level Cost: First level in which a goal appears
    - Very low estimate, since several actions can occur

        - Improvement: restrict to one action per level using *serial PG* (add mutex links between *every* pair of actions, except persistence actions).
- Cost of a conjunction of goals
    - Max-level: maximum first level of any of the goals
    - Sum-level: sum of first levels of all the goals
    - Set-level: First level in which all goals appear without being mutex
- The following is the GraphPlan Algorithm,
- Extract a solution directly from the PG

function GRAPHPLAN(*problem*) **return** *solution* or failure
    graph ← INITIAL-PLANNING-GRAPH(*problem*)
    goals ← GOALS[*problem*]
    loop do
        if *goals* all non-mutex in last level of graph **then do**
            solution ← EXTRACT-SOLUTION(*graph*, *goals*, LENGTH*(graph)*)
            if *solution* ≠ failure **then return** *solution*
            else if NO-SOLUTION-POSSIBLE(*graph*) **then return** failure
        graph ← EXPAND-GRAPH(*graph*, *problem*)

- Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si
- EXPAND-GRAPH also looks for mutex relations
    - Inconsistent effects
        - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and **not** At(Spare, Ground)
    - Interference
        - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and **not** At(Flat,Axle) as EFFECT
    - Competing needs
        - E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and **not** At(Flat, Axle)
    - Inconsistent support
        - E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

- In S2, the goal literals exist and are not mutex with any other
    - Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
    - Initial state = last level of PG and goal goals of planning problem
    - Actions = select any set of non-conflicting actions that cover the goals in the state
    - Goal = reach level S0 such that all goals are satisfied
    - Cost = 1 for each action.



3.3.2 Termination of GraphPlan:-

- Termination? YES
- PG are monotonically increasing or decreasing:
  - Literals increase monotonically: - Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; Once a literal shows up, persistence actions cause it to say forever.
  - Actions increase monotonically:- Once a literal appears at a given level, it will appear at all subsequent levels. This is a consequence of literals increasing; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus will the action
  - Mutexes decrease monotonically:- If two actions are mutex at a given level A_i, then they will also be mutex for all previous levels at which they both appear.
- Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off

3.4 Planning and Acting in the Real World:

- In which we see how more expressive representation and more interactive agent architectures lead to planners that are useful in the real world.
- Planners that are used in the real world for tasks such as scheduling,
  - Hubble Space Telescope Observations
  - Operating factories
  - handling the logistics for military campaigns

3.4.1 Time, Schedules and Resources:

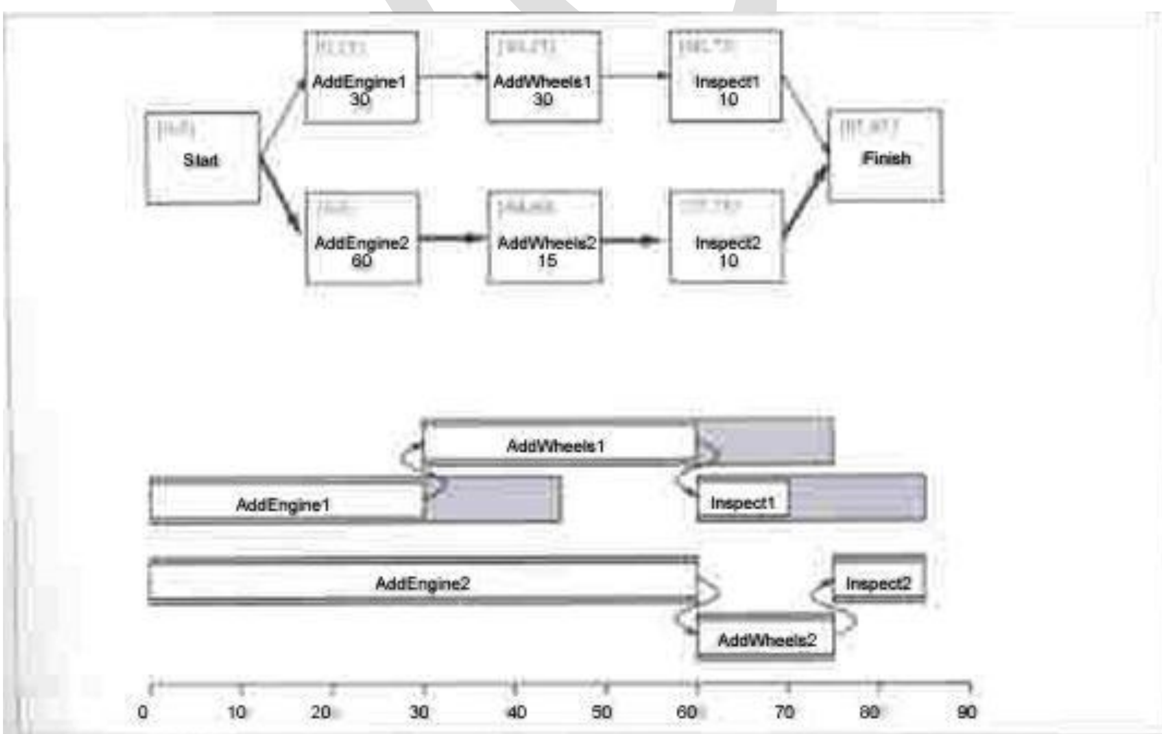
- Time is the essence in the general family of applications called **Job Shop Scheduling.**
- Such a tasks require completing a set of jobs, each of which consists of a sequence of actions, where each action has a given duration and might require some resources.
- The problem is to determine a schedule that minimizes the total time required to complete all the jobs, while respecting the resource constraints.
- For Example:- The following problem is a job shop scheduling.

Init (chassis(C1) $\wedge$ chassis(C2)

$\wedge$ Engine (E1,C1,30) $\wedge$ Engine (E2,C2,60)
$\wedge$ Wheels (W1,C1,30) $\wedge$ Wheels (W2,C2,15))
Goal (Done(C1) $\wedge$ Done(C2))

Action (AddEngine(e,c,m),
PRECOND: Engine(e,c,d) $\wedge$ chassis(c) $\wedge \neg$ EngineIn(c),
EFFECT: EngineIn(c) $\wedge$ Duration (d))
Action (AddWheels(w,c),
PRECOND: Wheels(w,c,d) $\wedge$ chassis(c),
EFFECT: WheelsOn(c) $\wedge$ Duration (d))
Action (Inspect(c),
PRECOND: EngineIn(c) $\wedge$ WheelsOn (c) $\wedge$ chassis (c),
EFFECT: Done (c) $\wedge$ Duration(10))

- The above table shows the Job Shop scheduling problem for assembling two cars.
- The notation Duration (d) means that an action takes d minutes to execute.
- Engine(E1,C1,30) means that E1 is an Engine that fits into chassis C1 and takes 30 minutes to Install
- The problem can be solved by POP (Partial order planning).
- We must now determine when each action should begin and end.
- The following diagram shows the solution for the above problem
- To find the start and end times of each action apply the Critical Path Method CPM.
- The critical path is the one that is the longest and upon which the other parts of the process cannot be shorter than.



- At the top, the solution is given as a partial order plan.

- The duration of each action is given at the bottom of each rectangle, with the earliest and latest start time listed as [ES, LS] in the upper left.
- The difference between these two numbers is the slack of an action
- Action with zero slack are on the critical path, shown with bold arrows.
- At the bottom of the figure the same solution is shown as 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 grey rectangle indicates the slack.
- The following formula serve as a definition for ES and LS and also as the outline of a dynamic programming algorithm to compute them:

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

- The complexity of the critical path algorithm is just O(Nb).
- where N is the number of actions and b is the branching factor.

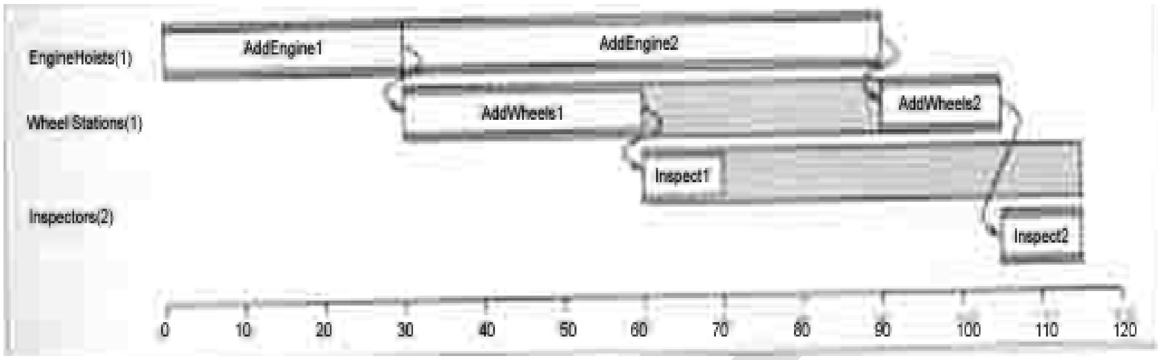
Scheduling with resource constraints:

- Real scheduling problems are complicated by the presence of constraints on resources.
- Consider the above example with some resources.
- The following table shows the job shop scheduling problem for assembling two cars, with resources.

Init (chassis(C1) $\wedge$ chassis(C2)
$\wedge$ Engine (E1,C1,30) $\wedge$ Engine (E2,C2,60)
$\wedge$ Wheels (W1,C1,30) $\wedge$ Wheels (W2,C2,15)
$\wedge$ EngineHoists (1) $\wedge$ WheelStations (1) $\wedge$ Inspectors (2))
Goal (Done(C1) $\wedge$ Done(C2))

Action (AddEngine(e,c,m),
PRECOND: Engine(e,c,d) $\wedge$ chassis(c) $\wedge \neg$ EngineIn(c),
EFFECT: EngineIn(c) $\wedge$ Duration (d)
RESOURCE: EngineHoists (1))
Action (AddWheels(w,c),
PRECOND: Wheels(w,c,d) $\wedge$ chassis(c),
EFFECT: WheelsOn(c) $\wedge$ Duration (d),
RESOURCE: WheelStations (1))
Action (Inspect(c),
PRECOND: EngineIn(c) $\wedge$ WheelsOn (c) $\wedge$ chassis (c),
EFFECT: Done (c) $\wedge$ Duration(10),
RESOURCE: Inspectors (1))

- The available resources are on engine assembly station, one wheel assembly station, and two inspectors.
- The notation RESOURCE: means that the resource r is used during execution of an action, but becomes free again when the action is complete.
- The following diagram shows the solution to the job shop scheduling with resources.



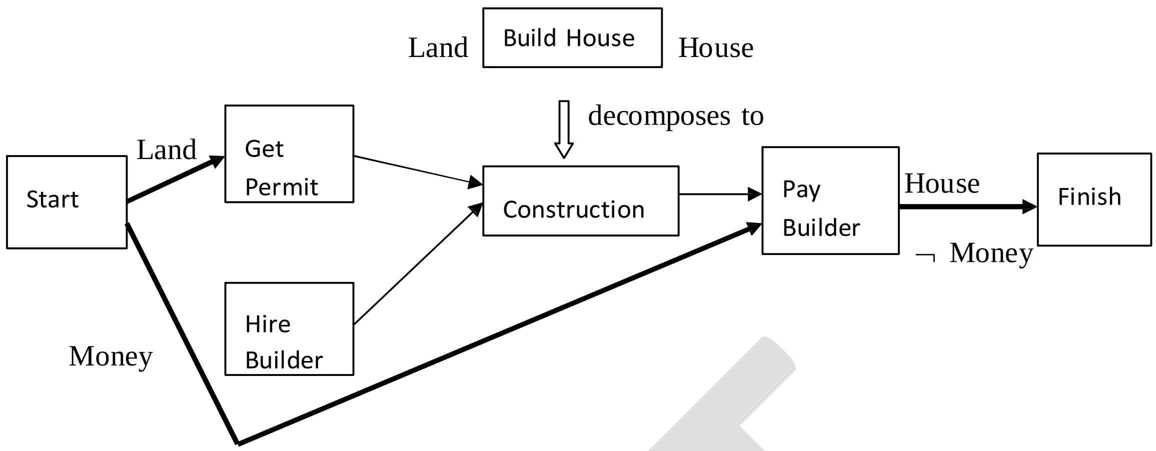
- The left hand margin lists the three resources
- Actions are shown aligned horizontally with the resources they consume.
- There are two possible schedules, depending on which assembly uses the engine station first.
- One simple but popular heuristic is the minimum slack algorithm.
- it schedules actions in a greedy fashion.
- On each iteration, it considers the unscheduled actions that have had all their predecessors scheduled and schedules the one with the least slack for the earliest possible start.
- It then updates the ES and LS times for each affected action and repeats.
- The heuristics is based on the same principle as the most-constrained variable heuristic in constraint satisfaction.

3.4.2 Hierarchical Task Network Planning:

- One of the most pervasive ideas for dealing with complexit y is Hierarchical Decomposition.
- The key benefit of hierarchical structure structure is that, at each level of the hierarchy is reduced to a small number of activities at the next lower level
- So that the computational cost of finding the correct way to arrange those activities for the current problem is small.
- A planning method based on Hierarchical Task Networks or HTNs.
- This approach we take combines ideas from both partial-order planning and the area known as "HTN planning".
- In HTN planning, the initial plan, which describes the problem, is viewed as very high-level description of what is to be done. **For Example: -** Building a House.
- Plans are refined by applying a action decompositions.
- Each action decompositions reduces a high-level action to a partially ordered set of lower-level actions

3.4.2.1 Representing action decompositions:

- The following diagram shows the decomposition of a Building a house action.



- In pure HTN planning, plans are generated only by successive action decompositions.
- Therefore the HTN views planning as a process of making an activity description more concrete, rather than a process of constructing an activity description, starting from the empty activity.
- The action decompositions are represented as, action decompositions methods are stored in a plan library
- From which they are extracted and instantiated to fit the needs of the plan being constructed.
- Each method is an expression of the form Decompose (a, d).
- It means that an action a can be decomposed into the plan d, which is represented as a partial ordered plan.
- The following table shows the action descriptions for the house-building problem and a detailed decomposition for the BuildHouse action.
- The start action of the decomposition supplies all those preconditions of actions in the plan that are not supplied by other actions, such a things called external preconditions.
- In our example external preconditions are land and money.
- Similarly, the external effects, which are the preconditions of Finish, are all those effects of actions in the plan that are not negated by other actions.

Action (BuyLand, PRECOND: Money, EFFECT: Land $\land \neg$ Money)
Action (GetLoan, PRECOND: GoodCredit, EFFECT:Money $\land$ Mortgage)
Action (BuildHouse, PRECOND: Land, EFFECT: House)

Action (GetPermit, PRECOND: Land, EFFECT: Permit)
Action (HireBuilder, EFFECT: Contract)
Action (Construction, PRECOND: Permit $\land$ Contract, EFFECT: HouseBuilt $\land \neg$ Permit)
Action (PayBuilder, PRECOND: Money $\land$ HouseBuilt, EFFECT: $\neg$ Money $\land$ House $\land$ $\neg$ Contract)

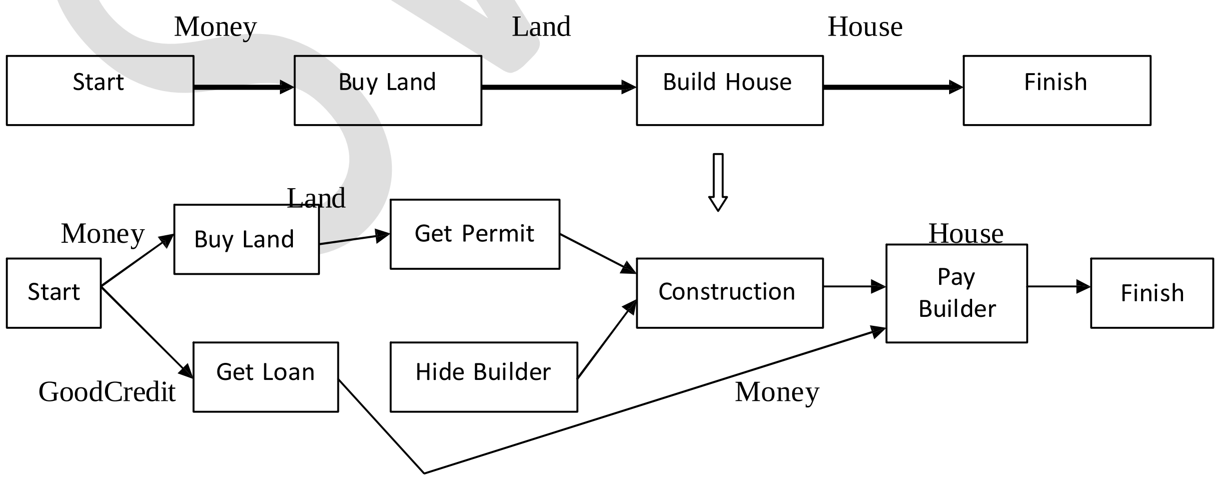
Decompose (BuildHouse,
    Plan (Steps : {S1: GetPermit, S2: HireBuilder, S3: Construction, S4: PayBuilder}
        ORDERINGS: {Start $\prec$ S1 $\prec$ S3 $\prec$ S4 Finish, Start $\prec$ S2 $\prec$ S3},
        Links: {Start $\xrightarrow{Land}$ S1, Start $\xrightarrow{Money}$ S4, S1 $\xrightarrow{permit}$ S3, S2 $\xrightarrow{Contract}$ S3, S3 $\xrightarrow{HouseBuilt}$ S4,
            S4 $\xrightarrow{House}$ Finish, S4 $\xrightarrow{\neg Money}$ Finish}))

- Decomposition should be a correct implementation of the action.

- A plan library could contain several decompositions for any given high-level action.
- Decomposition should be a correct plan, but it could have additional preconditions and effects beyond those stated in the high-level action description.
- The precondition of the high-level action should be the intersection of the external preconditions of its decomposition.
- In which two other forms of information hiding should be noted as,
- First the high-level description completely ignores all internal effects of the decompositions
- Second the high-level description does not specify the intervals “inside” the activity during which the high-level preconditions are effects must hold.
- Information hiding of this kind is essential if hierarchical planning is to reduce complexity.

3.4.2.2 Modifying the planner for decomposition:

- In this we will see how to modify the Partial Order Planning to incorporate HTN planning.
- We can do that by modifying the POP successor function to allow decomposition methods to be applied to the current partial plan P.
- The new successor plans are formed by first selecting some non-primitive action a’ in P and then, for any Decompose (a, d) method from the plan library such that a and a’ unify with substitution θ, replacing a’ with d’ = SUBST (θ, d)
- The following diagram shows the decomposition of a high-level action within an existing plan.
- Where The BuildHouse action is replaced by the decomposition from the above example.
- The external precondition land is supplied by the existing causal link from BuyLand.
- The external precondition Money remains open after the decomposition step, so we add a new action, GetLoan.
- To be more precise follow the below steps,
    - First the action a’ is removed from P.Then for each step S in the decomposition d’
    - Second step is to hook up the ordering constraints for a’ in the original plan to the steps in d’.
    - Third and final step is to hook up casual links.



- This completes the additions required for generating decompositions in the context of the POP Planner.

3.4.3 Planning and Acting in Non-deterministic domains:

- So far we have considered only classical planning domains that are fully observable, static and deterministic.
- Furthermore we have assumed that the action descriptions are correct and complete.
- Agents have to deal with both incomplete and incorrect information.
- Incompleteness arises because the world is partially observable, non-deterministic or both.
- Incorrectness arises because the world does not necessarily match my model of the world.
- The possibility of having complete or correct knowledge depends on how much indeterminacy there in the world.
- **Bounded indeterminacy** actions can have unpredictable effects, but the possible effects can be listed in the action description axioms.
- **Unbounded indeterminacy** the set of possible preconditions or effects either is unknown or is too large to be enumerated completely.
- **Unbounded indeterminacy** is closely related to the **qualification problem.**
- There are four planning methods for handling indeterminacy.
- The following planning methods are suitable for bounded indeterminacy,
  - **Sensorleses Planning:-**
    - Also called as **Confront Planning**
    - This method constructs standard, sequential plans that are to be executed without perception.
    - This algorithm must ensure that the plan achieves the goal in all possible circumstances, regardless of the true initial state and the actual action outcomes.
    - It relies on **coercion –** the idea that the world can be forced into a given state even when the agent has only partial information about the current state.
    - Coercion is not always possible.
  - **Conditional Planning:-**
    - Also called as **Contingency planning**
    - This method constructing a conditional plan with different branches for the different contingencies that could arise.
    - The agent plans first and then executes the plan was produced.
    - The agents find out which part of the plan to execute by including **sensing actions** in the plan to test for the appropriate conditions.
- The following planning methods are suitable for Unbounded indeterminacy,
  - **Execution Monitoring and Replanning:-**
    - In this, the agent can use any of the preceding planning techniques to construct a plan.
    - It also uses **Execution Monitoring** to judge whether the plan has a provision for the actual current situation or need to be revised.
    - **Replanning** occurs when something goes wrong.
    - In this the agent can handle unbounded indeterminacy.
  - **Continuous Planning:-**
    - It is designed to persist over a lifetime.
    - It can handle unexpected circumstances in the environment, even if these occur while the agent is in the middle of constructing a plan.

- It can also handle the abandonment of goals and the creation of additional goals by **goal formulation.**

3.4.4 Conditional Planning:-

- Conditional planning is a way to deal with uncertainty by checking what is actually happening in the environment at predetermined points in the plan.
- Conditional planning is simplest to explain for fully observable environments
- The partially observable case is more difficult to explain in this conditional planning.

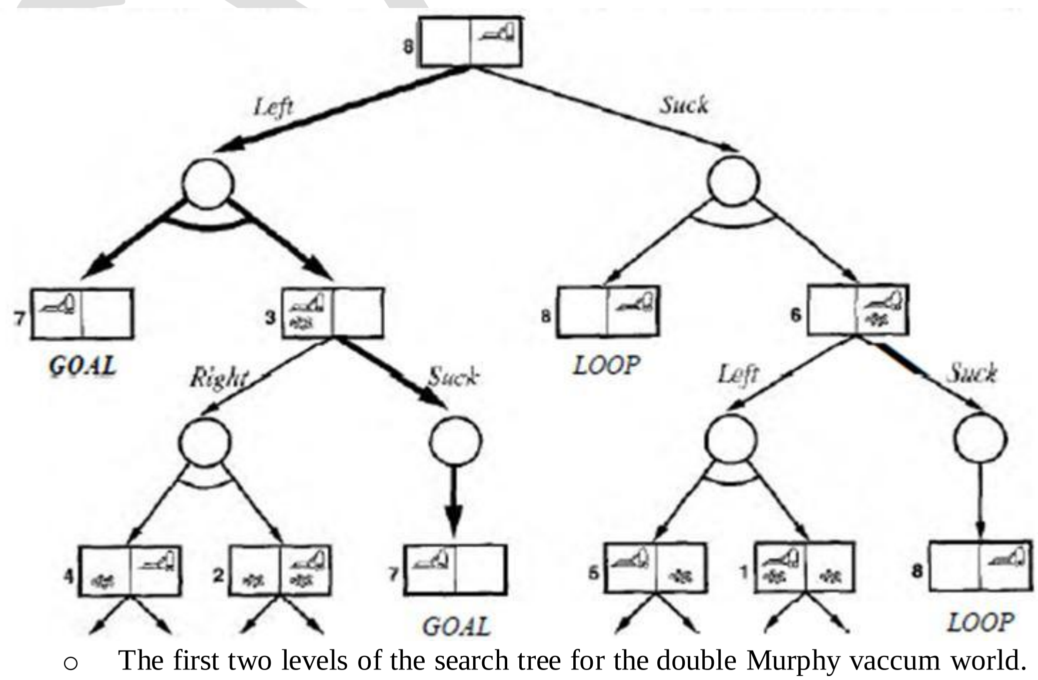
3.4.4.1 Conditional planning in fully observable environments:

- Full observability means that the agent always knows the current state.
- CP in fully observable environments (FOE)
  - initial state : the robot in the right square of a clean world;
  - the environment is fully observable: *AtR ∧CleanL∧CleanR.*
  - The goal state : the robot in the left square of a clean world.
    - Vacuum world with actions *Left*, *Right*, and *Suck*
    - Disjunctive effects: Action (Left, PRECOND : AtR, EFFECT : AtL ∧ ¬ AtR)
    - Modified Disjunctive effects : Action (Left, PRECOND : AtR, EFFECT : AtL v AtR)
    - Conditional effects: Action(Suck, Precond: , Effect: (when AtL: CleanL) ^ (when AtR: CleanR)

      Action (Left, Precond: AtR, Effect: AtL v (AtL ^ when CleanL: !ClearnL)
  - Conditional steps for creating conditional plans:

    if test then planA else planB

    e.g., if AtL ^ CleanL then Right else Suck
  - The search tree for the vacuum world is shown in the following figure



  - The first two levels of the search tree for the double Murphy vaccum world.
  - State nodes are OR nodes where some action must be chosen.

- Chance nodes, shown as circles, are AND nodes where every outcome must be handled, as indicated by the arc linking the outgoing branches.
- The solution is shown as **bold lines** in the tree.

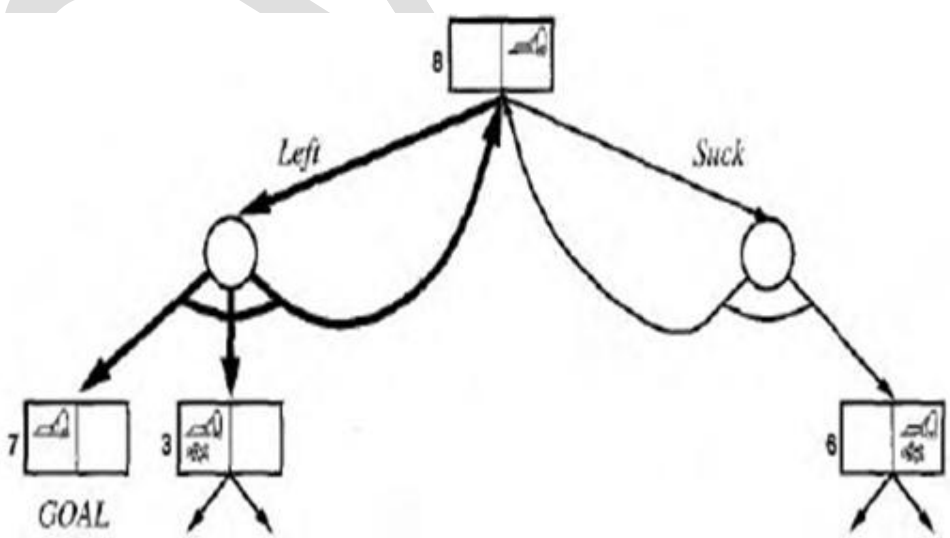
- The following table shows the recursive depth first algorithm for AND-OR graph search.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(INITIAL-STATE[problem], problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if GOAL-TEST[problem](state) then return the empty plan
  if state is on path then return failure
  for each action, state-set in SUCCESSORS[problem](state) do
      plan ← AND-SEARCH(state_set, problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(state_set, problem, path) returns a conditional plan, or failure
  for each s_i in state-set do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan = failure then return failure
  return [if s_1 then plan, else if s_2 then plan, else ... if s_{n-1} then plan,-, else plan,]
```

- The following figure shows the part of the search graph,
- clearly there are no longer any acyclic solutions, and AND-OR-GRAPH-SEARCH would return with failure, there is however a, cyclic solution, which is keep trying Left until it works.



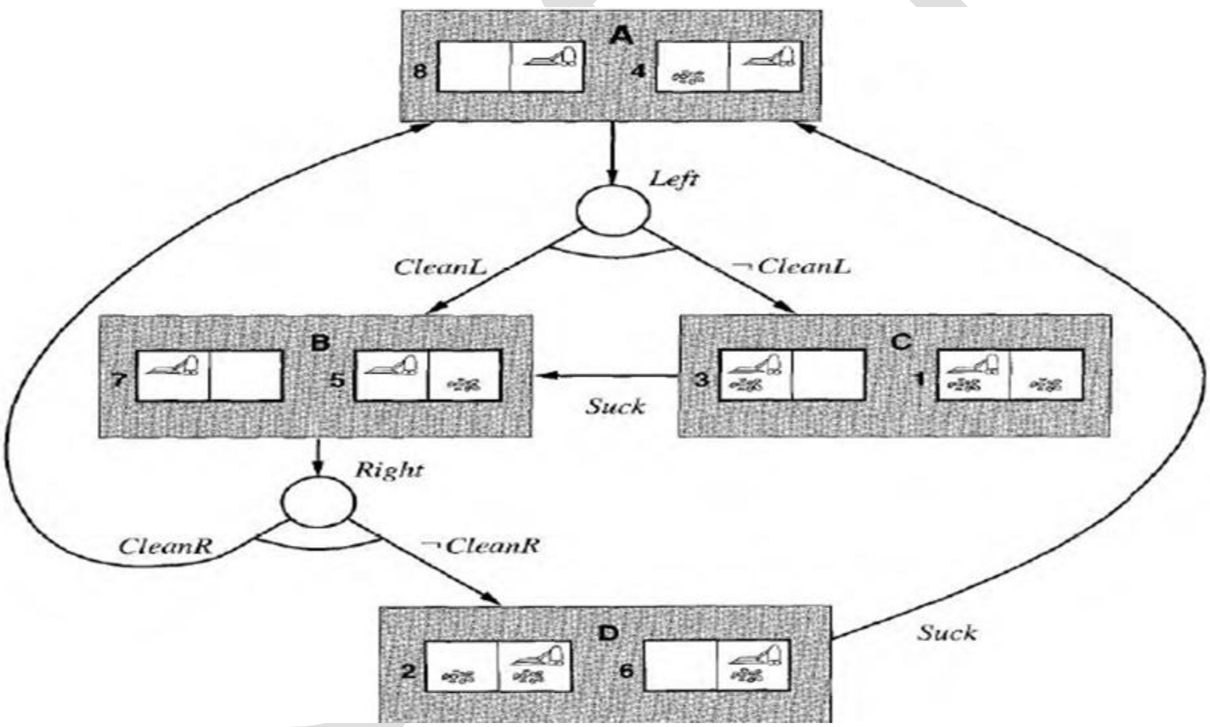
- The first level of the search graph for the triple Murphy vacuum world, where we have shown cycles explicitly.
- All solutions for this problem are cyclic plans.

- The cyclic solution is as follows,

$$[L_1 : \text{ Left, } \textbf{if } AtR \textbf{ then } L_1 \textbf{ else if } CleanL \textbf{ then } [\,] \textbf{ else } Suck]$$

Conditional Planning in partially observable environments

- In the initial state of a partially observable planning problem, the agent knows only a certain amount about the actual state.
- The simplest way to model this situation is to say that the initial state belongs to a **state!set**
- The state set is a way of describing the agents initial belief state.
- Determine "both squares are clean" with local dirt sensing
  - o the vacuum agent is AtR and knows about R, how about L?
- The following graph shows part of the AND-OR graph for the alternate double Murphy vaccum world,
- In which Dirt can sometimes be left behind when the agent leaves a clean square



- The agent cannot sense dirt in other squares.
- Sets of full state descriptions
  - o { *(AtR ⋀ CleanR ⋀ CleanL), (AtR ⋀ CleianR ⋀ ¬CleanL)* }
- Logical sentences that capture exactly the set of possible worlds in the belief state.
  - o *AtR ⋀ CleanR*
- **Knowledge propositions** describing the agent's knowledge

$$K(AtR) \wedge K(CleanR)$$

- **closed-world assumption -** if a knowledge proposition does not appear in the list, it is assumed false.
- Now we need to decide how sensing works.

- There are two choices here,
  - **Automatic sensing:-** Which means that at every time step the agent gets all the variable percepts
  - **Active sensing:-** Which means the percepts are obtained only by executing specific **sensory actions** such as
    - **CheckDirt**
    - **CheckLocation**

Action(Left, PRECOND: AtR,
EFFECT: K(AtL) ⋀¬K (AtR) ⋀ when CleanR: ¬K(CleanR) ⋀
when *CleanL: K (CleanL) ⋀*
when ¬ *CleanL: K(¬ CleanL)) .*

Action(CheckDirt, EFFECT:
when AtL⋀CleanL: K(CleanL) ⋀
when AtL ⋀ ¬CleanL: K (¬CleanL) ⋀
when AtR ⋀ CleanR: K(CleanR) ⋀
when AtR ⋀ ¬CleanR: K(¬CleanR))

3.4.4.2 Execution Monitoring and Replanning:

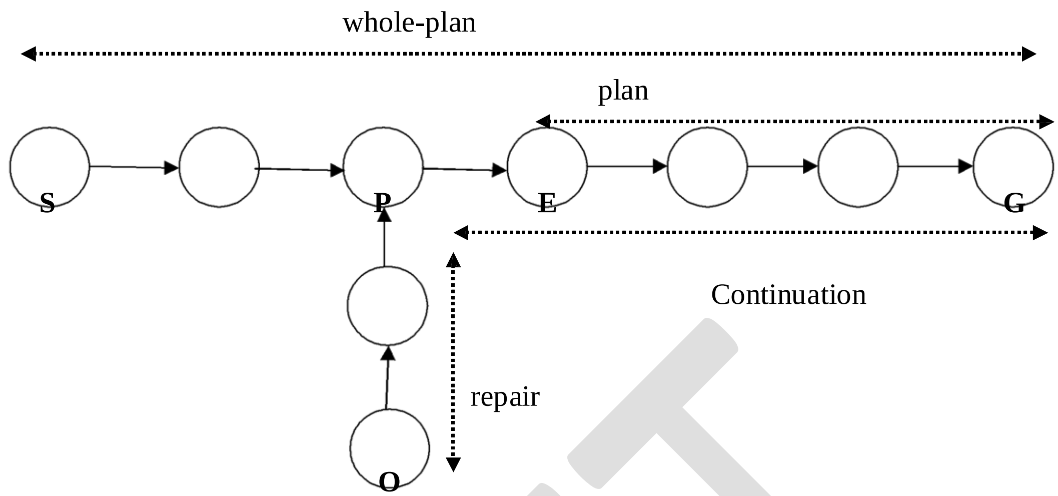
- An execution monitoring agent checks its percepts to see whether everything is going to according plan.
- Murphy's law tells us that even the best-laid plans of mice, men and conditional planning agents frequently fail.
- The problem is unbounded indeterminacy – some unanticipated circumstances will always arise for which the agents action description are incorrect.
- Therefore, execution monitoring is a necessity in realistic environments.
- we will consider two kinds of execution monitoring,
  - Simple, but weak form called action monitoring – whereby the agent checks the environment to verify that the next action will work.
  - more complex, but more effective form called plan monitoring – in which the agent verifies the entire remaining plan.
- A **replanning** agent knows what to do when something unexpected happens, call a planner again to come up with a new plan to reach the goal.
- To avoid spending too much time planning, this is usually done by trying to repair the old plan – to find a way from the current unexpected state back onto the plan
- Together **Execution Monitoring and replanning** form a general strategy that can be applied to both fully and partially observable environments
- It can be applied to a variety of planning representations as state-space, partial-order and conditional plans.
- The following table shows a simple approach to state-space planning.
- The planning agent starts with a goal and creates an initial plan to achieve it.
- The agent then starts executing actions one by one.
- The replanning agent keeps track of both the remaining unexpected plan segment plan and the complete original plan whole-plan
- It uses **action monitoring:** before carrying out the next action of plan, the agent examines its percepts to see whether any preconditions of the plan have unexpectedly become unsatisfied.

- If they have, the agent will try to get back on track by replanning a sequence of actions that should take it back to some point in the whole-plan.
- The following table **has an agent that does action monitoring and replanning**
- It uses a complete state-space planning algorithm called PLANNER as a subroutine.
- If the preconditions of the next action are not met, the agent loops through the possible point p in whole-plan, trying to find one that PLANNER can plan a path to.
- This path is called repair.
- If PLANNER succeeds in finding a repair, the agent appends repair and the tail of the plan after p, to create the new plan.
- The agent then returns the first step in the plan.

```
Function REPLANNING-AGENT(percept)  returns an action
    Static: KB, a Knowledge base (includes action descriptions)
            Plan, a plan, initially []
            Whole-plan, a plan, initially []
             Goal, a goal

    TELL(KB,MAKE-PERCEPT-SENTENCE(percept,t))
    Current ← STATE-DESCRIPTION(KB,t)
    If plan  =  [] then
          whole-plan ← plan ← PLANNER(current,goal,KB)
    If PRECONDITIONS(FIRST(plan))  not currently true in KB then
          Candidates ← SORT(whole-plan, ordered by distance to current)
          Find state s in candidates such that
               Failure   repair ← PLANNER(current,s,KB)
         Continuation ← the tail of whole-plan starting at s
         Whole-plan ← plan ← APPEND(repair, continuation)
    Return POP(plan)
```

- The following diagram shows the schematic illustration of the process.
- The illustration of process is also called as Plan Monitoring.
- The replanner notices that the preconditions of the first action in plan are not satisfied by the current state.
- It then calls the planner to come up with a new subplan called repair that will get from the current situation to some state s on whole-plan.



- Before execution, the planner comes up with a plan, here called whole-plan, to get from **S** to **G**.
- The agent executes the plan until the point Marked **E.**
- Before executing the remaining plan, it checks preconditions as usual and finds that it is actually in state **O** rather than state **E.**
- It then calls its planning algorithm to come up with repair, which is a plan to get from **O** to some point **P** on the original whole-plan.
- The new plan now becomes the concatenation of repair and continuation.
- For example:-
    - Problem of achieving a chair and table of matching color

$$
\begin{aligned}
&Init(Color(Chair, Blue) \wedge Color(Table, Green) \\
&\quad \wedge ContainsColor(BC, Blue) \wedge PaintCan(BC)) \\
&\quad \wedge ContainsColor(RC, Red) \wedge PaintCan(RC)) \\
&Goal(Color(Chair, x) \wedge Color(Table, x)) \\
&Action(Paint(object, color), \\
&\quad \text{PRECOND:} HavePaint(color) \\
&\quad \text{EFFECT:} Color(object, color)) \\
&Action(Open(can), \\
&\quad \text{PRECOND:} PaintCan(can) \wedge ContainsColor(can, color) \\
&\quad \text{EFFECT:} HavePaint(color)
\end{aligned}
$$

- The agents PLANNER should come up with the following plan as,

$$[Start, Open(BC); Paint(Table, Blue); Finish]$$

- If: the agent constructs a plan to solve the painting problem by painting the chair and table red. only enough paint for the chair
- Plan monitoring
  - Detect failure by checking the *preconditions for success* of the *entire remaining* plan
  - Useful when a goal is serendipitously achieved
    - While you're painting the chair, someone comes painting the table with the same color
  - Cut off execution of a doomed plan and don't continue until the failure actually occurs
    - While you're painting the chair, someone comes painting the table with a different color
- If one insists on checking every precondition, it might never get around to actually doing anything
- RP - monitors during execution

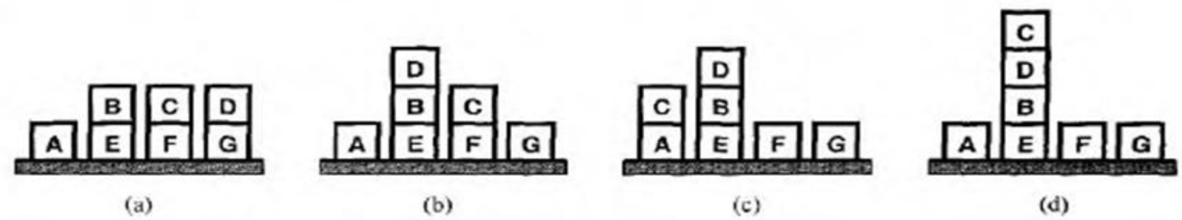
3.4.4.3 Continuous Planning

- Continuous planning agent
  - execute some steps ready to be executed
  - refine the plan to resolve standard deficiencies
  - refine the plan with additional information
  - fix the plan according to unexpected changes
    - recover from execution errors
    - remove steps that have been made redundant
- Goal ->Partial Plan->Some actions-> Monitoring the world -> New Goal
- The continuous planning agent monitors the world continuously, updating its world model from new percepts even if its deliberations are still continuing.
- For example:-
  - use the blocks world domain problem
  - The action we will need is Move(x, y), which moves block x onto block y, provided that both are clear.
  - The following is the action schema,

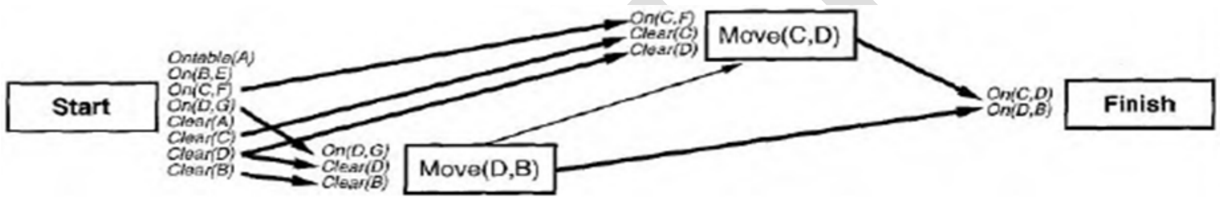
    Action (*Move(x, y),*

    PRECOND: $Clear(x) \wedge Clear(y) \wedge On(x, z)$,

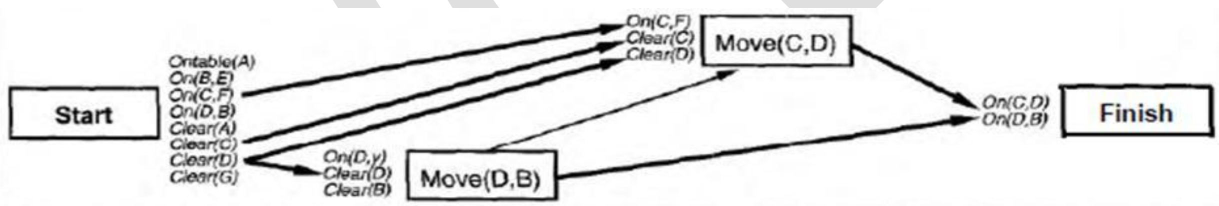
    EFFECT: $On(x, y \wedge Clear(z) \wedge \neg Clear(y) \wedge \neg On(x, z))$
  - Goal: $On(C, D) \wedge On(D, B)$
  - *Start* is used as the label for the current state
  - The following seven diagram shows the continuous planning agent approach towards the goal
  - Plan and execution
    - Steps in execution:
      - Ordering - *Move(D,B),* then *Move(C,D)*
      - Another agent did *Move(D,B) - change the plan*
      - Remove the redundant step
      - Make a mistake, so *On(C,A)*
        - Still one open condition
      - *Planning one more time - Move(C,D)*
      - Final state: start -> finish



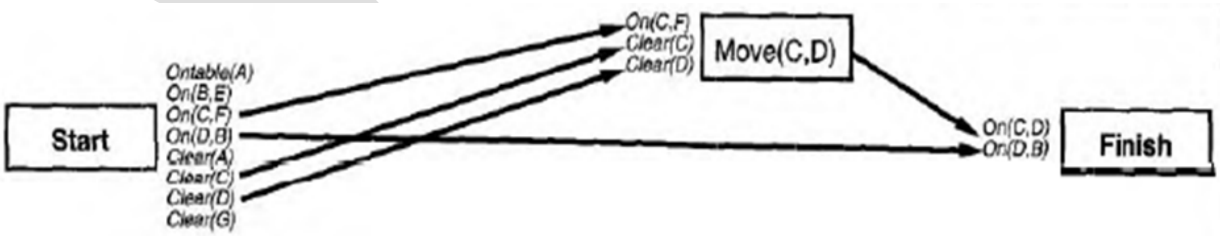
- The sequences of states as the continuous planning agent tries to reach the goal state *On(C, D)* $\wedge$ *On(D ,B) as shown in (d).*
- The start state is (a).
- At (b), another agent has interfered, putting D on B.
- At (c), the agent has executed Move(C, D) but has failed, dropping C on A instead.
- It retries Move(C, D), reaching the goal state (d).



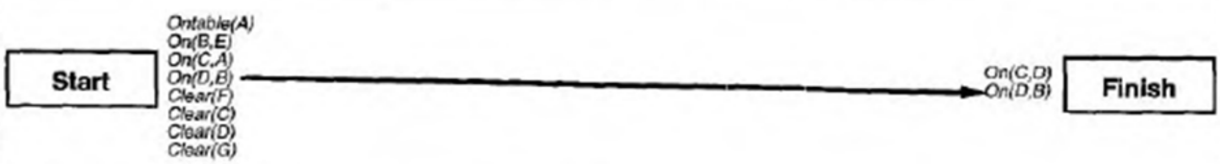
- The initial plan constructed by the continuous planning agent.
- The plan is indistinguishable, so far, from that produced by a normal POP.



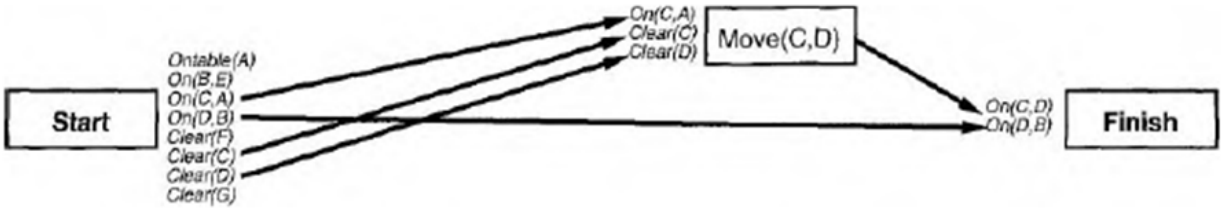
- After someone else moves D onto B, the unsupported links supplying Clear(B) and On(D, G) are dropped, producing this plan.



- The link supplied by Move(D, B) has been replaced by one from Start, and the new-redundant step Move(D, B) has been dropped.



- After Move(C, D) is executed and removed from the plan, the effects of the Start step reflect the fact that C ended up on A instead of the intended D.
- The goal precondition On(C, D) is still open.



- The open condition is resolved by adding Move(C, D) back in.

- After Move(C, D) is executed and dropped from the plan, the remaining open condition On(C, D) is resolved by adding a causal link from the new start step.
- Now the plan is completed.
- From this example, we can see that continuous planning is quite similar to POP.
- On each iteration, the algorithm finds something about the plan that needs fixing a so-called **plan-flaw** and fixes it.
- The POP algorithm can be seen as a flaw-removal algorithm where the two flaws are open preconditions and causal conflicts.
- On the other hand, the continuous planning agent addresses a much broader range of flaws as follows,
  - Missing goals
  - Open precondition
  - Causal conflicts
  - Unsupported links
  - Redundant actions
  - Unexecuted actions
  - Unnecessary historical goal
- The following table shows the continuous-POP-Agent algorithm

Function CONTINUOUS-POP-AGENT (percept) **returns** an action
    Static: plan, a plan, initially with just Start, Finish

    Action ← NoOp (the default)
    EFFECTS [Start] = UPDATE(EFFECTS [Start], percept)
    REMOVE-FLAW (plan) // possibly updating action
    Return action

- It has a cycle of "perceive, remove flaw act"
- It keeps a persistent plan in its KB, and on each turn it removes one ,flaw from the plan.
- It then takes an action and repeats the loop.

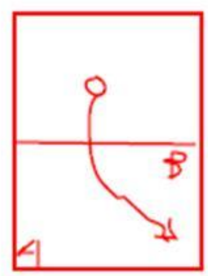
- It is a continuous partial-order planning agent.
- After receiving a percept the agent removes flaw from its constantly updated plan and then returns an action.
- Often it will take many steps of flaw-removal planning, during which it returns NoOp, before it is ready to take a real action.

3.4.4.4 Multiagent Planning

- So far we have dealt with **single-agent environments**
- Multiagent environments can be **cooperative** or **competitive.**
- For example:-
  - the problem is team planning in double tennis.
- Plans can be constructed that specify actions for both players on the team
- Our objective is to construct plans efficiently.
- To do this we need requires some form of **coordination,** possibly achieved by **communication.**
- The following table shows the double tennis problem,

Agents(*A*, *B*) ← declares that there are two agents
Init(*At*(*A*, [*Left*, *Baseline*]) ∧ *At*(*B*, [*Right*, *Net*]) ∧
Approaching(*Ball*, [*Right*, *Baseline*])) ∧ *Partner*(*A*, *B*) ∧ *Partner*(*B*, *A*)
Goal(*Returned*(*Ball*) ∧ *At*(*agent*, [*x*, *Net*]))
Action(*Hit*(*agent*, *Ball*),
PRECOND: *Approaching*(*Ball*, [*x*,*y*]) ∧ *At*(*agent*, [*x*,*y*]) ∧
Partner(*agent*, *partner*) ∧ ¬ *At*(*partner*, [*x*,*y*])
EFFECT: *Returned*(*Ball*))
Action(*Go*(*agent*, [*x*,*y*]),
PRECOND: *At*(*agent*, [*a*,*b*]),
EFFECT: *At*(*agent*, [*x*,*y*]) ∧ ¬ *At*(*agent*, [*a*,*b*]))

Ini



- In the above table, Two agents are playing together and can be in one of four locations as follows,
  - [Left, Baseline]
  - [Right, Baseline]
  - [Left, Net]
  - [Right, Net]
- The ball can be returned if exactly one player is in the right place.

Cooperation: Joint goals and plans

- **An agent (A, B) declares** that there are two agents, A and B who are participating in the plan.
- Each action explicitly mentions the agent as a parameter, because we need to keep track of which agent does what.
- A solution to a multiagent planning problem is a **joint plan** consisting of actions for each agent

- A joint plan is a solution if the goal will be achieved when each agent performs its assigned actions.
- The following plan is a solution to the tennis problem
  - PLAN 1 :
    - *A* : *[Go(A,[Right, Baseline]),Hit(A, Ball)]*
    - *B* : *[NoOp(B),NoOp(B)]*.
- If both agents have the same KB, and if this is the only solution, then everything would be fine; the agents could each determine the solution and then jointly execute it.
- Unfortunately for the agents, there is another plan that satisfies the goal just as well as the first
  - PLAN 2:
    - *A* : *[Go(A, [Left, Net]), NoOp(A)]*
    - *B* : *[Go (B,[Right,baseline]),H it(23, Ball)]*
- 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 probably collide with each other; no one returns the ball and the net may remain uncovered.
- So the agents need a mechanism for **coordination** to reach the same joint plan

Multibody Planning:

- concentrates on the construction of correct joint plans, deferring the coordination issue for the time being, we call this **Multibody planning**
- Our approach to multibody planning will be based on partial-order planning
- we will assume full observability, to keep things simple
- There is one additional issue that doesn't arise in the single-agent case; the environment is no longer truly **static.**
- Because other agents could act while any particular agent is deliberating.
- Therefore we n**eed synchronization**
- We will assume that each action takes the same amount of time and that actions at each point in the joint plan are simultaneous.
- At any point in time, each agent is executing exactly one action.
- This set of concurrent actions is called a **joint action.**
- For example, Plan 2 for the tennis problem can be represented as this sequence of joint actlons:

$$\langle Go(A,[Left,Net]), Go(B,[Right,baseline])\rangle$$
$$\langle NoOp(A), Hit(B,Ball)\rangle$$

Coordination Mechanisms:

- The simplest method by which a group of agents can ensure agreement on a joint plan is to adopt a **convention** prior to engaging in joint activity.
- A convention is any constraint on the selection of joint plans, beyond the basic constraint that the joint plan must work if all agents adopt it
- For example
  - the convention "stick to your side of the court" would cause the doubles partners to select plan 2

  - the convention "one player always stays at the net" would lead them to plan 1
- In the absence of an applicable convention, agents can use communication to achieve common knowledge of a feasible join plan
- For example:
  - a doubles tennis player could shout "Mine!" or "Yours!" to indicate a preferred joint plan.

Competition:

- Not all multiagent environments involve cooperative agents
- Agents with conflicting utility functions are in **competition** with each other
- One example: chess-playing. So an agent must
  (a) recognize that there are other agents
  (b) compute some of the other agent's possible plans
  (c) compute how the other agent's plans interact with its own plans
  (d) decide on the best action in view of these interactions

****

THIRD UNIT-I PLANNING FINISHED

GOOD LUCK

****