CMU 15-781

Lecture 7:
Planning III

Teacher:
Gianni A. Di Caro

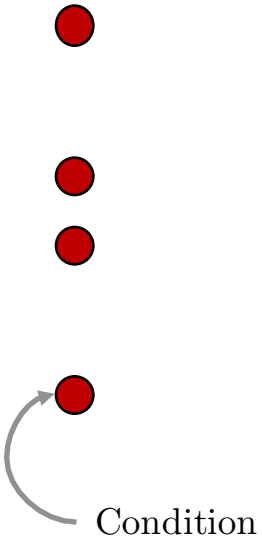
Planning graphs

- Graph-based data structure representing a polynomial-size/time approximation of the exponential search tree
- Can be used to automatically produce good heuristic estimates (e.g., for A*)
- Can be used to search for a solution using the GRAPHPLAN algorithm

Planning graphs

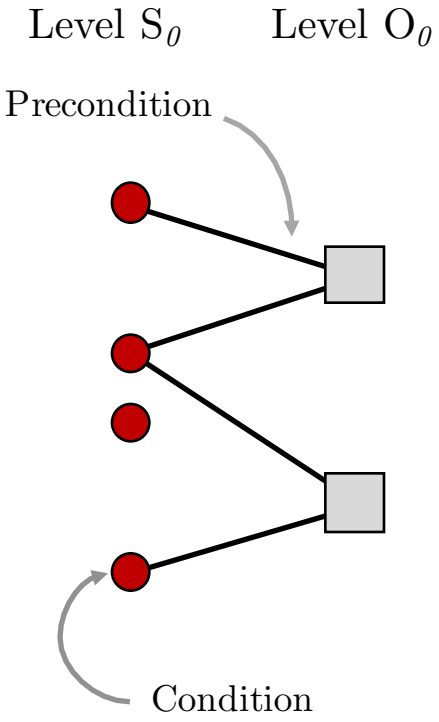
- Leveled graph: vertices organized into levels/stages, with edges only between levels
- Two types of *vertices* on alternating levels:
  - Conditions
  - Operations
- Two types of *edges*:
  - Precondition: from condition to operation
  - Postcondition: from operation to condition

Generic planning graph



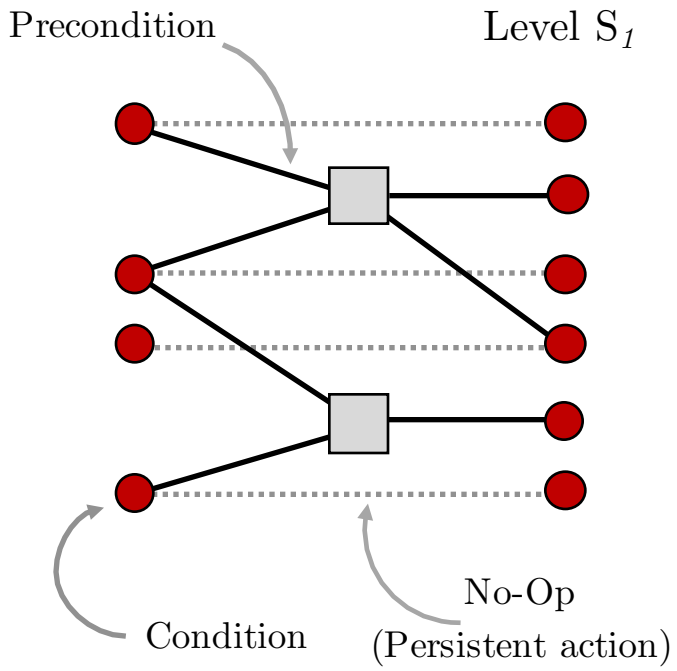
- S_0 contains all the conditions that hold in initial state

Generic planning graph



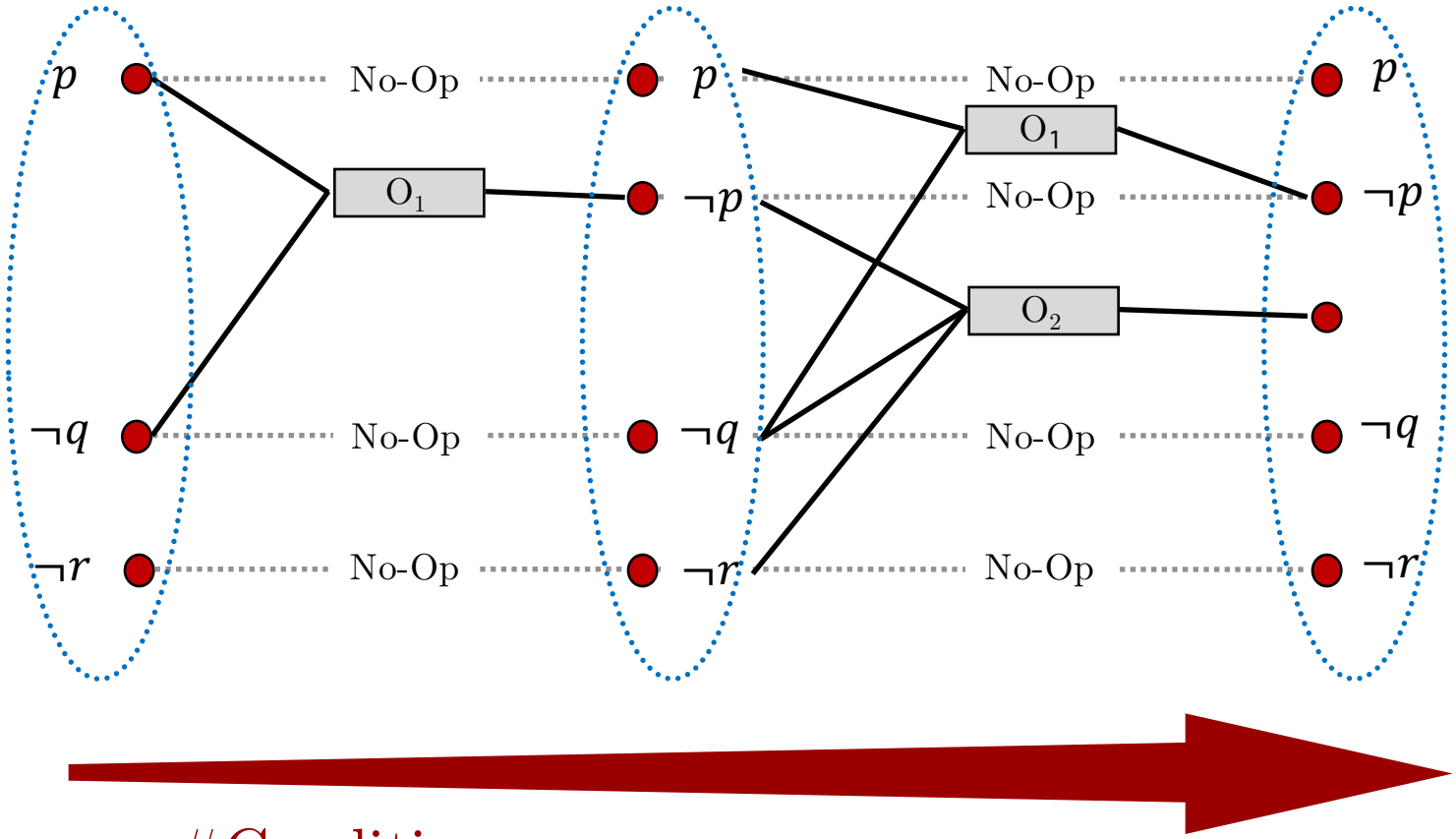
- Add operation to level O_i if its preconditions appear in level S_i

Generic planning graph



- Add condition to level S_i if it is the postcondition of an operation (it is in ADD or DELETE lists) in level O_{i-1}
- Keep a previous condition of no action negates it (*persistence, no-op action*)

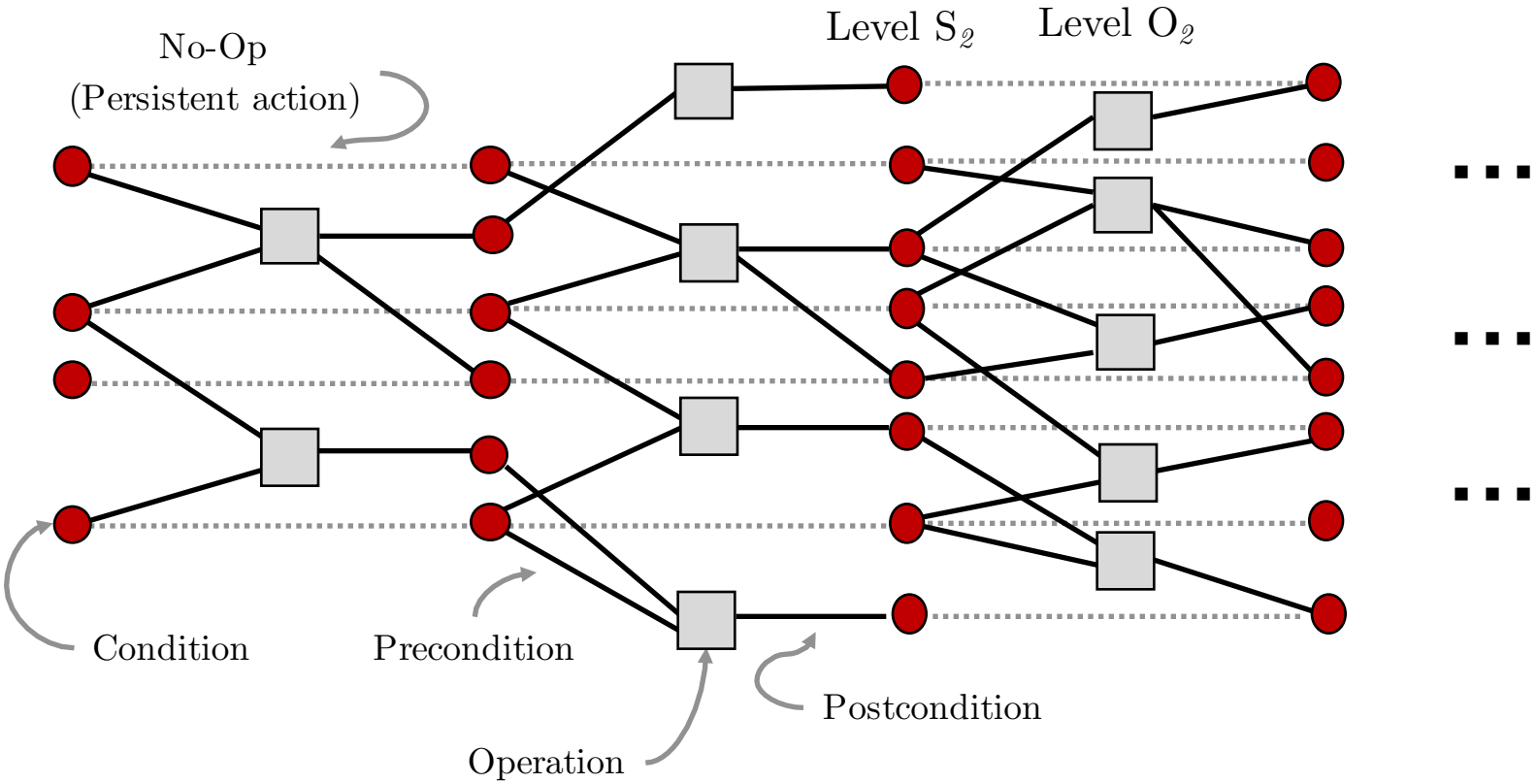
Conditions monotonically increase



#Conditions
(always carried forward by no-ops)

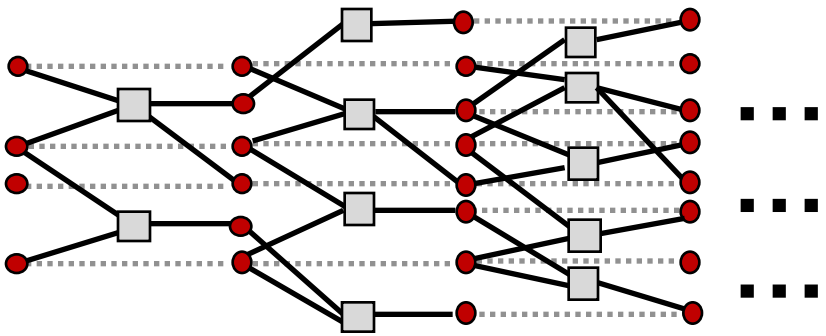
Slide based on Brafman

Generic planning graph



- Repeat ...

Generic planning graph

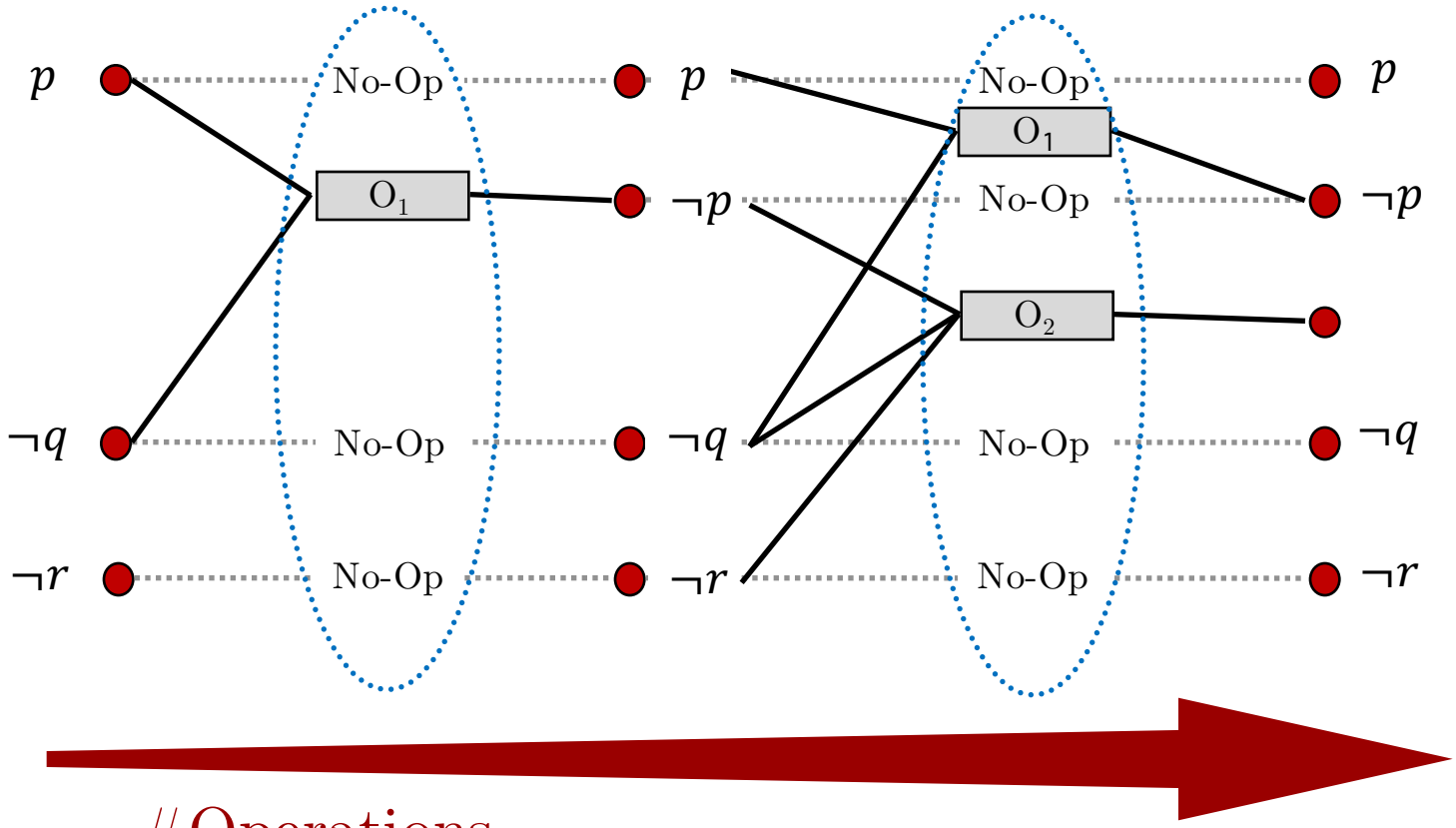


Idea: S_i contains all conditions that *could* hold at stage i based on past actions; O_i contains all operations that *could* have their preconditions satisfied at time i

No ordering among the operations is assumed at each stage, they could be executed in parallel

- $\rightarrow$ The level j at which a condition first appears is a (good) estimate of how difficult is to achieve that condition
- $\rightarrow$ Can optimistically estimate how many steps it takes to reach a goal g (or sub-goal g_i) from the initial state

Operations monotonically increase



#Operations

(as a result of conditions monotonic increase, that keep previous preconditions hold and set new preconditions true)

Slide based on Brafman

Mutual exclusion links

- As it is the graph would be too optimistic!
- The graph also *records conflicts* between actions or conditions: two operations or conditions are mutually exclusive (mutex) ***if no valid plan can contain both at the same time***
- A bit more formally:
  - *Two operations* are mutex if their preconditions or postconditions are mutex (*inconsistent effects*, *competing needs*, *interference*)
  - *Two conditions* are mutex if one is the negation of the other, or all action pairs that achieve them are mutex (*inconsistent support*)

A running example

- "*Have cake and eat cake too*" problem

Initial state: $Have(Cake)$

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

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

$Bake(Cake)$:
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

A running example

Initial state: $Have(Cake)$

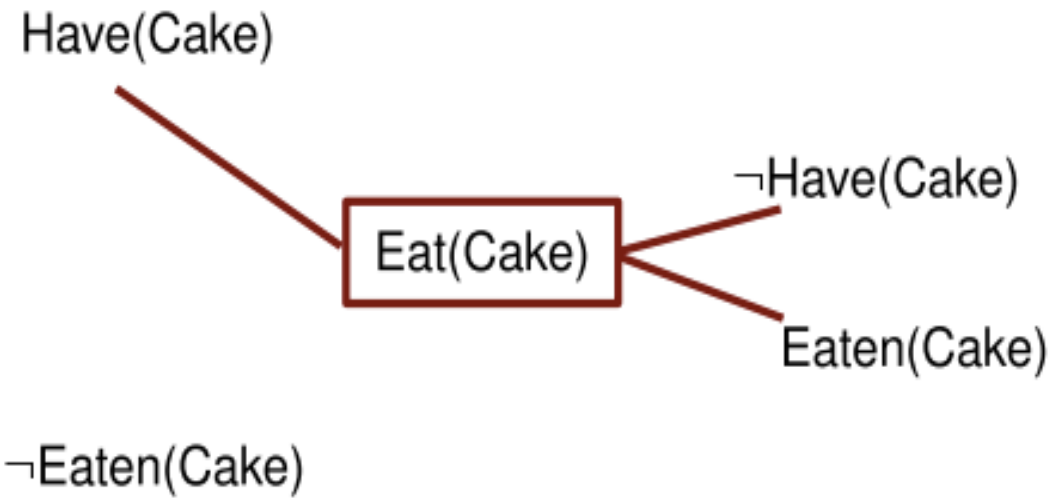
Goal: $Have(Cake) \wedge Eaten(Cake)$

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

$Bake(Cake)$:
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

S_0 A_0

- Only Eat(Cake) is applicable



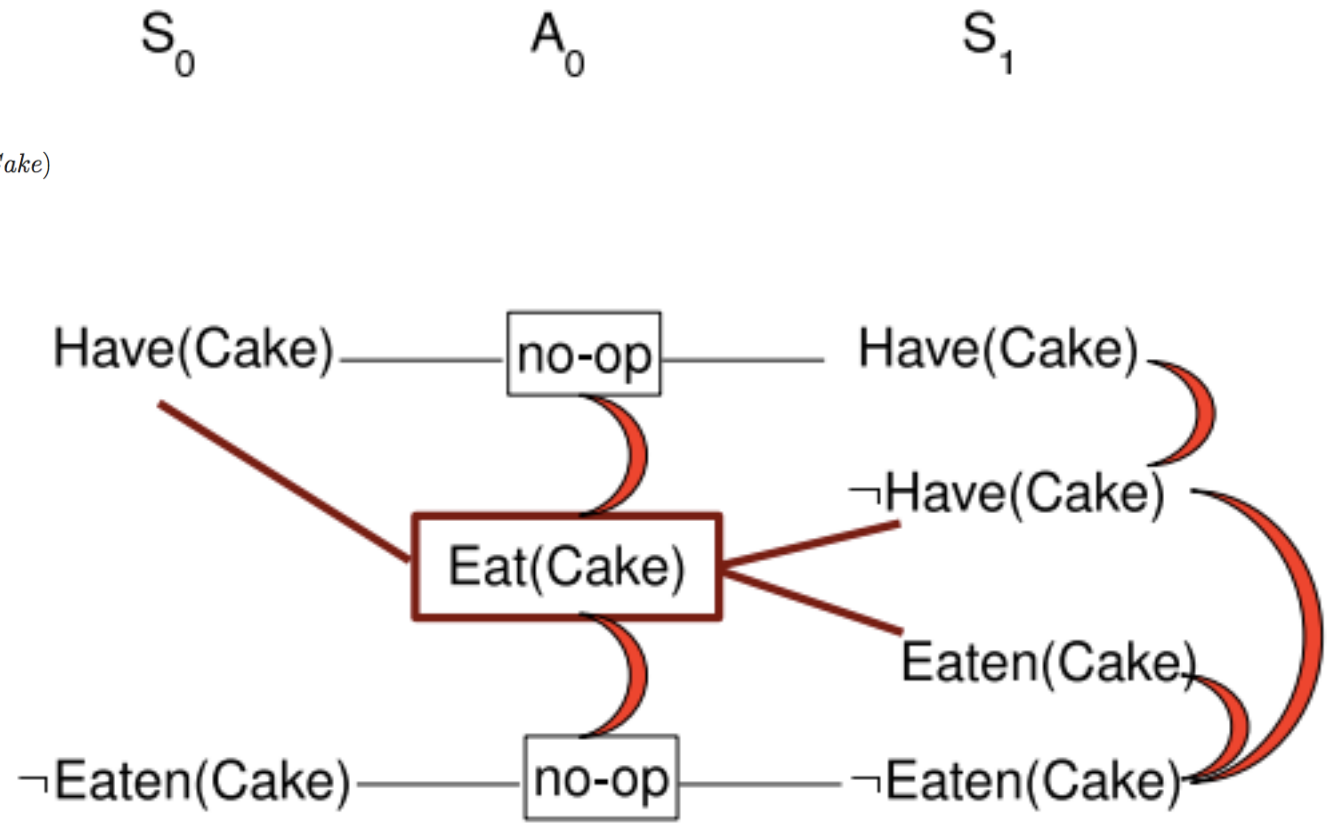
A running example

Initial state: $Have(Cake)$

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

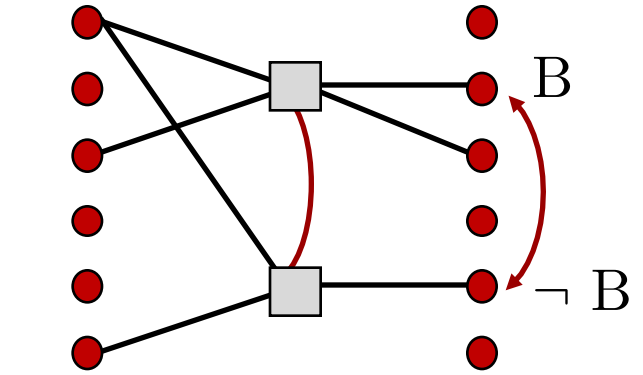
$Eat(Cake)$:
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake)$

$Bake(Cake)$:
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$



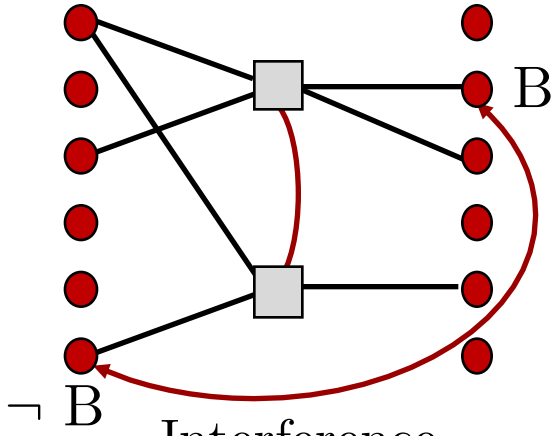
Mutex cases

- Inconsistent postconditions (two ops): one operation negates the effect of the other; *Eat(Cake)* and no-op *Have(Cake)*



Inconsistent Postconditions

- Interference (two ops): a postcondition of one operation negates a precondition of other; *Eat(Cake)* and no-op *Have(Cake)* (issue in parallel execution, the order should not matter but here it would)



Interference

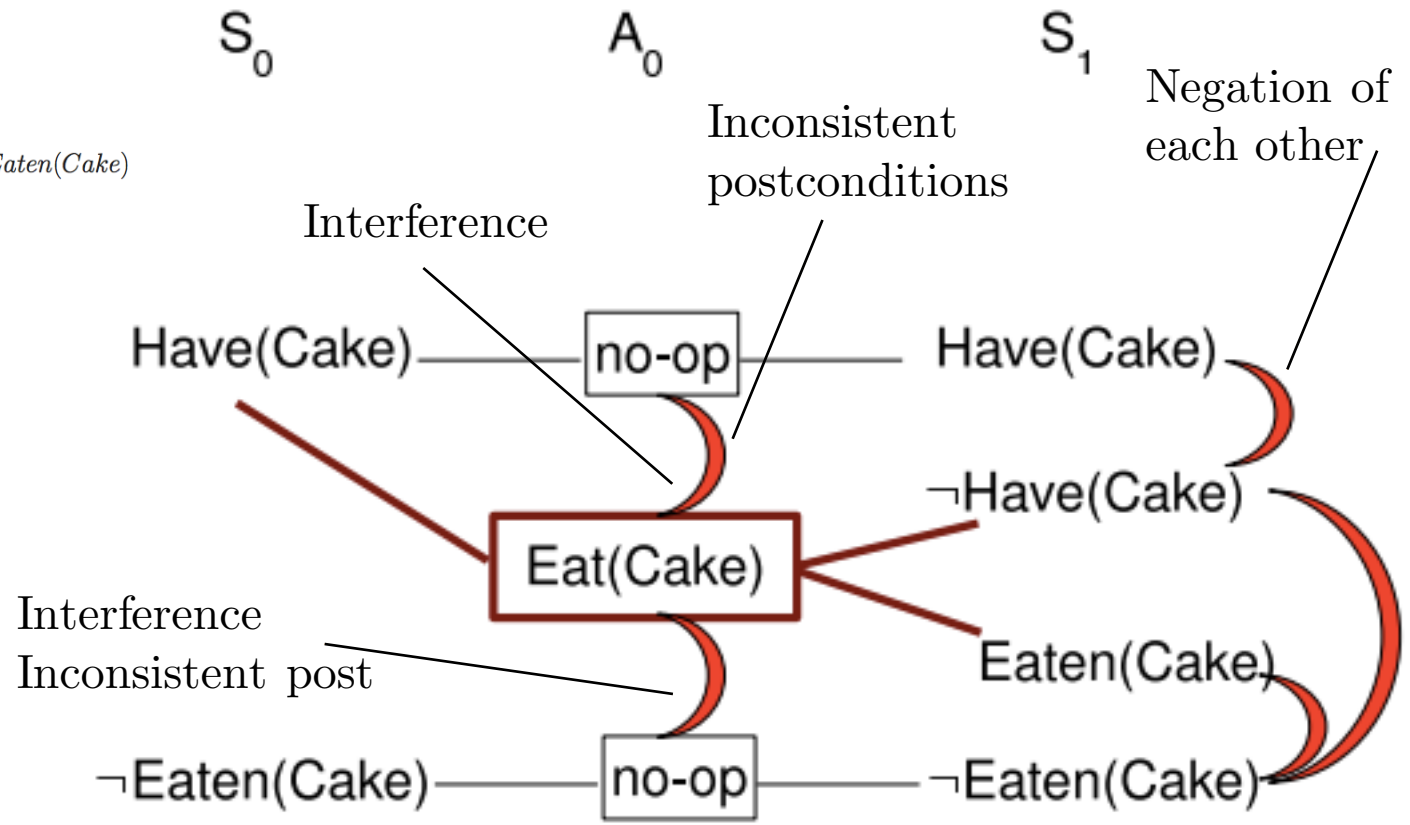
A RUNNING EXAMPLE

Initial state: $Have(Cake)$

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

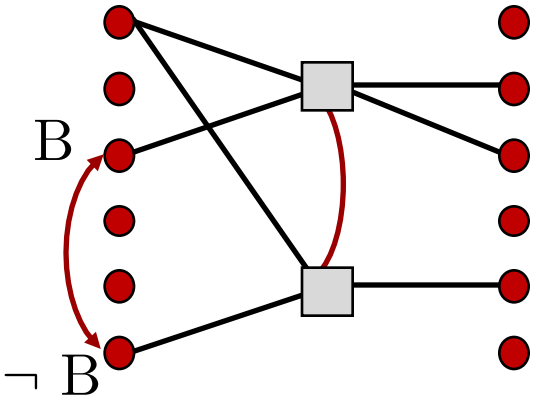
$Eat(Cake)$:
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake)$

$Bake(Cake)$:
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$



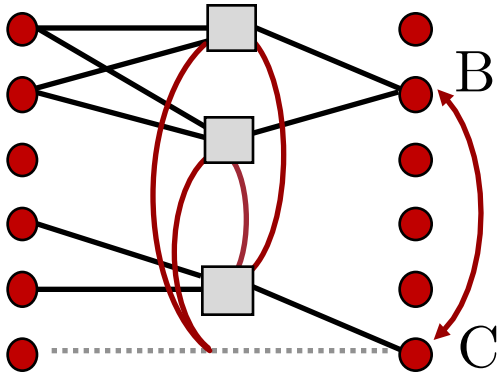
Mutex cases

- Competing needs (two ops): a precondition of one operation is mutex with a precondition of the other because they are the negate of each other, like for *Bake(Cake)* and *Eat(Cake)*, or because they have inconsistent support



Competing Needs

- Inconsistent support (two conditions): each possible pair of operations that achieve the two conditions is mutex, *Have(Cake)* and *Eaten(Cake)*, are mutex in S_1 but not in S_2 because they can be achieved by *Bake(Cake)* and *Eaten(Cake)*



Inconsistent Support

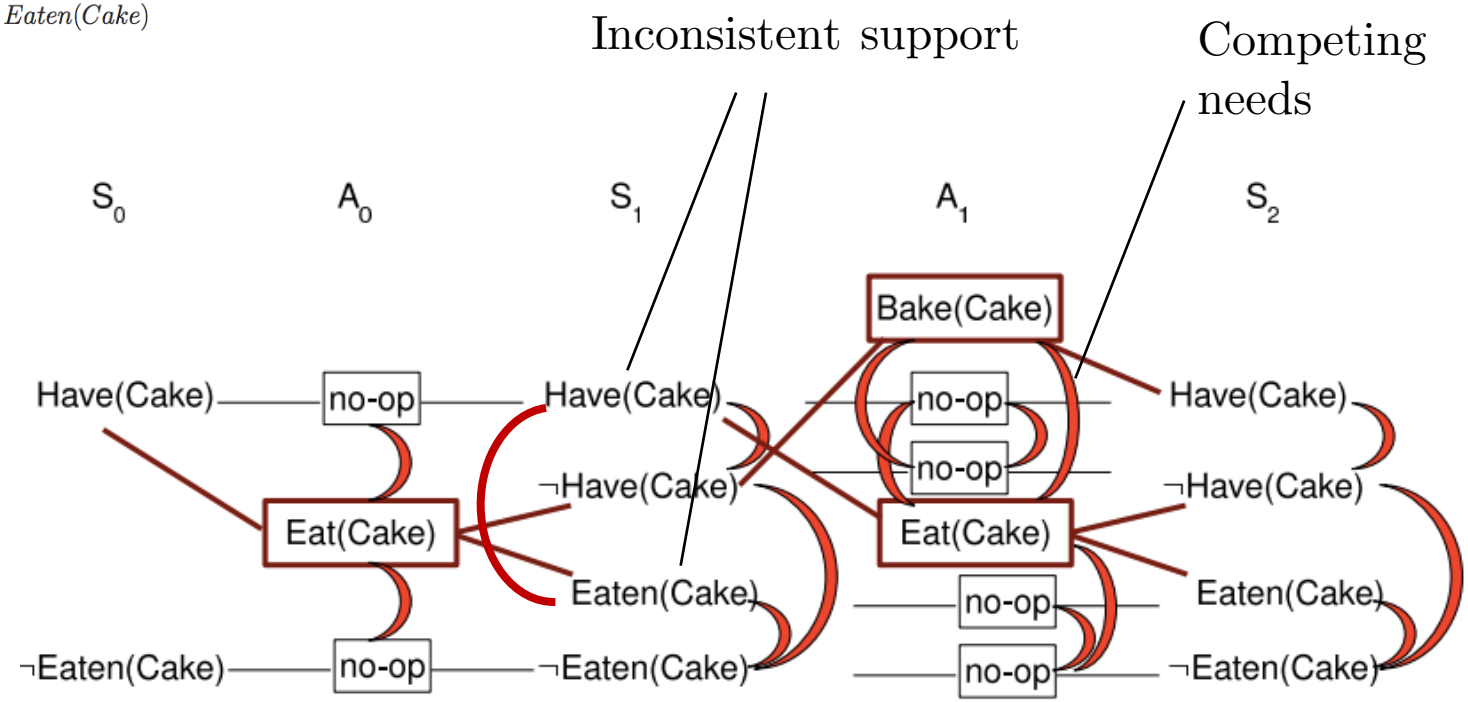
A RUNNING EXAMPLE

Initial state: $Have(Cake)$

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

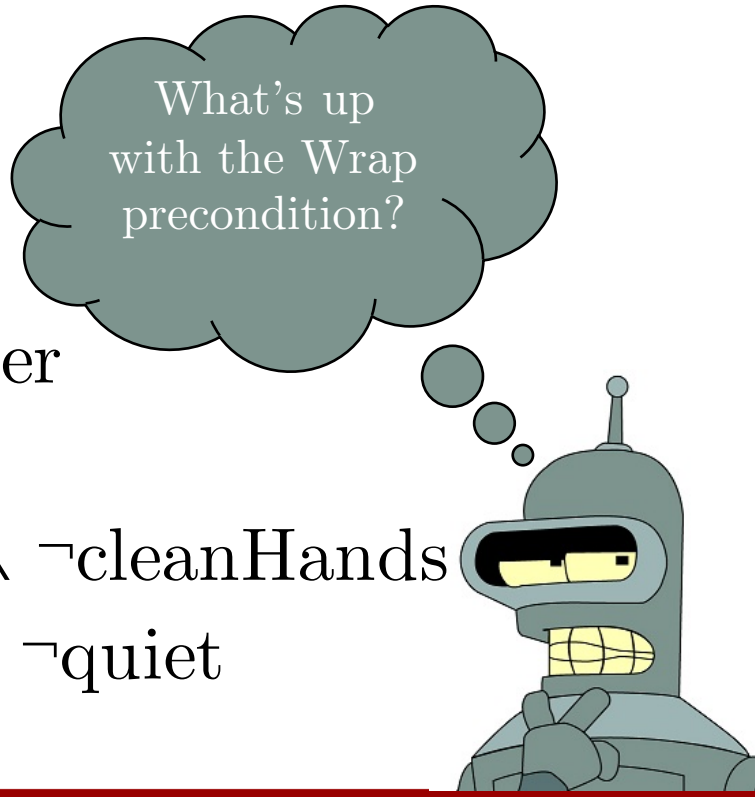
$Eat(Cake)$:
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake)$

$Bake(Cake)$:
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

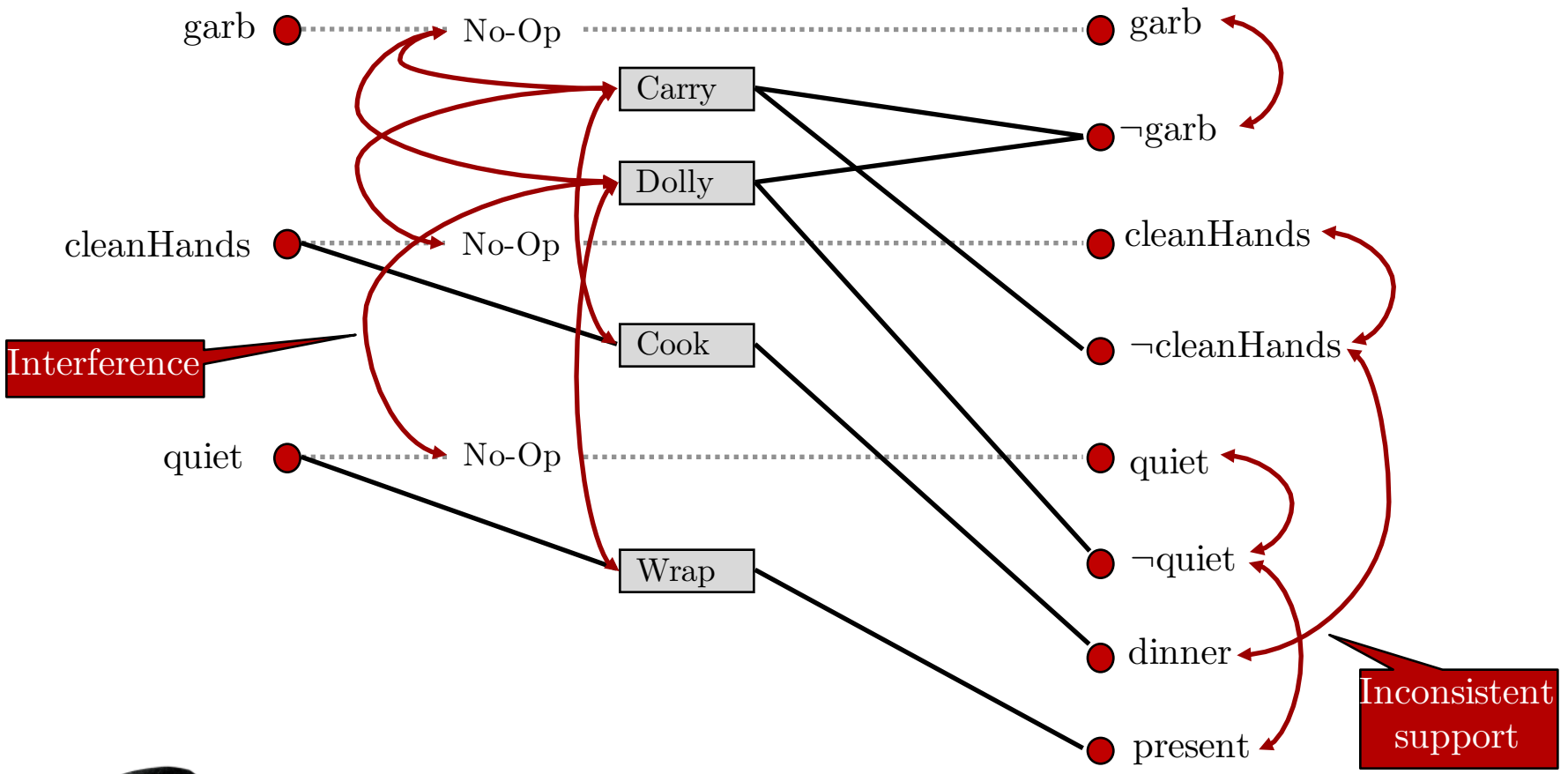


Dinner date example

- Initial state:
  garbage ∧ cleanHands ∧ quiet
- Goals: dinner ∧ present
  ∧ ¬garbage
- Actions:
  - Cook: cleanHands ⇒ dinner
  - Wrap: quiet ⇒ present
  - Carry: none ⇒ ¬garbage ∧ ¬cleanHands
  - Dolly: none ⇒ ¬garbage ∧ ¬quiet

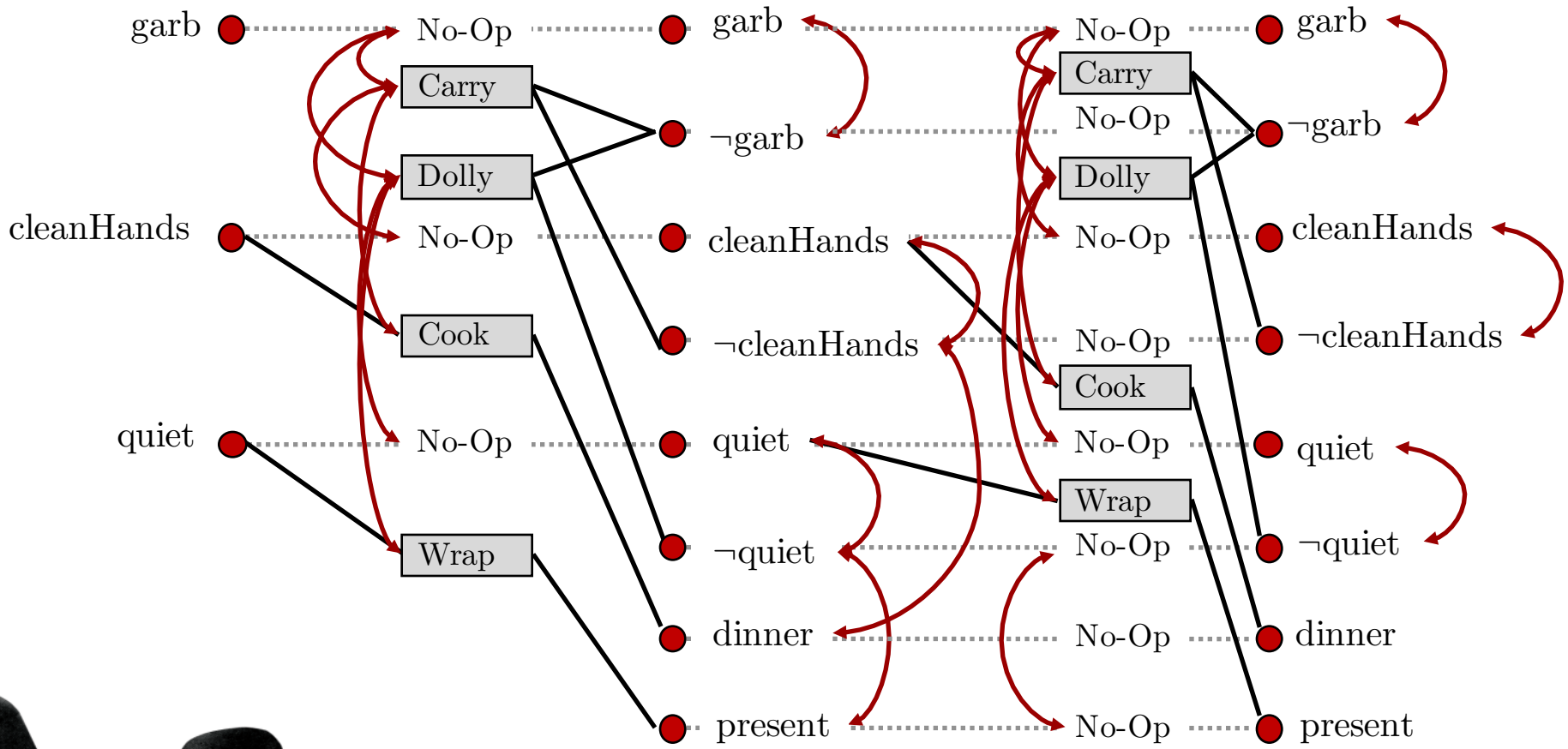


Dinner date example*



* Slide based on Brafman

Dinner date example*



Slide based on Brafman

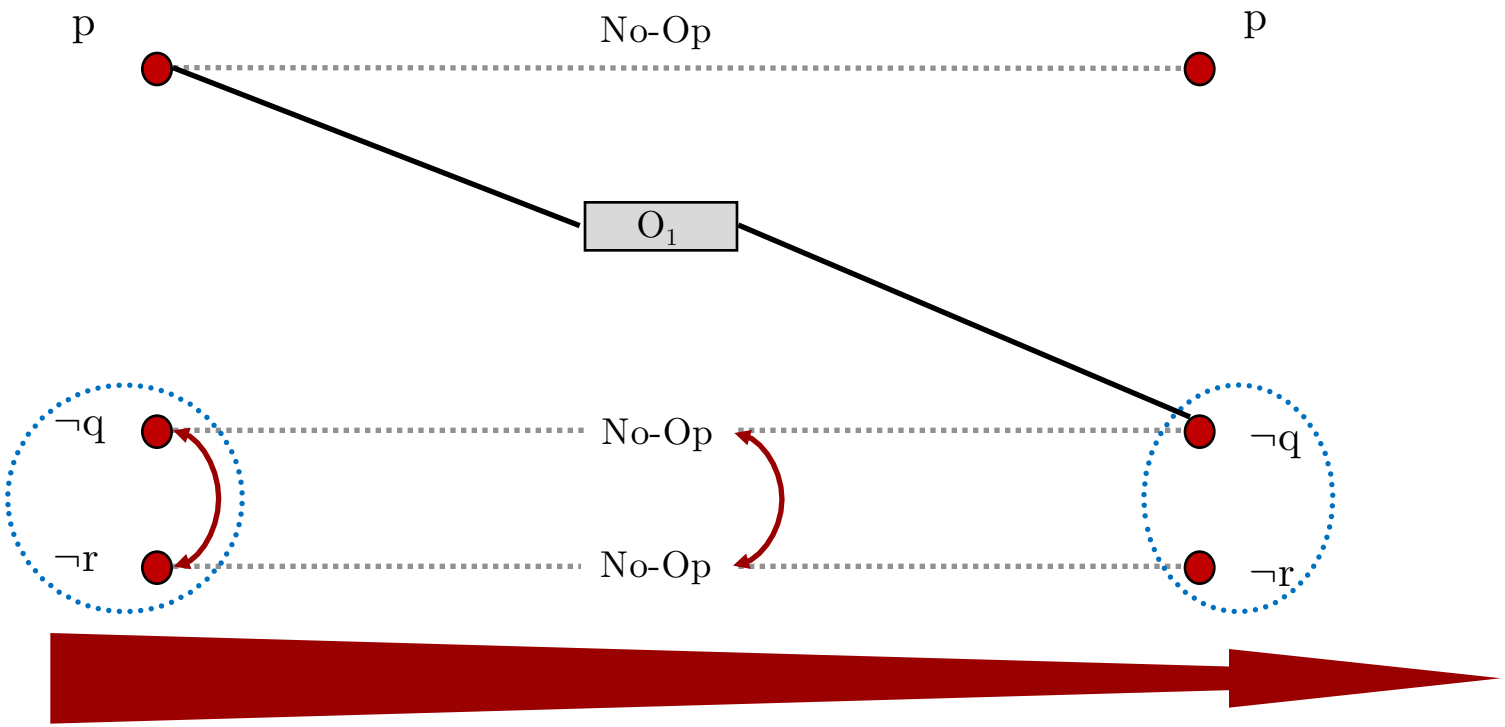
A constructive process with no choices involved

- The process of constructing the planning graph does not require to choose among the actions
- $\rightarrow$ No combinatorial search is involved!
- At each O level, identify applicable actions, including the persistence ones
- At each S level identify all conditions that could result from the actions at the previous level
- Check for mutexes and add mutex link relations
- Check for level-off

#MUTEXES MONOTONICALLY DECREASE

- Intuitive proof:
  - If two operations or conditions are a mutex at a given level, they will be also mutex for all previous levels (because of less conditions or actions to use to possibly avoid their conflicts)
  - At each level we should also think of all operations that cannot be executed and all actions that do not hold: these are in mutex with everything
  - By adding them over time the total # of mutexs decreases

Condition mutex relationships monotonically decrease



#Conditions mutex

(new operations appear that can resolve inconsistent support conflicts)

Operation mutexes monotonically decrease

- Proof idea:
  - Different behavior for the different types of mutexes
  - Inconsistent postconditions and interference mutex are properties of the operations themselves ⇒ hold at every level
  - Competing needs depend on conditions at level S_i: if two conditions are mutex because they are the negation of each other, then the mutex will hold at every level and so for the operations. Instead, if the conditions are mutex because of inconsistent support, by induction the monotonic increasing in #operations monotonically implies decrease in #mutexes

Leveling off

- As a corollary of the previous properties, we see that the planning graph levels off
  - Consecutive levels become identical
- Proof:
  - Limit in the # of construction steps
  - Upper bound on #operations and #conditions based on problem definition
  - Lower bound of 0 on #mutexes ■

Planning graph complexity

- The graph is polynomial in the size of the planning problem *P(L,A)*
  - Each S_i has at most L nodes and L^2 mutex links
  - Each O_i has at most $A+L$ nodes, $(A+L)^2$ mutex links, and $2(AL+L)$ pre- and post-condition links
  - A graph with n levels has a size of $O(n(A+L)^2)$
  - The time to build the graph has the same complexity

Planning graph ≅ Relaxed problem

- The relaxed problem provides an *optimistic estimate* of the number of steps to reach each sub-goal g_i → Heuristics!
- Level cost of sub-goal (literal) g_i = level where g_i first appears
- Level cost is an *admissible* heuristic estimate but can be quite bad because level ≠ number of actions
- If any sub-goals g_i fail to appear in the final level of the graph ⇒ *The problem is unsolvable* (∞ LB)
- If all sub-goals appear → "Possibly" there is a plan that achieves g, conditional that no mutexes exist

Heuristics from planning graphs

- To estimate the cost of a conjunction of sub-goals:
  - Max level: max level cost of any goal (clearly admissible)
  - Level sum: sum of level costs for all goals
  - Set level: level at which all sub-goals appear without any *pair* of them being mutex
- Poll 3: Which is admissible:
  1. Level sum
  2. Set level
  3. Both
  4. Neither

The Graphplan algorithm

1. Grow the planning graph until all sub-goals are reachable (appear as conditions) and *not mutex* (If planning graph levels off first, fail)
2. Call Extract-Solution on current planning graph
3. If none found, Expand-Graph by adding a level to the planning graph and try again

The Graphplan algorithm

```
function GraphPlan(problem) returns solution or failure
    graph ← Initial-Planning-Graph(problem)
    goals ← Conjuncts(problem.Goal)
    loop do
        if goals all non-mutex in last level of graph then do
            solution ← Extract-Solution(graph,goals, NumLevels(graph))
            if solution ≠ failure then return solution
            else if No-solution-possible(graph) then return failure
        graph ← Expand-Graph(graph,problem)
```

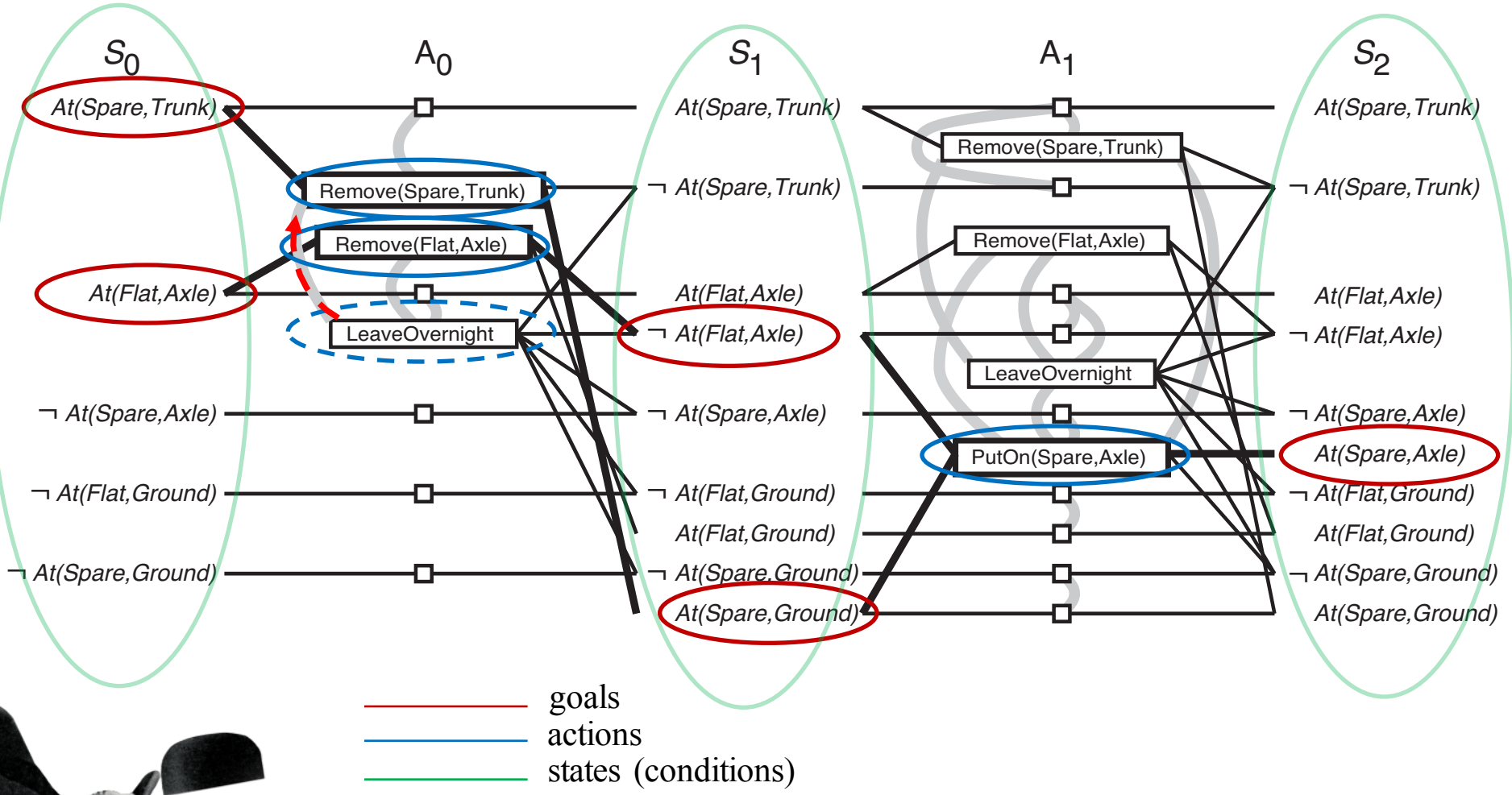
Extract-Solution

- Backward search where each state corresponds to an S level of the planning graph and a set of unsatisfied goals
- Initial state is the last level of the planning graph, along with the goals of the planning problem
- Actions available at level S_i are to select any conflict-free subset of operations in A_{i-1} whose effects cover the goals in the state
- Resulting state has level S_{i-1} and its goals are the preconditions for selected actions
- Goal is to reach a state at level S_0

Flat tire problem

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
PRECOND: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
PRECOND: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Backward search



Backward search, steps explained

- Starting from S_0, that does not include the goal's literals, Expand-Graph is called twice. The literals from the goal are then present in the S_2, and Expand-Solution is called
- Backward search starts at state S_2 with goal $At(Spare, Axle)$
- The only action choice for achieving the goal is $PutOn(Spare, Axle)$
- The regression brings to a search state S_1 with goals $At(Spare, Ground)$, $\neg At(Flat, Axle)$
- $At(Spare, Ground)$ can only be achieved by $Remove(Spare, Trunk)$
- $At(Flat, Axle)$ achieved either by $Remove(Flat, Axle)$ or $LeaveOvernight()$
- $LeaveOvernight()$ is mutex with $Remove(Spare, Trunk)$ $\rightarrow$ choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$
- New search state at S_0 with the goals $At(Spare, Trunk)$, $At(Flat, Axle)$, that are both in the initial conditions $\rightarrow$ *Solution found!*
- **Plan:** $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0, $PutOn(Spare, Axle)$ in A_1.

Heuristics for backward search

- How to choose among the actions?
- One approach: *Greedy, based on the level cost of the literals in the set of sub-goals*
  - Pick first the sub-goal with the highest level cost
  - Prefer actions with *easier preconditions* to achieve the selected sub-goal
  - Easier preconditions = Sum (or max) of the level costs of its preconditions is smallest

GRAPHPLAN GUARANTEES

- Observation: The size of the t-level planning graph and the time to create it are polynomial in t, #operations, #conditions
- Theorem: GRAPHPLAN returns a plan if one exists, and returns failure if one does not exists

Summary

- Terminology:
  - Planning graphs
  - Set level heuristic
- Algorithms:
  - Graphplan
  - Forward, Backward search
- Big ideas:
  - Planning is search, but admits domain-independent heuristics