



# Planning

---



# Introduction

---

- Planning is required to reach a particular destination.
- It is necessary to find the best route in Planning, but the tasks to be done at **a particular time and why** they are done also very important.
- Planning is considered the **logical side of taking action**.
- Planning is about deciding the tasks to be performed by the artificial intelligence system and the system's functioning under **domain-independent** conditions.
- We require **domain description, task specification, and goal description** for any planning system.
- A plan is considered a sequence of actions, and each action has its preconditions that must be satisfied before it can act and some effects that can be positive or negative.



# FSSP


---

- Forward State Space Planning (FSSP) and Backward State Space Planning (BSSP) at the basic level.
- FSSP behaves in the same way as forwarding state-space search.
- It says that given an initial state  $S$  in any domain, we perform some necessary actions and obtain a new state  $S'$ , called a progression.
- It continues until we reach the target position.
- **Disadvantage:** Large branching factor
- **Advantage:** The algorithm is Sound



# BSSP

---

- BSSP behaves similarly to backward state-space search.
  - Move from the target state  $g$  to the sub-goal of  $g$ , tracing the previous action to achieve that goal.
  - This process is called regression.
  - These sub-goals should also be checked for **consistency**. The action should be relevant in this case.
  - **Disadvantages:** not sound algorithm (sometimes inconsistency can be found)
  - **Advantage:** Small branching factor (much smaller than FSSP)
  - So for an efficient planning system, we need to combine the features of FSSP and BSSP, which gives rise to target stack planning which will be discussed in the next article.
- 



# Components of the planning system

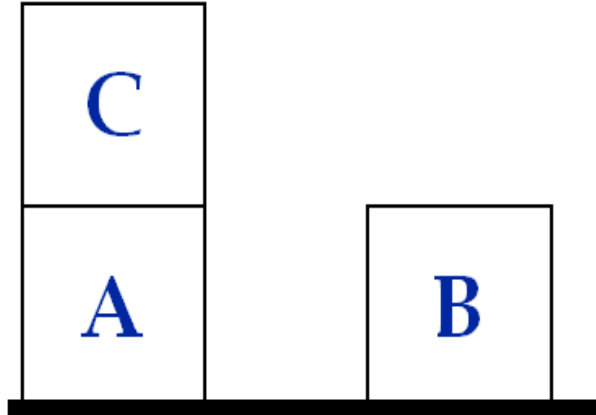
---

- Choose the best rule to apply the next rule based on the best available guess.
- Apply the chosen rule to calculate the new problem condition.
- Find out when a solution has been found.
- Detect dead ends so they can be discarded and direct system effort in more useful directions.
- Find out when a near-perfect solution is found.

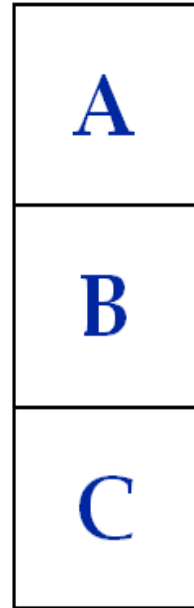
# Example Problem Instance: Sussman Anomaly

- The Sussman Anomaly shows the limitations of non-interleaved planning methods.
- The anomaly will show that naively pursuing one subgoal X after you satisfy the other subgoal Y, may not work because steps required to accomplish X may undo steps of subgoal Y.

**Initial State:**

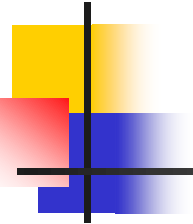


**Goal:**



Initial State: (on-table A) (on C A) (on-table B)

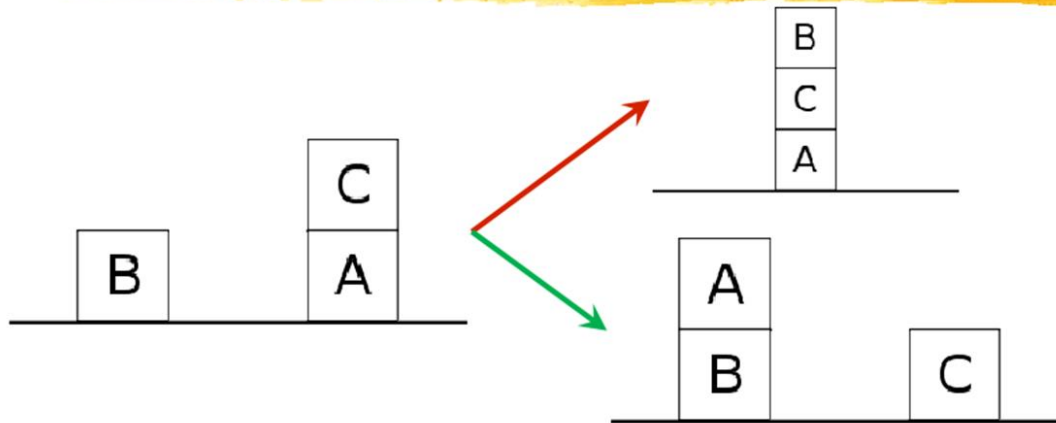
(clear B) (clear C) Goal: (on A B) (on B C)



Goal:



## Sussman Anomaly

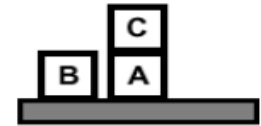


- Final state requires  $\text{On}(A,B)$  and  $\text{On}(B,C)$
- Red arrow focus on subgoal:  $\text{On}(B,C)$  -- Now trying to put A on top of B cannot be done without undoing  $\text{On}(B,C)$
- Bottom diagram tries to focus on subgoal:  $\text{On}(A,B)$  first; but now trying to put B on top of C would cause  $\text{On}(A,B)$  to be undone!

# Anomaly Illustrates the Need for Interleaved Plans

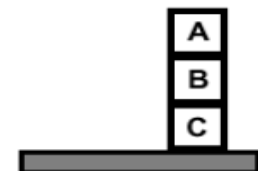
START

*On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)*



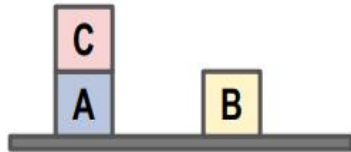
*On(A,B) On(B,C)*

FINISH





# Partial Solutions

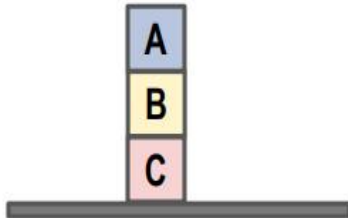


$\text{On}(C, A), \text{On}(A, \text{Table}), \text{On}(B, \text{Table}), \text{Clear}(C), \text{Clear}(B)$

$\text{Clear}(C), \text{Clear}(B)$

**Move(B, C)**

$\text{On}(A, B), \text{On}(B, C)$



## ACTIONS:

**Move(X, Y)**

**Precond:**  $\text{Clear}(X), \text{Clear}(Y)$

**Effect:**  $\text{On}(X, Y)$

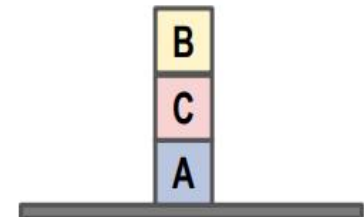
**Move(X, Table)**

**Precond:**  $\text{Clear}(X)$

**Effect:**  $\text{On}(X, \text{Table})$

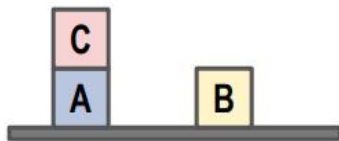
We use  $\text{Move}(B, C)$  to achieve the sub-goal,  $\text{On}(B, C)$ .

But if we apply this move at the beginning, we get:



Which is not what we want !!

# Partial Solutions

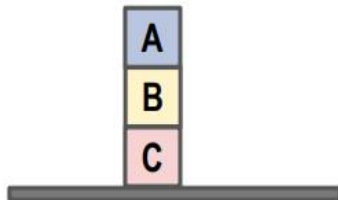


On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C), Clear(B)

Move(B, C)

On(A, B), On(B, C)



## ACTIONS:

Move(X, Y)

Precond: Clear(X), Clear(Y)

Effect: On(X, Y)

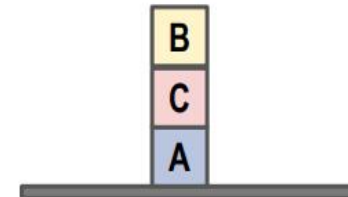
Move(X, Table)

Precond: Clear(X)

Effect: On(X, Table)

We use Move(B, C) to achieve the sub-goal, On(B, C).

But if we apply this move at the beginning, we get:



Which is not what we want !!

## Partial Solutions



On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C)

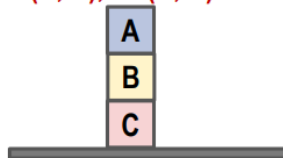
Move(C, Table)

Clear(A), On(C, Table)

Clear(A), Clear(B)

Move(A, B)

On(A, B), On(B, C)



## ACTIONS:

Move(X, Y)

Precond: Clear(X), Clear(Y)

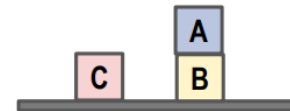
Effect: On(X, Y)

Move(X, Table)

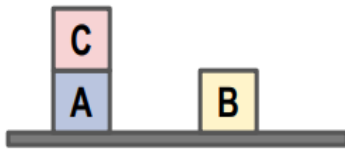
Precond: Clear(X)

Effect: On(X, Table)

The sub-goal On(A, B) is achieved by moving C to the table and then moving A to top to B. But this gives us:



But this too is not what we want !!



$\text{On}(C, A), \text{On}(A, \text{Table}), \text{On}(B, \text{Table}), \text{Clear}(C), \text{Clear}(B)$

$\text{Clear}(C)$

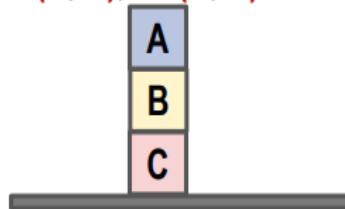
**Move(C, Table)**

$\text{Clear}(A), \text{On}(C, \text{Table})$

$\text{Clear}(A), \text{Clear}(B)$

**Move(A, B)**

$\text{On}(A, B), \text{On}(B, C)$



## ACTIONS:

**Move(X, Y)**

Precond:  $\text{Clear}(X), \text{Clear}(Y)$

Effect:  $\text{On}(X, Y)$

**Move(X, Table)**

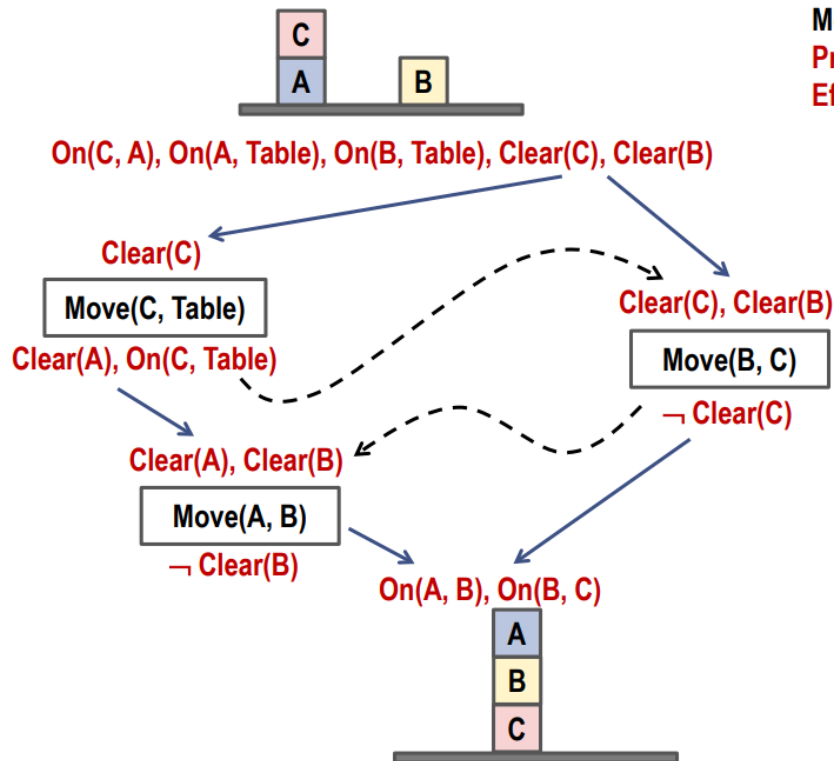
Precond:  $\text{Clear}(X)$

Effect:  $\text{On}(X, \text{Table})$

**Move(B, C)** removes the  $\text{Clear}(C)$  predicate which is essential for **Move(C, Table)**. Hence **Move(C, Table)** must precede **Move(B, C)**.

Can **Move(B, C)** and **Move(A, B)** be executed in any order?

## Ordering Partial Solutions



### ACTIONS:

**Move(X, Y)**

Precond:  $\text{Clear}(X)$ ,  $\text{Clear}(Y)$

Effect:  $\text{On}(X, Y)$

**Move(X, Table)**

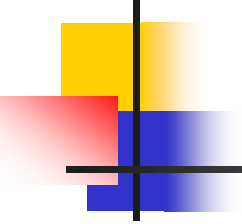
Precond:  $\text{Clear}(X)$

Effect:  $\text{On}(X, \text{Table})$

$\text{Move}(A, B)$  removes the  $\text{Clear}(B)$  predicate which is essential for  $\text{Move}(B, C)$ . Hence  $\text{Move}(B, C)$  must precede  $\text{Move}(A, B)$ .

Therefore the only total order is:

1.  $\text{Move}(C, \text{Table})$
2.  $\text{Move}(B, C)$
3.  $\text{Move}(A, B)$



---

## Partial-order planning

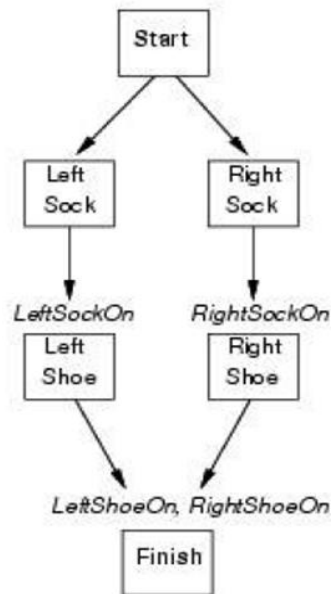
- Progression and regression planning are *totally ordered plan search* forms.
  - They cannot take advantage of problem decomposition.
    - Decisions must be made on how to sequence actions on all the subproblems
- Least commitment strategy:
  - Delay choice during search

- **Basic Idea: Make choices only that are relevant to solving the current part of the problem**

## Partial-order planning

- Any planning algorithm that can place two actions into a plan without commitment about which comes first is a Partial Order Plan

Partial Order Plan:

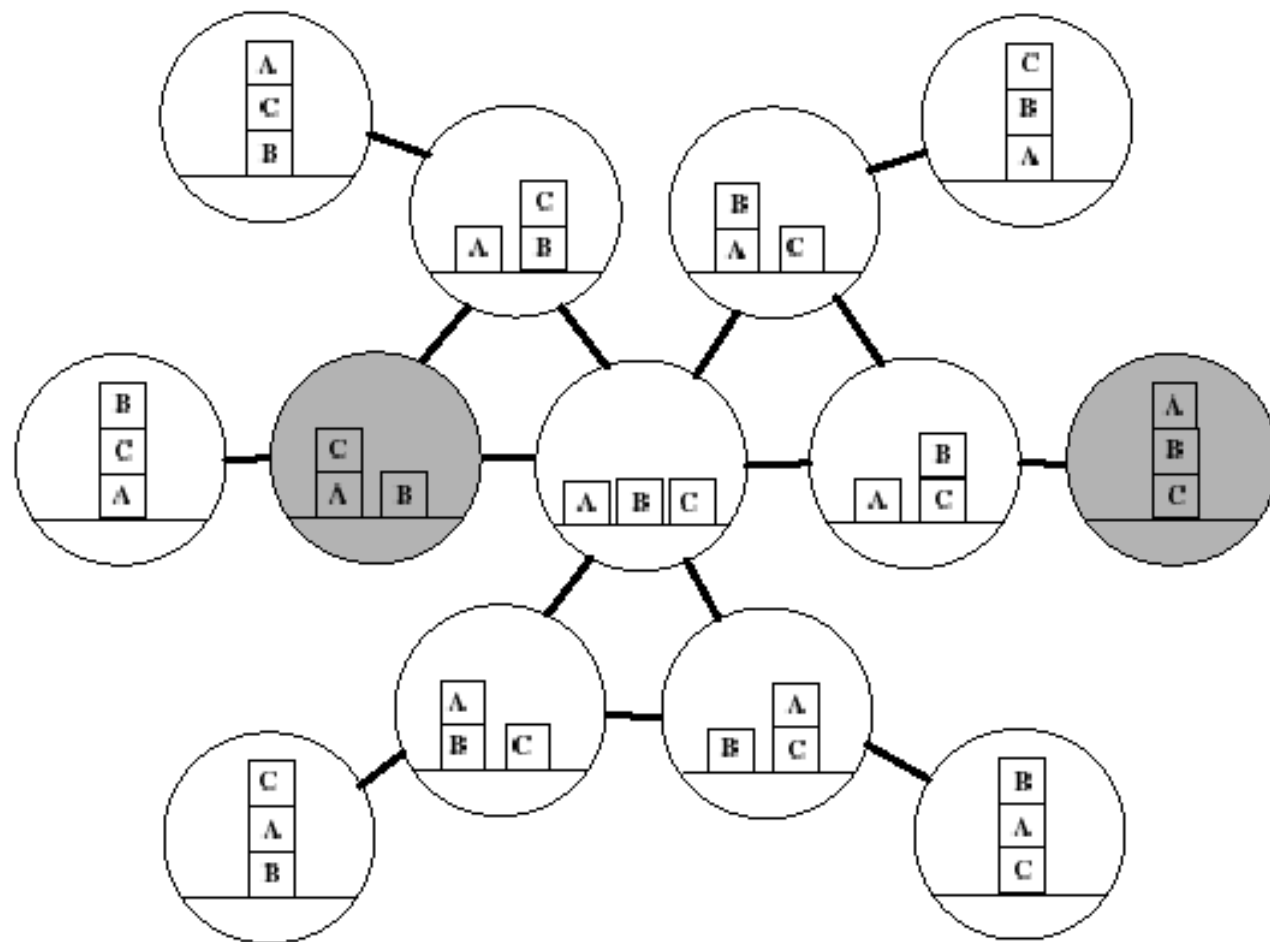


Total Order Plans:



- A partial order plan is executed by repeatedly choosing *any* of the possible next actions.
  - This flexibility is a benefit in non-cooperative environments.

# Search Space: Blocks World







# Target stack plan

---

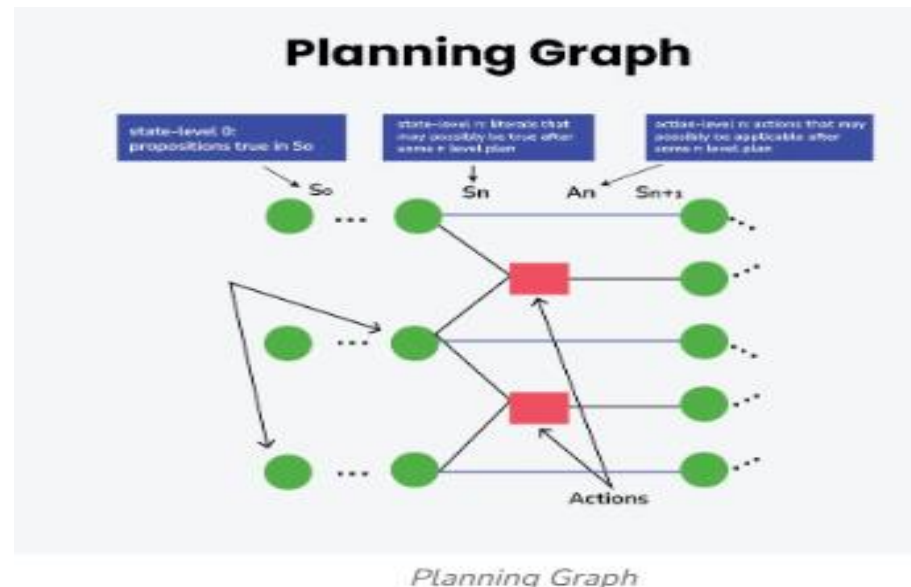
- It is one of the most important planning algorithms used by STRIPS. • **STanford Research Institute Problem Solver**
- Stacks are used in algorithms to capture the action and complete the target.
- A knowledge base is used to hold the current situation and actions.
- A target stack is similar to a node in a search tree, where branches are created with a choice of action.
- Construct and analyze the compact structure called graph planning.

# What is a Planning Graph?

- A Planning Graph is a data structure primarily used in automated planning to find solutions to planning problems.
- Represents a planning problem's progression through a series of levels that describe states of the world and the actions that can be taken.
- **State levels** consist of nodes representing logical propositions or facts about the world. Each successive state level contains all the propositions of the previous level plus any that can be derived by the actions of the intervening action levels.
- **Action** levels contain nodes representing actions. An action node connects to a state level if the state contains all the preconditions necessary for that action. Actions in turn can create new state conditions, influencing the subsequent state level.

Two Types of **Edges or Arcs**: One connecting state nodes to actions  
another connecting action to state nodes

- **Mutual Exclusion (Mutex) Relationships**: At each level, certain pairs of actions or states might be mutually exclusive, meaning they cannot coexist or occur together due to conflicting conditions or effects.
- These mutex relationships are critical for reducing the complexity of the planning problem by limiting the combinations of actions and states that need to be considered.





# Plan Generation:

## Search space of world states

---

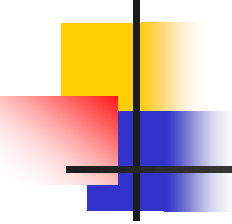
Planning as a (graph) search problem

- Nodes: world states
- Arcs: actions
- Solution: path from the initial state to one state that satisfies the goal
  - Initial state is fully specified
  - There are many goal states



## Planning Graphs

- Consists of a sequence of levels that correspond to time steps in the plan
- Each level contains a set of actions and a set of literals that *could* be true at that time step depending on the actions taken in previous time steps
- For every +ve and -ve literal *C*, we add a *persistence action* with precondition *C* and effect *C*



## Planning graphs

- Used to achieve better heuristic estimates.
  - A solution can also directly extracted using GRAPHPLAN.
- Consists of a sequence 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.
    - *Literals* = all those that *could* be true at that time step, depending upon the actions executed at the preceding time step.
    - *Actions* = all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.



# Cake Example

---

- They work only for propositional problems.
- Example:

Init(Have(Cake))

Goal(Have(Cake)  $\wedge$  Eaten(Cake))

Action(Eat(Cake), PRECOND: Have(Cake)

EFFECT:  $\neg$ Have(Cake)  $\wedge$  Eaten(Cake))

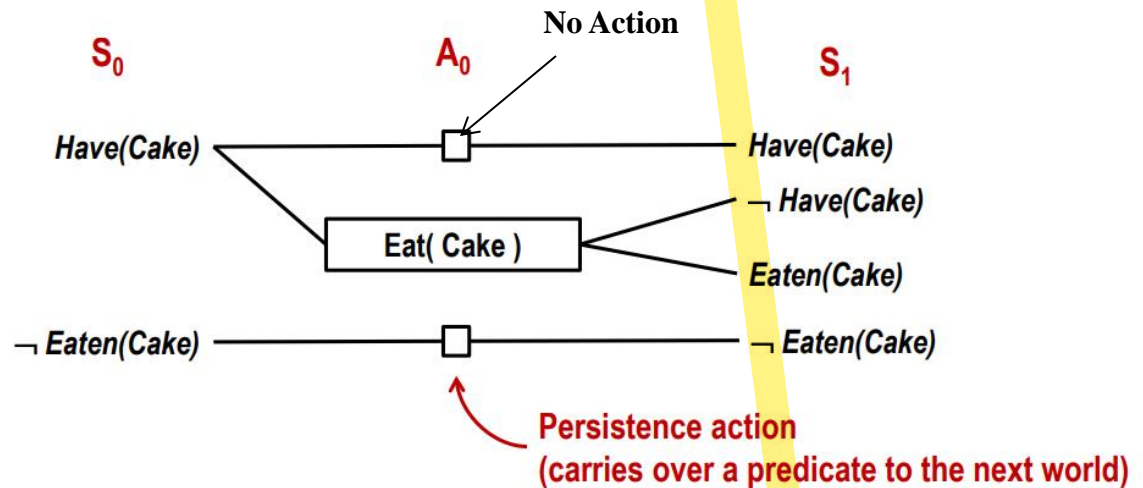
Action(Bake(Cake), PRECOND:  $\neg$  Have(Cake)

EFFECT: Have(Cake))

# Planning Graph

Start: Have(Cake)  
Finish: Have(Cake)  $\wedge$  Eaten(Cake)

Op( **ACTION**: Eat(Cake),  
**PRECOND**: Have(Cake),  
**EFFECT**: Eaten(Cake)  $\wedge$   $\neg$ Have(Cake))  
Op( **ACTION**: Bake(Cake),  
**PRECOND**:  $\neg$ Have(Cake),  
**EFFECT**: Have(Cake))

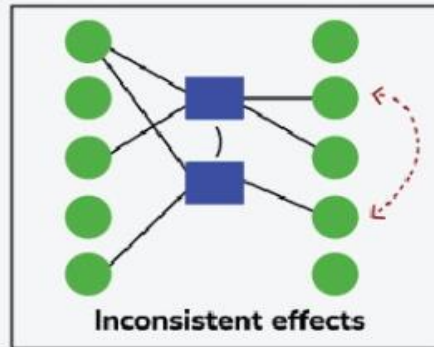
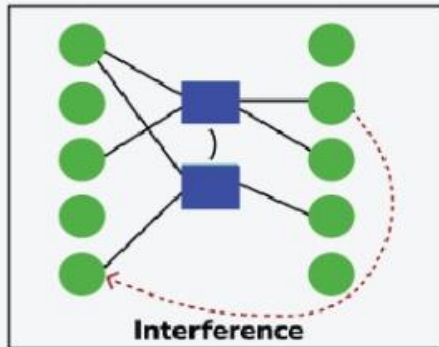




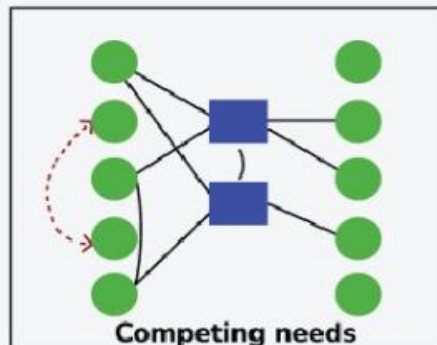
# Mutex Conditions Between Actions

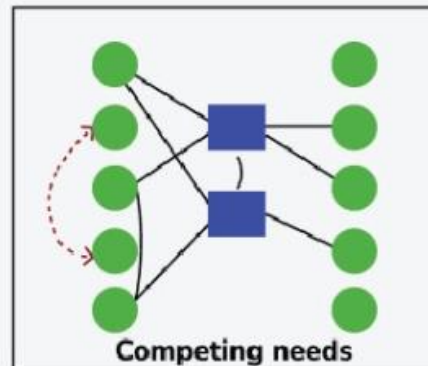
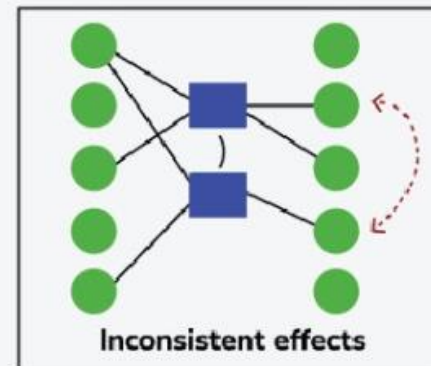
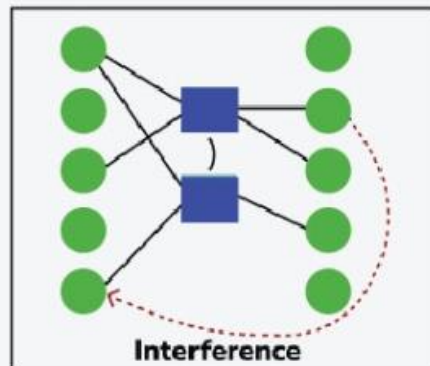
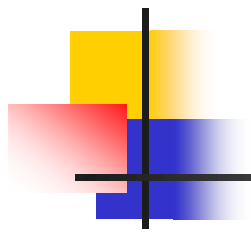
**Inconsistent Effects:** One action negates the effect of another.

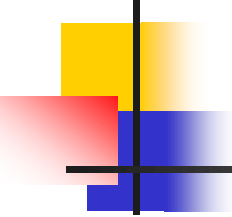
- **Interference:** One action deletes a precondition or creates an add-effect of another.
- **Competing Needs:** Precondition of action a and precondition of action b cannot be true simultaneously.



Two literals are mutually exclusive if one is the negation of the other.



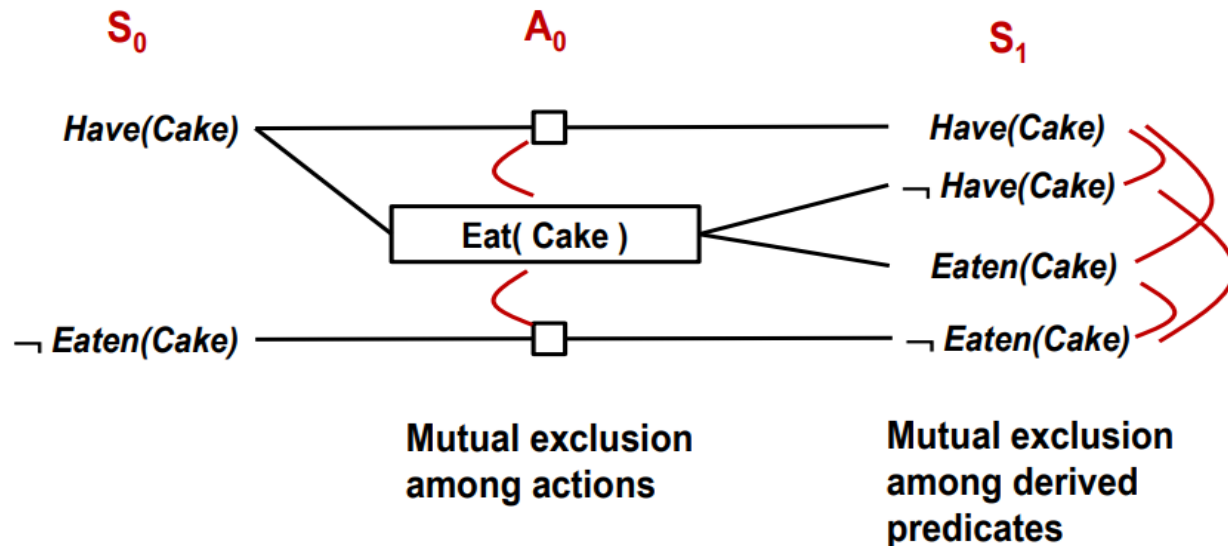




The mutex links define the set of states, revealing which combinations of literals are not possible together.

## Mutex Links in a Planning Graph

If  $\text{Eat}(\text{Cake})$  is not performed,  $\neg\text{Have}(\text{Cake})$  and  $\text{Eaten}(\text{Cake})$  cannot be true together.



if we eat the cake, we cannot have the cake at the same time.

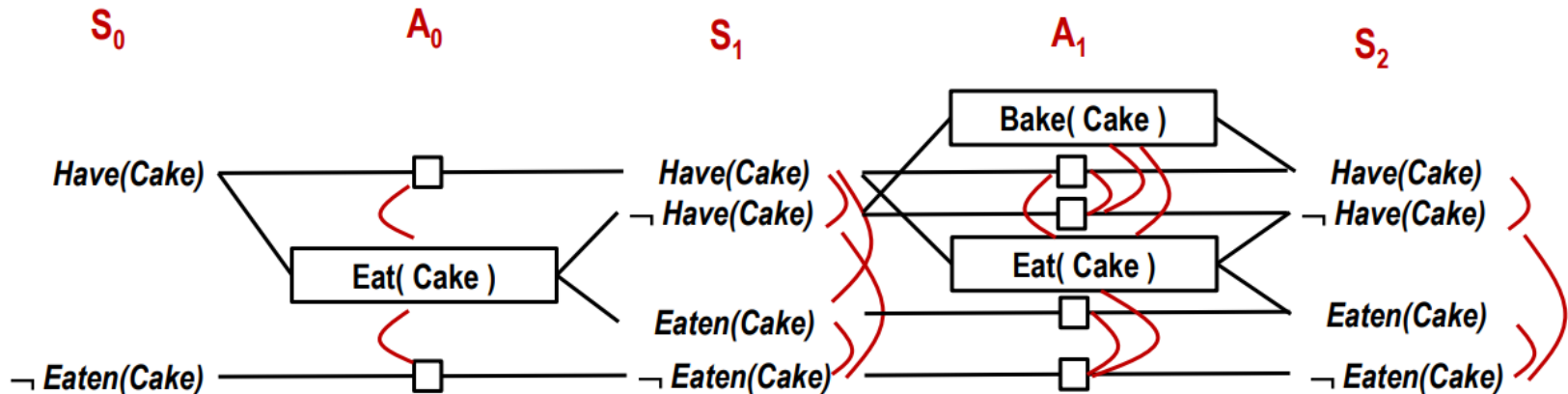
Thus, **Have(Cake)** and **Eaten(Cake)** would be mutually exclusive.

# Second Action Level (A1) and State Level (S2)

## Planning Graph

Op( **ACTION:** Eat(Cake),  
**PRECOND:** Have(Cake),  
**EFFECT:** Eaten(Cake)  $\wedge$   $\neg$ Have(Cake))

Op( **ACTION:** Bake(Cake),  
**PRECOND:**  $\neg$ Have(Cake),  
**EFFECT:** Have(Cake))



Start:  $\text{Have(Cake)}$   
 Finish:  $\text{Have(Cake)} \wedge \text{Eaten(Cake)}$

In the world  $S_2$  the goal predicates exist without mutexes, hence we need not expand the graph any further

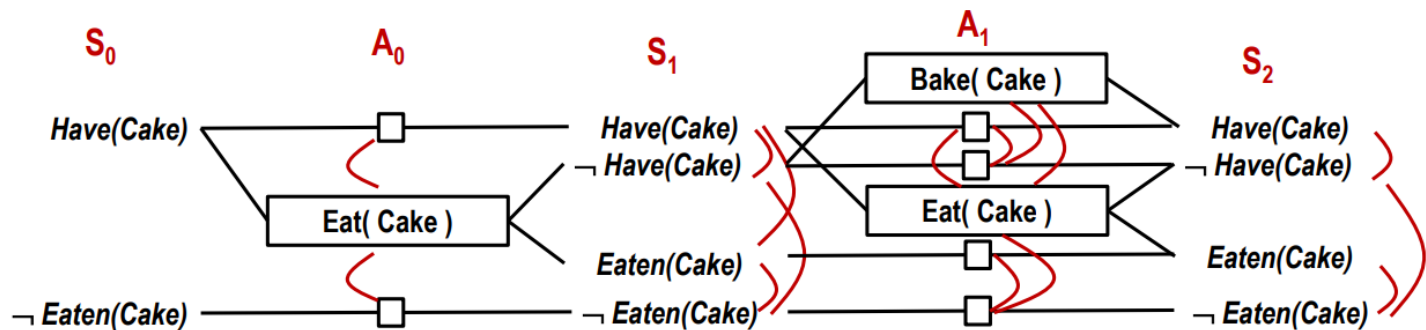
**Bake(Cake):** The precondition for this action is  $\neg \text{Have(Cake)}$ . If  $\neg \text{Have(Cake)}$  is true in  $S_1$ , applying  $\text{Bake(Cake)}$  results in **Have(Cake)** in  $S_2$ .

This action does not affect the **Eaten(Cake)** or  $\neg \text{Eaten(Cake)}$  states.

**Eat(Cake):** The precondition for this action is **Have(Cake)**. If **Have(Cake)** is true in  $S_1$ , applying  $\text{Eat(Cake)}$  results in  $\neg \text{Have(Cake)}$  and **Eaten(Cake)** in  $S_2$ .

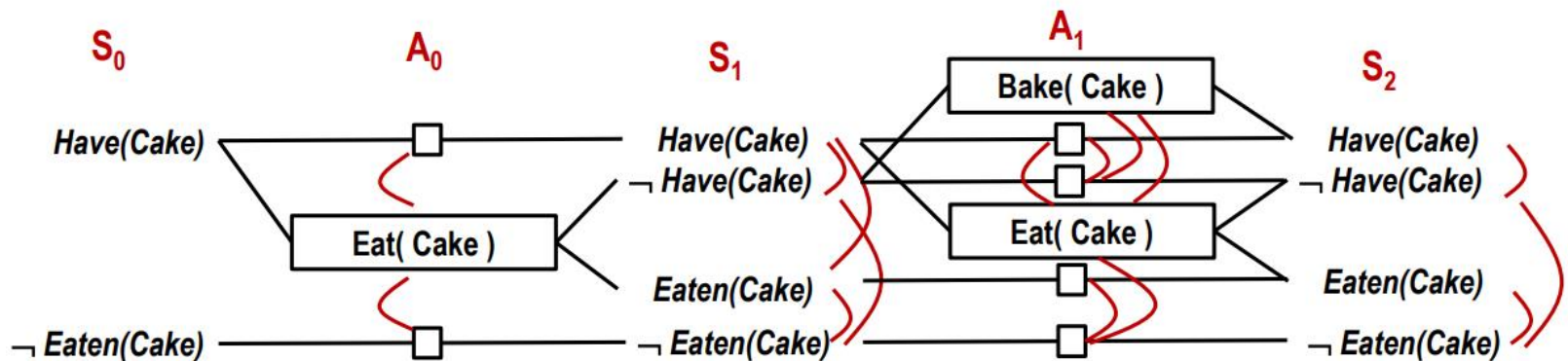
## Mutex Actions

- Mutex relation exists between two actions if:
  - **Inconsistent effects** – one action negates an effect of the other  
 $\text{Eat( Cake )}$  causes  $\neg \text{Have(Cake)}$  and  $\text{Bake( Cake )}$  causes  $\text{Have(Cake)}$
  - **Interference** – one of the effects of one action is the negation of a precondition of the other  
 $\text{Eat( Cake )}$  causes  $\neg \text{Have(Cake)}$  and the persistence of  $\text{Have( Cake )}$  needs  $\text{Have(Cake)}$
  - **Competing needs** – one of the preconditions of one action is mutually exclusive with a precondition of the other  
 $\text{Bake( Cake )}$  needs  $\neg \text{Have(Cake)}$  and  $\text{Eat( Cake )}$  needs  $\text{Have(Cake)}$



## Mutex Literals

- Mutex relation exists between two literals if:
  - One is the negation of the other, or
  - Each possible pair of actions that could achieve the two literals is mutually exclusive (inconsistent support)





---

## Function GraphPLAN( problem )

*// returns solution or failure*

graph  $\leftarrow$  Initial-Planning-Graph( problem )

goals  $\leftarrow$  Goals[ problem ]

do

if goals are all non-mutex in last level of graph then do

    solution  $\leftarrow$  Extract-Solution( graph )

    if solution  $\neq$  failure then return solution

    else if No-Solution-Possible (graph )

        then return failure

graph  $\leftarrow$  Expand-Graph( graph, problem )



---

## Function GraphPLAN( problem )

*// returns solution or failure*

graph  $\leftarrow$  Initial-Planning-Graph( problem )

goals  $\leftarrow$  Goals[ problem ]

do

if goals are all non-mutex in last level of graph then do

    solution  $\leftarrow$  Extract-Solution( graph )

    if solution  $\neq$  failure then return solution

    else if No-Solution-Possible (graph )

        then return failure

graph  $\leftarrow$  Expand-Graph( graph, problem )



## Finding the plan

- Once a world is found having all goal predicates without mutexes, the plan can be extracted by solving a constraint satisfaction problem (CSP) for resolving the mutexes
- Creating the planning graph can be done in polynomial time, but planning is known to be a PSPACE-complete problem. The hardness is in the CSP.
- The plan is shown in blue below

