Planning

The task of coming up with a sequence of action to achieve a goal is called Planning.

It consists of:

- States - Conjunction of Literals
- Action - It depends on
  - Pre Condition
  - Effects

Two Types of Planning

Classical Planning

Non Classical Planning

classical Planning: The environment which are fully observable, deterministic, finite, static & discrete

Non classical Planning: The environment which is Partially observable or stochastic Environment
The straightforward approach is state space search. Description of action in Planning
Problem specify both preconditions & effects, but in state space it is possible to search
in either forward from the initial state or backward from the Goal. Explicit goal &
action representation to drive heuristic automatically

Forward & Backward Search

**Forward Search**

Planning with Forward State space

search called Progression Planning

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 which are not appearing are False.

(i) The action that are applicable to a state are all these whose preconditions are satisfied.

(ii) The Goal test checks whether the state satisfies the goal of the planning problem.

(iii) The step cost of each action is typically.

Although it would be easy to allow different costs for different action.

Forward Search does not address the irrelevant action problem.
Algorithm
Forward Search
Progress \((\text{State}, \text{goal}, \text{Action}, \text{Path})\)

if state satisfies goal
return Path;
else a = choose \((\text{Action})\)

Preconditions \((a)\) satisfied in state
if no such \(a\), then return false
else return

Progress \((\text{apply}(a, \text{state}), \text{goals, actions, concatenate}(\text{Path}, a)\)\)

Backward Search

Backward Search can be difficult to implement when the Goal States are described by a set of constraints
rather than being listed explicitly.

The advantage of backward search is to consider only relevant action. An action is relevant to a conjunctive goal if it is achieved one of the conjunction of the goal.

Algorithm for Backward:

Regress (IS, Current Goal, Actions, Path)

if IS satisfies Current Goals. then
   return Path
else a = choose (action)  
Si Some effect of a satisfies one of Current Goal
if no such a, then return failure
if some effect of a contradicts some current Goal
then return failure
CG' = CG - effects [a] + Pre Condition [a]
of CG < CG' then return failure
Reg ws (IS, CG', action, concatenate (a, Path))
**GRAPHPLAN:**

The graphplan algorithm repeatedly adds a level to a planning graph with `EXPAND-GRAPH`. Once all the goals show up as non-nested in the graph, `GRAPHPLAN` calls `ExtractSolution` to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

**Algorithm:**

```plaintext
function GRAPHPLAN(problem) returns solution or failure

graph <- InitialPlanningGraph(problem)
goals <- ConjoinGoals(problem, Goal)
for i = 0 to oo do
  if goals all non-nested in St of graph then
    solution <- ExtractSolution(graph, goals, NumLevels), (graph, nogoods)
    if solution != failure then return solution
  if graph and nogoods change both delete off
  then return failure

graph <- Expand-Graph(graph, problem)
```

**Example:**

Init (In Flat) ∧ In (Space) ∧ At (Flat, Axle) ∧ At (Space, Trunk)
Goal (At (Space, Axle))
Action (Remove (obj, loc),
  PRECOND: At (obj, loc),
  EFFECT: ¬ At (obj, loc) ∧ At (obj, Ground))
Action (PutOn (k, Axle),
  PRECOND: In (k) ∧ At (k, Ground) ∧ ¬ At (Flat, Axle),
  EFFECT: ¬ At (k, Ground) ∧ At (k, Axle))
Action (Leave Overnight,
  PRECOND:,
  EFFECT: ¬ At (Space, Ground) ∧ ¬ At (Space, Axle) ∧ ¬ At (Space, Trunk) ∧ At (Flat, Ground) ∧ ¬ At (Flat, Axle) ∧ ¬ At (Flat, Trunk))
In the above problem we have the full component of actions, it is worthwhile to look at some of the examples of mutex relations and their causes.

(i) **Inconsistent Effects:**
Remove (Space, Trunk) is mutex with Leave Overnight because one has effect 
At (Space, Ground) and another has its negation.

(ii) **Interference:**
Remove (Flat, Axle) is mutex with Leave Overnight because one has the precondition At (Flat, Axle) and other

(iii) **Competing Needs:**
PutOn (Space, Axle) is mutex with Remove (Flat, Axle) because one has At (Flat, Axle) as a precondition and other has its negation.

(iv) **Inconsistent Support:**
At (Space, Axle) is mutex with At (Flat, Axle) because only way of achieving At (Space, Axle) is by PutOn (Space, Axle) and that is mutex with the persistent action that is the only way of achieving At (Flat, Axle). Thus mutex relations detect immediate conflict that arises from trying to put two objects in same place at same time.

**Formal Solution:**
We can formulate a solution as a Boolean constraint satisfaction problem (CSP) whose variables are actions at each state, the values for are in or out of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.
Alternatively, we can define \textit{Exhaustive Solution} as a backward search problem, where each state in search contains a pointer to a level in planning graph and a set of unsatisfied goals.

**Termination of Graphplan:**

Graphplan will in fact terminate and return failure when there is no solution. We can eliminate with failure because there is no possibility of a subsequent change that could add a solution.

The properties are:

(i) \textbf{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 action cause it to stay forever.

(ii) \textbf{Actions increase monotonically:}

Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of monotonic increase of literals.

(iii) \textbf{Mutual decrease monotonically:}

If two actions are mutual at a given level, then they will also be mutual at all previous levels at which they both appear.
The following diagram describes about the planning graph for space tree epistemic after expansion to level $S_2$.

- Maker lines are shown in double lines.
- Solution is indicated by bold lines & outlines.

**Planning Graph for Chemist's Estimation:**

A planning graph, once constructed, is a rich source of information about a problem. First, if any goal literal fails to appear in final level of graph then problem is unsolvable.

Second, we can estimate cost of achieving any goal literal $g_i$ from state $s$ at the level at which $g_i$ first appears in planning graph constructed from initial state $s$. We call this as level cost of $g_i$. 

www.studentsfocus.com
Approaches:

The following use the three approaches to estimate the cost of a conjunction of goals:

1. Max level
2. Level sum
3. Set level

(i) Max level:
The max-level heuristic simply takes the maximum level cost of any of goals, this is admissible but not necessarily accurate.

(ii) Level sum:
In the level sum heuristic, the following subgoals are due to be independent assumption, results the sum of level cost of goals, this can be inadmissible but works well in quadratic for problems that are largely decomposable.

(iii) Set level:
The set-level heuristic finds the level at which will literals in the conjunction of goal appear in the planning graph without any pair of them being mutually exclusive.
Hierarchical Task Networks (HTN) Planning:

- In Hierarchical Task Networks (HTN) planning, it has some primitive actions. The following are the primitive actions in HTN planning:
  - Full Observability and Determinism
  - Availability of set of actions
  - Standard precondition-effect schemas.

- The Hierarchical Decomposition is implemented in this planning, is to reduce the complexity.
- At each level of hierarchy, a computational task is reduced to a small number of activities at the next lower level.
- The computational task of arranging these activities is low.

 Hierarchical Task Network (HTN) planning uses a refinement of actions through decomposition.

Types of HTN Planning:

There are two types of HTN planning. They are:

- Pure HTN Planning
- Hybrid HTN Planning
Hierarchical Planning:

⇒ Hierarchical Planning is a plan at higher level of abstraction.
⇒ Here, the planning can occur both before and during the execution of plan.
⇒ In Hierarchical Planning, particular action will remain at an abstract level prior to the execution phase.
⇒ In Hierarchical Planning, we concentrate on the aspect of Hierarchical Decomposition.

Hierarchical Decomposition:

⇒ Hierarchical Decomposition is an idea that pervades almost all attempts to manage complexity.
⇒ Armies operate on hierarchy of units.
⇒ Complex software is created from a hierarchy of subroutines or object classes.
⇒ The basic formalism we adopt to understand hierarchical Decomposition comes from the area of Hierarchical Task Networks (HTN) planning.
Start action supplies all preconditions of actions, not supplied by other actions

Finish action has all effects of actions not present in other actions

Example: "Building a House"

Action (BuyLand, PRECOND: Money, EFFECT: Land \land Money)
Action (GetLoan, PRECOND: GoodCredit, EFFECT: Money \land Mortgage)
Action (BuildHouse, PRECOND: Land, EFFECT: House)
Action (GetPermit, PRECOND: Land, EFFECT: Permit)
Action (HireBuilder, PRECOND: Contract)
Action (Construction, PRECOND: Permit \land Contract, EFFECT: House Built \land Permit)
Action (PayBuilder, PRECOND : Money ∧ House Built,  
EFFECT: 7Money ∧ House ∧ 7 (Contract)  

Decompose (Build House, Plan : :  
STEPS { S1: Get Permit, S2: Hire Builder, S3: Construction,  
S4: Pay Builder }  
ORDERINGS: { Start < S1 < S3 < S4 < Finish, Start < S2 < S3 }  
LINKS { Start → S1, Start → S4, S1 → Permit → S3, S2 → Contract → S3, S3 → S4, House Built → S4  
S4 → Finish, S4 → 7Money → Finish }  

High level actions:  
⇒ The key additional concept in the Hierarchical Task Network is High-level action (HLA).  
⇒ Each HLA has one or more possible refinements into a sequence of actions, each of which may be an HLA or a primitive action.  
For Eg:  
"Build House" is an action, which has many refinements such as Get Permit, Hire Builder, Construction, Pay Builder in the example mentioned above.  
⇒ High level Plan achieves the goal from a given state, if atleast one of its implementations achieves the goal from that state.
Partially Ordered Planning:

⇒ Partially ordered plan is a set of actions and a set of constraints of the form Before (a_i, a_j), saying that one action occurs before another.

⇒ Partially ordered plans are created by a search through the space of plans rather than through the state space.

⇒ It starts with the empty plan consisting of just the initial state and Goal.

⇒ It has no actions in between.

Planning to be possible in four Components:

⇒ The partially Ordered planning could be possible by using the following four components:

1. Action:
   * It makes up the step, by using the two dummy actions Start, Stop.

2. Ordering Constraint:
   * The Ordering Constraint in the Partially Ordered Planning is represented as below:
   
   \[ A \leq B \text{ (A before B)} \]

   * The constraint states that, A must be executed before B. But, not necessarily immediately.
3. Casual Link -

Casual Link in Partially Ordered Planning is represented as below:

\[ A \xrightarrow{p} B \quad (A \text{ achieves } p \text{ for } B) \]

4. Open precondition:

* A set of preconditions are open, if they are not achieved by some actions in the plan.

Example:

Let us consider the spare tire problem for an example.

(i) The tire problem expressed as an empty plan:

```
At (Spare, Trunk)    At (Spare, Trunk)    Finish
Start                At (Flat, Ankle) At (Spare, Trunk)
```

(ii) An incomplete partially ordered plan for the tire problem:

```
At (Spare, Trunk) Remove (Spare, Trunk) At (Spare, Trunk) Puton (Spare, Ankle) At (Spare, Ankle) Finish
Start At (Spare, Trunk) At (Flat, Ankle)
```

(Boxes represent Actions and arrows indicate that one action must occur before another)
Finding Solution for a problem using Partially Ordered Planning:

1. Detecting Flaws:
   - The search procedure looks for a flaw in the plan, and makes an addition to the plan to correct the flaw.
   - A flaw is anything that keeps the partial plan from being a solution.

For Eg:
- One flaw present in the empty plane (i) is, no action achieves At (Space, Axle).
- One way to correct the flaw is to insert into the plan, the action Put On (Space, Axle).
- It also introduces some new flaws, "The preconditions of the new action are not achieved."
(ii) Removing the flaw:

⇒ For removing the flaw, we make the least commitment possible to fix the flaw.

For Eg:

⇒ In adding the action Remove (Spaie, Trunk) we need to commit, to having it occur before PutOn (Space, Axle)

⇒ But, we make no other commitment that places it before or after other action.

(iii) Backtracking (if necessary):

⇒ If again there exists any flaw, move to the step 1 (Detecting Flaw)

⇒ If no flaw exists, then return the solution of Partially Ordered Planning.
Artificial Intelligence.

Planning and acting in non-deterministic domains:

* In an uncertain environment, agent must use its perceptions to discover what is happening while the plan is being executed.
* Agents have to deal with both incomplete and incorrect information.
* Incompleteness arises because the world is partially observable, non-deterministic.
* Incorrectness arises because the world does not necessarily match my model of the world.
* The possibility of having complete or correct knowledge represents/depends on how much indeterminacy is there in the world.
* Bounded Indeterminacy: Actions can have unpredictable effects.
* Unbounded Indeterminacy: The set of possible preconditions or effects either is unknown or too large to be enumerated completely.

Four planning methods for handling indeterminacy:

1. Sensorless Planning
2. Conditional Planning
3. Execution monitoring & replanning
4. Continuous Planning

Sensorless Planning:

* Also called as Conformant Planning.
* It constructs standard, sequential plans that are to be executed without perceptions.
* This planning algorithm must ensure that the plan achieves the goal in all possible circumstances.
* This relies on coercion.
Constrain:
The idea that the world can be forced into a given state even when the agent has only partial information about the current state. Sensoryless planning is often inapplicable due to coercion.

Conditional Planning:
* Also called as contingency planning
* It constructs a conditional plan with different branches for the different contingencies that could arise.
* The agent finds out which part of the plan to execute by including sensing actions in the plan to test for the appropriate conditions.
* The agent plans first and executes the plan that was produced.

Execution, Monitoring and Replanning:
* In this planning, the agent can use any of the planning preceding techniques to construct a plan.
* It uses execution monitoring to judge whether the plan has a provision for actual current situations or to be revised again.
* Replanning occurs when something goes wrong.

Continuous Planning:
* Continuous planning is designed to persist over a lifetime.
* It can handle unexpected circumstances in the environment, even if these occur in the middle of constructing a plan.
* It also handles abandonment of goals and the creation of additional goals by goal formulations.
Critical path Method (CPM):

1. To minimize a makespan, find the earliest start time for all the actions consistent with the ordering constraint. [Ordering constraints as a directed graph related to their action.]

CPM: CPM is a graph to determine the possible start and end times of each action.

> A path through a graph representing a partial-ordered plan in a linear ordered sequence beginning with 'start' and ending with 'finish'.

> In partial ordered structure: a plan is a set of actions and set of constraints of a form before (ai, aj). It causes some ambiguity. To overcome this ambiguity and minimize the makespan by using critical path method.

Critical Path: Critical path is a path, whose total duration is longest, the path is said to be critical because it determines the duration of the entire plan.

> Delaying the start of any action on the critical plan slow down the whole plan. Critical path have a window of time where they executed.

Slack: The window is specified in terms of an earliest possible start time (ES) and latest possible start time (LS). The quantity LS-ES is known as slack of an action. ES and LS times for an action on the critical path constitute a schedule for the problem.
The definition for ES and LS, and also as the outline of dynamic-programming algorithm to compute them:

\[ ES(\text{Start}) = 0 \]
\[ ES(A) = \max_{A \leq B} ES(A) + \text{Duration}(A) \]
\[ LS(\text{Finish}) = ES(\text{Finish}) \]
\[ LS(A) = \min_{B > A} LS(B) - \text{Duration}(A) \]

Example:

Jobs: [AddEngine1 < AddWheels1 < Inspect1], [AddEngine2 < AddWheels2 < Inspect2]

Resources: (Engine Hold (1), WheelStation (1), Inspectors (2), Lug Nuts (500))

Action (AddEngine1, Duration: 30, Use: Engine Hold (1))
Action (AddEngine2, Duration: 30, Use: Engine Hold (1))
Action (AddWheel1, Duration: 10, Use: WheelStation (1))
Action (AddWheel2, Duration: 15, Use: WheelStation (1))
Action (Inspect1, Duration: 10, Use: Inspectors (1))

* The complexity of critical path algorithm is just \( O(Nb) \) where \( N \) is the number of actions and \( b \) is the maximum branching factor into or out of an action.