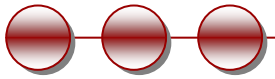
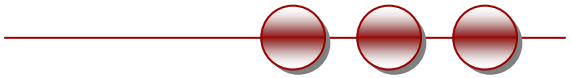


CMSC 671
Fall 2010

Thu 10/19/10

State space planning
Graph-based planning: GraphPlan
Chapter 10

Prof. Laura Zavala, laura.zavala@umbc.edu, ITE 373, 410-455-8775



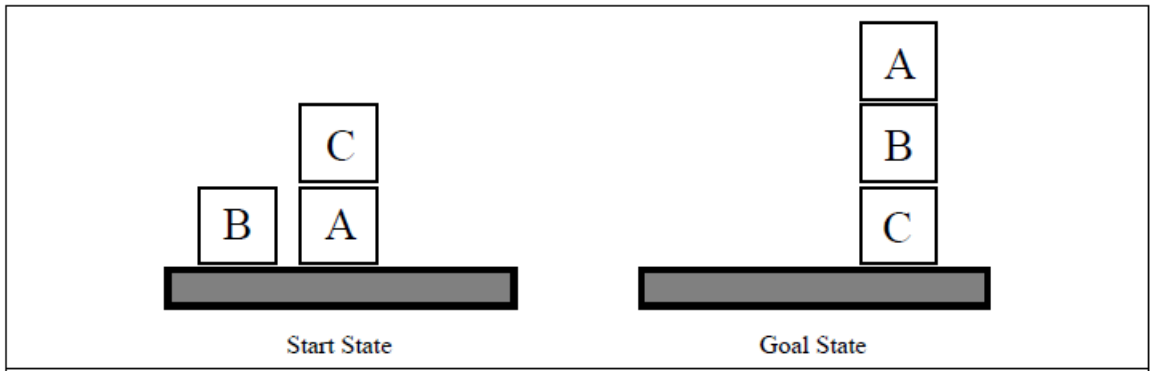
Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of goals to be achieved

Planning

- Planning is finding and choosing a sequence (or a "**program**") of actions to achieve goals.
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it



- Move C to the table
- Put B on top of C
- Put A on top of B

Typical assumptions

- **Atomic time**: Each action is indivisible
- **No concurrent actions** are allowed (though actions do not need to be ordered with respect to each other in the plan)
- **Deterministic actions**: The result of actions are completely determined—there is no uncertainty in their effects
- Agent is the **sole cause of change** in the world
- Agent is **omniscient**: Has complete knowledge of the state of the world
- **Database semantics**: close world assumption and unique names assumption

PDDL: Planning Domain Definition Language
A way to represent states, actions and goals

- **States** represented as a conjunction of **ground** literals
  - at(Home) ∧ have(Milk) ∧ have(bananas) ...
- **Actions** have
  - **Action Name and Parameter List**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied

Action(Fly(p,from, to),
PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT: ¬AT(p,from) ∧ At(p,to))

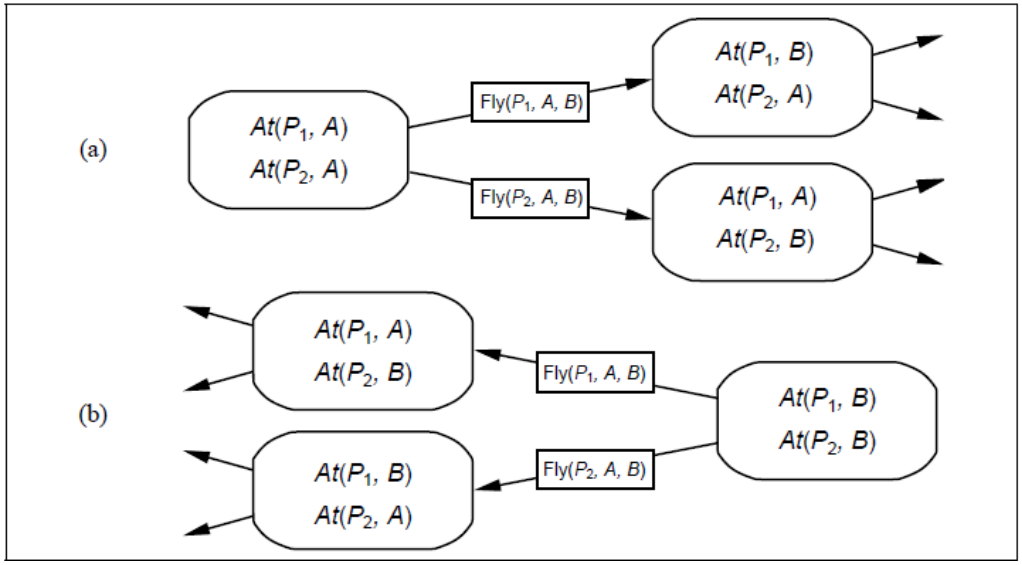
- **Aplicability** - States that satisfies the precondition
- **Result** - After performing an action we can obtain:
  - Delete list and add list: what should be removed and added to the state

Actions are specified in terms of what changes; everything that stays the same is left unmentioned

Approaches

- State-space search
  - The plan is a solution found by "searching" through the situations to get to the goal
    - Forward state-space search
    - Backward relevant state search
- Graph-based search: GraphPlan
- Hierarchical Planning
- Multiagent Planning

State-space planning

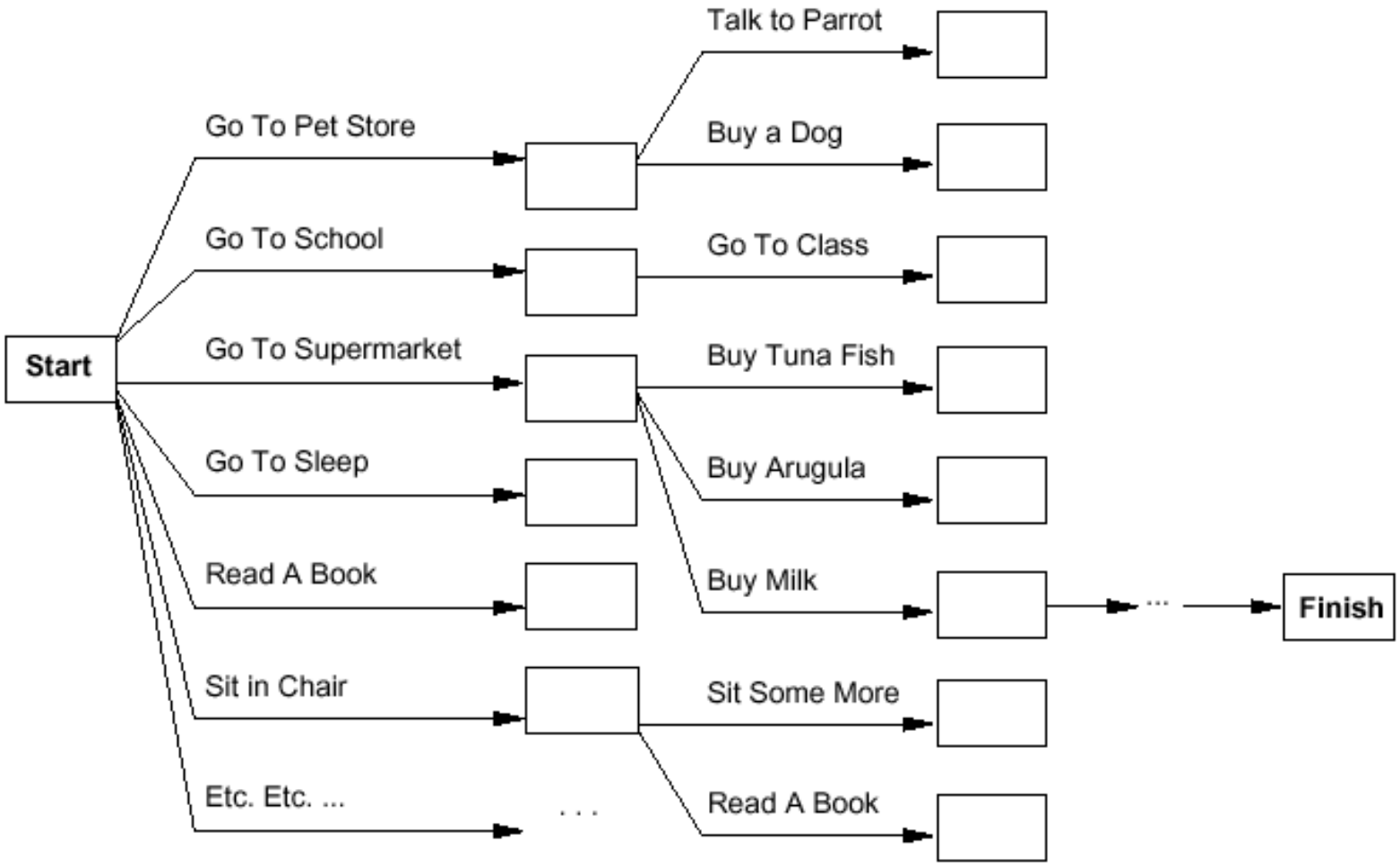


- Progression planners
  - Forward state-space search
  - Consider the effect of all possible actions in a given state
  - Choose **applicable** actions
- Regression planners
  - Backward state-space search
  - Determine what must have been true in the previous state in order to achieve the current state
  - Choose **relevant** actions

Progression algorithm

- Formulation as state-space search problem:
  - Initial state and goal test: obvious
  - Successor function: generate from applicable actions
  - Step cost = each action costs 1
- Any complete graph search algorithm is a complete planning algorithm.
  - E.g. A*
- Inherently inefficient:
  - (1) irrelevant actions lead to very broad search tree
  - (2) good heuristic required for efficient search

Forward search explores irrelevant actions



Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors are as follows:
    - Any positive effects of A that appear in G are deleted.
    - Each precondition literal of A is added , unless it already appears.
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.

Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?

    Goal state = $At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$

    Relevant action for first conjunct: $Unload(C1,p,B)$

    Works only if pre-conditions are satisfied.

    $Action\ (Unload(C,p',B),$

    $Precond: In(C,p') \wedge At(p', B) \wedge Cargo(C) \wedge Plane(p') \wedge Airport(B)$

    $Effect: At(C,B) \wedge \neg In(C,p')$

    Previous state g'= $In(C1, p') \wedge At(p', B) \wedge Cargo(C) \wedge Plane(p') \wedge Airport(B)$

    Subgoal At(C1,B) should not be present in this state.

- **Relevant actions**:
  - At least one of the action's effects must **unify** with an element of the goal
  - Must not have any effect that negates an element of the goal

GraphPlan

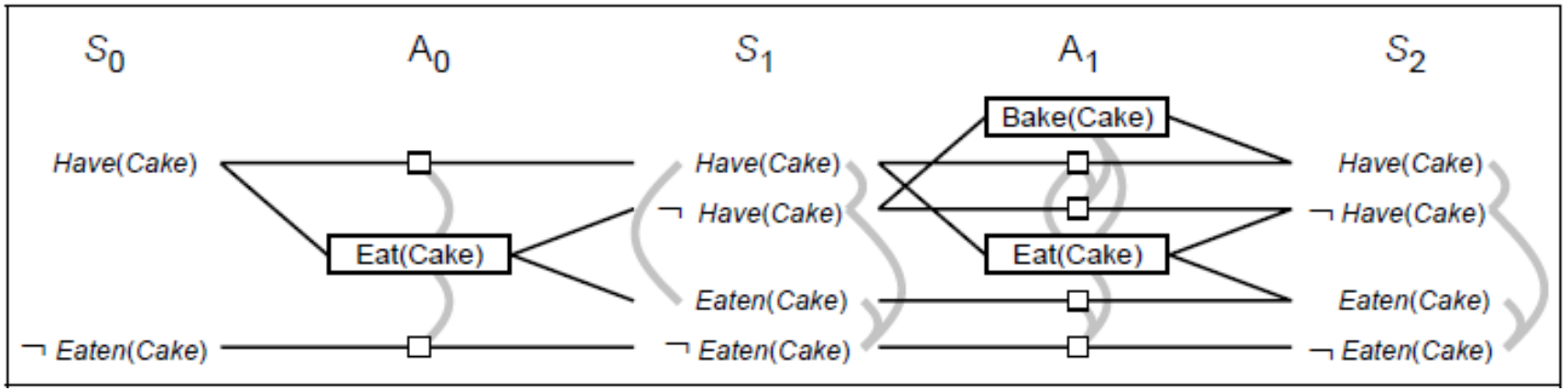
- Construct a graph that encodes constraints on possible plans and use it to constrain search for a valid plan

Planning graph

- Directed, leveled graph with alternating layers of nodes
- Odd layers ("**state levels**") represent candidate propositions that could possibly hold at step *i*
- Even layers ("**action levels**") represent candidate actions that could possibly be executed at step *i*, including maintenance actions [do nothing]
- **Arcs** represent preconditions, adds and deletes
- We can only execute one real action at any step, but the data structure keeps track of **all actions and states that are *possible***

GraphPlan example

- Initial State: *Have(Cake)*
- Goal state: *Have(Cake)* ∧ *Eaten(Cake)*
- *Action Eat(Cake)*
  - Precondition: *Have(Cake)*
  - Effect: ¬*Have(Cake)* ∧ *Eaten(Cake)*
- *Action Bake(Cake)*
  - Precondition: ¬ *Have(Cake)*



What actions and what literals?

- Add an action in level A_i if ***all*** of its preconditions are present in level S_i
- Add a literal in level S_i if it is the effect of ***some*** action in level A_{i-1} (*including no-ops*)
- Level S_0 has all of the literals from the initial state

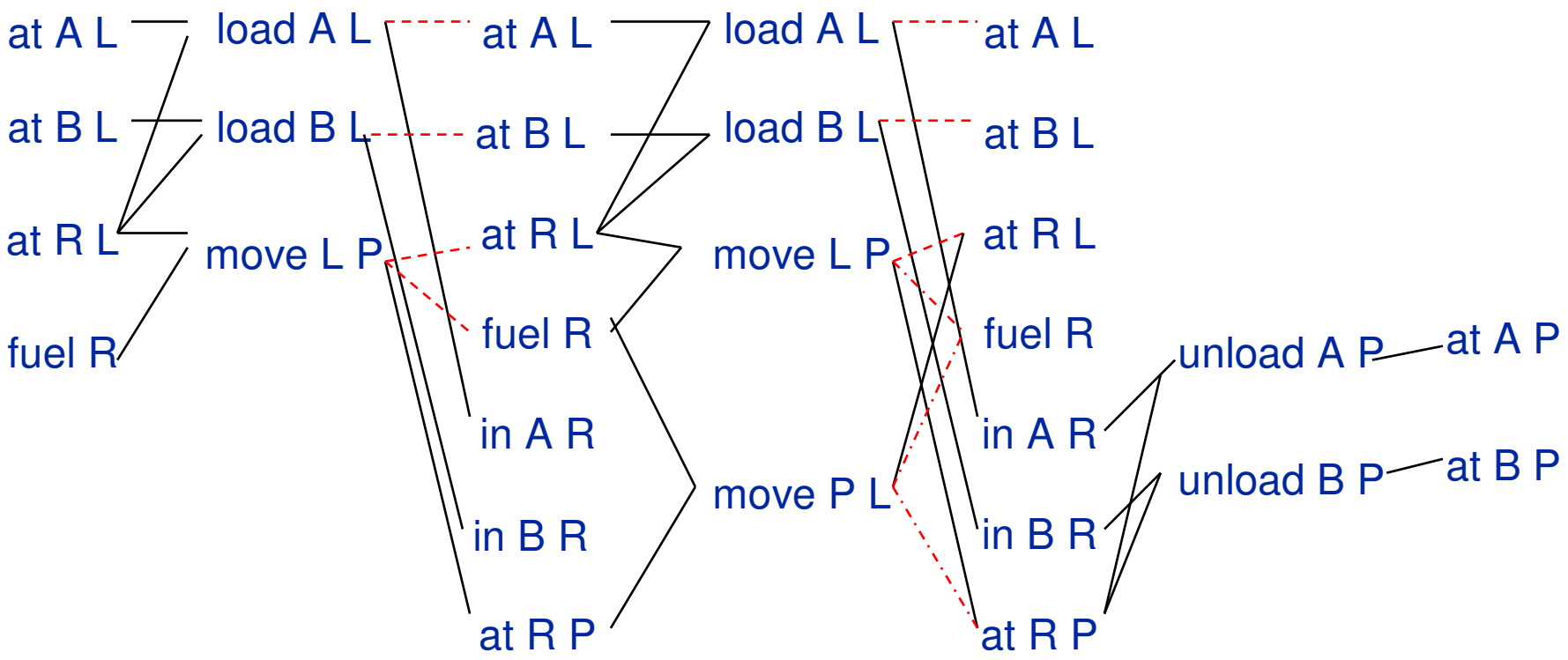
Exclusion relations (mutexes)

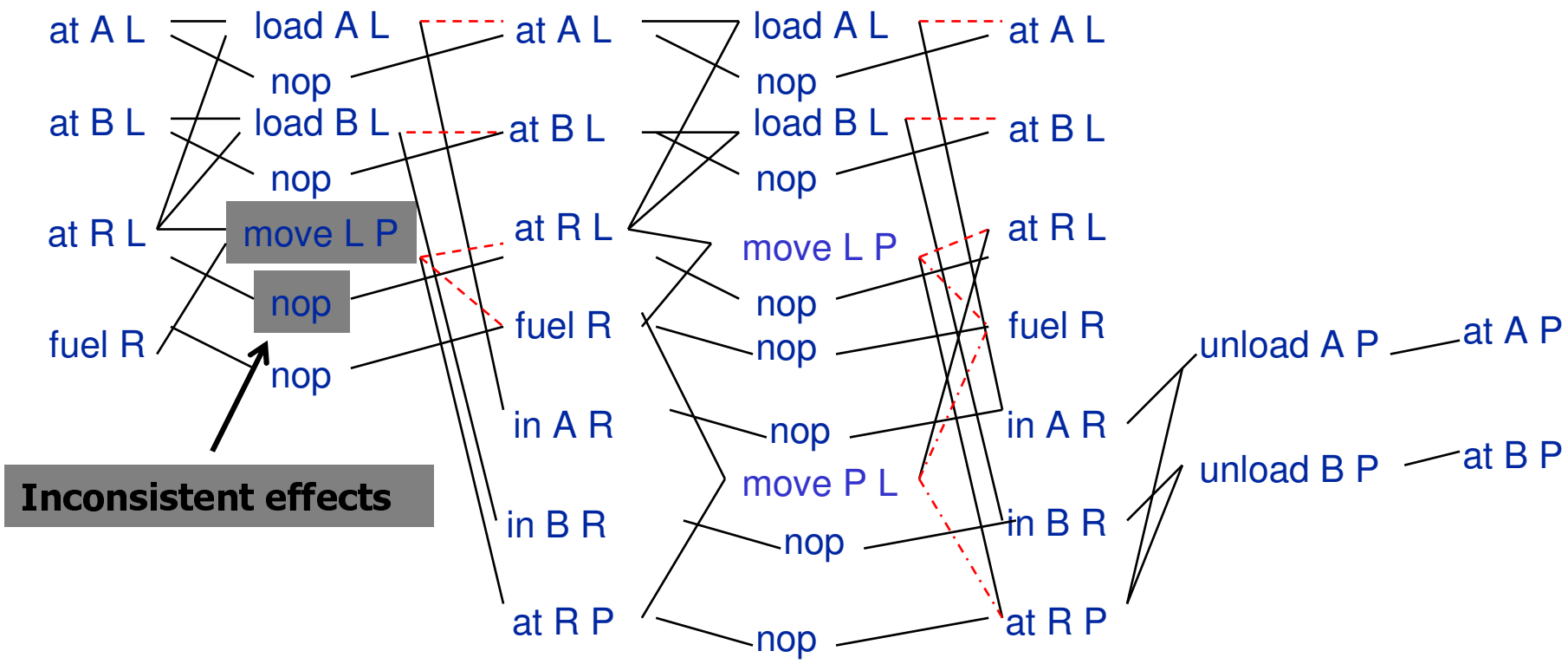
- Two actions (or literals) are **mutually exclusive ("mutex")** at step *i* if no valid plan could contain both actions at that step
- Can quickly find and mark *some* mutexes:
  - **Inconsistent effects**: Two actions whose effects are mutex with each other
  - **Interference**: Two actions that interfere (the effect of one negates the precondition of another) are mutex
  - **Competing needs**: Two actions are mutex if any of their preconditions are mutex with each other
  - **Inconsistent support**: Two literals are mutex if all ways of creating them both are mutex

Simple domain

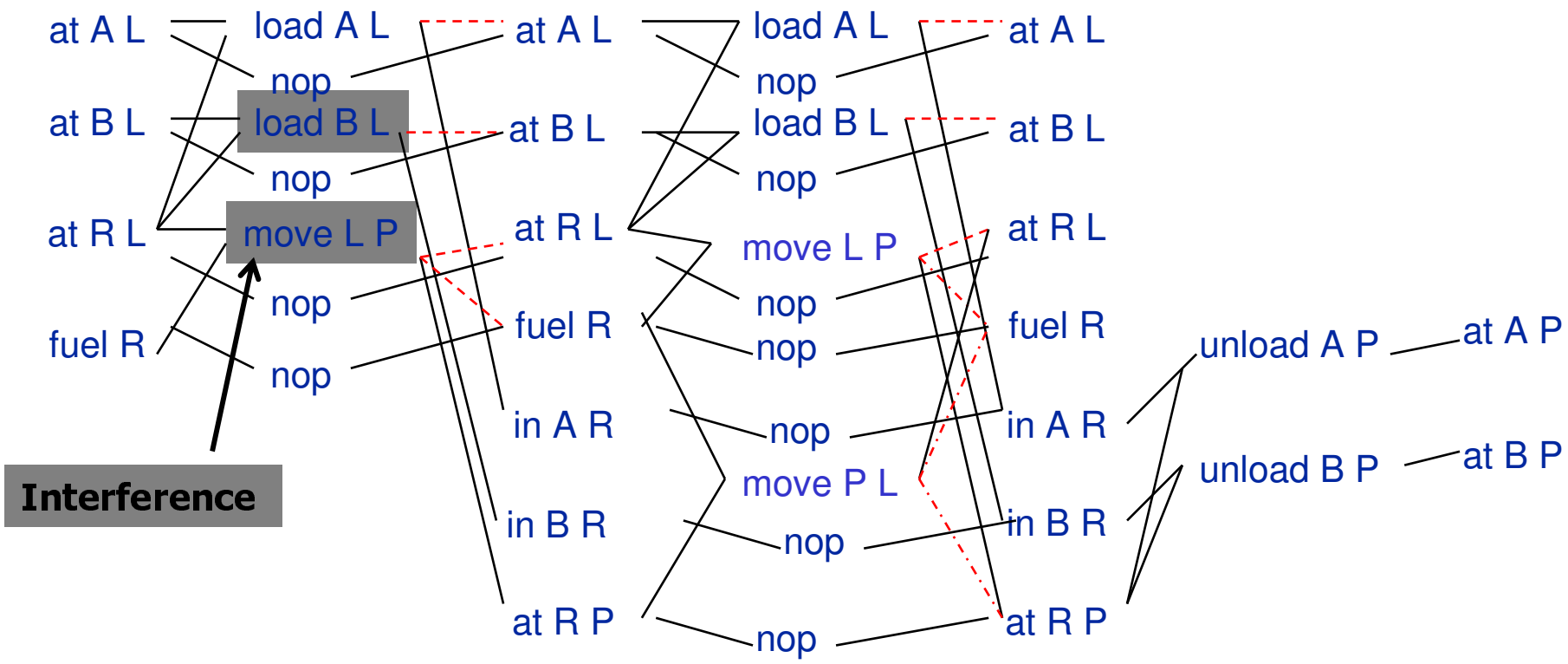
- Literals:
  - at X Y — X is at location Y
  - fuel R — rocket R has fuel
  - in X R — X is in rocket R
- Actions:
  - load X L — load X (onto R) at location L (X and R must be at L)
  - unload X L — unload X (from R) at location L (R must be at L)
  - move X Y — move rocket R from X to Y (R must be at X and have fuel)
- Graph representation:
  - Solid black lines: preconditions/effects
  - Dotted red lines: negated preconditions/effects

Example planning graph

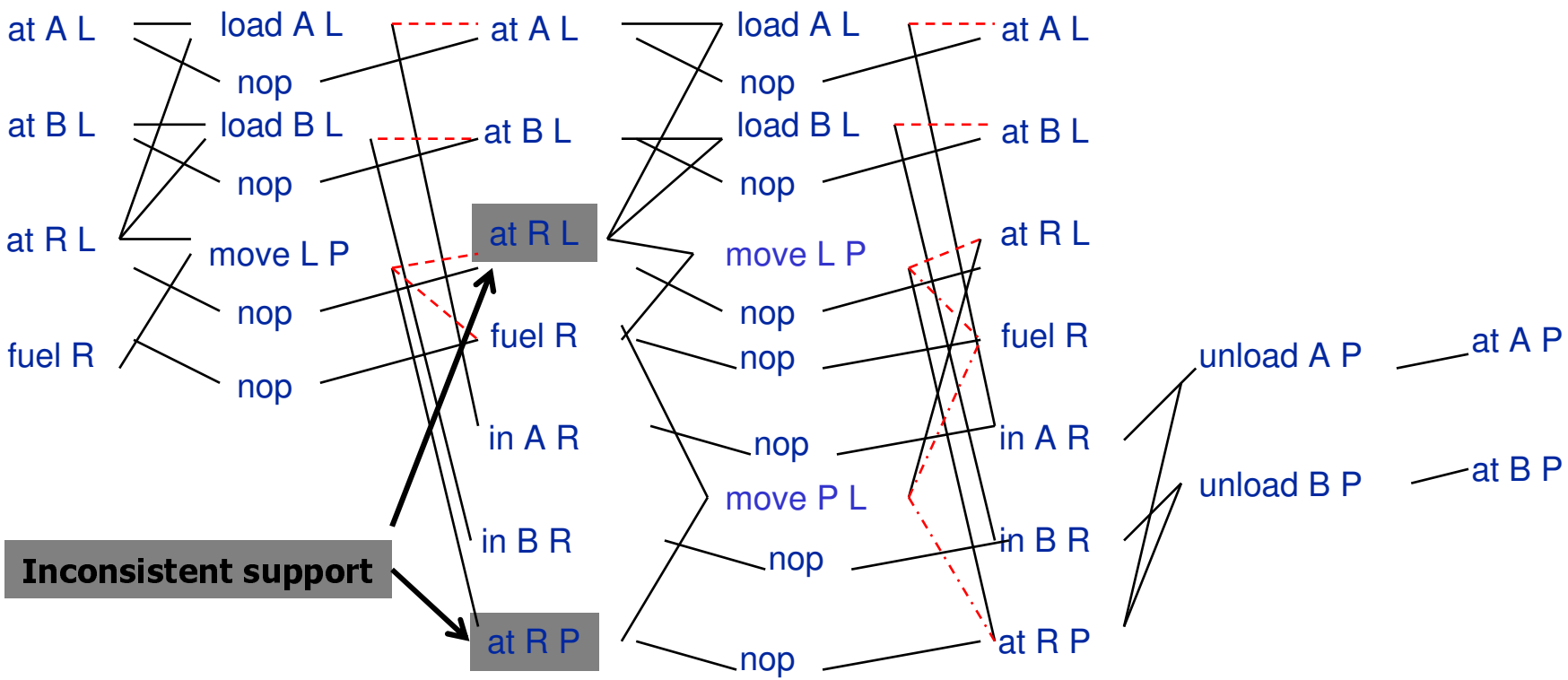



at A L
at B L
at R L
fuel R
load A L
nop
load B L
nop
move L P
nop
nop
Inconsistent effects
at A L
at B L
at R L
fuel R
in A R
in B R
at R P
load A L
nop
load B L
nop
move L P
nop
nop
nop
move P L
nop
nop
at A L
at B L
at R L
fuel R
in A R
in B R
at R P
unload A P
unload B P
at A P
at B P
States
S_0
Actions
A_0
States
S_1
Actions
A_1
States
S_2
Actions
A_2
States
S_3
(Goals!)

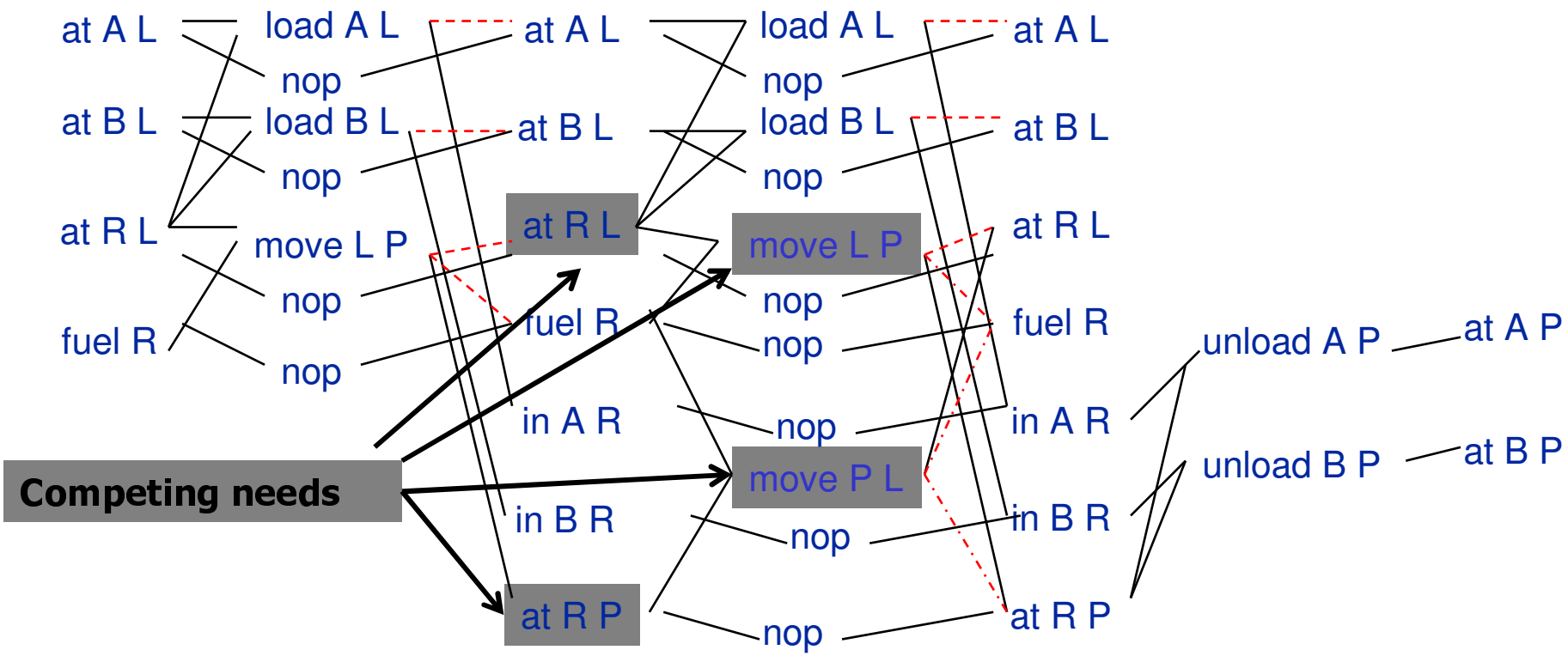
Example: Mutex constraints



Example: Mutex constraints



Example: Mutex constraints



States S_0 | Actions A_0 | States S_1 | Actions A_1 | States S_2 | Actions A_2 | States S_3 (Goals!)

Valid plans

- A **valid** plan is a planning graph in which:
  - Actions at the same level don't interfere (delete each other's preconditions or add effects)
  - Each action's preconditions are true at that point in the plan
  - Goals are satisfied at the end of the plan

Basic GraphPlan algorithm

- **Grow** the planning graph (PG) until all goals are reachable and none are pairwise mutex. (If PG **levels off** [reaches a steady state] first, fail)
- **Search** the PG for a **valid plan**
- If none found, **add a level** to the PG and try again

Creating the planning graph is usually fast

- Theorem:

  The size of the t-level planning graph and the time to create it are polynomial in:

  – t (number of levels),

  – n (number of objects),

  – m (number of operators), and

  – p (number of propositions in the initial state)

Searching for a plan

- Backward chain on the planning graph
- Complete all goals at one level before going back
- At level *i*, pick a non-mutex subset of actions that achieve the goals at level *i+1*. The preconditions of these actions become the goals at level *i*
  - Various heuristics can be used for choosing which actions to select
- Build the action subset by iterating over goals, choosing an action that has the goal as an effect. Use an action that was already selected if possible. Do forward checking on remaining goals.

Summary

- Problem solving
  - Atomic representations of states
- Planning combines search and logic
  - Problem solving algorithms that operate on explicit propositional or relational representations of states and actions.
- PDDL describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State space planning performs forward or backward search on the state space
  - Progression planners choose **applicable** actions
  - Regression planners choose **relevant** actions
- A planning graph encodes constraints on possible plans which can be used to constrain the search for a valid plan