B.S. Abdur Rahman ™ Crescent Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956

CSD 3102- ARTIFICIAL INTELLIGENCE TECHNIQUES

MODULE 4

PLANNING GRAPHS

TOPIC: Planning-Planning

MODULE IV PLANNING 9

Planning Problem – Simple Planning agent –Blocks world - Goal Stack Planning- Means Ends Analysis- Planning as a Statespace Search - Partial Order **Planning-Planning Graphs-Hierarchical Planning - Non- linear Planning -Conditional Planning-Reactive Planning - Knowledge based Planning-Using Temporal Logic – Execution Monitoring and Re-planning- Continuous Planning-Multi-agent Planning-Job shop Scheduling Problem**.

Graph Planning

- All of the heuristics we have suggested for total-order and partial-order planning can suffer from inaccuracies.
- P**lanning graph** can be used to give better heuristic estimates.
- Alternatively, we can extract a solution directly from the planning graph, using a specialized algorithm such as the one called **GRAPHPLAN**.

- **GRAPHPLAN** is an algorithm used in artificial intelligence for **automated planning**, designed to efficiently find a sequence of actions (a plan) that will transform an initial state into a desired goal state. It works by constructing a **planning graph**, which is a data structure that alternates between layers of facts (propositions) and actions, representing the evolving state of the world over time.

Key Features of GRAPHPLAN:

1.Planning Graph Construction:

- The graph alternates between fact layers and action layers.
- **Fact Layers**: These represent the conditions (facts) that are true at a specific point in time.
- **Action Layers**: These represent the possible actions that can be taken based on the previous fact layer.

2.Mutex (Mutual Exclusion) Relationships:

- GRAPHPLAN detects **mutexes**, or mutually exclusive actions or facts, to prevent the inclusion of conflicting actions or states.
- These mutexes help prune impossible or invalid plans, speeding up the search process.

3.Plan Extraction:Once the planning graph is built, GRAPHPLAN attempts to extract a valid plan by searching backwards from the goal state.

If no plan is found, the planning graph is expanded further, and the process repeats until a solution is found or the algorithm determines that no solution exists.

4. Parallelism:GRAPHPLAN allows actions to be executed in parallel (simultaneously) as long as they don’t interfere with each other (i.e., they are not mutex).

Steps of GRAPHPLAN:

- **Initialize**: Start with an initial state and a goal state.

- **Graph Expansion**: Expand the planning graph by alternating between action layers and fact layers.

- **Mutex Detection**: Mark actions and facts that are mutually exclusive (cannot coexist).

- **Plan Extraction**: Search for a valid plan from the graph using backward search.

- **Repeat if Necessary**: If no plan is found, expand the graph further and
- repeat the extraction.

Example: Simple Robot and Box Problem

A **robot** is trying to move a **box** to a specific location.

- Problem Definition:

Initial State: Robot is at location A, and the box is at location B.

Goal State: The box needs to be at location C.

Actions Available:

Move(robot, X, Y): Move the robot from location X to Y.

PickUp(robot, box, X): Robot picks up the box at location X.

PutDown(robot, box, Y): Robot puts the box down at location Y.

Solution

- Step 1: Building the Planning Graph

Layer 0 (Initial State):

Robot is at A, Box is at B.

Layer 1 (Action Layer): Possible actions:

- Move(robot, A, B)
- Move(robot, A, C)

Layer 2 (Fact Layer): Results of the actions in the previous layer:

- Robot is at B (if it moved from A to B)
- Robot is at C (if it moved from A to C)
- Box is still at B (since no action was taken on the box)

Layer 3 (Action Layer): Possible actions now:

- PickUp(robot, box, B) (if the robot is at B)
- Move(robot, B, C) (if the robot is at B)
- Move(robot, C, B) (if the robot is at C)

Layer 4 (Fact Layer): Results:

- Robot is holding the box at B
- Robot is at C (if it moved)
- Box is still at B
- ...and so on.

- Step 2: Mutex Detection

As the planning graph grows, GRAPHPLAN marks pairs of actions or facts that cannot coexist in the same layer. For example, the robot cannot both be at A and C simultaneously, so a mutex is introduced between these propositions.

- Step 3: Plan Extraction

Once the planning graph is built, GRAPHPLAN attempts to extract a valid plan by selecting actions that achieve the goal state, while ensuring that none of the actions or facts violate mutex conditions.

- For this example, GRAPHPLAN might extract the following plan:

1. **Move(robot, A, B)**
2. **PickUp(robot, box, B)**
3. **Move(robot, B, C)**
4. **PutDown(robot, box, C)**

- This sequence of actions moves the robot and the box to the correct locations, achieving the goal.

Problem Example 2: Making Breakfast

- Task is to Make a breakfast, which involves:

1. Toasting bread.
2. Frying an egg.

- **Initial State**:

  -Bread is not toasted.

  -Egg is not fried.

- **Goal State**:

  -Bread is toasted.

  -Egg is fried.

- **Available Actions**:

1. **ToastBread**: Toast the bread if it’s not already toasted.
2. **FryEgg**: Fry the egg if it’s not already fried.

Step 1: Building the Planning Graph

GRAPHPLAN starts by constructing a **planning graph** with alternating layers of facts (propositions) and actions.

Layer 0 (Initial Fact Layer):

- The planning graph begins with the facts representing the initial state:
- **Bread is not toasted**.
- **Egg is not fried**.

Layer 1 (Action Layer):

- Next, the algorithm lists all possible actions based on the current facts:
- **ToastBread**: This action is possible because "Bread is not toasted" is true.
- **FryEgg**: This action is possible because "Egg is not fried" is true.

Layer 2 (Fact Layer):

- After executing the actions, we create the new fact layer with the results of those actions:
- **Bread is toasted** (result of the ToastBread action).
- **Egg is fried** (result of the FryEgg action).

- At this point, the planning graph has expanded, and the next action layer would consider additional actions. But since our goal is already achieved (bread toasted, egg fried), we can stop here and start **extracting the plan**.

Step 2: Plan Extraction

- GRAPHPLAN tries to extract a plan (a valid sequence of actions) that leads to the goal while ensuring that no **mutexes** (mutual exclusions) are violated.

Goal Layer:

- Our goal is that "Bread is toasted" and "Egg is fried".
- Both of these facts are present in the final fact layer (Layer 2).

Since **no mutexes** exist between these facts (toasting bread and frying an egg do not conflict), GRAPHPLAN backtracks to find the actions that achieved these goals.

Layer 1 (Action Layer):

- **ToastBread** leads to "Bread is toasted".
- **FryEgg** leads to "Egg is fried".

- These actions are valid and can be performed independently (i.e., no mutexes between them), so both actions are selected.

Step 3: Final Plan

The plan extracted by GRAPHPLAN is:

1. **ToastBread**.
2. **FryEgg**.

- This sequence of actions achieves the goal of making breakfast, and since the actions are not mutually exclusive, the plan is valid.

Persistence Actions

- Definition: **A persistence (no-op) action** is a special kind of action that maintains the current state of a fact without changing it. If a fact is true in one layer, a persistence action allows it to remain true in the next layer.
- **Purpose**: Persistence actions are used in planning graphs to propagate facts forward in time. This means if a fact is true at one stage of the planning, and no other action changes it, the persistence action ensures that the fact remains true at subsequent stages.

Example:

Suppose we have the fact:

- **Robot is at A** in fact layer 0.

If no action moves the robot away from A, we need to ensure that this fact remains true in the next fact layer. To do this, GRAPHPLAN includes a **persistence action** that keeps the fact "Robot is at A" true in the next layer.

- So, for each fact, a persistence action exists that essentially says:
- "Keep this fact true if no action changes it."

Understanding Mutex Detection in GRAPHPLAN

- **Two Types of Mutexes**

1. **Mutex between Actions**: These indicate actions that cannot be performed simultaneously because they interfere with each other.
2. **Mutex between Propositions (Facts)**: These represent facts that cannot be true at the same time.

Mutex Between Actions

- There are three main types of action mutexes in GRAPHPLAN:

1. **Inconsistent Effects**: Two actions have mutually exclusive effects if one action undoes the effects of another. For example, if one action **Move(box, A, B)** makes the fact "Box is at B" true, but another action **Move(box, B, A)** makes it false, they are mutex and cannot occur simultaneously.

2.Interference: One action's effect negates the precondition of another. For example:

- Action 1: **PickUp(robot, box, B)** requires that the robot is at B.
- Action 2: **Move(robot, B, A)** would move the robot away from B. These two actions interfere because moving the robot to A would make it impossible to pick up the box at B.

3. Competing Needs: Two actions are mutex if their preconditions are mutually exclusive. For example:

- Action 1: **PickUp(robot, box, B)** requires the robot to be at B.
- Action 2: **Move(robot, A, B)** requires the robot to be at A. Since the robot cannot be at both A and B simultaneously, these actions are mutex.

Example of Action Mutex in the Robot and Box Problem

Imagine two actions at the same action layer:

- **Move(robot, A, B)**: Moves the robot from A to B.
- **PickUp(robot, box, B)**: Robot picks up the box at location B.

If the robot is not already at B (e.g., it's at A), these actions are mutex because **PickUp** requires the robot to be at B, but the robot isn't there until after the **Move** action.

Mutex Between Propositions (Facts)

- In GRAPHPLAN, propositions (facts) can also be mutex. A pair of facts is considered mutex if achieving both at the same time is logically impossible.
- There are two types of fact mutexes:

1. Inconsistent Support: Two facts are mutex if all ways of achieving them involve mutex actions. For example, achieving "Robot is at A" and "Robot is at B" at the same time requires actions that are mutually exclusive, so these facts are marked as mutex.

2.Negation: A fact and its negation are always mutex. For example, "Box is at B" and "Box is not at B" cannot both be true at the same time, so they are marked as mutex.

Example of Fact Mutex in the Robot and Box Problem

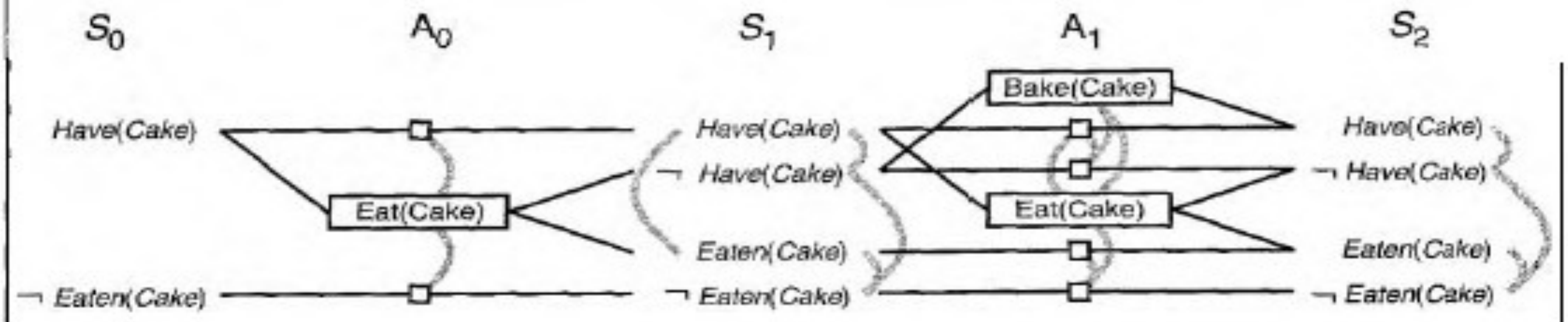
Consider two facts in a fact layer:

- **Robot is at A**.
- **Robot is at B**.

Since the robot cannot be at two locations at the same time, these facts are mutex and GRAPHPLAN will prevent actions that would lead to both being true in the same fact layer.

Init(Have(Cake))
Goal(Have(Cake) A Eaten(Cake))
Action(Eat(Cake)
  PRECOND: *Have(Cake)*
  EFFECT: ¬ *Have(Cake) A Eaten(Cake))*
Action(Bake(Cake)
  PRECOND: ¬ *Have(Cake)*
  EFFECT: *Have(Cake)*

The "have cake and eat cake too" problem.



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.

- The classic "have cake and eat cake too" problem in planning, along with its corresponding planning graph. Let’s break down the components shown and explain how the GRAPHPLAN approach is applied to this problem.

Problem Setup:

- **Initial State**:Have(Cake)
- **Goal**:Have(Cake) $\wedge$ Eaten(Cake) i.e., you want to have the cake and eat it too).
- **Actions**:

  Eat(Cake):

  - **Precondition**:Have(Cake)
  - **Effect**: ¬Have(Cake) $\wedge$ Eaten(Cake) (after eating the cake, you no longer have the cake, but the cake has been eaten).

Bake(Cake):

- **Precondition**: ¬Have(Cake) (you can only bake a cake if you don’t already have one).
- **Effect**: Have(Cake) (after baking, you now have the cake).

Planning Graph Explanation:

1. **S_0** (Fact Layer 0):

Represents the initial state:Have(Cake)and ¬Eaten(Cake) (the cake exists but hasn't been eaten yet).

2. **A_0** (Action Layer 0):

Two actions are possible at this stage:

- **Eat(Cake)**: This action is possible because Have(Cake) is true, but it will result in ¬Have(Cake) and Eaten(Cake)
- **Persistence Actions** (No-Op): These are the implicit actions that allow the facts Have(Cake) and ¬Eaten(Cake) to remain true in the next layer if no action changes them

3. **S_1** (Fact Layer 1):

This layer represents the state of the world after applying the actions in A_0:

Have(Cake) persists, or it could be negated if the cake is eaten.

Eaten(Cake) can become true if the Eat(Cake) action is taken.

Mutex relationships (shown by the curved lines) indicate that certain facts/actions cannot co-occur (e.g., you cannot both have the cake and have eaten it at the same time).

4. **A_1** (Action Layer 1):

Additional actions are possible:

Bake(Cake): If you no longer have the cake (¬Have(Cake)) you can bake one to get it again.

- **Eat(Cake)**: If you have the cake, you can eat it.
- Persistence actions forHave(Cake) and Eaten(Cake)

5. **S_2** (Fact Layer 2):

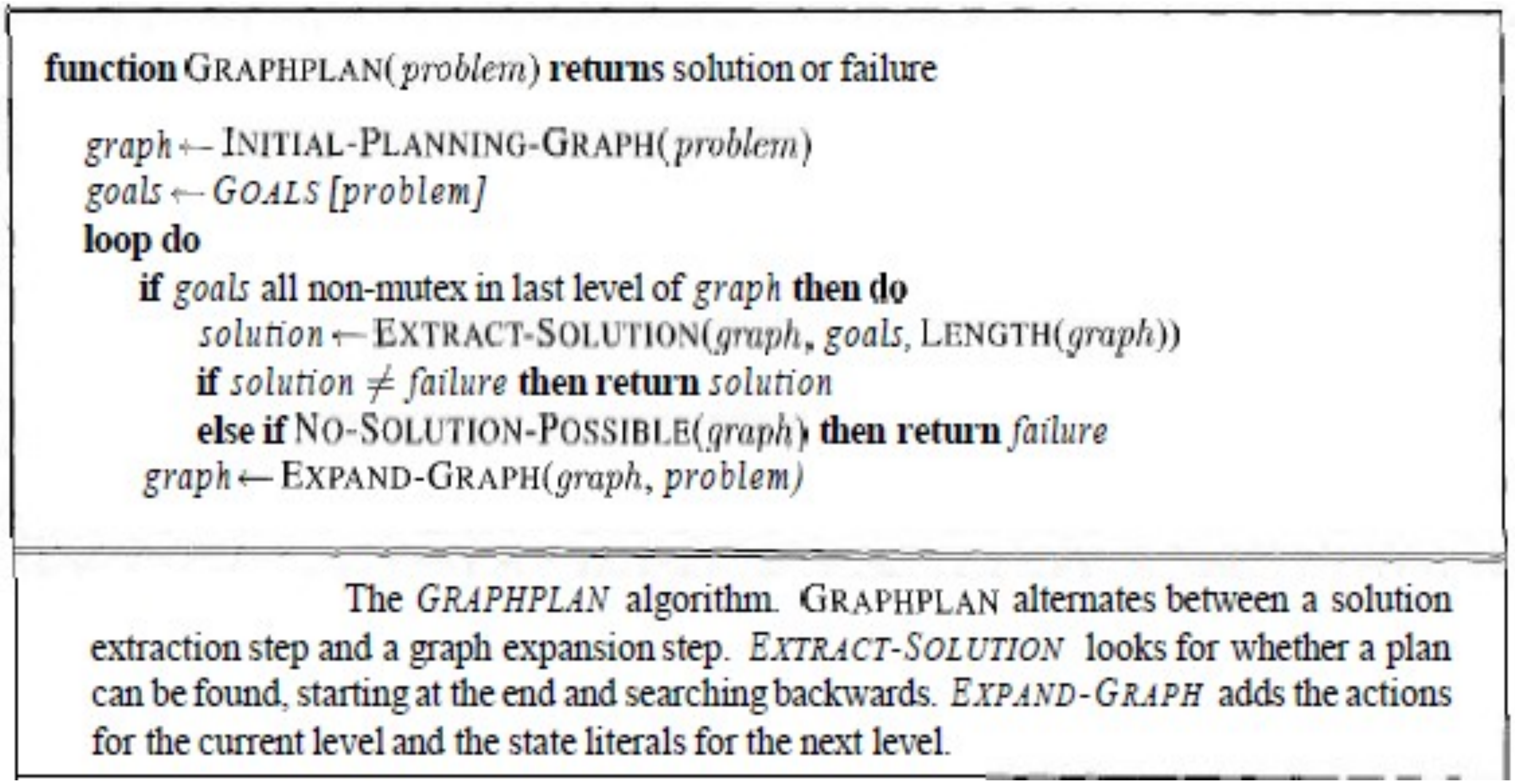
This layer represents the state after the second round of actions.

Have(Cake) and Eaten(Cake) could potentially both be true, depending on the actions chosen.

Leveled Off

- At this point, we say that the graph has **leveled off.**
- Every subsequent level will be identical, so further expansion is unnecessary.
- What we end up with is a structure where every A, level contains all the actions that are applicable in S, along with constraints saying which pairs of actions cannot both be executed.

The GRAPHPLAN algorithm



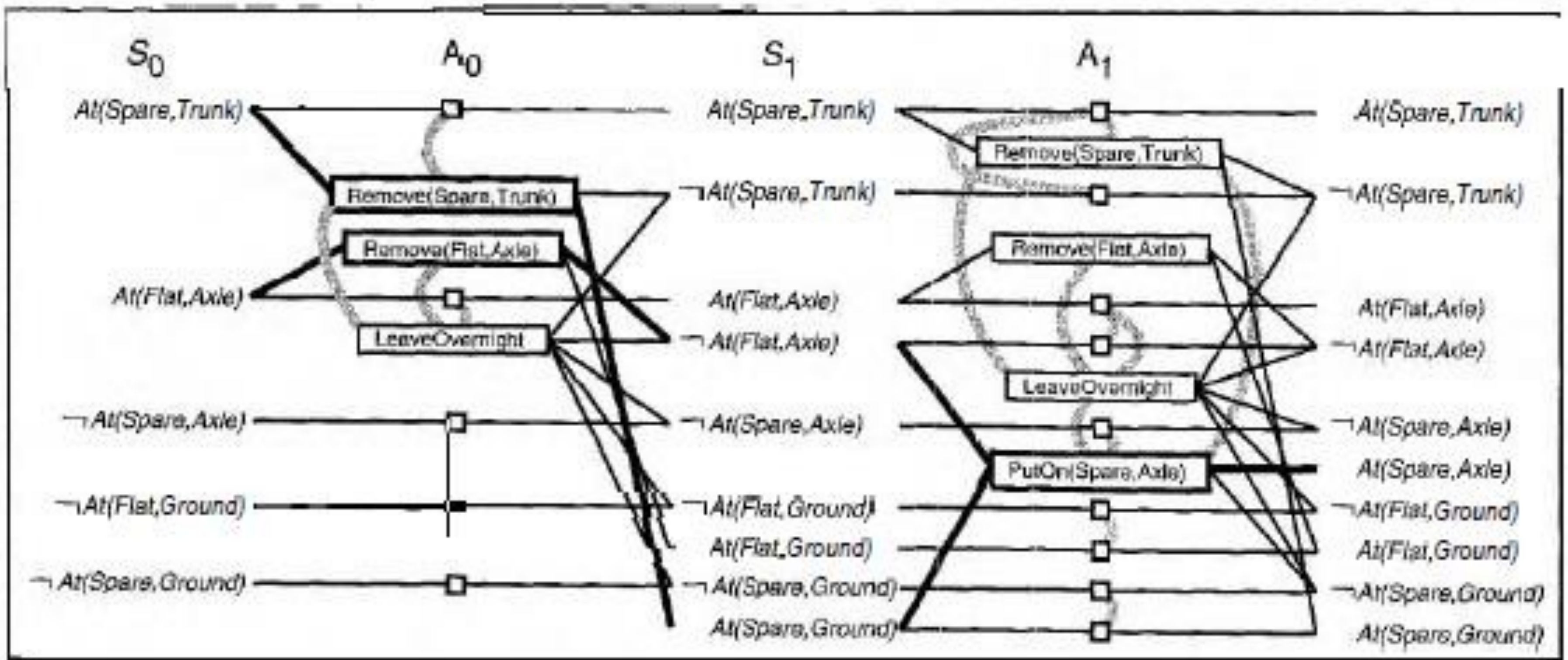
```
function GRAPHPLAN(problem) returns solution or failure

  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)
```

The *GRAPHPLAN* algorithm. GRAPHPLAN alternates between a solution extraction step and a graph expansion step. *EXTRACT-SOLUTION* looks for whether a plan can be found, starting at the end and searching backwards. *EXPAND-GRAPH* adds the actions for the current level and the state literals for the next level.

The GRAPHPLAN algorithm

- The GRAPHPLAN algorithm has two main steps, which alternate within a loop.
- First, it checks whether all the goal literals are present in the current level with no mutex links between any pair of them.
- If this is the case, then a solution ***might*** exist within the current graph, so the algorithm tries to extract that solution.
- Otherwise, it expands the graph by adding the actions for the current level and the state literals for the next level.
- The process continues until either a solution is found or it is learned that no solution exists.

Example: Spare Tire problem



The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

Graph Planning : Spare Tire problem

- The first line of GRAPHPLAN initializes the planning graph to a one-level (So) graph consisting of the five literals from the initial state.
- The goal literal *At(Spare, Axle)* is not present in So, so we need not call EXTRACT-SOLUTION-we are certain that there is no solution yet.
- Instead, EXPAND-GRAPH adds the three actions whose preconditions exist at level So (i.e., all the actions except *PutOn(Spare, Axle).*

- Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:
  - ***Inconsistent effects:*** *Remove (Spare, Trunk) is mutex with LeaveOvernight because one has the effect At (Spare, Ground) and the other has its negation.*
  - ***Inteference:*** *Remove(Flat, Axle) is mutex with LeaveOvernight because one has the precondition At(Flat, Axle) and the other has its negation as an effect.*
  - ***Competing needs:*** *Put On (Spare, Axle) is mutex with Remove (Flat, Axle) because one has At (Flat, Axle) as a precondition and the other has its negation.*
  - ***Inconsistent support:*** *At(Spare, Axle) is mutex with At(Flat, Axle) in S2 because the only way of achieving At(Spare, Axle) is by PutOn(Spare, Axle), and that is mutex with the persistence action that is the only way of achieving At(Flat, Axle). Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.*