COL864: Special Topics in AI
Semester II, 2020-21

Task Planning

Rohan Paul

Outline

- Last Class
  - A* search
- This Class
  - Symbolic Representations for Task Planning
    - How can we represent such problems?
    - How can we (efficiently) search for a plan?
- Reference Material
  - Primary reference are the lecture notes. For basic background refer to AIMA Classical Planning Ch. 10 (Sec 10.1 - 10.3)

Acknowledgements

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Nicholas Roy, Wolfram Burgard, Dieter Fox, Sebastian Thrun, Siddharth Srinivasa, Dan Klein, Pieter Abbeel, Max Likhachev and others.

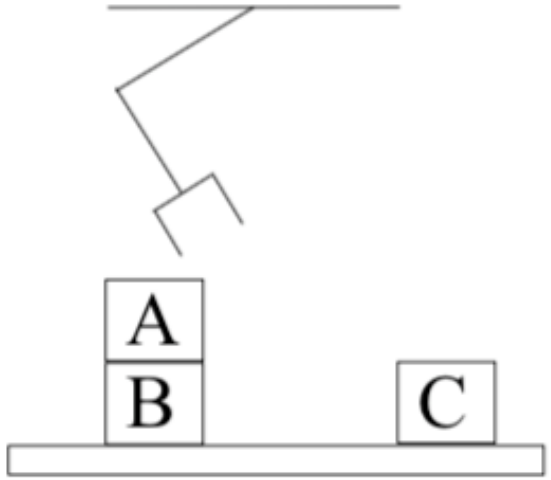
Task Planning

- Motion planning
  - Generating collision free trajectories.
- Task Planning
  - Presence of Semantic constraints
    - Till now, generating collision free trajectories.
    - Now, consider when an action a_i must be performed before action a_j (opening a box before placing and object inside it)
  - Need to scale decision making
    - Consider an assembly task: packing objects in a container and transporting it.
    - Intuitively, we solve such problems by thinking about abstract actions “picking an object and placing in the box” assuming that precise motions can be determined later.
    - Characteristic of long-horizon tasks.
- Planning vs. Scheduling
  - Scheduling
    - Tasks are fixed (scheduling classes in a week). There may be constraints on the tasks. Don’t need to determine ”which” tasks are to be done.
  - Planning
    - We need to decide “which” set of tasks or steps we need to them as well as to schedule them.

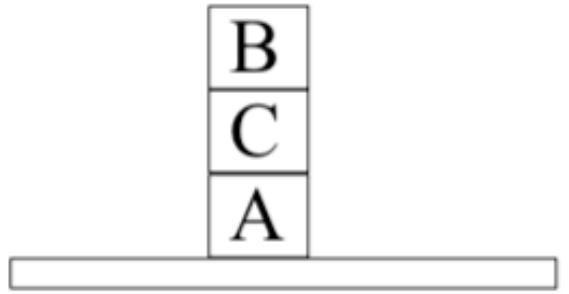
Other examples:
https://www.youtube.com/watch?v=lY4PKBqp9ZM&t=179s

Example: Blocksworld

- Problem
  - Re-order the blocks from the start state to the goal state.
  - Assume that the arm can reach/move all the top blocks.
  - Planning task: determining the order of actions.
- Abstraction
  - The precise poses of B and C are less relevant, what matters is whether B is on C or not.
  - The precise motion of the gripper is less relevant. Its symbolic effect matters, i.e., the block went on top of another.
- Symbolic Representation
  - States: “On(X, Y)” that aggregate low level positions that represent this relationship.
  - Actions: “Move(X, Y)” to denote all ways in which the robot can move X on Y.



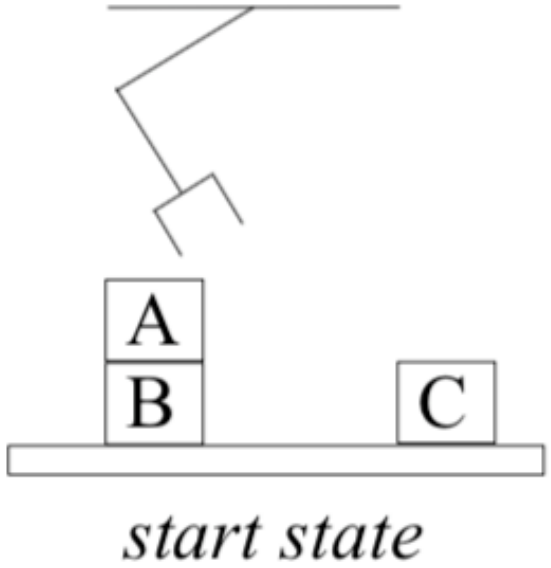
start state



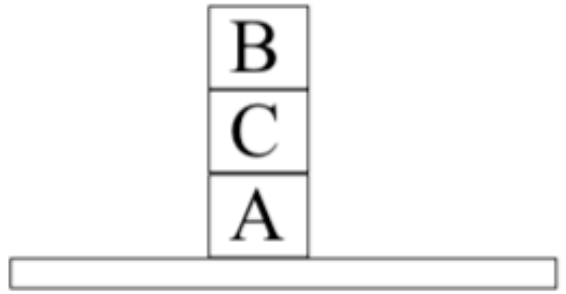
goal state

Symbolic Planning

- World or Domain
  - Describe the world (domain) using logic.
- Actions
  - Describe the actions available to the agent as
    - When they can be executed.
    - What happens if they are.
- Initial and Goal states
- Task
  - Find a plan that moves the agent from start state to goal



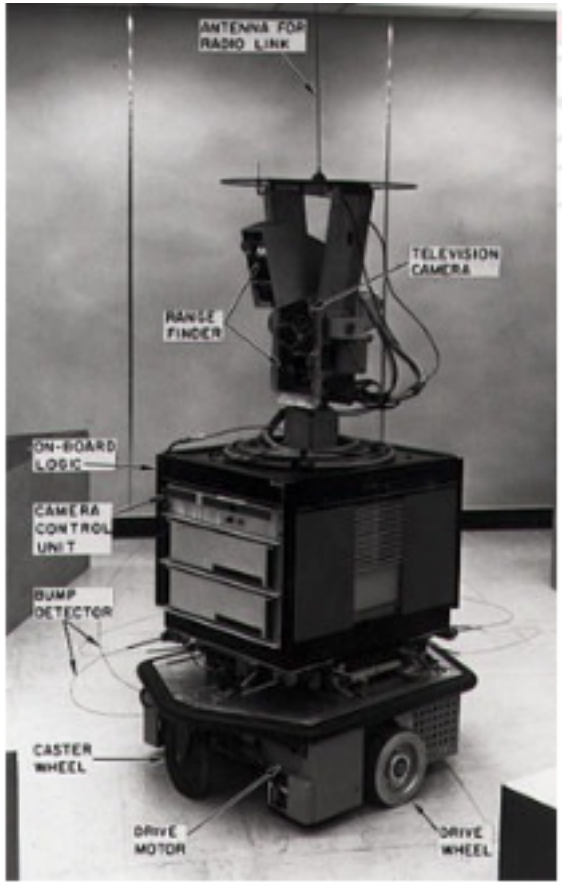
start state



goal state

STRIPS Planning

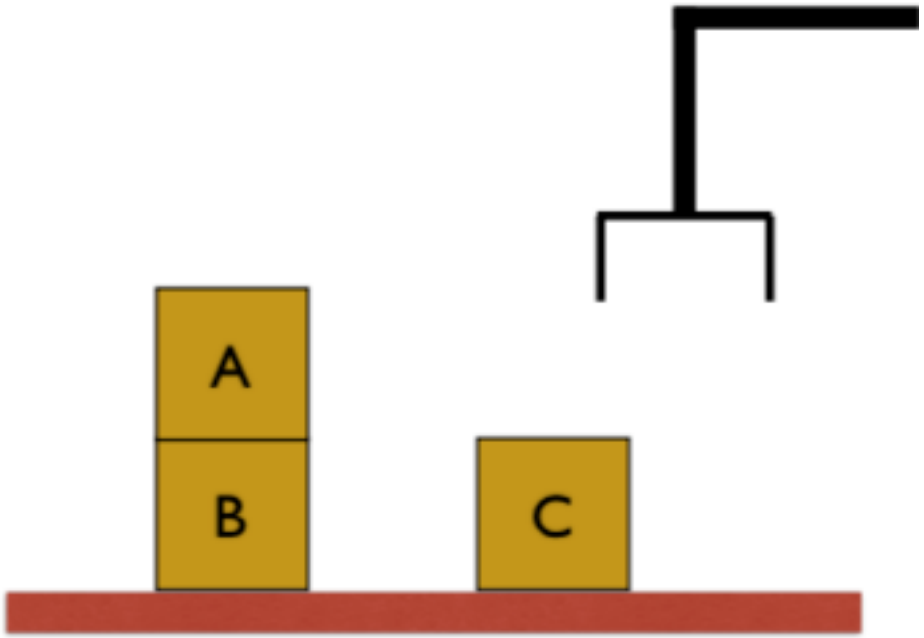
- STRIPS: Stanford Research Institute Problem Solver
  - Represent the world using a knowledge-base of first-order logic.
  - Actions change what is currently true.
  - Describe the actions available, defined by preconditions and effects
- Planning Domain Description Language
  - Standard language for planning domains
  - International programming competitions
- Separate definitions of:
  - A domain, which describes a class of tasks.
    - Predicates and operators.
- A task, which is an instance of domain.
  - Objects.
  - Start and goal states.
- A predicate is a first-order logic function returning True or False, given a set of objects.



PDDL: Predicates

- A predicate returns True or False given a set of objects.

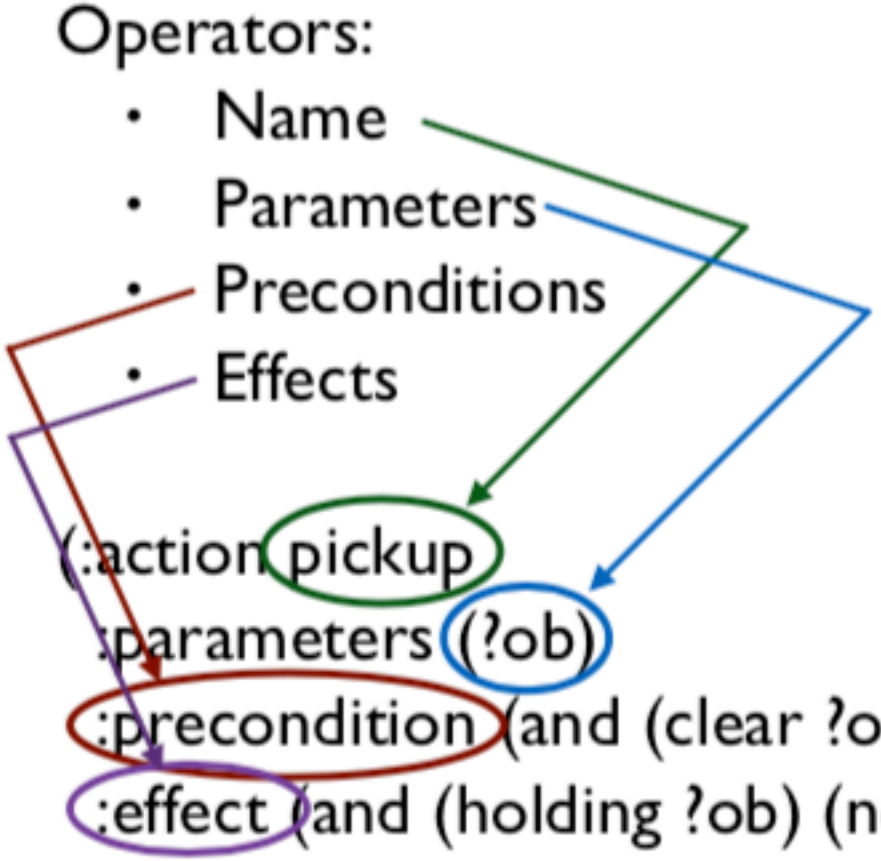
```
(define (domain blocksworld)

(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))
```



PDDL: Operators

Operators:

- Name
- Parameters
- Preconditions
- Effects



```
(:action pickup
 :parameters (?ob)
 :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
 :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
              (not (arm-empty))))
```

PDDL: Problem Instance

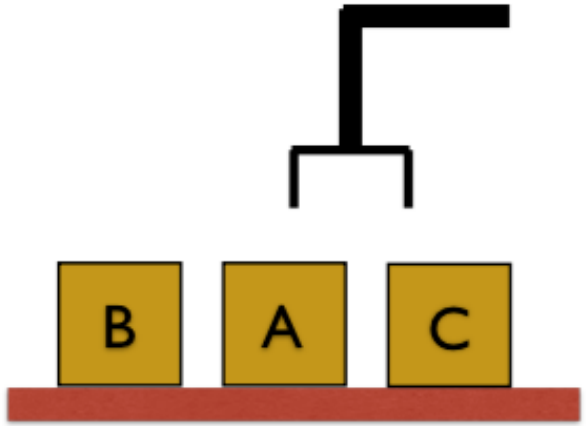

```
(define (problem pb3)
   (:domain blocksworld)
   (:objects a b c)
   (:init (on-table a) (on-table b)   (on-table c)
          (clear a)  (clear b) (clear c) (arm-empty))
   (:goal (and (on a b) (on b c))))
```

Start state

Goal state

PDDL: States

- A state describes the configuration of the world at a moment of time.
- A conjunction of positive literal predicates.
- Closed world assumption
  - Those not mentioned are assumed to be False.
  - Implications
    - Avoid inference
    - No uncertainty about which actions can be executed.
    - No uncertainty about the goal.
    - Consistency with knowledge base semantics.



(on-table a)
(on-table b)
(on-table c)
(clear a)
(clear b)
(clear c)
(arm-empty)

PDDL: Operators

- Implicit Markov assumption.

```
(:action putdown
  :parameters  (?ob)
  :precondition (and (holding ?ob))
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))
```

PDDL: Goals

- A conjunction of literal predicates.
  - **(and (on a b) (on b c))**
- Predicates not listed are don’t cares.
- Each goal is thus a *partial* state expression.
  - Implies a set of goal states.

PDDL: Action Execution

Start state:

(on-table a) (on-table b) (on-table c)
(clear a) (clear b) (clear c) (arm-empty)

Action: pickup(a)

- Check preconditions
- Decide to execute.
- Delete negative effects.
- Add positive effects.

```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))
```

Next state:

~~(on-table a)~~ (on-table b) (on-table c)
~~(clear a)~~ (clear b) (clear c) ~~(arm-empty)~~
(holding a)

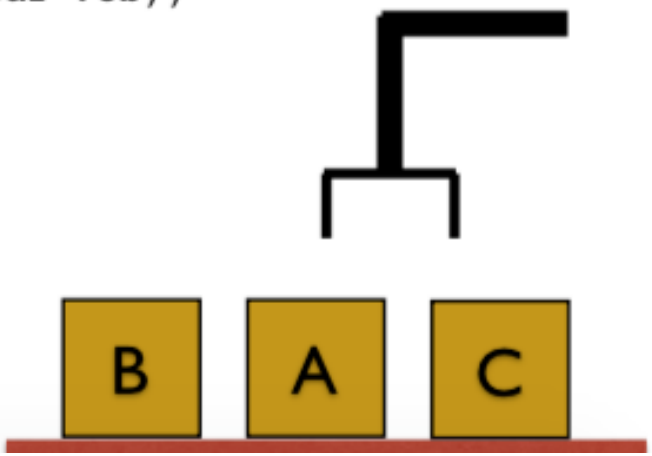
PDDL: Example

State: (on-table a) (on-table b) (on-table c)
(clear a) (clear b) (clear c) (arm-empty))

Goal: (and (on a b) (on b c))

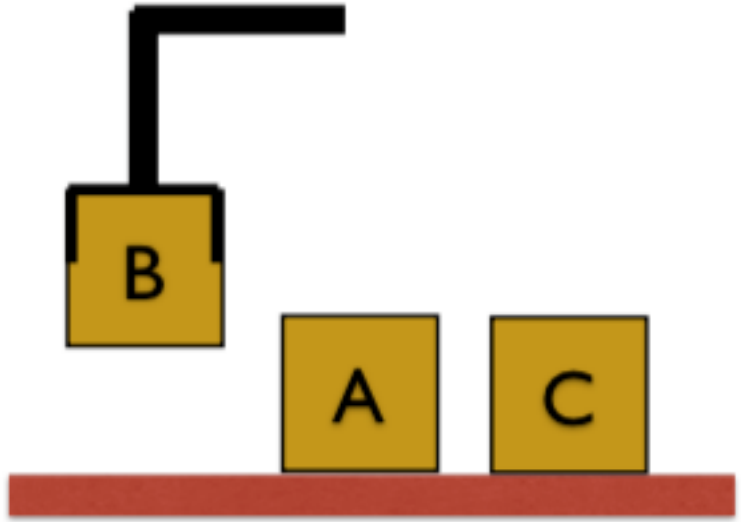
```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob))
               (not (on-table ?ob))
               (not (arm-empty))))
```

pickup(b)



PDDL: Example

after pickup(b) …



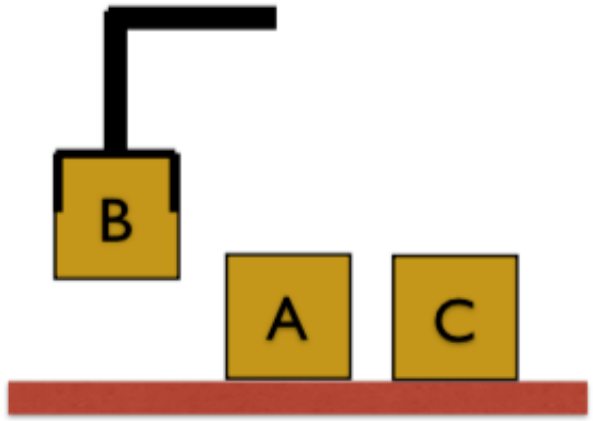
State: (on-table a) ~~(on-table b)~~ (on-table c)
(clear a) ~~(clear b)~~ (clear c) ~~(arm-empty)~~ (holding b))
Goal: (and (on a b) (on b c))

PDDL: Example

State: (on-table a) (on-table c)
(clear a) (clear c) (holding b))
Goal: (and (on a b) (on b c))

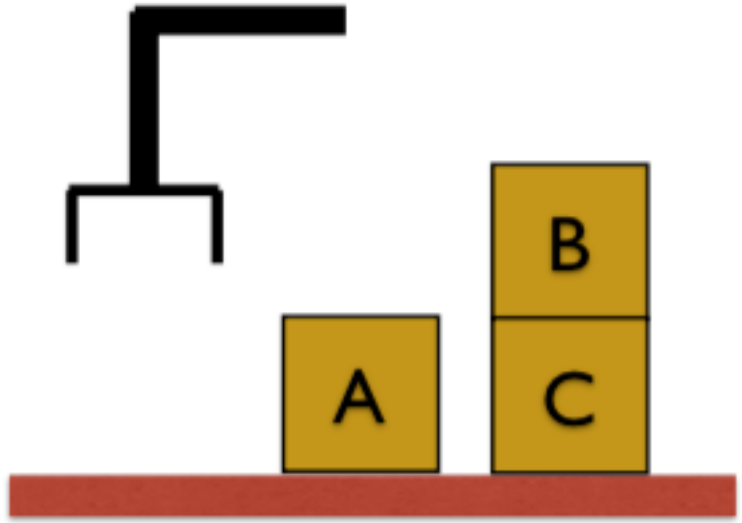
```
(:action stack
  :parameters  (?ob ?underob)
  :precondition (and  (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))
```

stack(b, c)



PDDL: Example

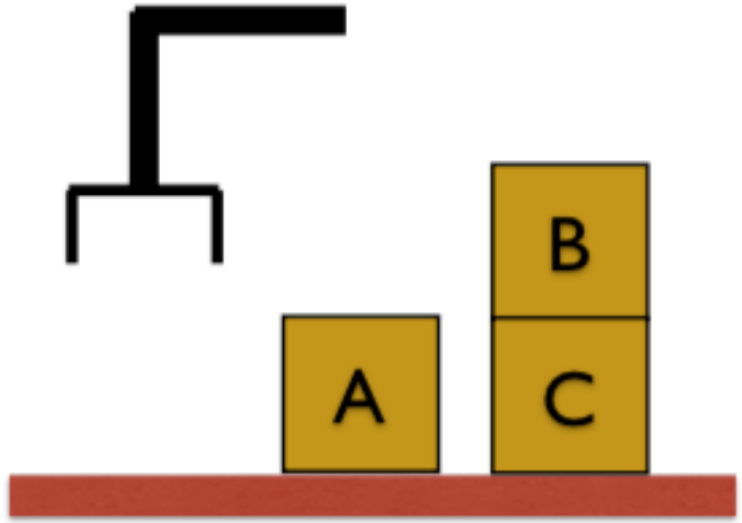
after stack(b, c) …



State: (on-table a) (on-table c)
(clear a) ~~(clear c) (holding b)~~
(arm-empty) (clear b) (on b, c))
Goal: (and (on a b) (on b c))

PDDL: Example

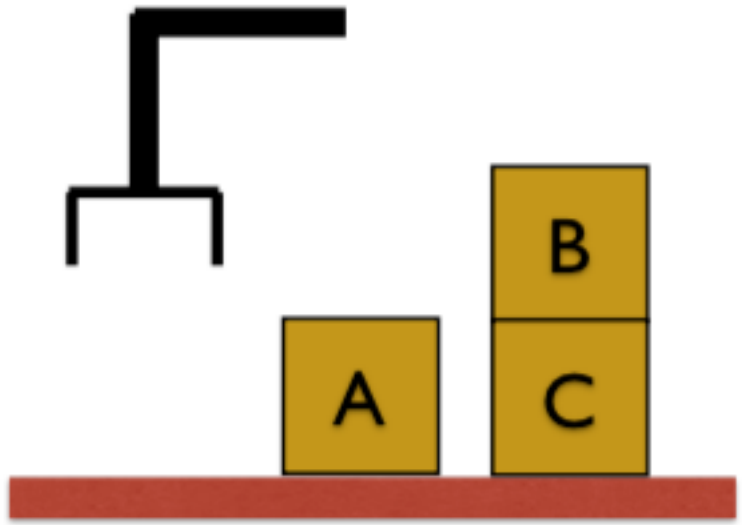
after stack(b, c) …



State: (on-table a) (on-table c)
(clear a) ~~(clear c) (holding b)~~
(arm-empty) (clear b) (on b, c))
Goal: (and (on a b) (on b c))

PDDL: Example

after stack(b, c) …



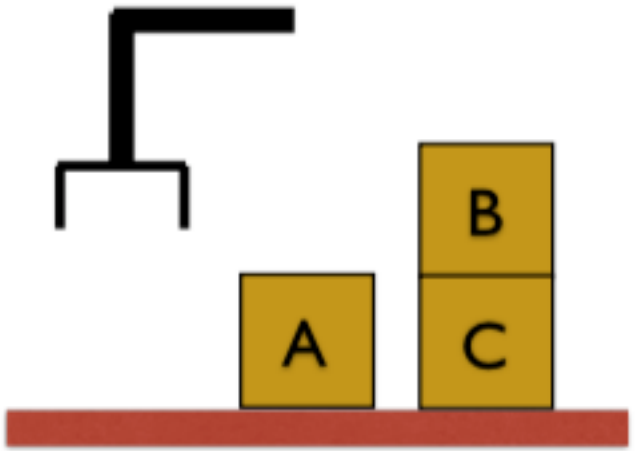
State: (on-table a) (on-table c)
(clear a) ~~(clear c) (holding b)~~
(arm-empty) (clear b) (on b, c))
Goal: (and (on a b) (on b c))

PDDL: Example

State: (on-table a) (on-table c)
(clear a) (arm-empty) (clear b) (on b, c))
Goal: (and (on a b) (on b c))

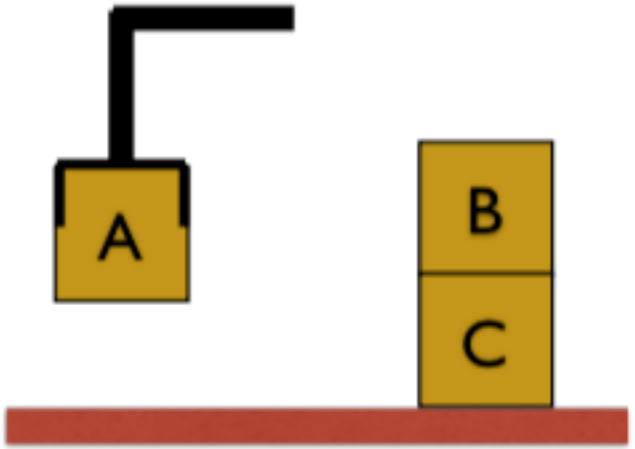
```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))
```

pickup(a)



PDDL: Example

after pickup(a)



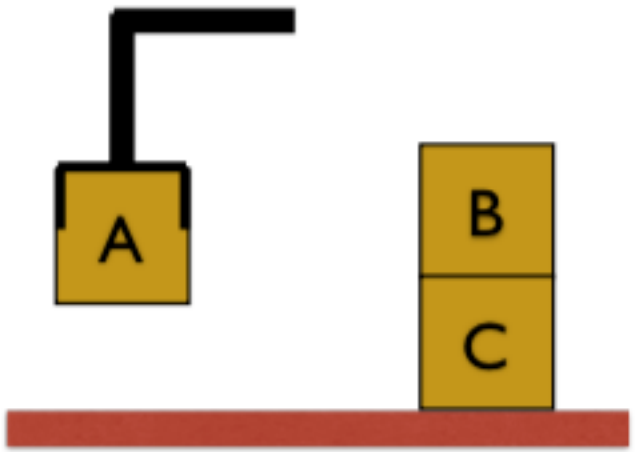
State: ~~(on-table a)~~ (on-table c)
~~(clear a) (arm-empty)~~ (clear b) (on b, c) (holding a)
Goal: (and (on a b) (on b c))

PDDL: Example

State: (on-table c)
(on b, c) (clear b) (holding a))
Goal: (and (on a b) (on b c))

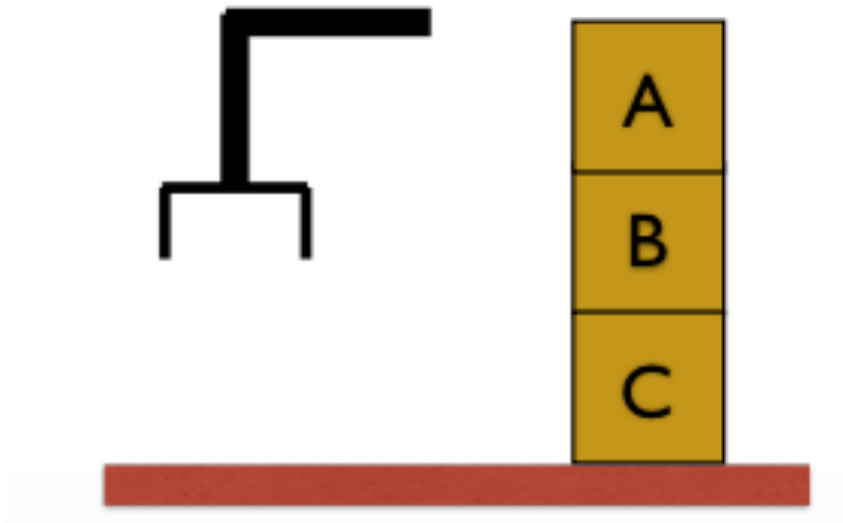
```
(:action stack
  :parameters  (?ob ?underob)
  :precondition (and  (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))
```

stack(a, b)



PDDL: Example

State: (on-table c)
(on a b) (clear b) (on b, c) (holding a))
Goal: (and (on a b) (on b c))

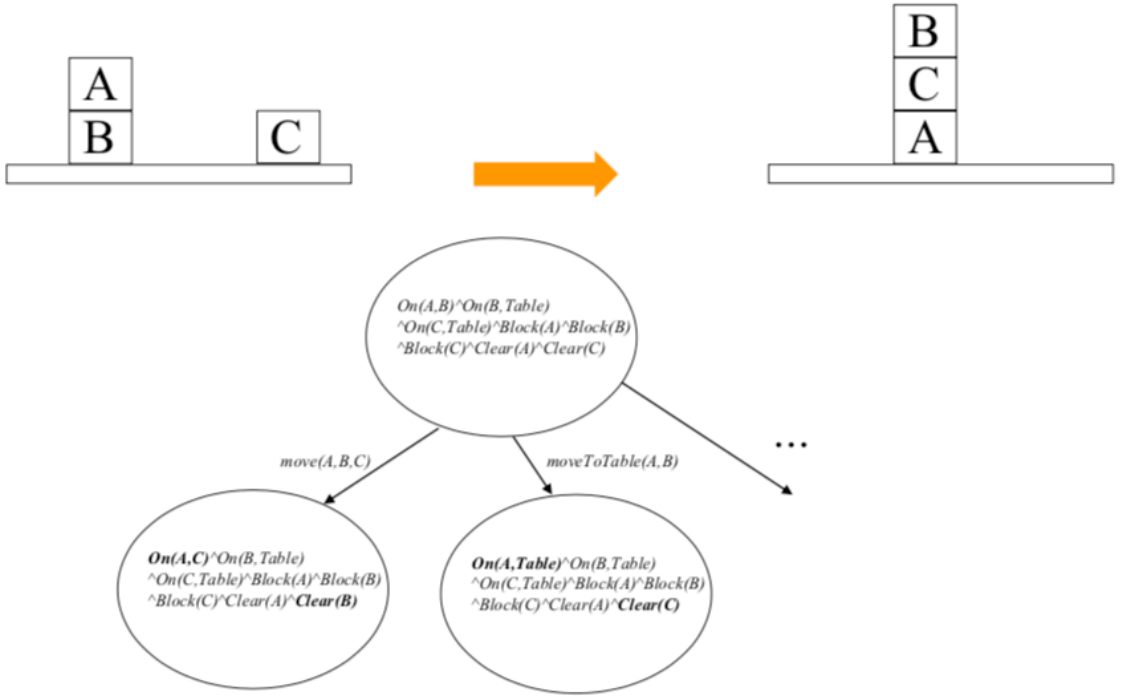


Formal Specification

- Predicates *P*
  - A set of predicates *P*, each with p_n parameters.
- Objects *O*
- Literal predicates *L*
  - A set of predicates from *P* with bound parameters from *O*.
- States
  - A list of positive ground literals, $s \subseteq L$
  - Goal test : a list of positive ground literals, $g \subseteq L$
- Operator List:
  - Name
  - Parameters
  - Preconditions
  - Effects

Planning: As a Graph Search

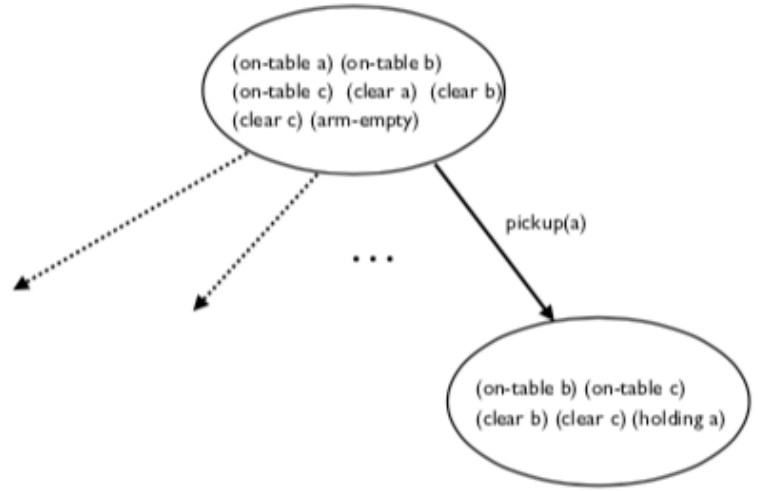
- Search Problem
  - Nodes are states
  - Actions are applicable operators
  - Goal expression as a goal test
- How to search the graph for a plan?
  - Direct search
  - Informed search
    - Domain independent heuristics.



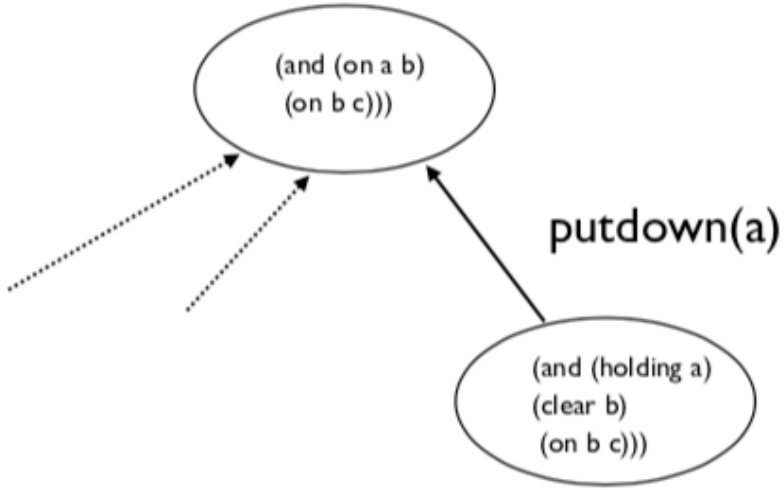
Searching for a Plan

- Forward planning
  - Begin from the initial state and examine the effects of all actions applicable on that state.
  - Determine *successor* states and continue the process till you reach the goal state.
  - Often the branching factor is large.
- Backward or regression search
  - Start at the goal state, apply actions backward until we find the sequence of actions that reaches the initial state.
  - Computes the *predecessor* state s' for a final state s reached by action a.
  - Check for only those actions that are relevant for the goal
    - At least one of the action's effects (positive or negative) should unify with the goal.
  - Often the branching factor is low.

Forward search

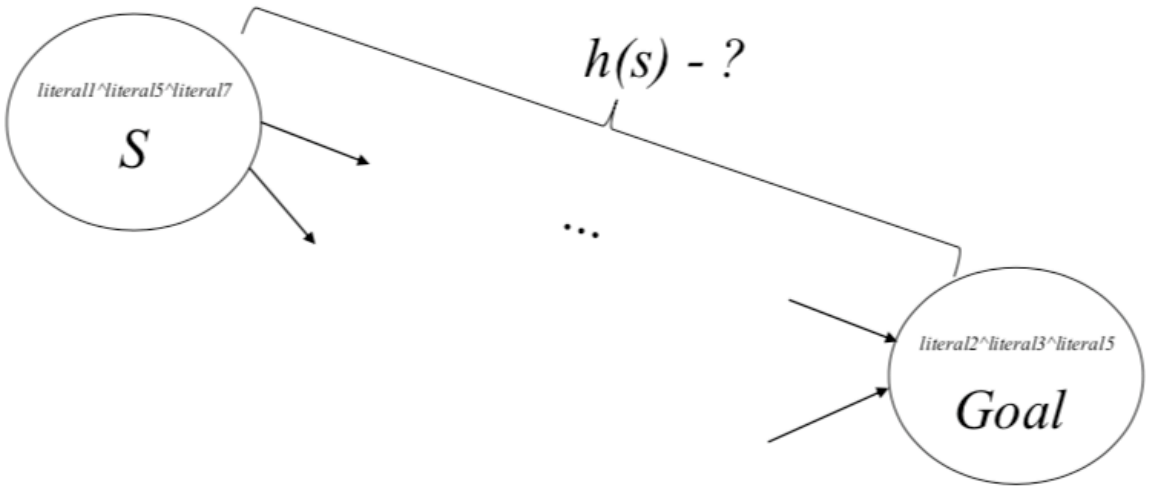


Backward search



Heuristics for Planning

- Informed Search
  - Can try to search with A* by constructing a heuristic.
  - *Domain-specific heuristics* can be derived from the problem structure.
  - Requires careful engineering.
- *Domain-independent* heuristics
  - Once a planning problem is encoded in the PDDL form then can the search be independent of the domain.
  - The number of literals that are NOT yet satisfied.
  - Obtain a relaxation on the problem
  - Work for any problem encoded in PDDL.



Fast Forward (FF) Planner

- Recap: Notion of a relaxed problem
  - Make simplifying assumption on the original planning problem i.e., address a relaxed problem.
  - Solve the relaxed problem optimally. Use the optimal plan in the relaxed problem as a heuristic for the hard problem.
- FF Planner (2000)
  - Relax the problem by deleting the negative effects of actions.
  - Solve the relaxed problem using a planner.

```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))
```

FF Planner

- Deleting negative effects of actions leads to a relaxation of the problem.
- Setup
  - Goal: conjunction of positive literals.
  - Actions
    - Precondition (conjunction of positive literals)
    - Effects (adds and deletes)
- Monotonic progress towards the goal
  - Each action execution monotonically adds the applicable actions.
    - Once a literal is made true, it is not deleted. Progress made is not undone.
  - Some of the the complex interactions between actions is ignored
    - Intuitively, If there is an action that deletes the preconditions for another action then the plan length will be longer as effort is needed to set the deleted predicate as true.
    - By ignoring delete effects, the problem gets relaxed as the actual plan is going to be at least as long.
- Planning
  - Central idea: FF Planner searches for a plan making use of the heuristic computed from the relaxed problem.
  - Other strategies are also applied to aid plan search.

FF Planner

- Still NP hard to compute the optimal solution in the relaxed problem
  - Approximate solution can be found via hill climbing.
- Figure
  - Visualizes state space for two problems using the ignore delete list heuristic.
  - Dots (states), edges (actions) and height (heuristic cost).
  - Hill climbing search will provide an approximate solution.

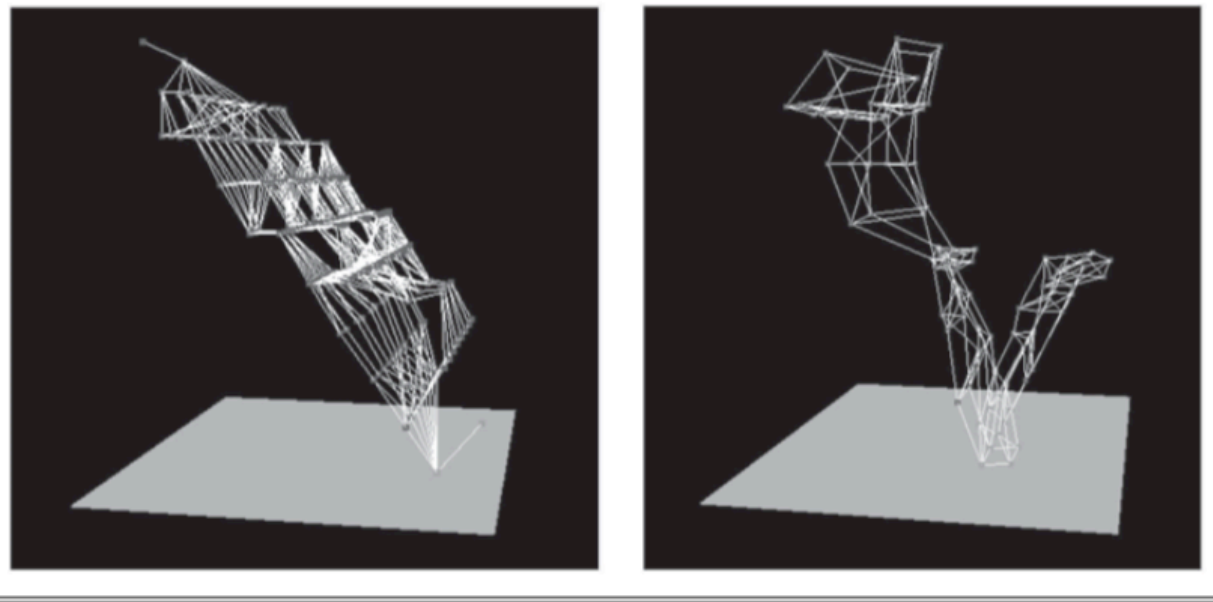


Figure 10.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

AIMA Ch 10

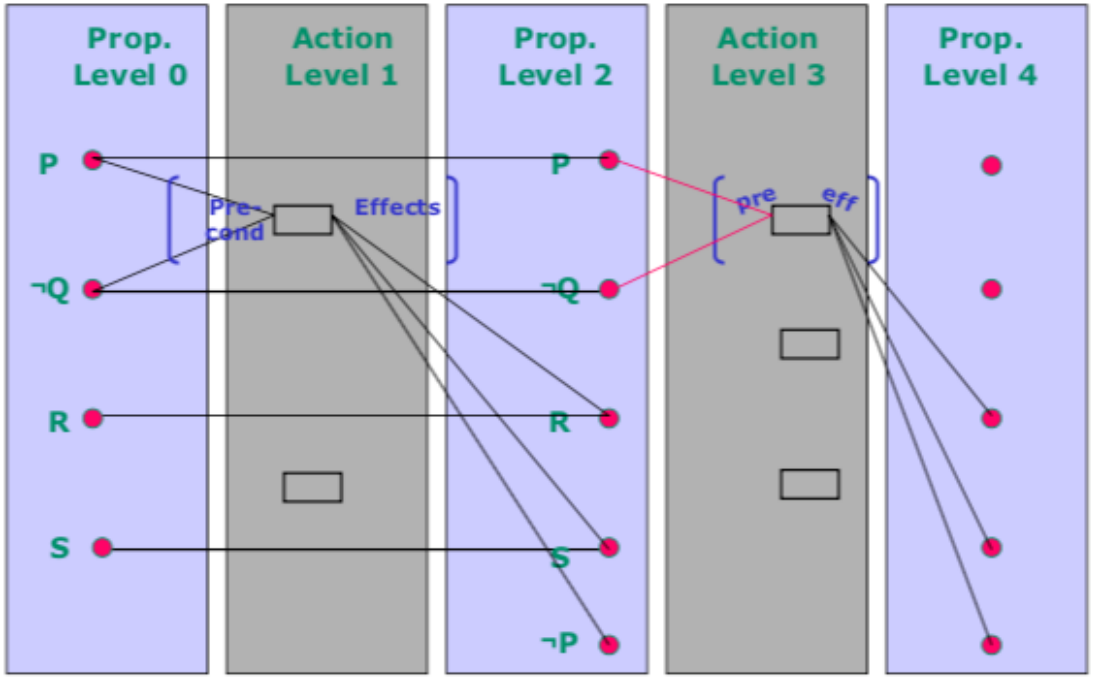
Planning Graphs

- Motivation
  - A planning problem asks if we can reach a goal state from the initial state.
  - If we have a tree of all possible actions from the initial state to successor states and so on.
    - We can determine if there is a plan from start to the goal.
    - ***Problem: this tree is exponential in size.***
- Planning Graphs
  - Planning graph is a polynomial-size approximation to this tree. Trade off is that the planning graph indicates states that can ***possibly*** be reached.
  - It cannot definitively answer if the goal G is attainable from s0
    - But, it can estimate how may steps it takes to reach G.
  - The estimate is always correct when it reports the goal is not reachable.
    - It never overestimates the number of steps (hence an admissible heuristic)

AIMA Ch 10 Sec 10.3

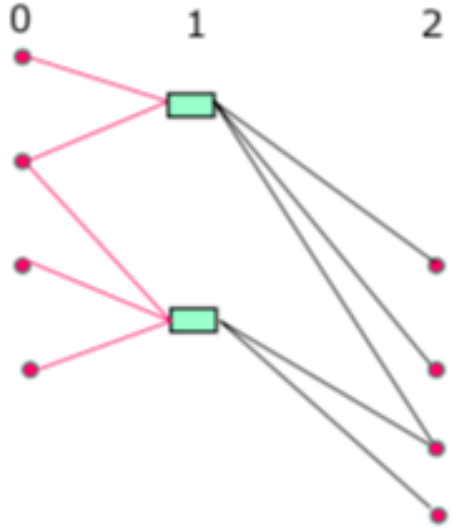
Planning Graphs

- Construction
  - Layered Graph
  - S_i contains all the literals that could hold at time i.
  - A_i contains all the actions that could have their pre-conditions satisfied at time i.
  - No variables. All grounded literals and grounded actions.
- What does a planning graph encode?
  - A planning graph only records a restricted subset of negative interaction between actions.
  - Allows quick elimination of some impossible alternatives in the search process.
  - The level at which a literal appears is a good estimate of how difficult it is to attain a literal from the initial state.
- How are planning graphs used?
  - Computing a heuristic
  - Extracting a plan (GraphPlan)

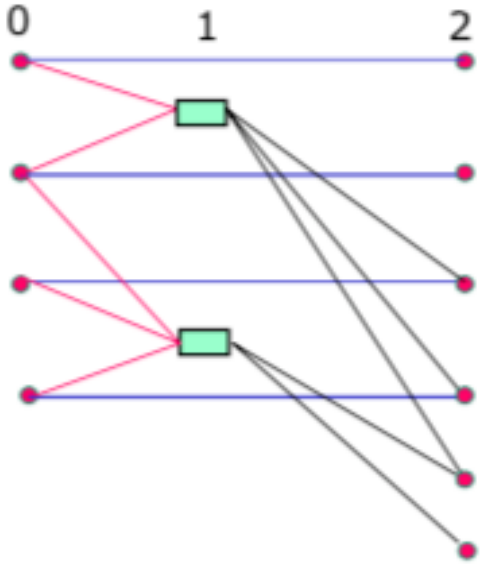


Planning graph: Construction

- Start with the initial state (given)
- Add applicable actions and effects
  - Add actions with satisfied pre-conditions
  - Add all effects of actions at previous levels
  - The action layer will contain all actions whose pre-conditions are satisfied.
- Add maintenance actions
  - Ensures that once a literal is reached it is “maintained” in the planning graph for every subsequence layer.



Add actions with satisfied pre-conditions

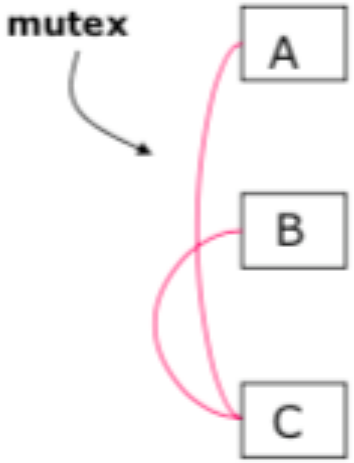


Add maintenance operations.

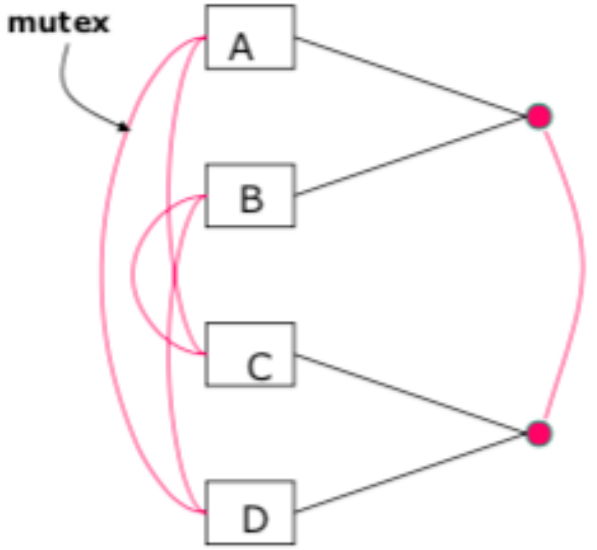
Mutually Exclusive Actions

- Two action instances at level i are mutex if:
  - *Inconsistent effects*
    - The effect of one action is negation of another.
  - *Interference*
    - One action deletes the precondition of another.
  - *Competing needs*
    - The actions have preconditions that are mutex at level i-1
- What do the mutexes model?
  - Some conditions under which two actions cannot be performed together (i.e., only one of them must be selected)
  - Detection of certain obvious flaws (there may be other conflicts that are not encoded by the planning graph)

Actions considered as mutex



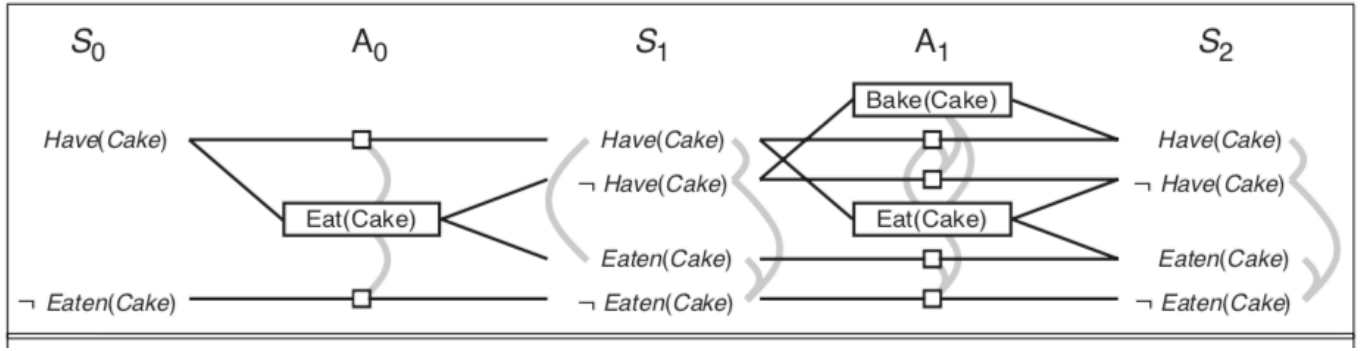
Inconsistent support



Mutually Exclusive Actions

- Inconsistent effects
  - Eat(Cake) and the persistence of Have(cake) have inconsistent effects.
- Interference
  - Eat(Cake) interferes with the persistence of Have(Cake) by negating its pre-conditions.
- Competing needs
  - Bake(Cake) and Eat(Cake) are mutex because they compete on the value of the Have(Cake) precondition.

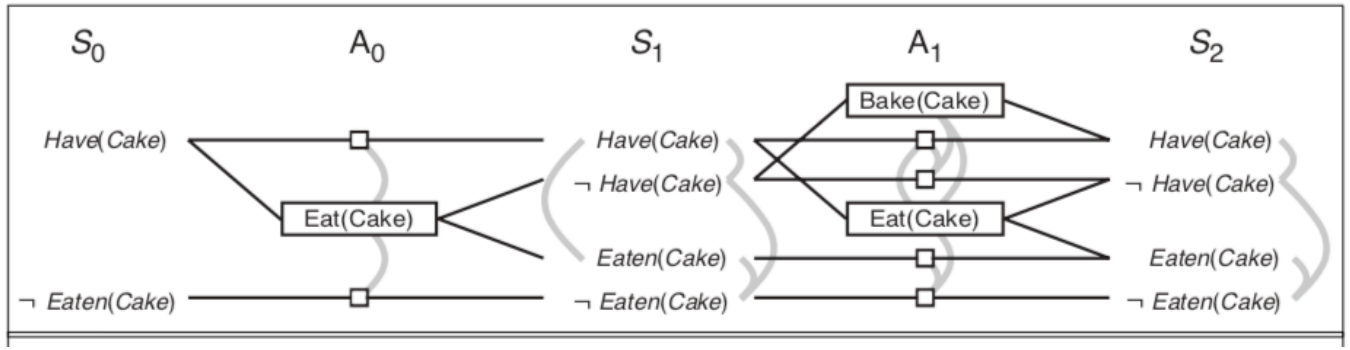
$$
\begin{aligned}
&Init(Have(Cake))\\
&Goal(Have(Cake) \wedge Eaten(Cake))\\
&Action(Eat(Cake)\\
&\quad \text{PRECOND: } Have(Cake)\\
&\quad \text{EFFECT: } \neg Have(Cake) \wedge Eaten(Cake))\\
&Action(Bake(Cake)\\
&\quad \text{PRECOND: } \neg Have(Cake)\\
&\quad \text{EFFECT: } Have(Cake))
\end{aligned}
$$



Mutually Exclusive Propositions

- Two propositions at level l are mutex if:
  - Negation
    - They are negations of one another
  - Inconsistent support
    - All ways of achieving the propositions at level i-1 are pairwise mutex.
- Example
  - Inconsistent support
    - Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of attaining Have(Cake) which is maintenance action is mutex with the only way of achieving Eaten(Cake) which is Eat(Cake)

$$
\begin{aligned}
&Init(Have(Cake))\\
&Goal(Have(Cake) \wedge Eaten(Cake))\\
&Action(Eat(Cake)\\
&\quad \textsc{Precond}: Have(Cake)\\
&\quad \textsc{Effect}: \neg Have(Cake) \wedge Eaten(Cake))\\
&Action(Bake(Cake)\\
&\quad \textsc{Precond}: \neg Have(Cake)\\
&\quad \textsc{Effect}: Have(Cake))
\end{aligned}
$$

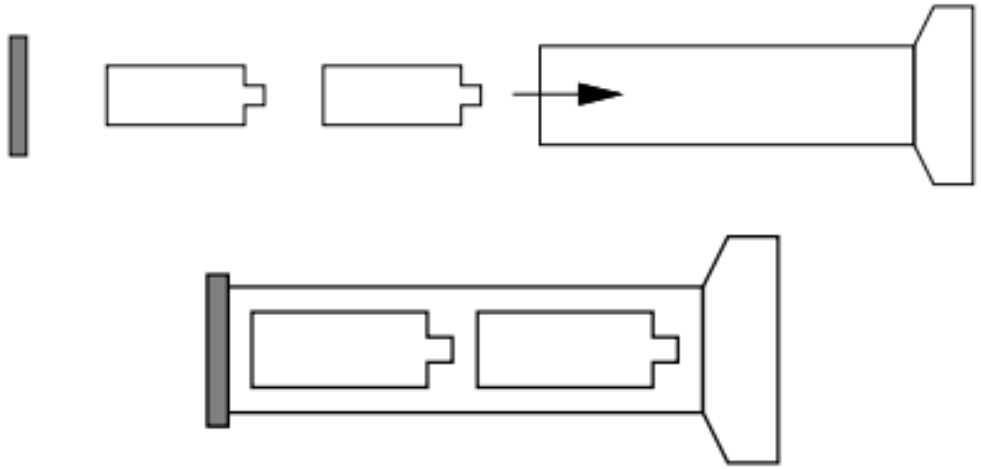


Planning Graphs for Heuristic Estimation

- Using Planning Graphs for Heuristic Estimation
  - The planning graph is polynomial in size (hence tractable to compute)
  - The planning graph can be used to estimate the cost to go from a current state s to the goal, serving as a heuristic.
  - If any goal literal fails to appear in the final level of the planning graph, then the problem is unsolvable.
- Level-cost
  - Cost of attaining any goal g_i from a state s as the level at which g_i first appears in the planning graph constructed from the initial state s.
- Cost of attaining Multiple Goals
  - Max-level heuristic
  - Level-sum heuristic
  - Set level heuristics

Example: flashlight domain

- Problem involves putting batteries into a torchlight.



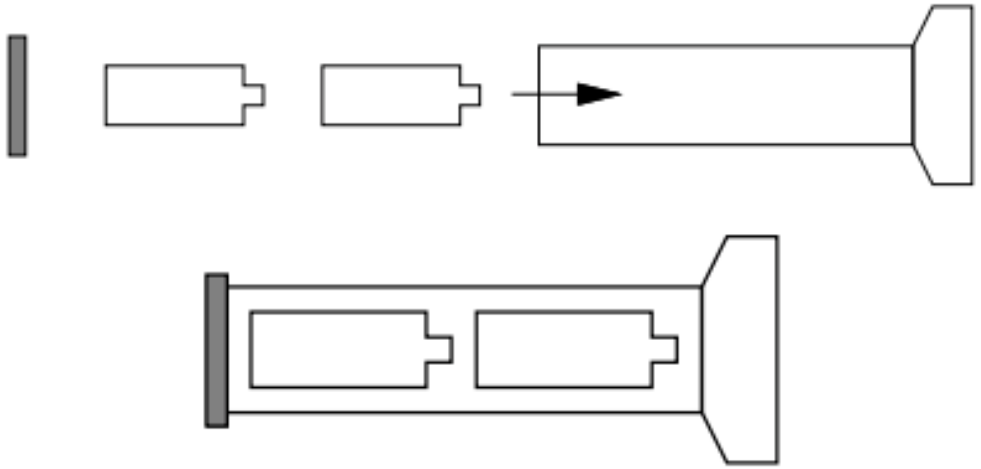
Object Instances

$$I = \{Battery1, Battery2, Cap, Flashlight\}$$

Actions

Name	Preconditions	Effects
$PlaceCap$	$\{\neg On(Cap, Flashlight)\}$	$\{On(Cap, Flashlight)\}$
$RemoveCap$	$\{On(Cap, Flashlight)\}$	$\{\neg On(Cap, Flashlight)\}$
$Insert(i)$	$\{\neg On(Cap, Flashlight), \neg In(i, Flashlight)\}$	$\{In(i, Flashlight)\}$

Example: flashlight domain



State $S = \{On(Cap, Flashlight)\}$

$(On(Cap, Flashlight), \neg In(Battery1, Flashlight1), In(Battery2, Flashlight))$

Goal $G = \{On(Cap, Flashlight), In(Battery1, Flashlight),$

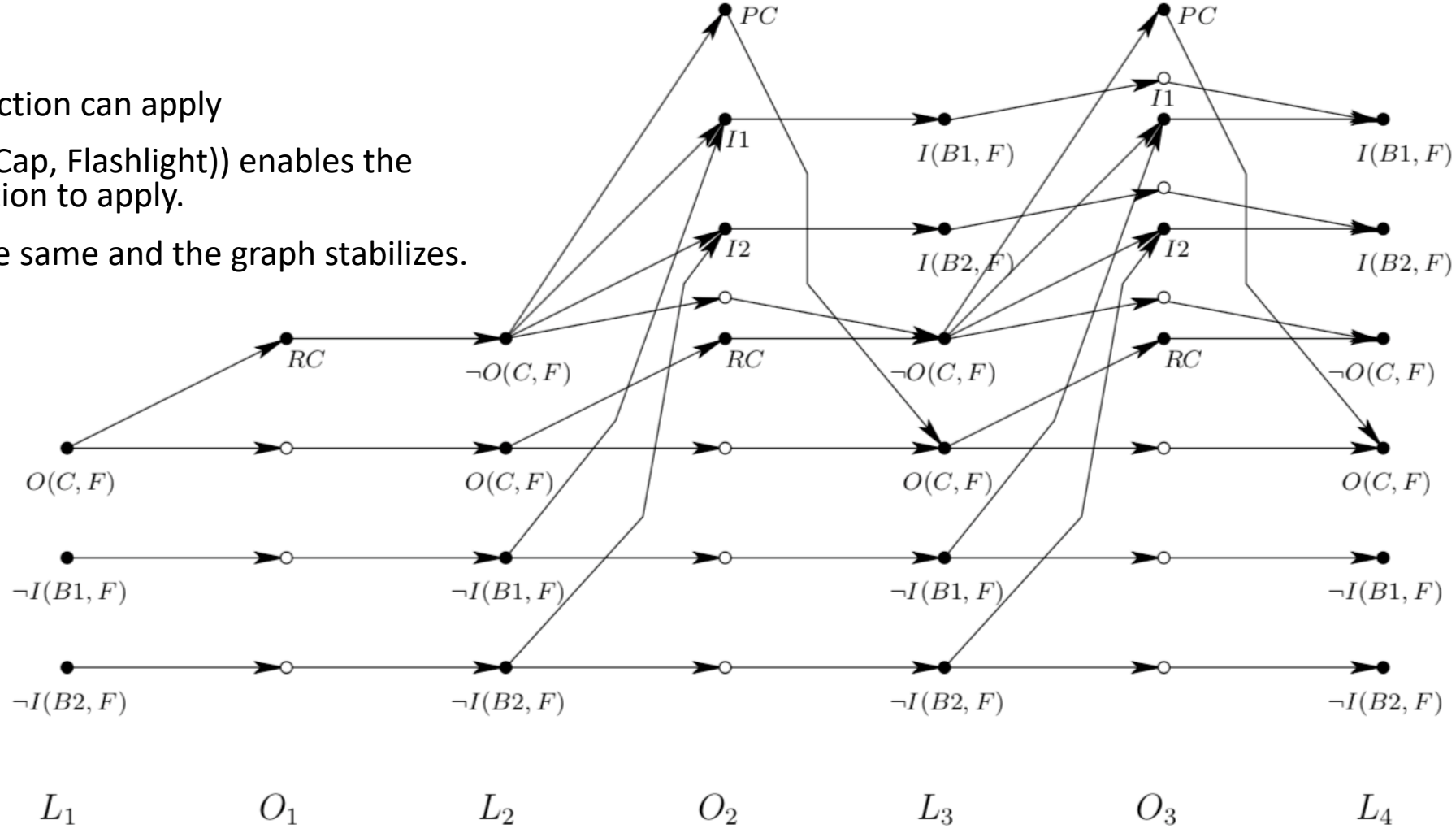
$In(Battery2, Flashlight)\},$

Plan $(RemoveCap, Insert(Battery1), Insert(Battery2), PlaceCap)$

Example: flashlight domain

Planning graph

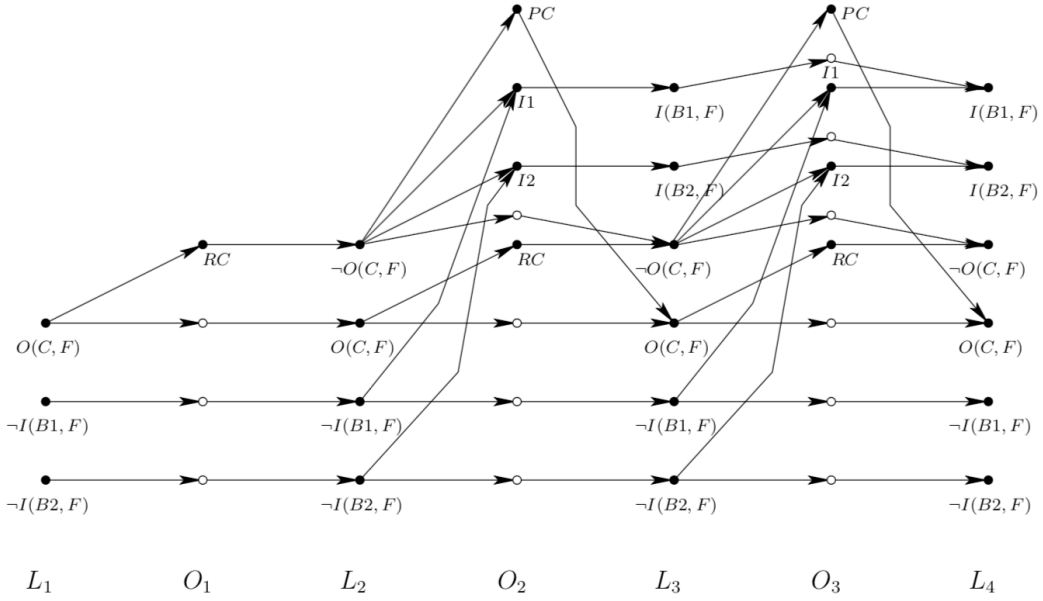
- In L1 only remove cap action can apply
- Appearance of Not(On(Cap, Flashlight)) enables the battery insertion operation to apply.
- Finally, L3 and L4 are the same and the graph stabilizes.



Graph Plan

- Central Idea
  - Use the planning graph to extract a plan
  - Instead of using the graph for providing a heuristic.

- Graph Plan
  - Look for a plan of depth K.
  - Then search for a solution.
  - If you succeed return a plan, else increase the plan depth to K+1



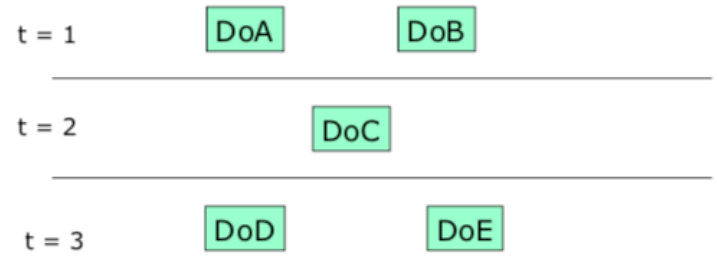
- Interleaves graph extension and plan search
- Once all the goals appear as non-mutex in the graph then call a plan search.

Plan Extraction

- There can be several actions in an action layer.

Planning graph has multiple actions at a level.

A plan of depth k
- has k times steps
- may have multiple parallel actions per time step



- How to extract the plan?
  - Start from layer k and search backwards.
  - Perform an And/Or search.
  - All literals in the target state are to be satisfied (AND part)
  - Try the possible operators under mutex constraints (OR part)

Procedure for plan extraction

- If all the literals in the goal appear at the deepest level and not mutex, then search for a solution for each subgoal at level i
  - For each subgoal at level i
    - Choose an action to achieve it
    - If it's mutex with another action, Fail
  - Repeat for preconditions at level i-2