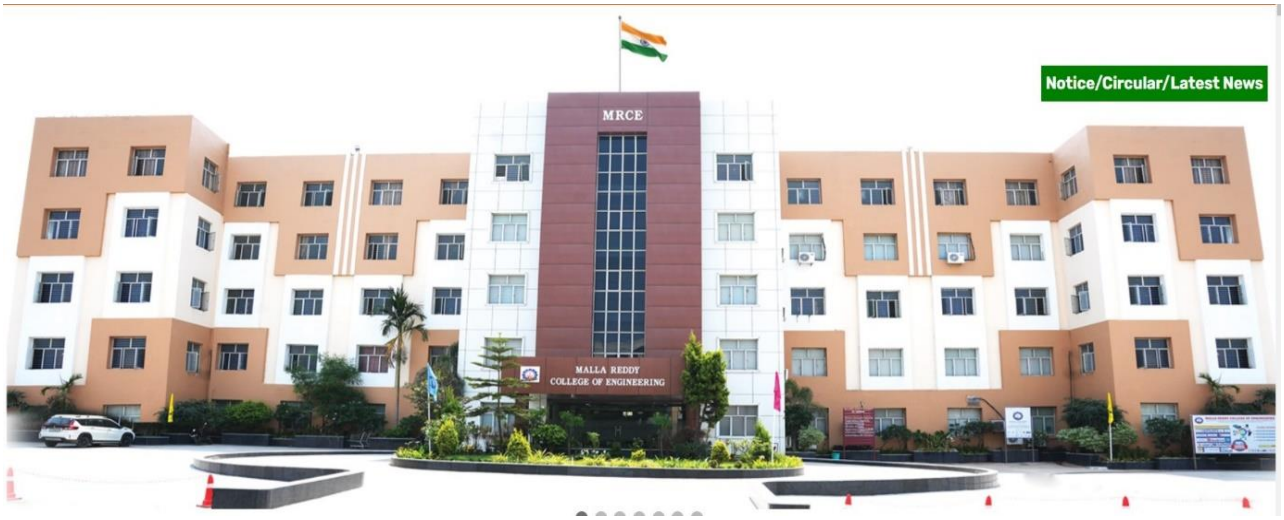
MALLA REDDY COLLEGE OF ENGINEERING

(Approved by AICTE(New Delhi), Permanently Affiliated to JNTUH)

Recognised under Section 2(f) & 12(B) of the UGC Act 1956, An ISO 9001:2015 Certified Institution.

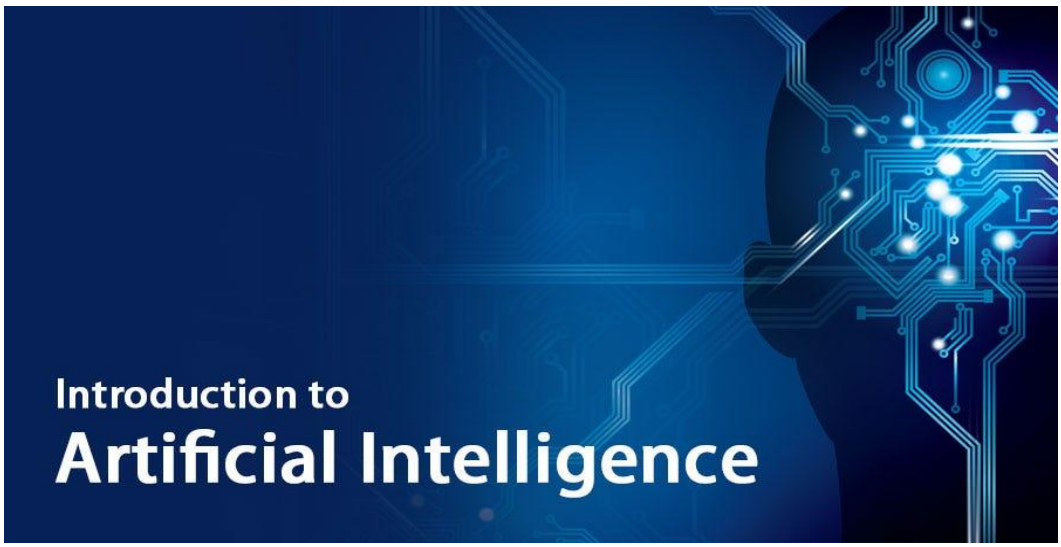
NATIONAL BOARD OF ACCREDITATION (CSE & ECE)



Subject: Introduction to Artificial Intelligence

Class Notes: 4th UNIT

Topic: Planning



Prepared By

Dr K Madan Mohan Ph. D. (JNTUH), MISTE, MIEEE
Associate Professor,
Department of CSE (AI&ML)
MallaReddy College Of Engineering (MRCE)
Maisammaguda, Dhulapally, post via Kompally, Secunderabad - 500100

IAI- Class Notes

UNIT-4

Planning

Syllabus: Definition of Classical Planning, Algorithms for Planning with State Space Search, Planning Graphs, Other Classical Planning Approaches, Analysis of Planning Approaches, Hierarchical Planning

Topic-1. Classical Planning

Classical planning involves agents devising action plans to achieve goals efficiently. It uses structured representations of problems that enable scaling to complex scenarios. Key elements include:

- A defined language for problem representation.
- Utilization of forward and backward search algorithms with effective heuristics.
- Planning graphs for enhancing plan search efficiency.
- Focus on fully observable, deterministic environments with single agents, contrasting with more complex scenarios covered in later chapters.

1.1 DEFINITION OF CLASSICAL PLANNING:

1. Classical planning finds sequences of actions to reach a goal state, relying on domain-specific heuristics due to its atomic state representation.
2. In contrast, the hybrid propositional logical agent uses domain-independent heuristics but requires ground propositional inference, which can become overwhelming with many actions and states.
3. To address these challenges, planning researchers use factored representations where states are described using variables. PDDL (Planning Domain Definition Language) is a standard for expressing initial states, available actions, action outcomes, and goal tests, using a fluent-based representation.
4. PDDL evolved from the STRIPS planning language, which was less flexible as it couldn't handle negative literals in preconditions and goals.

   In planning, actions are defined using schemas that specify what changes when an action is executed, solving the frame problem by focusing on changes rather than everything that stays the same.

   For instance, flying a plane from one airport to another can be represented as:

   Action (Fly(p, from, to),

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

   EFFECT: ¬At(p, from) ∧ At(p, to))

Here, PRECOND lists conditions necessary for the action, and EFFECT describes changes after execution. Actions can be generalized (lifted) or instantiated for specific objects. Resulting states after executing actions are determined by adding and removing fluents (state components). This approach simplifies reasoning and planning in dynamic environments.

- Planning involves defining a domain with action schemas. A specific problem within this domain adds an initial state and a goal.
- The initial state is a set of true atoms under the closed-world assumption. The goal is a set of positive or negative literals with variables treated as existentially quantified.
- The aim is to find a sequence of actions leading from the initial state to a state that satisfies the goal.

1.2 Example: Air cargo transport

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

The Explanation of the above Algorithm:

1. **Initial State (Init)**:
    - C1 and C2 are initially at airports SFO and JFK respectively.
    - P1 and P2 are initially at airports SFO and JFK respectively.
    - C1 and C2 are cargoes.
    - P1 and P2 are planes.
    - JFK and SFO are airports.
2. **Goal State (Goal)**:
    - C1 should be at JFK.

- C2 should be at SFO.

3. **Actions**:
    - **Load(c, p, a)**: Load cargo c onto plane p at airport a.
        - Preconditions (PRECOND): Cargo c and plane p must be at airport a.
        - Effects (EFFECT): Cargo c is no longer at airport a, and cargo c is now in plane p.
    - **Unload(c, p, a)**: Unload cargo c from plane p at airport a.
        - Preconditions (PRECOND): Cargo c must be in plane p, and plane p must be at airport a.
        - Effects (EFFECT): Cargo c is now at airport a, and cargo c is no longer in plane p.
    - **Fly(p, from, to)**: Fly plane p from airport from to airport to.
        - Preconditions (PRECOND): Plane p must be at airport from.
        - Effects (EFFECT): Plane p is no longer at airport from, and plane p is now at airport to.
4. These actions allow the transportation of cargoes (C1 and C2) between airports (JFK and SFO) using planes (P1 and P2), aiming to achieve the specified goal configuration.
5. These actions define how the state of the world changes based on the preconditions being met and the effects that occur after each action is executed.
6. An air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load, Unload, and Fly.

The actions affect two predicates:

- In(c,p) means that cargo c is inside plane p, and At(x, a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the At predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it.
- In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be A_t anywhere when it is In a plane; the cargo only becomes At the new airport when it is unloaded. So A_t really means "available for use at a given location."

The following plan is a solution to the problem:

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

Finally, there is the problem of spurious actions such as Fly(P1,JFK,JFK), which should be a no-op, but which has contradictory effects (according to the definition, the effect would include At(P1,JFK) ∧¬At(P1,JFK)). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the from and to airports must be different;

1.3 Example: The spare tire problem

$$
\begin{array}{l}
Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk)) \\
Goal(At(Spare, Axle)) \\
Action(Remove(obj, loc), \\
\quad \text{PRECOND: } At(obj, loc) \\
\quad \text{EFFECT: } \neg At(obj, loc) \wedge At(obj, Ground)) \\
Action(PutOn(t, Axle), \\
\quad \text{PRECOND: } Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle) \\
\quad \text{EFFECT: } \neg At(t, Ground) \wedge At(t, Axle)) \\
Action(LeaveOvernight, \\
\quad \text{PRECOND:} \\
\quad \text{EFFECT: } \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk) \\
\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))
\end{array}
$$

Figure 10.2 The simple spare tire problem.

The Explanation of the above algorithm:

1. **Initial State:**
   - Flat tire is on the axle.
   - Spare tire is in the trunk.
   - Flat tire and spare tire are initially present.
2. **Goal:**
   - Move the spare tire from the trunk to the axle.
3. **Actions:**
   - **Remove(obj, loc):** Removes an object (tire) from a location (axle or trunk) to the ground.
     - Preconditions: The object must be at the specified location.
     - Effects: Moves the object from its initial location to the ground.
   - **PutOn(t, Axle):** Puts a tire onto the axle.

        - Preconditions: The tire must be on the ground and not currently on the axle.
        - Effects: Moves the tire from the ground onto the axle.
    - **LeaveOvernight:**
        - Preconditions: None (can be performed anytime).
        - Effects: Moves both tires (spare and flat) out of all locations (axle, trunk, and ground), assuming they were initially there.
4. These actions aim to manipulate the tire positions to achieve the goal of having the spare tire on the axle, considering the initial and potential states after each action.
5. Consider the 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. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications.

There are just four actions:

a) Removing the spare from the trunk,

b) Removing the flat tire from the axle,

c) putting the spare on the axle,

d) and leaving the car unattended overnight.

- We assume that the car is parked in a particularly bad neighbourhood so that the effect of leaving it overnight is that the tires disappear.
- A solution to the problem is [Remove (Flat, Axle), Remove (Spare, Trunk), PutOn (Spare, Axle)].

1.4 Example: The blocks world BLOCKS WORLD

The blocks world is a classic planning domain with cube-shaped blocks that can be stacked on a table. A robot arm can move one block at a time, either placing it on the table or on top of another block. Key predicates include On(b, x) indicating block b is on x, and Clear(x) meaning x has no blocks on it.

To move block b from x to y:

- **Preconditions:** On(b, x) ∧ Clear(b) ∧ Clear(y)
- **Effects:** On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y)

To handle moving a block to the table:

- **Preconditions:** On(b, x) ∧ Clear(b)
- **Effects:** On(b, Table) ∧ Clear(x) ∧ ¬On(b, x)

Clear(x) ensures there's space on x for a block. Introducing Block(b) ensures correct use of Move actions to maintain consistency and prevent unnecessary search space.

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

Explanation of Above Algorithm:

Initial State:

- A, B, and C are blocks.
- A and B are on the table, C is on A.
- B and C are clear.

Goal State:

- A on B, and B on C.

Actions:

- **Move(b, x, y):**
  - Precondition: Block `b` is on top of `x`, both `b` and `y` are clear.
  - Effect: Move `b` from `x` to `y`, making `b` on top of `y` and clearing `x`.
- **MoveToTable(b, x):**
  - Precondition: Block `b` is on top of `x`, `b` is clear.

- Effect: Move `b` from `x` to the table.

Solution Sequence:

1. Move block C from A to the table.
2. Move block B from the table to C.
3. Move block A from the table to B.

This sequence achieves the goal where block A is placed on top of block B, and block B is placed on top of block C, following the specified conditions and actions in the blocks world.

1.5 The complexity of classical planning

1.5.1 PlanSAT and Bounded PlanSAT:

- **PlanSAT:** Determines if there exists any plan for a given planning problem.
- **Example:** Finding a sequence of actions to move from one location to another in a grid, considering obstacles and movement costs.
- **Bounded PlanSAT:** Checks if there exists a solution of length kkk or less for a planning problem.
- **Example:** Finding the shortest sequence of steps to assemble a piece of furniture with a limited number of assembly steps allowed.

1.5.2 Complexity Class PSPACE:

- Both PlanSAT and Bounded PlanSAT are in PSPACE, which means they require polynomial space on a deterministic Turing machine.

1.5.3 Domain-Specific Challenges:

- In specific domains like the blocks-world or air cargo logistics:
  - Bounded PlanSAT can be NP-complete, like finding the shortest route to deliver all packages using a limited number of flights.
  - PlanSAT can be solvable in polynomial time (P), such as finding any valid sequence of moves to stack blocks without exceeding a certain height.
- **Key Insight:** While finding the optimal plan is generally hard, finding a good enough solution (sub-optimal planning) can be more feasible. Effective search heuristics are crucial for efficiently solving these problems.

2. ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Planning algorithms can be viewed as a search problem where we navigate through the space of states from the initial state to the goal state. Using action schemas, we can also search backward from the goal state to find the initial state. Figure 10.5 illustrates the comparison between forward and backward searches.

2.1 Forward (progression) state-space search:

A planning problem can be solved using heuristic search or local search algorithms. Forward state-space search was initially inefficient, but strong domain-independent heuristics can now be derived automatically, making forward search practical.

Example:

Imagine planning a route for a robot in a warehouse. Initially, without good heuristics, finding the best path was impractical. With automatically derived heuristics, the robot can now efficiently plan its route by estimating the remaining distance to the goal.

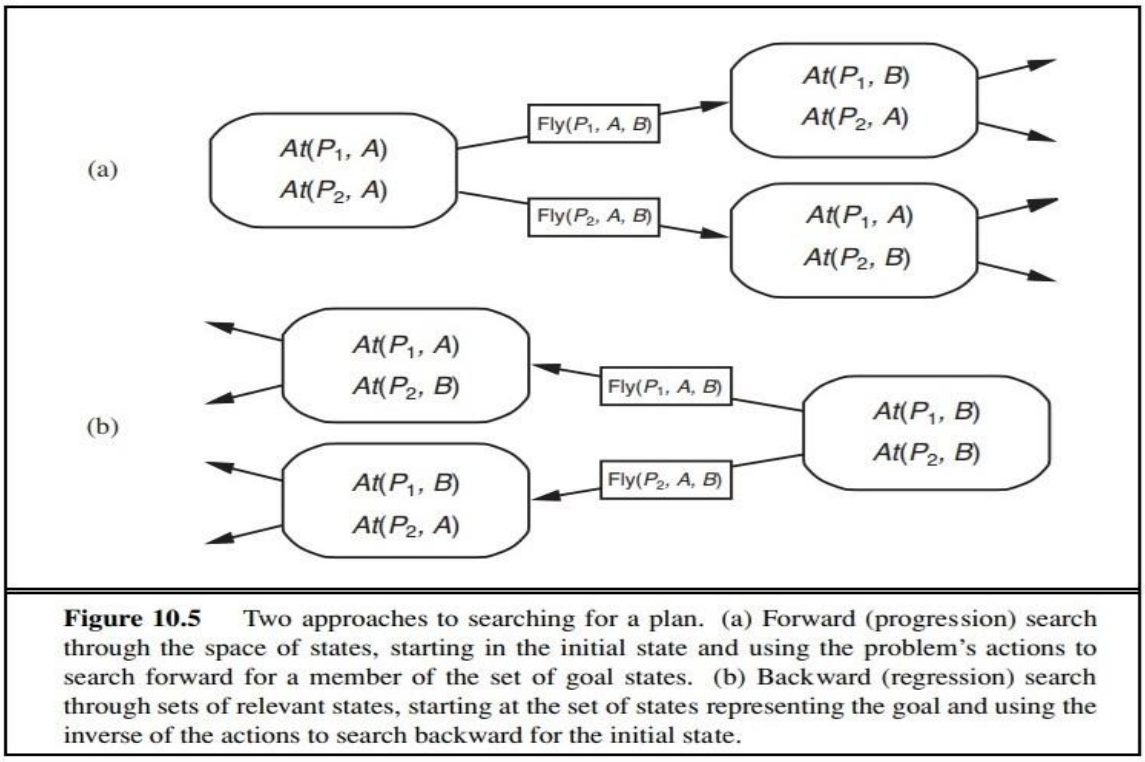


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

2.2 Backward (regression) relevant-states search

In backward (regression) relevant-states search, we work backwards from the goal to find a sequence of steps leading to the initial state. Here's a simplified explanation:

1. **Start from the Goal**: Begin with the goal, which is a description of the desired outcome. For example, the goal ¬Poor ∧ Famous means we want a state where "Poor" is false and "Famous" is true.

2. **Work Backwards**: Look for actions that can lead to the goal state. Only consider actions that are relevant to achieving the goal.
3. **Relevant States**: At each step, consider a set of possible states that could result from applying the actions. We don't look at just one state but all states that meet the goal conditions.
4. **Find a Path**: Continue working backward, applying relevant actions, until you find a sequence of steps that reaches the initial state.

Essentially, we trace backwards from the goal, considering all possible states and actions that can achieve the goal, until we map out a path back to where we started.

1. **Ground Fluents and States**: In a problem domain with n ground fluents (basic conditions), there can be 2^n possible ground states because each fluent can either be true or false.
2. **Descriptions of Goal States**: There are 3^n possible descriptions of sets of goal states. Each fluent can be positive, negative, or not mentioned in a description.
3. **Backward Search**: This type of search works by regressing from a goal state description to its predecessor state description. It's useful for finding solutions when we can easily describe the state before a given action.
4. **Example with PDDL**: PDDL (Planning Domain Definition Language) simplifies backward search by allowing regression over actions. For instance, from a goal like "deliver cargo C_2 to SFO," using the action "Unload(C_2, p^l, SFO)" regresses the goal state to "In(C_2, p^l) ∧ At(p^l, SFO) ∧ Cargo(C_2) ∧ Plane(p^l) ∧ Airport(SFO)."
5. **Forward vs Backward Search**: Forward search looks for applicable actions, while backward search looks for relevant actions that lead up to the current goal state. Backward search often has a lower branching factor but is harder to optimize with heuristics compared to forward search.
6. **Preference in Planning Systems**: Despite its advantages, many planning systems prefer forward search due to challenges in defining effective heuristics for backward search.

In backward search in planning focuses on tracing actions backward from a goal state to find a suitable plan, leveraging PDDL for efficient regression over actions.

2.3 Heuristics for planning

a) In search problems, like finding a path from a starting point to a goal, heuristic functions estimate how far we are from the goal. An "admissible" heuristic doesn't overestimate this distance. One way to create such a heuristic is by simplifying the problem—either by making it easier to move between states or by grouping similar states together.

b) Example, imagine a maze where you want to find the shortest path to a treasure. A heuristic could estimate the distance from your current position to the treasure, ignoring obstacles. This would be a relaxed or simplified version of the actual maze, but it still helps guide you towards the goal efficiently.

c) In more complex problems, like planning tasks with many possible actions and conditions, good heuristics often simplify by assuming actions can achieve goals directly without considering all the original conditions. This simplification helps in estimating how close a state is to achieving the goal state.

d) Heuristics make search algorithms like A* more efficient by guiding them towards solutions without exploring unnecessary paths.

In planning problems like the sliding block puzzle (like the 8-puzzle or 15-puzzle), actions are defined to move tiles around. Let's simplify this with an example:

Example: Moving a Tile in the Sliding Block Puzzle

Imagine you have an 8-puzzle where you need to slide tiles around to reach the goal configuration.

1. **Action Definition (Slide)**:
    - **Action:** Slide(t, s_1, s_2)
    - **Preconditions:**
        - On(t, s_1): Tile t is on square s1.
        - Tile(t): t is a tile.
        - Blank(s_2): Square s_2 is blank (no tile).
        - Adjacent(s_1, s_2): Squares s1 and s_2 are adjacent.
    - **Effects:**
        - On(t, s_2): Tile t moves to square s_2.
        - Blank(s_1): Square s1 becomes blank.
        - ¬On(t, s_1): Tile t is no longer on square s_1.
        - ¬Blank(s_2): Square s_2 is no longer blank.

2. **Heuristics Derived from Action Modifications**:

a) **Misplaced Tiles Heuristic**: Counts how many tiles are not in their goal positions by allowing any tile to move to any empty space in one action.

b) **Manhattan Distance Heuristic**: Measures the sum of horizontal and vertical distances from each tile's current position to its goal position by removing the precondition requiring a blank space.

c) **Factored Representation:** Factored representation simplifies modifying action rules like sliding a tile, enabling automatic derivation of different heuristics. This flexibility improves planning efficiency compared to rigid, atomic representations.
d) **Ignore-Delete-Lists Heuristic:** Focuses on planning without considering actions that reverse progress (like undoing previous steps in a road trip).
e) **State Abstraction:** Simplifies complex problems by focusing on key aspects and ignoring less important details (e.g., grouping warehouse items by type instead of tracking each individual item).
f) **Subgoal Independence:** Assumes solving parts of a problem independently provides a good estimate of solving the entire problem (e.g., planning each dish separately for a dinner party).
g) **Choosing Abstractions:** Selects which details to simplify to make planning more efficient without losing critical information (e.g., focusing on major cities and highways in travel planning).
h) **Pattern Databases and Planning Graphs:** Tools in AI planning that use precomputed data (like optimal routes in a delivery service) to speed up planning and improve decision-making.

These concepts help streamline planning algorithms by reducing complexity while maintaining effective heuristic guidance.

Topic-3: Planning Graphs:

1. Heuristics can be inaccurate.
2. A planning graph provides better heuristic estimates.
3. These heuristics work with any search technique.
4. GRAPHPLAN algorithm searches solutions within the planning graph.
5. A planning problem checks if a goal state is reachable from the initial state.
6. A planning graph is a quick, polynomial-size approximation.
7. It estimates the steps needed to reach the goal, though not definitively.

3.1 LEVELS:

A planning graph is a directed graph organized into levels:

1. **Initial State (S0):** Nodes represent each fluent that holds in S0.
2. **Action Level (A0):** Nodes for each applicable ground action in S0.
3. **Alternating Levels:** Si followed by Ai, until a termination condition is met.
4. **Negative Interactions:** Planning graphs record a restricted subset.
5. **Propositional Planning:** Planning graphs only work for these problems.
6. **Effectiveness:** Despite larger problem descriptions, planning graphs effectively solve hard planning problems.

Persistence Action: A condition remains true unless an action changes it. Represented as a no-op.

Mutual Exclusion (Mutex): Two conditions that cannot be true at the same time.

Example: Eating cake (Eat(Cake)) is mutually exclusive with having cake (Have(Cake)) or not having eaten cake (¬Eaten(Cake)).

Belief State: A collection of possible states.

3.2 Leveled Off:

1. Alternate between state level S_i and action level A_i until two consecutive levels are identical, indicating the graph has leveled off.
2. Ensure two actions cannot be executed at the same level.
3. Constructing the planning graph avoids combinatorial search by recording impossible choices with mutex links.

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.

- **Initial State**: You start with the state where you **Have(Cake)**.
- **Goal**: Your objective is to achieve both **Have(Cake)** and **Eaten(Cake)** simultaneously.
- **Actions**:
  - **Eat(Cake)**:
    - **Precondition**: You must **Have(Cake)**.
    - **Effect**: After eating the cake, you no longer **Have(Cake)** but you have **Eaten(Cake)**.
  - **Bake(Cake)**:
    - **Precondition**: You do **not Have(Cake)**.

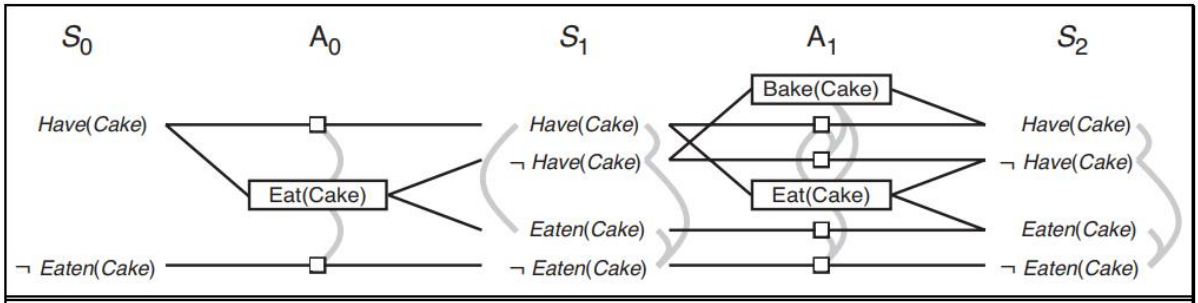


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

3.3 We now define mutex links for both actions and literals.

A mutex relation holds between two actions at a given level if any of the following three conditions holds:

1. **Inconsistent effects**: One action negates the effect of the other (e.g., Eat(Cake) and Have(Cake)).
2. **Interference**: One action's effect negates the precondition of the other (e.g., Eat(Cake) negates the precondition of Have(Cake)).

3. **Competing needs**: One action's precondition is mutually exclusive with the other's (e.g., Bake(Cake) and Eat(Cake) compete on Have(Cake)).

Inconsistent Support:

- A mutex relation exists between two literals at the same level if they negate each other or if all action pairs achieving the literals are mutually exclusive.
- The time to build the graph has the $O(n(a + l)^2)$ complexity.

3.4 Planning Graphs for Heuristic Estimation

Level Cost:

- Planning graphs allow several actions at each level.
- The heuristic counts just the level, not the number of actions.
- A serial planning graph, allowing only one action per time step, is often used for heuristics.

Max-Level Heuristic:

- Estimates the cost of a conjunction of goals by taking the maximum level cost of any goal.
- Admissible but not always accurate.

Level Sum Heuristic:

- Assumes subgoal independence and returns the sum of the level costs of the goals.
- Example: For the goal Have(Cake) ∧ Eaten(Cake), the level-sum heuristic gives 1 (0 + 1), but the correct answer is 2.

Set-Level Heuristic:

- Finds the level where all literals in the conjunctive goal appear without being mutually exclusive.
- Useful for generating accurate heuristics by viewing the planning graph as a relaxed problem.

Example Problem:

- An unsolvable problem that can't be recognized as such by a planning graph is the blocks-world problem with the goal of getting block A on B, B on C, and C on A.

3.5 The GRAPHLAN Algorithm:

1. **Extracting a Plan**: Extract a plan directly from the planning graph instead of using it just for heuristics.
2. **EXPAND-GRAPH**: GRAPHPLAN algorithm adds levels to a planning graph using EXPAND-GRAPH.

3. **Goal Check**: When all goals appear as non-mutex, GRAPHPLAN uses EXTRACT-SOLUTION to find a plan. If it fails, it expands another level and retries, stopping if it becomes futile.
4. **Initialization**: GRAPHPLAN starts with a one-level (S_0) graph representing the initial state.
5. **Mutex Relations**: EXPAND-GRAPH identifies and adds mutex relations to the graph.

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
    if goals all non-mutex in S_t of graph then
      solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Algorithm Explanation:

1. Initialize a planning graph based on the given problem description.
2. Extract individual goals from the problem's overall goal.
3. Begin a loop starting from level 0 of the planning graph.
4. Check if all goals can be achieved simultaneously in the current state of the graph without any mutual exclusion.
5. Attempt to find a solution path that achieves all goals using the planning graph.
6. If a solution is found, return it.
7. If the planning graph and the list of actions known not to achieve goals have stabilized without finding a solution, return failure.
8. Expand the planning graph by adding new levels and actions based on the problem description to continue towards achieving the goals.

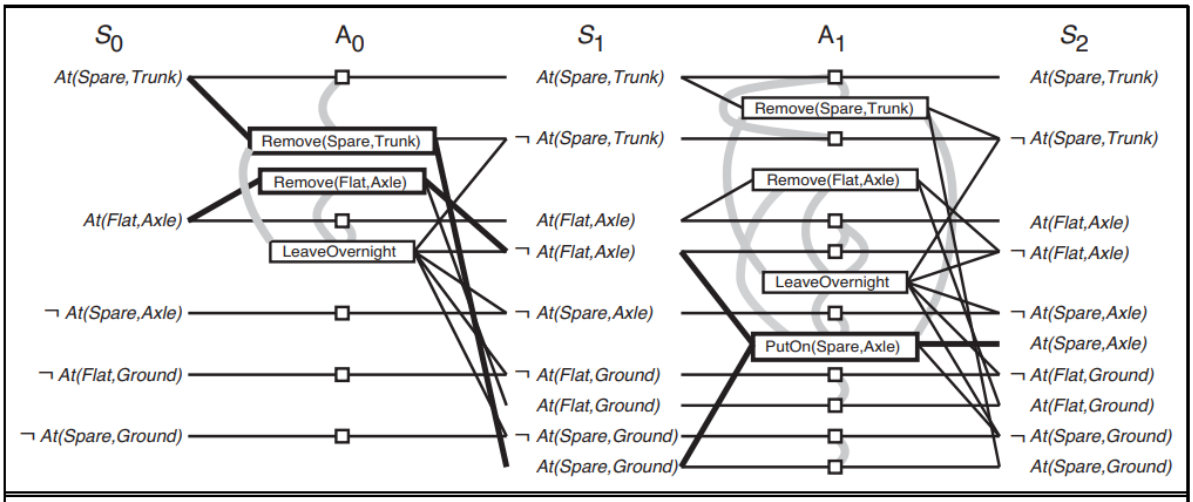


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

3.6 The mutex relations and their causes based on the spare-tire problem:

1. **Inconsistent effects**:
   - **Actions**: Remove(Spare, Trunk) vs. LeaveOvernight
   - **Cause**: One results in At(Spare, Ground) and the other negates it.
2. **Interference**:
   - **Actions**: Remove(Flat, Axle) vs. LeaveOvernight
   - **Cause**: One requires At(Flat, Axle) and the other negates it.
3. **Competing needs**:
   - **Actions**: PutOn(Spare, Axle) vs. Remove(Flat, Axle)
   - **Cause**: One needs At(Flat, Axle) and the other negates it.
4. **Inconsistent support**:
   - **States**: At(Spare, Axle) vs. At(Flat, Axle) in S2
   - **Cause**: Achieving At(Spare, Axle) through PutOn(Spare, Axle) is mutex with the persistence action required for At(Flat, Axle).

These mutex relations identify immediate conflicts when trying to perform actions or maintain states that are mutually exclusive.

3.7 Other Information:

1. EXTRACT-SOLUTION is formulated as a Boolean CSP (Constraint Satisfaction Problem) with actions as variables at each level.
2. Each action has a cost of 1.
3. If EXTRACT-SOLUTION fails for a set of goals at a given level, the (level, goals) pair is recorded as a no-good.

4. No-goods are used in the termination test.
5. Planning is PSPACE-complete, but constructing the planning graph takes polynomial time.

A practical approach is a greedy algorithm based on the level cost of literals:

1. Pick the literal with the highest-level cost.
2. Prefer actions with easier preconditions, i.e., choose an action with the smallest sum (or maximum) of the level costs of its preconditions.

3.8 Termination of GRAPHPLAN:

- If EXTRACT-SOLUTION fails, it indicates that some goals are unachievable and marked as no-good.
- Planning graphs have properties that change monotonically: a) Literals increase. b) Actions increase. c) Mutexes decrease. d) No-good decrease.
- If, at the final state, a goal is missing or mutex with another goal, GRAPHPLAN stops and returns failure.

4. OTHER CLASSICAL PLANNING APPROACHES

4.1 The most popular approaches to fully automated planning are:

1. **Translating to a SAT problem**: Converts the planning problem into a SAT problem, allowing efficient SAT solvers to find a solution by making a complex logical formula true.
2. **Forward state-space search with heuristics**: Explores future states from the current state by applying actions, using heuristics to prioritize states and find solutions faster.
3. **Search using a planning graph**: Builds a layered graph representing possible actions and their effects over time, enabling planners to identify efficient action sequences to reach the goal.

Other approaches include first-order logical deduction, constraint satisfaction, and plan refinement.

4.2 Winning Systems in International Planning Competitions:

1. **2008:**
    a. Optimal: GAMER (model checking, bidirectional search)
    b. Satisficing: LAMA (fast downward search with FF heuristic)
2. **2006:**
    a. Optimal: SATPLAN, MAXPLAN (Boolean satisfiability)
    b. Satisficing: SGPLAN (forward search; partitions into independent subproblems)
3. **2004:**
    a. Optimal: SATPLAN (Boolean satisfiability)
    b. Satisficing: FAST DIAGONALLY DOWNWARD (forward search with causal graph)
4. **2002:**
    a. Automated: LPG (local search, planning graphs converted to CSPs)
    b. Hand-coded: TLPLAN (temporal action logic with control rules for forward search)
5. **2000:**
    a. Automated: FF (forward search)
    b. Hand-coded: TALPLANNER (temporal action logic with control rules for forward search)
6. **1998:**
    a. Automated: IPP (planning graphs)
    b. Automated: HSP (forward search)

4.3 Top-Performing Systems in the International Planning Competition

- **Tracks:**
  - **Optimal:** Planners produce the shortest possible plan.
  - **Satisficing:** Non-optimal solutions are accepted.
  - **Hand coded:** Domain-specific heuristics are allowed.
  - **Automated:** Domain-specific heuristics are not allowed.

4.4 Classical Planning as Boolean Satisfiability (SAT)

- **SATPLAN:** Solves planning problems using propositional logic.
- **Translation Steps for PDDL to SATPLAN:**
  1. **Propositionalize Actions:** Replace each action schema with ground actions by substituting constants for variables.
  2. **Define Initial State:** Assert F_0 for every fluent F in the initial state, and ¬F for fluents not in the initial state.
  3. **Propositionalize Goal:** Replace goal literals with disjunctions over constants.
     Example: On(A,x) ∧ Block(x) becomes
     (On(A,A)∧Block(A))∨(On(A,B)∧Block(B))∨(On(A,C)∧Block(C))
  4. **Add Successor-State Axioms:** For each fluent F, add
     $F_{t+1} \Leftrightarrow$(Action Causes F∨(F_t∧¬Action Causes Not F)).
  5. **Add Precondition Axioms:** For each ground action A, add $A_t \Rightarrow PRE(A)_t$
  6. **Add Action Exclusion Axioms:** State that every action is distinct from every other action.

The translation results in a form that can be processed by SATPLAN to find a solution.

4.5 Planning as First-Order Logical Deduction and Situation Calculus:

4.5.1 PDDL Limitations:

- Can't express goals like “move all cargo from A to B regardless of amount.”
- Can't easily express constraints like “no more than four robots in one place.”

4.5.2 Propositional Logic Limitations:

- Time tied to fluent states (e.g., "South2" means “agent facing south at time 2”).
- Can't handle conditional time statements (e.g., “agent faces south at time 2 if turned right at time 1, otherwise east”).

4.5.3 First-Order Logic Advantages:

- Uses universal quantifiers for goals and constraints.
- Uses situation calculus for time and branching situations, representing actions and effects with different outcomes.

- First-order logic and situation calculus provide more flexibility for complex planning problems compared to PDDL.

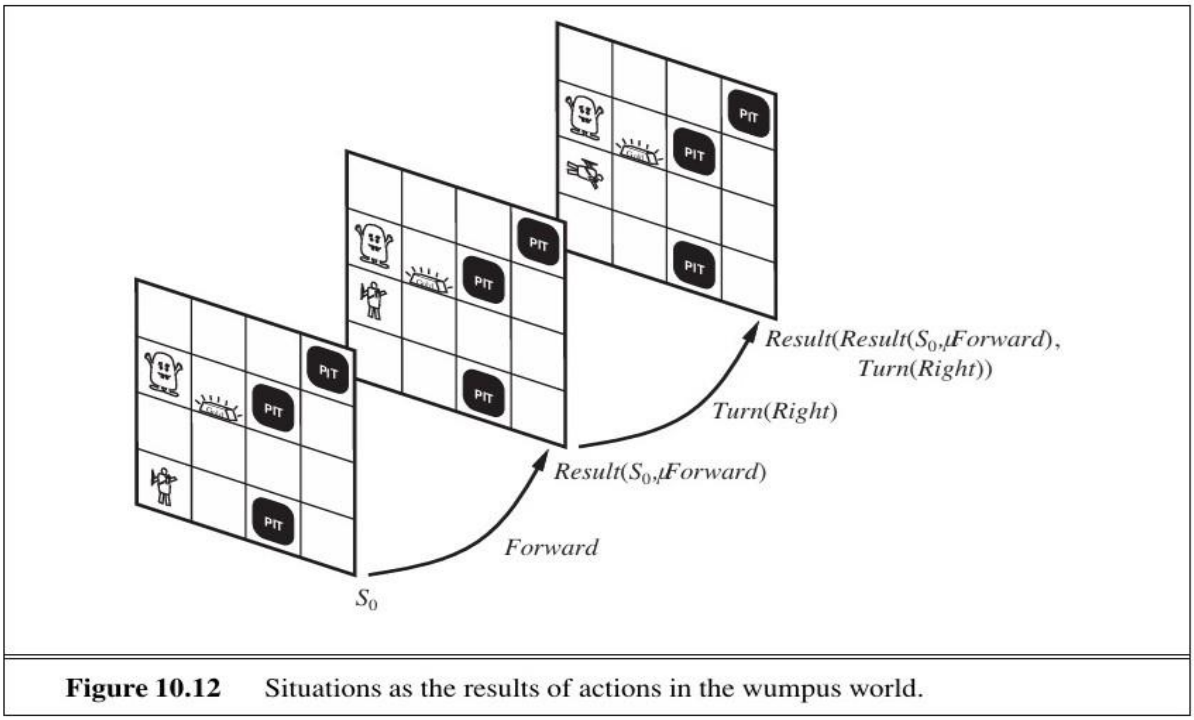


Figure 10.12 Situations as the results of actions in the wumpus world.

1. **Situation and Action**:
    - A situation is the initial state.
    - RESULT(s, a) is the state after applying action 'a' in situation 's'.
    - Two situations are the same if their starting state and actions are the same.
2. **Fluents**:
    - A fluent is a function that can change across different situations.
    - **Example:** At(x, l, s) means object x is at location l in situation s.
    - **Example:** Location(x, s) = l means x is at location l in situation s.
3. **Action Preconditions (Possibility Axiom)**:
    - Preconditions specify when an action can be taken.
    - **Example:** You can shoot if the agent is alive and has an arrow:
        - Alive (Agent, s) ∧ Have(Agent, Arrow, s) ⇒ Poss(Shoot, s).
4. **Fluent Change (Successor-State Axiom)**:
    - This axiom describes how a fluent changes with an action.
    - **Example:** The agent is holding gold g after action if it grabbed g or was already holding g and didn’t release it:
    - Poss (a, s) ⇒ (Holding(Agent, g, Result(a, s)) ⇔ a=Grab(g) ∨ (Holding(Agent, g, s) ∧ a != Release(g))).
5. **Action Uniqueness**:

- Each action must be uniquely identifiable.
- **Different actions must be distinct:** $A_i(x,...)= A_j(y,...)$.
- Identical actions have equal arguments: $A_i(x_1,...,x_n)=A_i(y_1,...,y_n) \Leftrightarrow x_1=y_1 \wedge ... \wedge x_n=y_n$.

6. **Solution**:
   - A solution is a sequence of actions that meets the goal.
   - Situation calculus defines planning semantics but lacks practical large-scale planning programs due to inference difficulty in FOL and lack of effective heuristics.

4.6 Planning as Constraint Satisfaction

1. Planning problems can be encoded as constraint satisfaction problems (CSPs).
2. Each time step uses a single variable, Actiont_tt, with possible actions as its domain.
3. This simplifies the encoding by removing the need for action exclusion axioms.
4. Planning graphs can also be encoded into CSPs, as done by GP-CSP (Do and Kambhampati, 2003).

4.7 Planning as Partially Ordered Plans

1. Traditional planning creates totally ordered sequences of actions.
2. Partially ordered plans represent actions and constraints without strict orderings.
3. Useful for independent subproblems (e.g., loading packages onto different planes).
4. Plans include actions and "Before" constraints (e.g., Before(action1, action2)).
5. **Example:** In the spare tire problem, "Remove(Spare, Trunk)" and "Remove(Flat, Axle)" can occur in any order before "PutOn(Spare, Axle)".

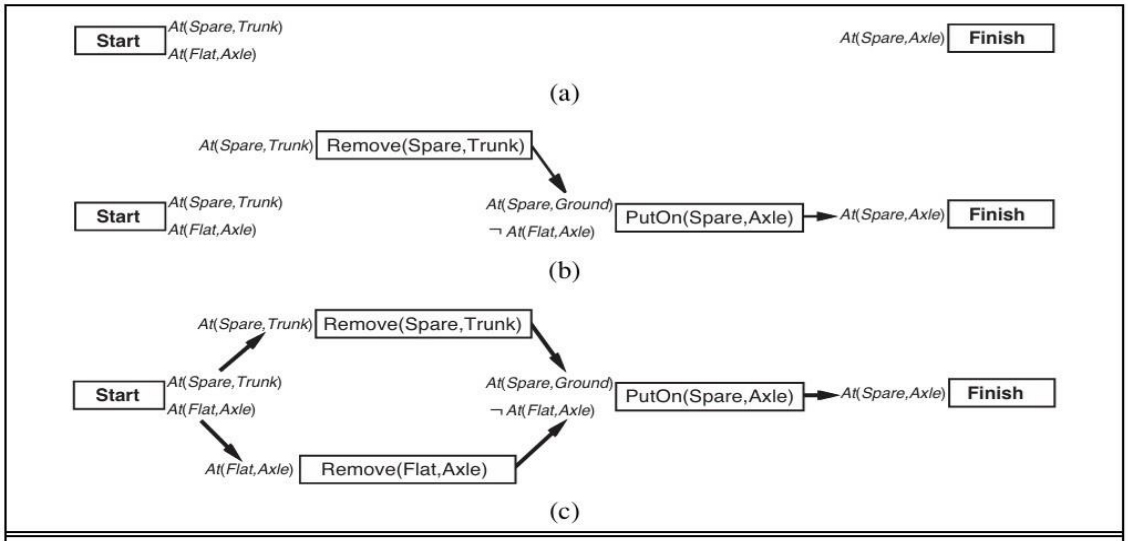


Figure 10.13 (a) the tire problem expressed as an empty plan. (b) an incomplete partially ordered plan for the tire problem. Boxes represent actions and arrows indicate that one action must occur before another. (c) a complete partially-ordered solution.

4.8 Partial-Order Planning

1. **Partial-Order Plans**:
    - Created by searching through the space of plans, not the state space.
    - Start with an empty plan (initial state and goal, no actions in between).
2. **Flaw Identification and Correction**:
    - A flaw is anything preventing the plan from being a solution.
    - Correct flaws by adding actions (e.g., if no action achieves At(Spare, Axle), add PutOn(Spare, Axle)).
    - Adding actions introduces new flaws (e.g., preconditions of new actions).
3. **Least Commitment Strategy**:
    - Make the minimum necessary commitments when adding actions.
    - For example, commit to ordering Remove(Spare, Trunk) before PutOn(Spare, Axle) but make no other commitments.
4. **Historical Context**:
    - Popular in the 1980s and 90s for handling planning problems with independent subproblems.
    - Less competitive by 2000 due to forward-search planners with excellent heuristics.
5. **Current Use**:
    - Still important for specific tasks like operations scheduling with domain-specific heuristics.
    - Used in domains where human understanding of plans is crucial (e.g., spacecraft operations).
6. **Advantages**:
    - Easier for humans to understand and verify plans.
    - Useful for operational plans in space missions.

4. ANALYSIS OF PLANNING APPROACHES

1. Overview of Planning: Planning in AI combines search and logic. It's about finding solutions to problems or proving their existence constructively. Over the last decade, ideas from both areas have significantly improved performance, making planners more useful in industries.

2. Challenges in Planning: One major challenge is dealing with combinatorial explosion. If a domain has nnn propositions, there can be 2n2^n2n possible states, making planning complex and computationally intensive.

3. Approaches to Address Challenges:

a) **Decomposability:** Breaking down problems into independent subproblems can dramatically speed up planning. However, negative interactions between actions can complicate this.

b) **Graphplan and SATPLAN:** These planning methods identify difficult interactions (called mutexes). Graphplan uses a specific structure to record these interactions, while SATPLAN uses a general form.

c) **Forward Search:** This heuristic approach looks for patterns in subsets of propositions that cover independent subproblems, even when they are not completely independent.

4. Serialized Subgoals: Some problems allow planners to achieve goals in a serializable order without needing to undo previous steps. For example, in a blocks world where you're stacking blocks (like A on B on C on the Table), achieving subgoals from bottom to top can avoid backtracking.

5. Real-World Applications: Planners like those used for NASA's Deep Space One spacecraft take advantage of serialized ordering of goals. By eliminating most of the search process, these planners can control spacecraft in real-time effectively, which was previously thought impossible.

6. Future Challenges and Directions: While current planning techniques have advanced performance, scaling to larger and more complex problems will likely require new approaches beyond current propositional representations. This could involve integrating first-order and hierarchical representations with efficient heuristics.

6. Hierarchical Planning:

Hierarchical planning is a method used to organize activities efficiently by breaking them down into smaller, manageable tasks. It relies on hierarchical decomposition, part of the Hierarchical Task Network (HTN), where actions are called primitive actions. Advanced concepts like High-Level Actions (HLAs) help refine action sequences.

1. Hierarchical Decomposition:

- **Definition:** Breaking down complex tasks into smaller, simpler tasks.
- **Example:** Planning a trip involves booking flights, accommodations, and planning activities.

2. Primitive Actions:

- **Definition:** Basic, indivisible actions that can be directly executed.
- **Example:** For the task "book flights," actions like "search for flights," "select a flight," and "make a booking" are primitive actions.

3. High-Level Actions (HLAs):

- **Definition:** Actions that represent a sequence of primitive actions.
- **Example:** "Plan trip" could be an HLA that includes actions like "book flights," "reserve hotel," and "create itinerary."

4. High-Level Plan:

- **Definition:** A sequence of HLAs that achieve a goal.
- **Example:** A high-level plan for a vacation could include HLAs like "plan trip," "pack luggage," and "arrange transportation."

5. Implementation:

- **Definition:** Carrying out the sequence of actions in a high-level plan.
- **Example:** Executing the "plan trip" HLA involves performing actions such as booking flights, reserving accommodations, and scheduling activities.

6. Success Criteria:

- **Definition:** Criteria to determine if a high-level plan is successful.
- **Example:** A successful vacation plan means all booked flights are confirmed, accommodations are secured, and activities are planned.

7. Search for Solutions:

- **Definition:** Finding either direct (primitive) solutions or more abstract (HLA) solutions.
- **Example:** If booking a flight directly isn't possible, searching for alternative flights or changing travel dates is necessary.

Hierarchical planning simplifies complex tasks by breaking them into smaller steps, using both primitive actions and higher-level plans to achieve goals efficiently.

Search for Primitive Solutions:

1. **Hierarchical Task Network (HTN)**: In HTN planning, we start with a high-level action called "Act" that we want to achieve. This action can be complex and needs to be broken down into simpler actions to achieve the goal.
2. **Primitive Action (ai)**: These are the basic, indivisible actions that the agent can directly perform. For example, in a cooking task, "chop vegetables" or "boil water" are primitive actions because they cannot be further decomposed into simpler actions.
3. **Refinement**: Refinement is the process of breaking down a complex action into simpler sub-actions or other high-level actions until we reach primitive actions that can be directly executed. This is like breaking down "prepare dinner" into actions like "chop vegetables," "boil rice," and "grill chicken."
4. **Goal Condition**: This specifies the state that we want to achieve. For instance, in a navigation task, the goal condition could be reaching a specific location.
5. **Hierarchical Level Actions (HLAs)**: These are high-level actions that can be decomposed further or directly refined into primitive actions or other HLAs. For example, "prepare meal" could be an HLA that includes actions like "cook main dish" and "prepare dessert."
6. **Plan Library**: This is a collection of known methods or strategies (plans) to achieve certain actions or goals. For example, a plan library for a robot might include methods for navigating through a room or picking up objects.

Example:

Task: Navigate through a maze to reach the exit.

- **High-level Action (Act):** "Navigate through maze"
- **Refinement:** Break down "Navigate through maze" into:
  - HLA: "Explore corridors"
    - Sub-actions: "Turn left," "Turn right," "Move forward"
  - HLA: "Reach exit"
    - Sub-actions: "Follow path," "Avoid obstacles"
- **Primitive Actions:**
  - "Turn left," "Turn right," "Move forward," "Follow path," "Avoid obstacles" are all actions that the agent can directly perform.

- **Goal Condition:** Reach the exit of the maze.
- **Plan Library:** Contains strategies for each action, such as algorithms for pathfinding, obstacle avoidance, and decision-making.

In HTN planning helps break down complex tasks into manageable steps, using a combination of high-level and primitive actions, along with a library of known methods to achieve goals efficiently.

Search for Abstract Solutions:

In hierarchical planning, solutions often involve describing high-level actions (HLAs) that outline the primitive actions needed to achieve a goal. This helps determine if a plan successfully achieves its goal or needs adjustment. Let's break down the concepts with simple examples:

1. **Effect Descriptions of HLAs:**

   HLAs describe higher-level actions that encompass several primitive actions. For example, "Make Breakfast" might include actions like "Boil Water" and "Toast Bread".

   Effect descriptions specify what changes occur when an HLA is executed. For instance, "Boil Water" changes the state from "Cold Water" to "Boiling Water".

2. **Downward Refinement Property:**

   A high-level plan has the downward refinement property if at least one of its HLAs, when implemented, achieves the overall goal.

   Example: If the goal is "Prepare Dinner," a plan might include HLAs like "Cook Main Course" and "Set the Table." If "Cook Main Course" successfully prepares a meal, the plan has downward refinement.

3. **Angelic Semantics:**

   Angelic semantics relate to the idea that from any state sss and a reachable set hhh of HLAs, we can determine a set of states that can be reached.

   Example: Starting from the state "Home" and considering HLAs like "Drive Car" or "Take Bus," angelic semantics help us predict reachable states such as "Work" or "School" depending on the chosen HLA.

These concepts in hierarchical planning simplify decision-making by clarifying how HLAs contribute to achieving goals and predicting possible outcomes from different actions.

Solve Exercises (From Text Book):

1. Describe the differences and similarities between problem solving and planning?
2. Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem?
3. Prove that backward search with PDDL problems is complete.
4. Differentiate between PlanSAT and Bounded PlanSAT decision problems?
5. Explain how planning graph data structure can be used to give a better heuristic for a planning problem?
6. What are the three conditions which are required to hold a relation between two actions in a mutex relation?
7. How totally ordered plans are different from partially ordered structures?
8. What is Boolean satisfiability? How can it be used in classical planning?