Artificial Intelligence Foundations and Applications

Planning – Part 2
Graphplan

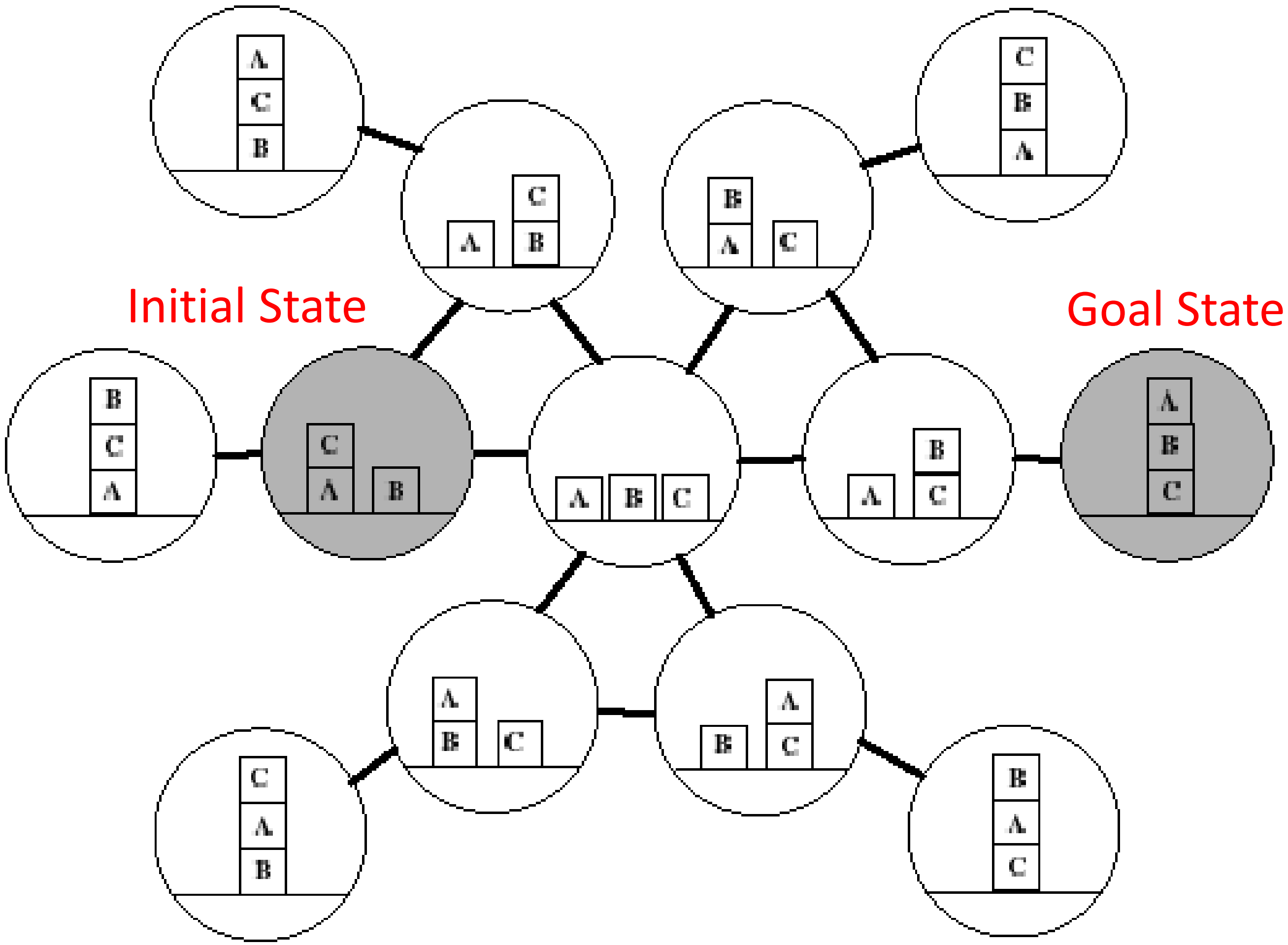
Sudeshna Sarkar
Oct 31 2022

Centre of Excellence in Artificial Intelligence, IIT Kharagpur

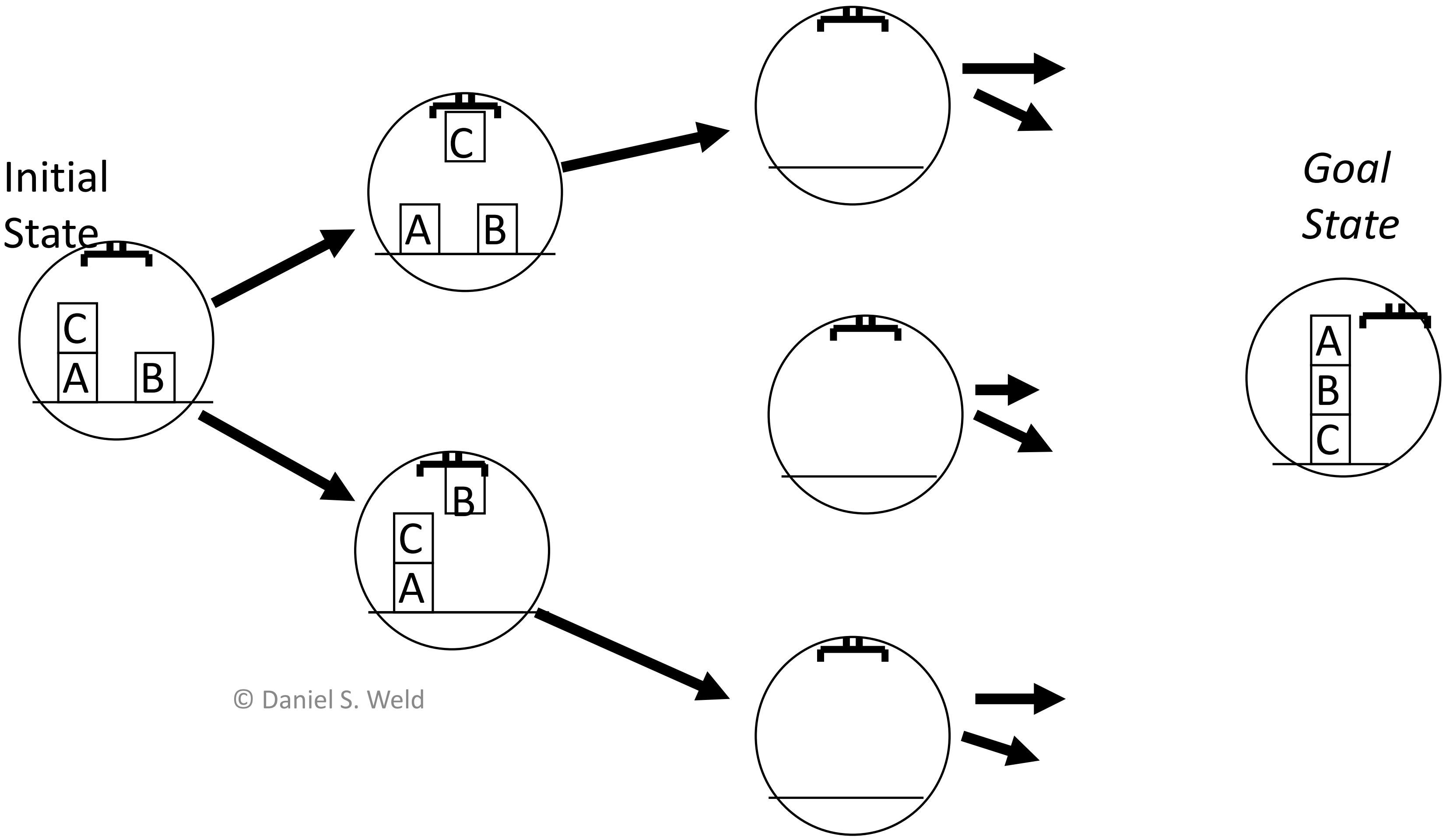
A General-purpose Planner

Search Space: Blocks World

Graph is finite



Forward State-Space Search



Initial state: set of positive ground literals

CWA: literals not appearing are false

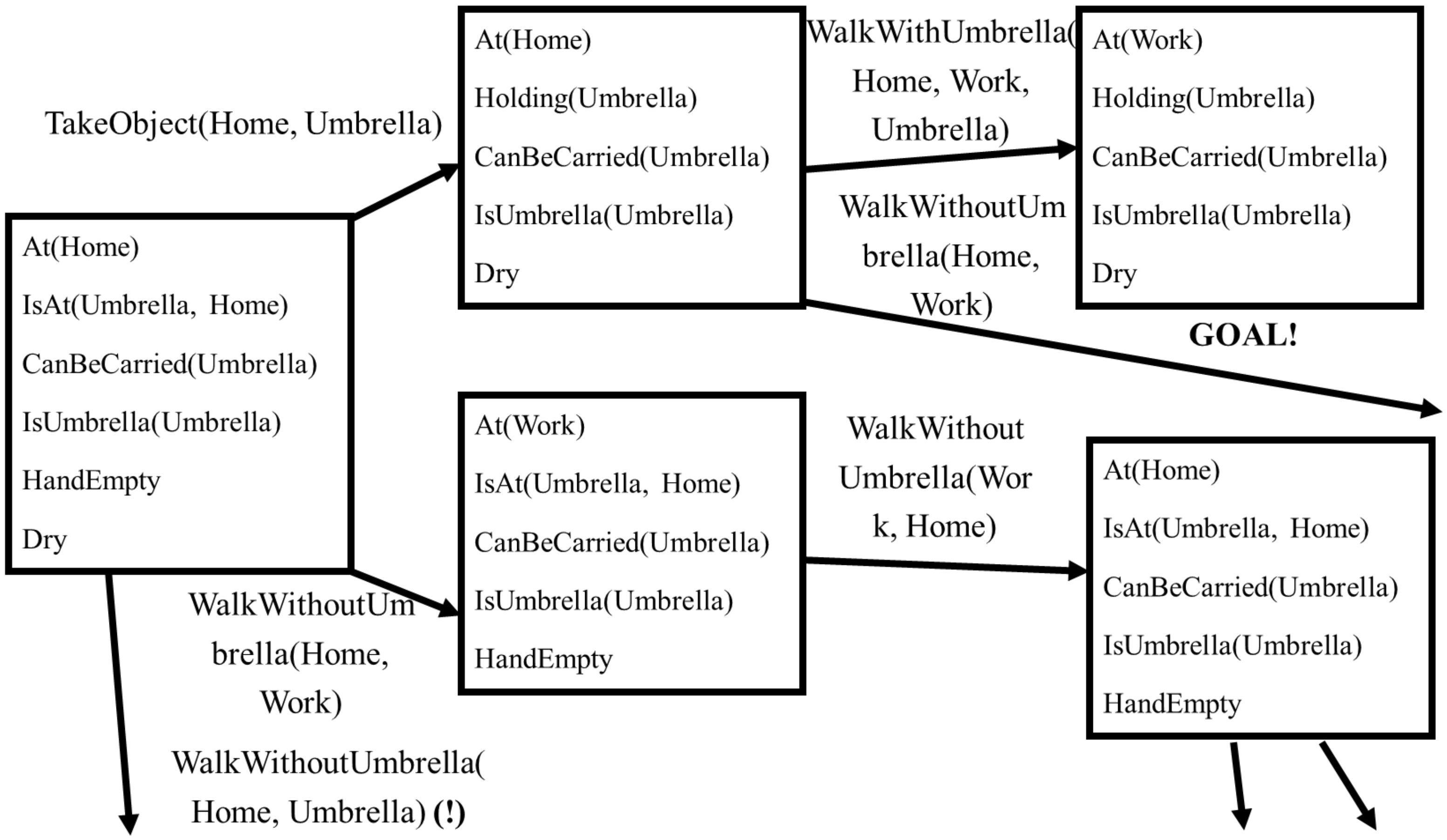
Actions:

- applicable if preconditions satisfied
- add positive effect literals
- remove negative effect literals

Goal test: does state logically satisfy goal?

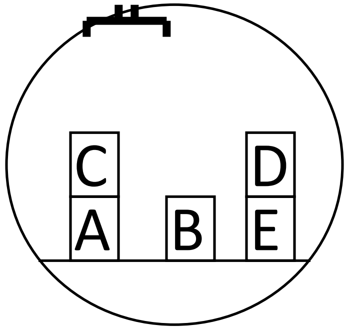
Successors: all states that can be reached with an action whose preconditions are satisfied in current state

Forward state-space search (progression planning)

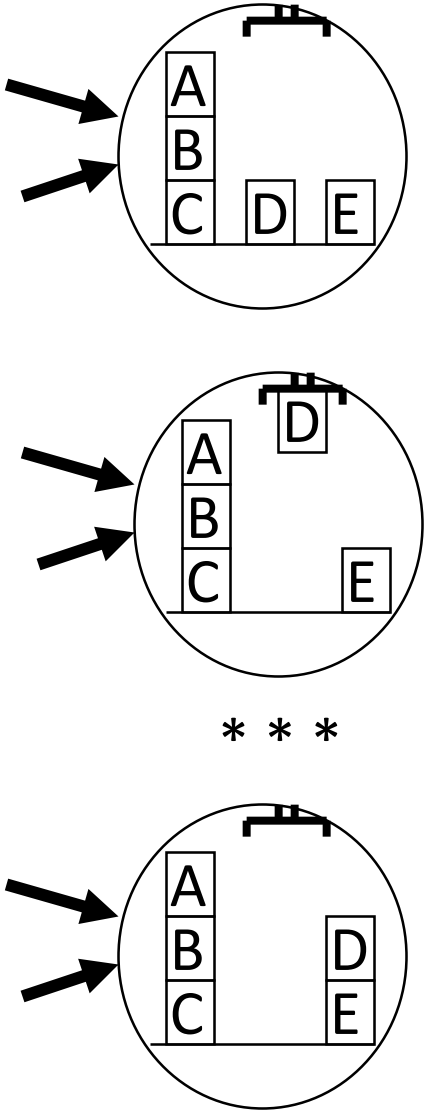


Backward Subgoal-Space Search

- Regression planning
- Problem: Need to find predecessors of state
- Problem: Many possible goal states are equally acceptable.
- From which one does one search?

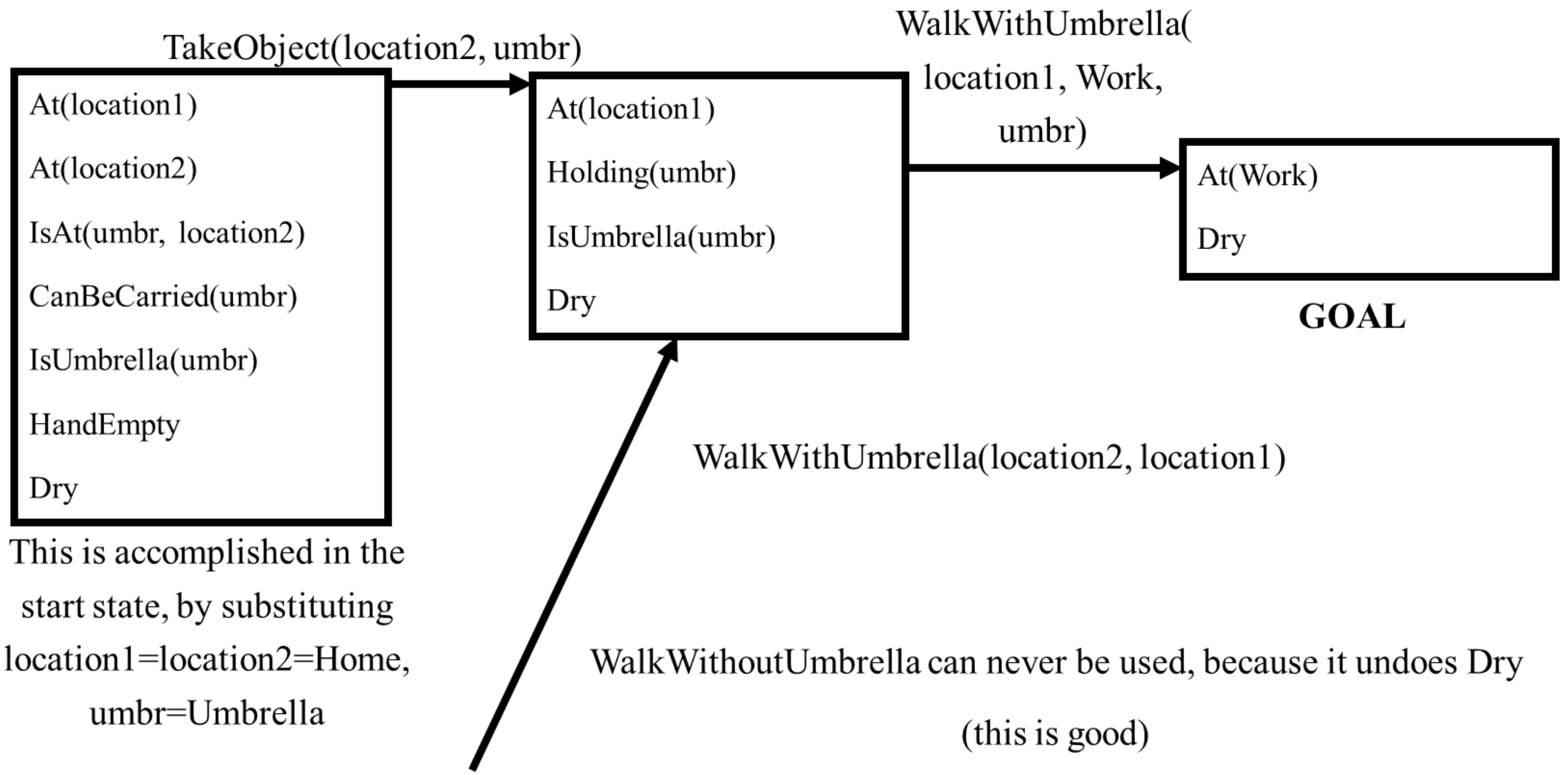


Initial State is completely defined



Backward state-space search (regression planning)

Predecessors: for every action that accomplishes one of the literals (and does not undo another literal), remove that literal and add all the preconditions



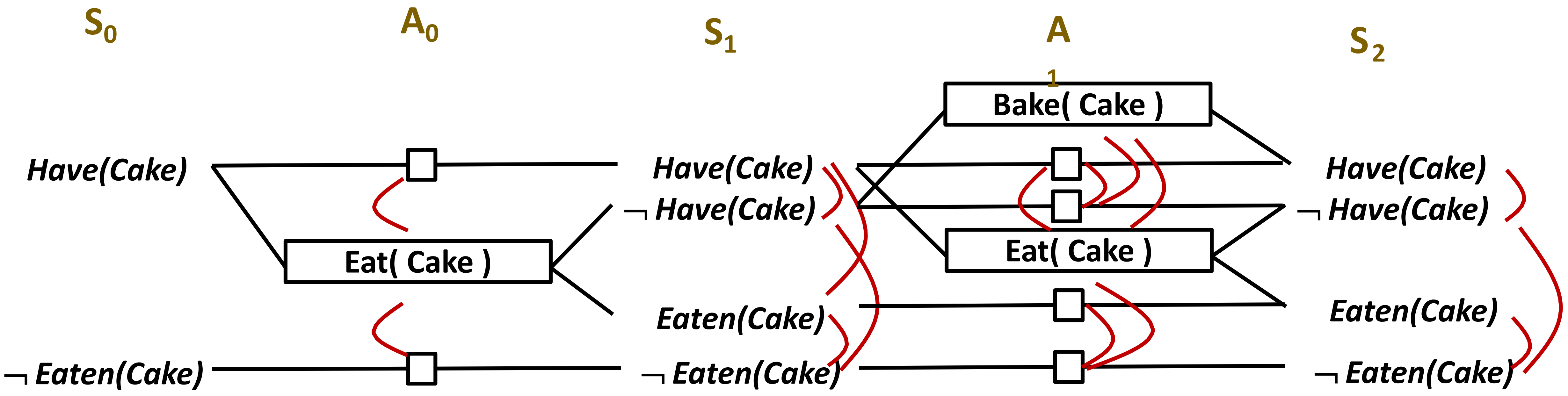
Other Kinds of Planners

- **Graphplan** converts a planning problem (the actions and initial/goal state) into a kind of graph that can be efficiently searched
- **SATplan** converts a planning problem into a SAT problem, and then uses a SAT solver to find a plan
- **Partial Order Planners** search the space of *plans* instead of the space of *states*
  - A partial order planner starts with an empty plan, and then inserts actions into the plan to make it better

Graphplan

Planning Graph

Initial state: Have(cake)
Goal: Have(cake), Eaten(cake)
Action Eat(cake):
  Preconditions: Have(cake)
  Effects: ¬Have(cake), Eaten(cake)
Action Bake(cake):
  Preconditions: ¬Have(cake)
  Effects: Have(cake)

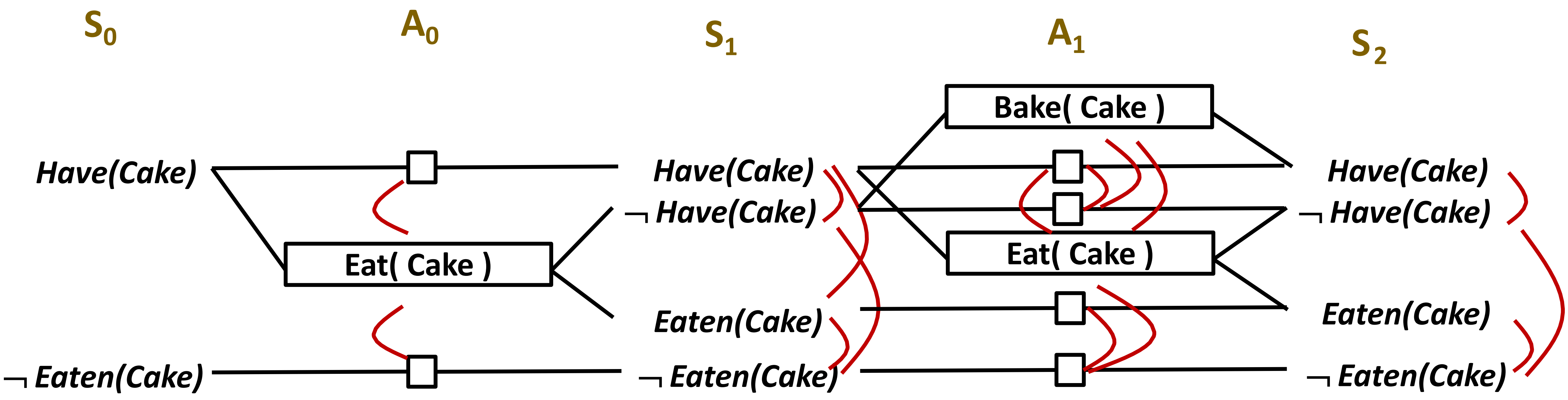


Graphplan

Phase 1: Construct a planning graph: encodes constraints on possible plans

Phase 2: Solution Extraction

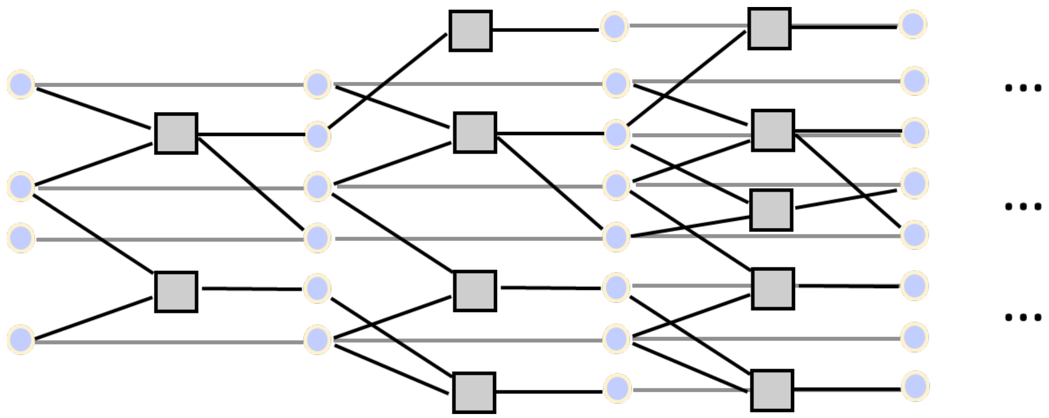
A Planning Graph



Gives a compact (but approximate) representation of states that are reachable by *n* level plans

Planning graph

Planning graphs consists of a sequence of levels that correspond to time steps in the plan.



Alternate levels

- **state levels** represent candidate propositions that *could* hold at that time step
- **action levels** represent candidate actions that could possibly be executed at the step

Arcs represent preconditions, adds and deletes
Can only execute one real action at a step, but the data structure keeps track of all actions & states that are possible

Graphplan

Phase 1: Construct a planning graph: encodes constraints on possible plans

- built from initial state
- contains actions and propositions that are possibly reachable from initial state
- does not include unreachable actions or propositions

Phase 2: Solution Extraction

- Backward search for the solution in the planning graph
  - backward from goal

Planning graph can be built for each problem in polynomial time

A Simple planning problem

Initial state: Have(cake)

Goal: Have(cake), Eaten(cake)

Action Eat(cake):
  Preconditions: Have(cake)
  Effects: ¬Have(cake), Eaten(cake)

Action Bake(cake):
  Preconditions: ¬Have(cake)
  Effects: Have(cake)

Solution:

Eat(cake)
Bake(cake)

Planning Graph for Cake Example: Initial State

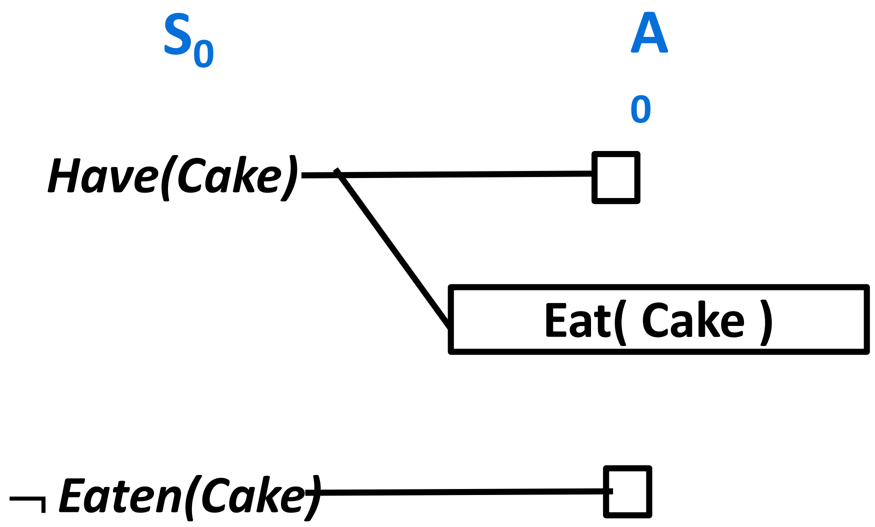
S_0

Have(Cake)

$\neg$ ***Eaten(Cake)***

Level S_0 has all literals from initial state

Add all applicable actions

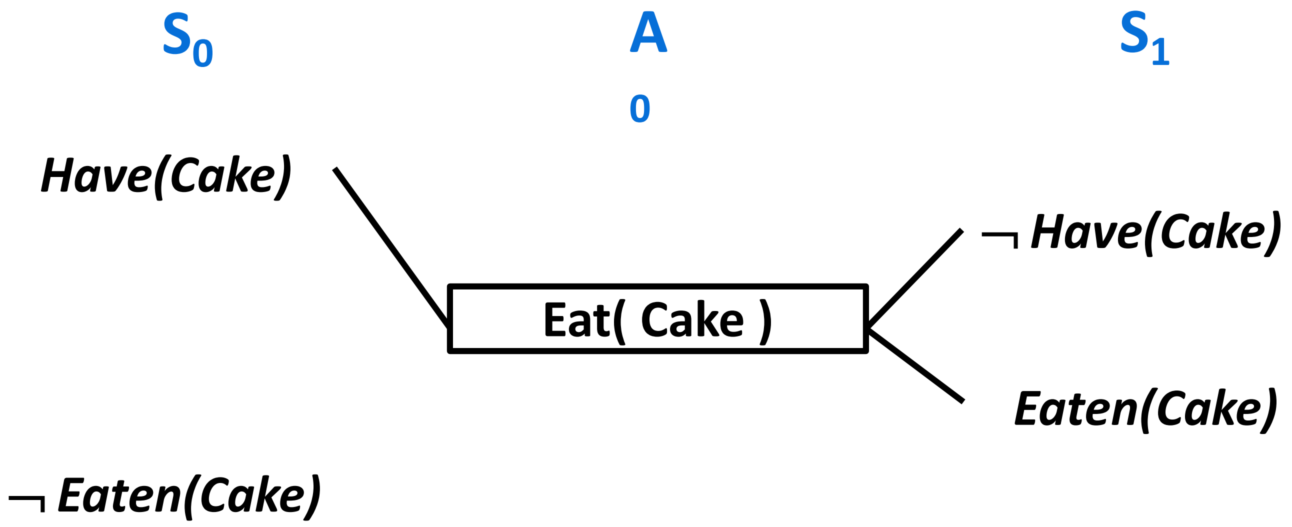


Proposition level S_0 : all literals from initial state

Action level A_0

- all actions whose preconditions are satisfied in S_0
- no-op for each proposition at level S_0

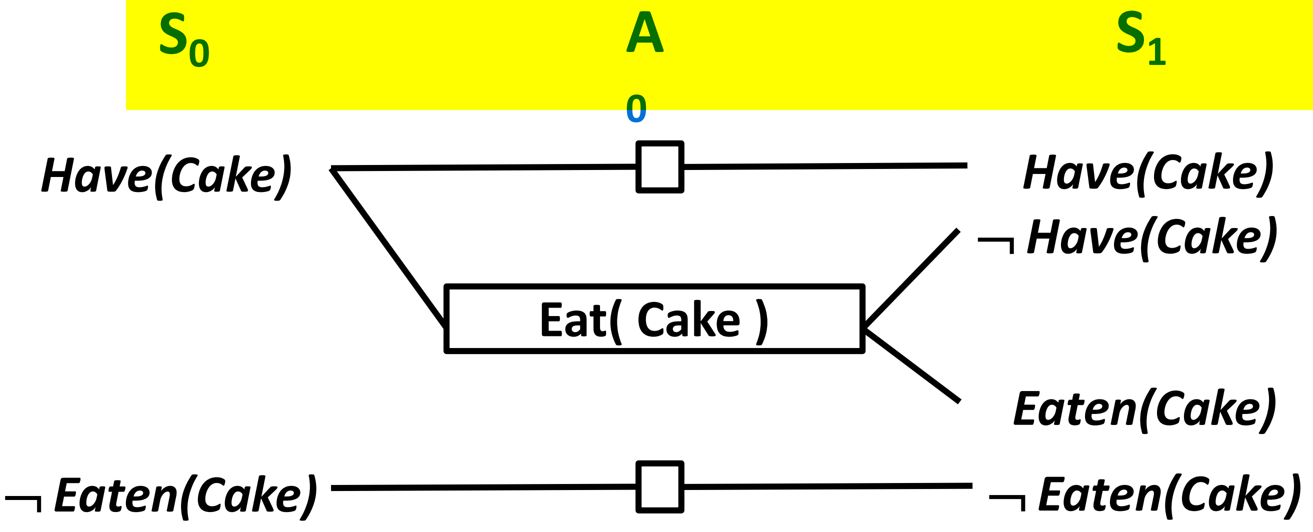
Add all effects



- Actions connect preconditions to effects

Add no-ops

Add no-ops to map all literals in state S_i to state S_{i+1}



No-op-action(P),
PRECOND: P
EFFECT: P
Have a no-op action for each ground fact

Graph Expansion Summary

Proposition level S_0

initial conditions

Action level A_i

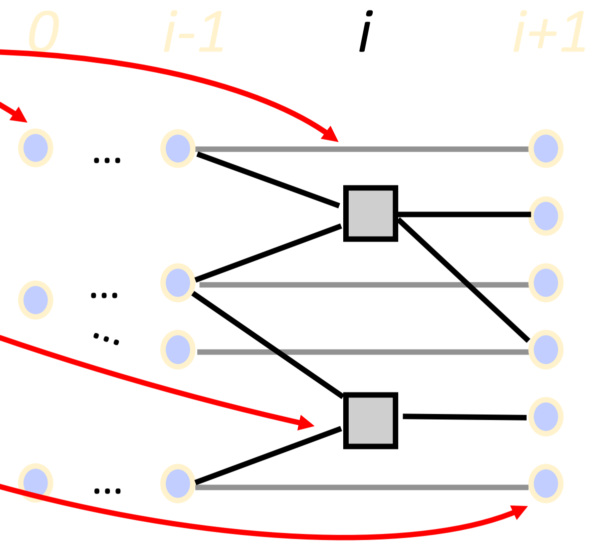
no-op for each proposition at level S_i

action for each operator instance whose preconditions exist at level S_i

Proposition level S_{i+1}

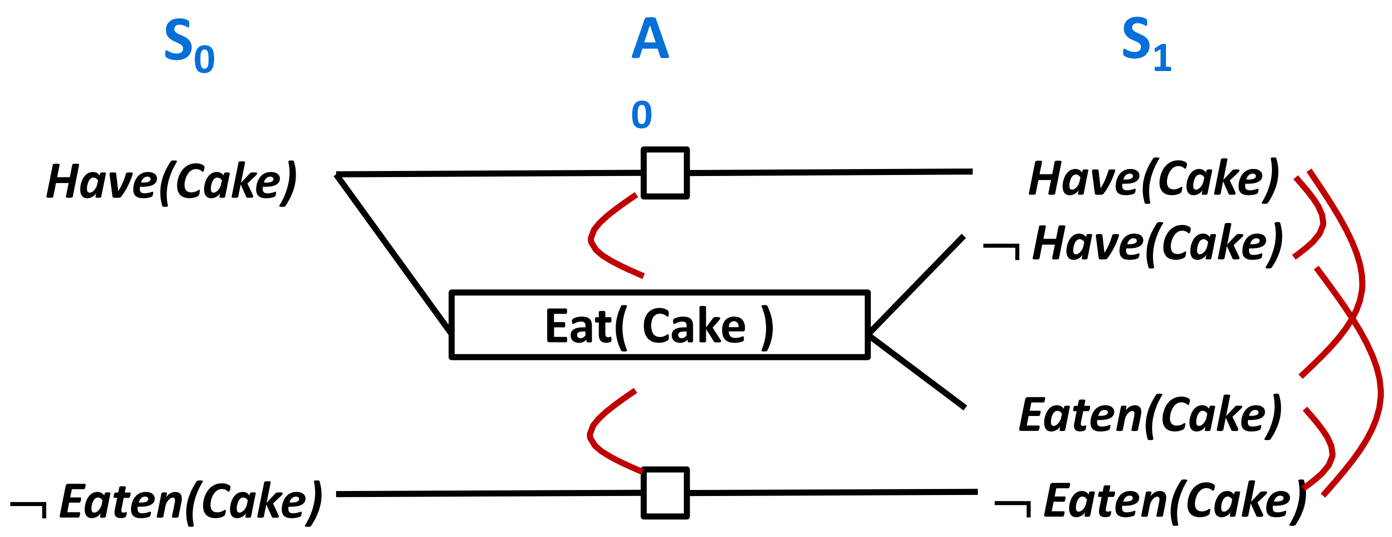
effects of each no-op and action at level A_i

Propagate mutex information



Mutual Exclusion

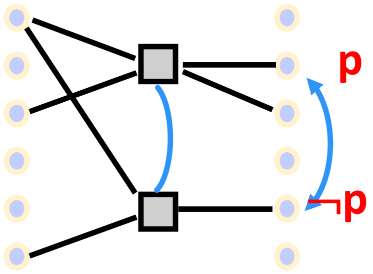
Identify *mutual exclusions* between actions and literals based on potential conflicts.

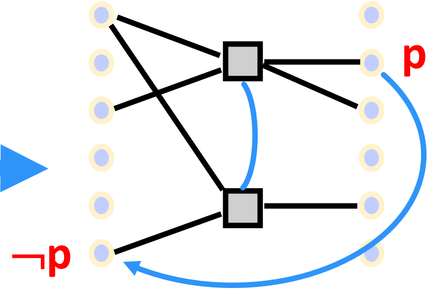


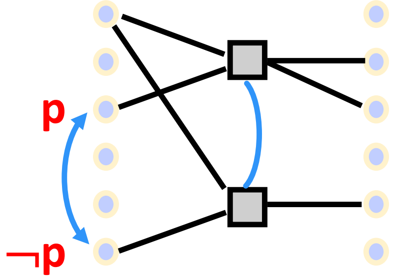
Mutual Exclusion

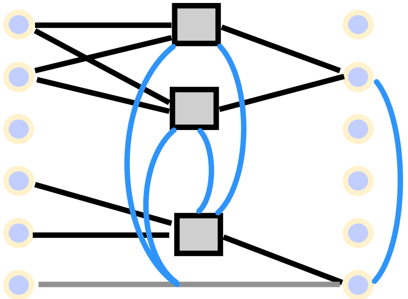
Two actions are mutex if
- Inconsistent effects: one action negates the effect of another.
- Interference: one of the effects of one action is the negation of a precondition of the other.
- Competing needs: one of the preconditions of one action is mutually exclusive with the precondition of the other.

- Two proposition are mutex if
  - Negation: one is the negation of the other
  - Inconsistent support: all ways of achieving them are mutex

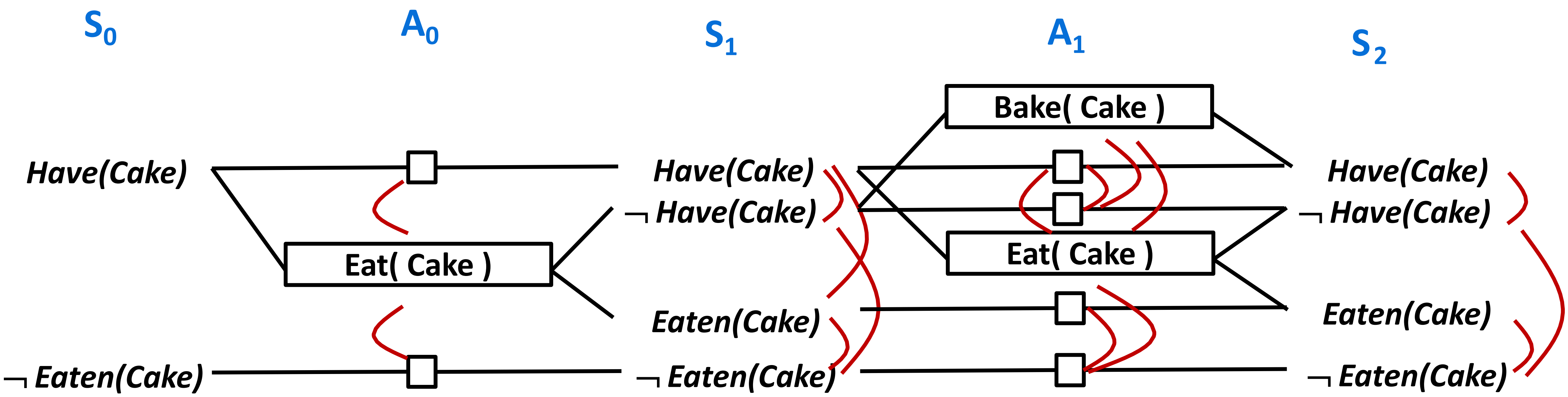








Mutex Actions



Inconsistent effects: one action negates an effect of another.

Eat(Cake) causes ¬Have(Cake) and Bake(Cake) causes Have(Cake)

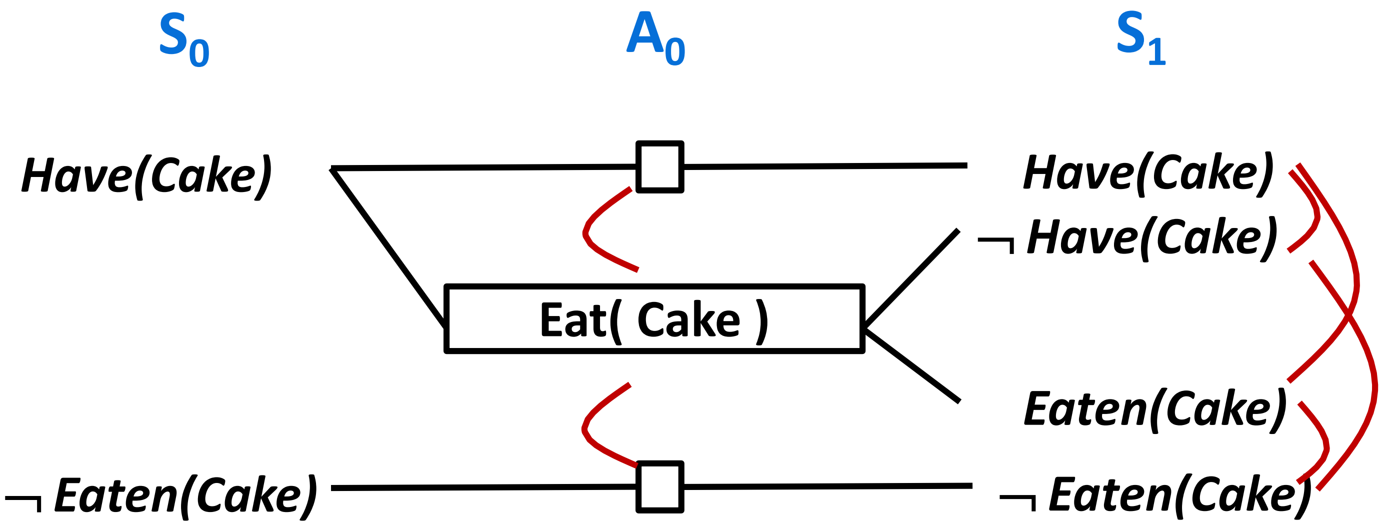
Interference: one of the effects of one action is the negation of the precondition for another.

Eat(cake) causes **¬Have(Cake)** negates the preconditions of the persistence actions for **Have(cake)**

Competing needs – one of the preconditions of one action is mutually exclusive with a precondition of the other

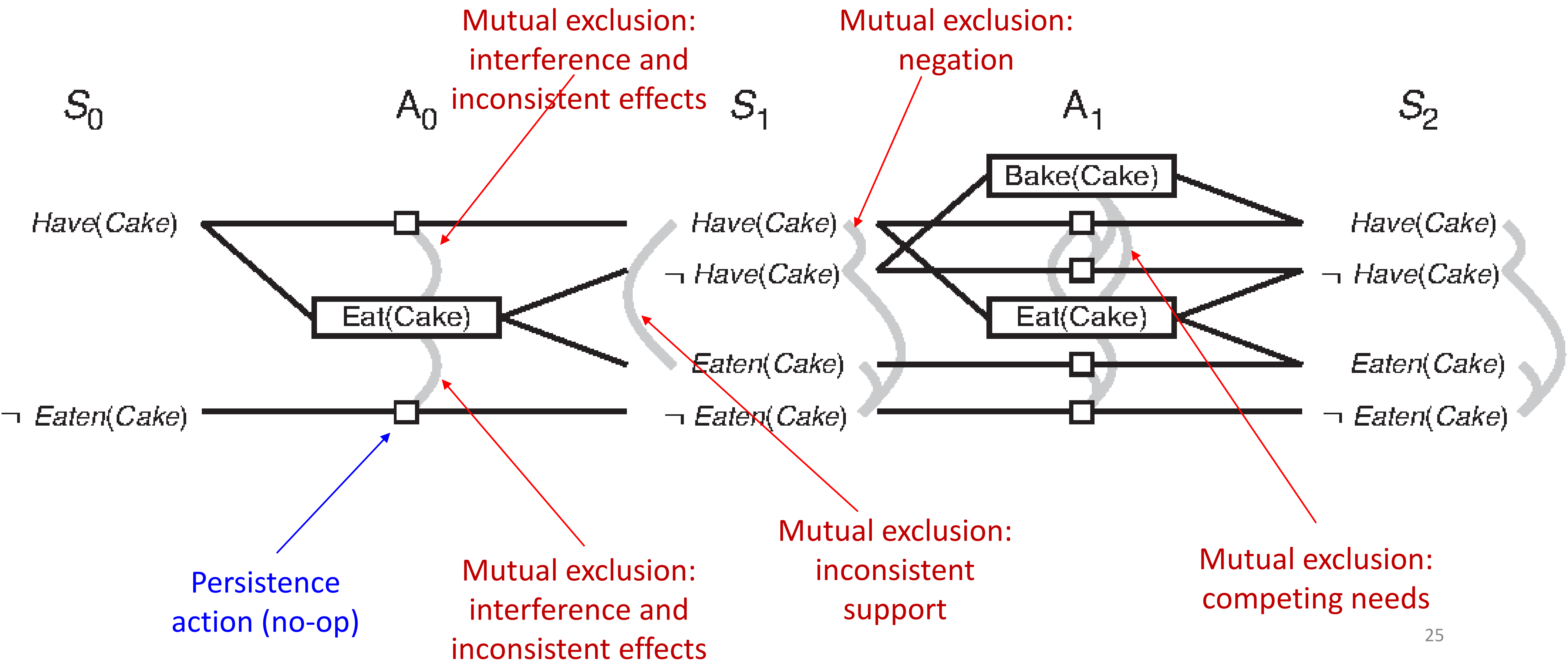
Bake(Cake) needs ¬Have(Cake) and Eat(Cake) needs Have(Cake)

Mutex Literals

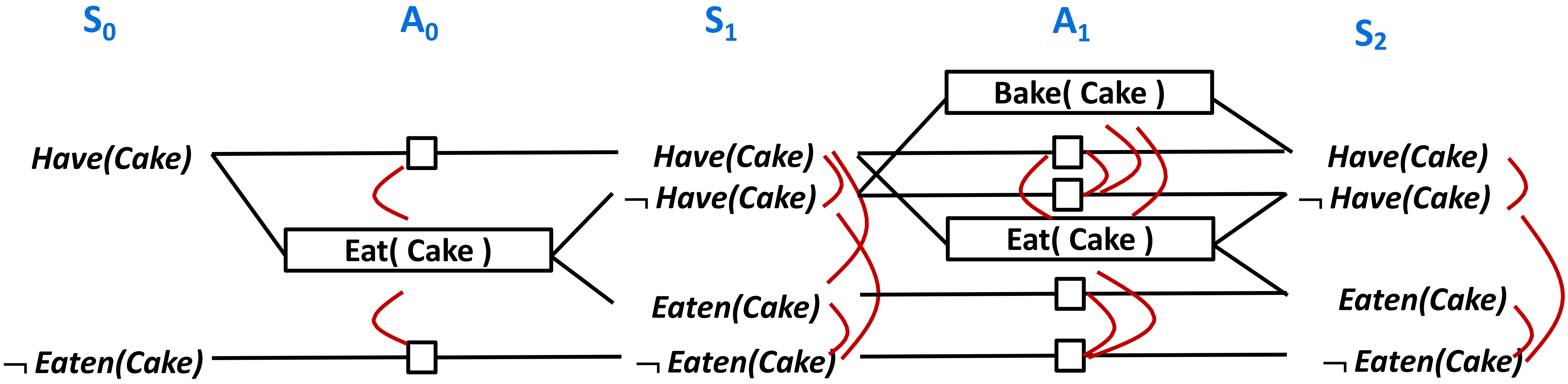


- A mutex relation holds between **two literals** when:
  - one is the negation of the other OR
  - each possible action pair that could achieve the literals is mutex (**inconsistent support**).

Mutex Relationships

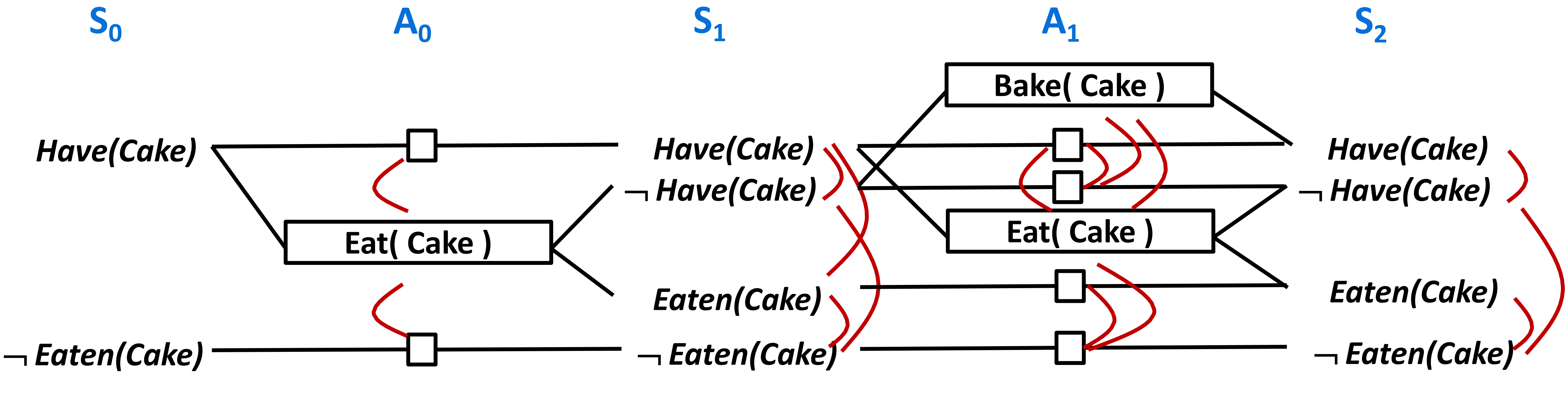


Planning Graph for Cake Example



Repeat process until graph levels off:

- two consecutive levels are identical or
- contain the same amount of literals



- If all of the literals in the goal are in the final state and are non-mutex …
- We can try to extract a plan from the plan graph

Properties of Planning Graph

- If goal is absent from last level?
  - Then goal cannot be achieved!

- If there exists a plan to achieve goal
  - Then goal is present in the last level &
  - No mutexes between conjuncts

- If goal is present in last level (w/ no mutexes) ?
  - There still may not exist any viable plan

Important Ideas

- Plan graph construction is polynomial time
- The plan graph captures important properties of the planning problem
  - Necessarily unreachable literals and actions
  - Possibly reachable literals and actions
  - Mutually exclusive literals and actions
- Significantly prunes search space compared to previously considered planners
- Plan graphs can also be used for deriving admissible (and good non-admissible) heuristics

GraphPlan Algorithm

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

graph ← INITIAL-PLANNING-GRAPH(*problem*)

goals ← CONJUNCTS(*problem*.GOAL)

nogoods ← an empty hash table

for t = 0 **to** ∞ **do**

    if *goals* all non-mutex in S_t of *graph* **then**

        solution ← EXTRACT-SOLUTION(*graph, goals,* NUMLEVELS(*graph*), *nogoods*)

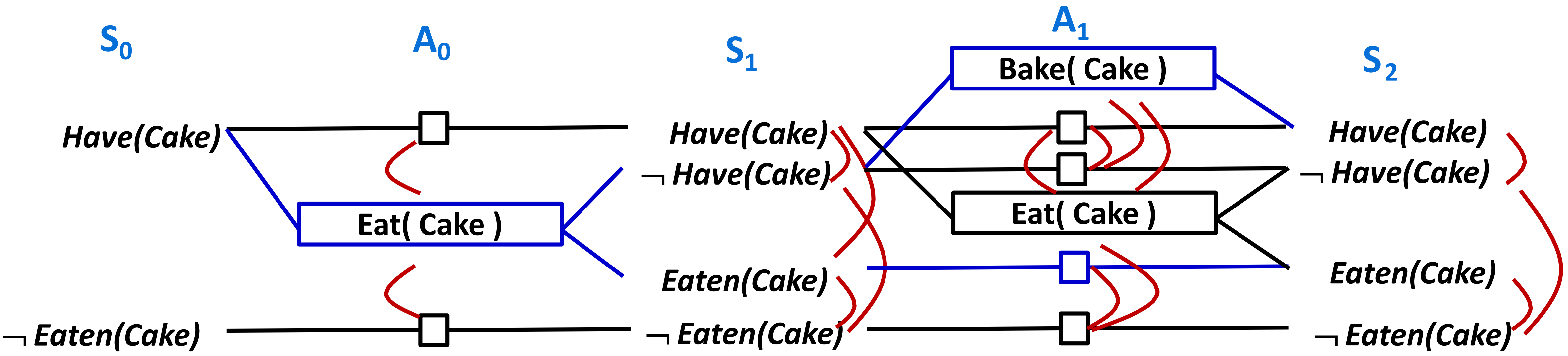
    if *graph* and *nogoods* have both leveled off **then return** *failure*

    graph ← EXPAND-GRAPH(*graph*, *problem*)

From Fig. 10.9, p. 383

Finding the plan

Once a world is found having all goal predicates without mutexes, the plancan be extracted by solving a constraint satisfaction problem (CSP) for resolving the mutexes



Solution Extraction: Backward Search

Search problem:

Start state: goal set at last level

Actions: conflict-free ways of achieving the current goal set

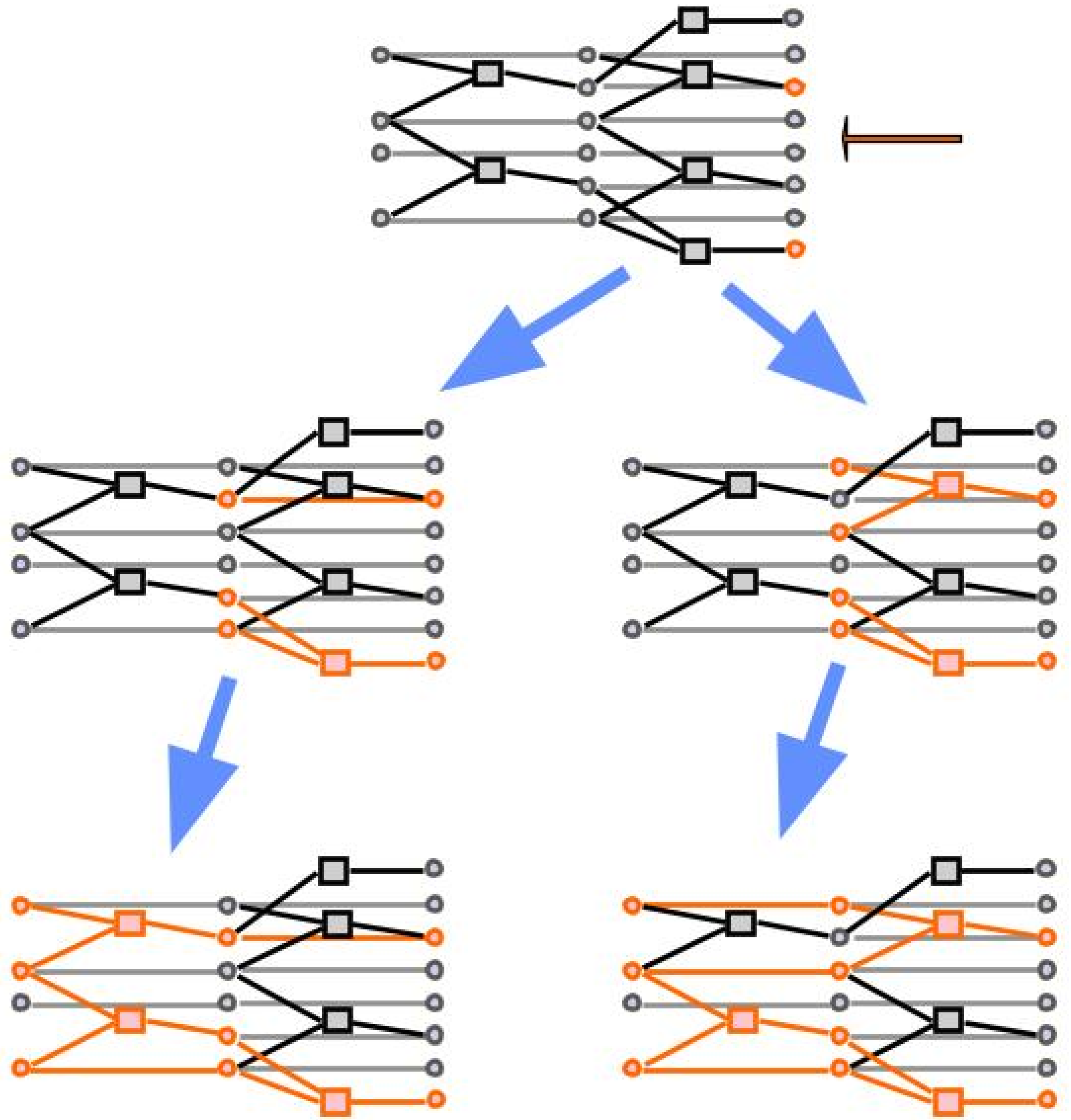
Terminal test: at S0 with goal set entailed by initial planning state

Repeat until goal set is empty

If goals are present and non-mutex:

1) Choose set of non-mutex actions to achieve each goal

2) Add preconditions to next goal set



Note: may need to start much deeper than the leveling-off point!

Nogood table for planning graph upto layer k

- For each layer j define nogood(j)
- A set of sets of goal propositions
  - Inner set: One combination of propositions that cannot be achieved
  - Outer set: All combinations that cannot be achieved at layer j.

- before searching for set g in S_j:
  - Check whether g is in nogood(j)
- when search for set g in S_j has failed:
  - Add g to nogood(j)

EXTRACT-SOLUTION(graph, goals, k=NUMLEVELS(graph))

if k=0 then return <>

if goals in nogood(i) then return FAIL

Π_k = gpSearch (graph, goals, {}, k)

if (PI != FAIL) then return PI

nogood(i) = nogodd(i) + goals

return FAIL

gpSearch(graph, goals, Π, i)

inputs: planning graph G, remaining sub-goals g, and set of actions already committed to π, both at level i

If goals={} then

Π= extract(graph, $\cup_{a\in\Pi}\,\mathrm{precond}(a)$, Π, i-1)

if Π = FAIL then return FAIL

return $\Pi \cdot < \pi >$

p = goals.selectOne()

resolvers = {a∈A_i | p∈addlist(a) and ¬∃a'∈π: (a,a')∈μAi }

if resolvers={} then return FAIL

a = resolvers.chooseOne()

return gpSearch (graph, goals−adlist(a), π+a, i)

Termination of GraphPLAN

- Literals increase monotonically
- Actions increase monotonically
- Mutexes decrease monotonically

This guarantees the existence of a fixpoint. Planning Graph ‘levels off’.

- After some time k all levels are identical