Lecture 7

CS 6260 Automated Planning and Learning

Heuristics for Planning

Department of Computer Science and Engineering

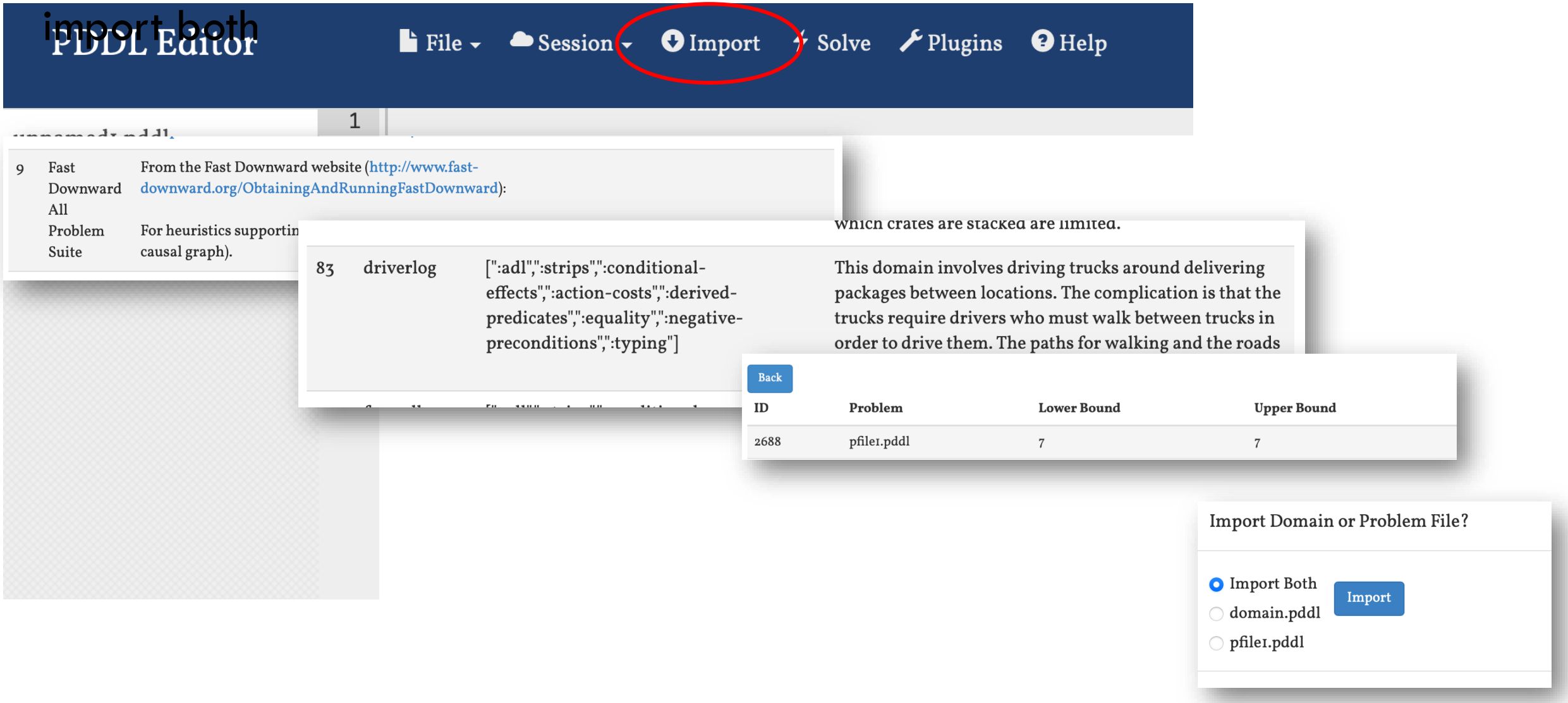
Indian Institute of Technology Madras

Some material based on lecture slides for ASU's CSE571 Artificial Intelligence by Siddharth Srivastava.

Recap: PDDL

Another Example: Driverlog

Import→ Fast Downward All problem suite → driverlog → pfile 1→ import both



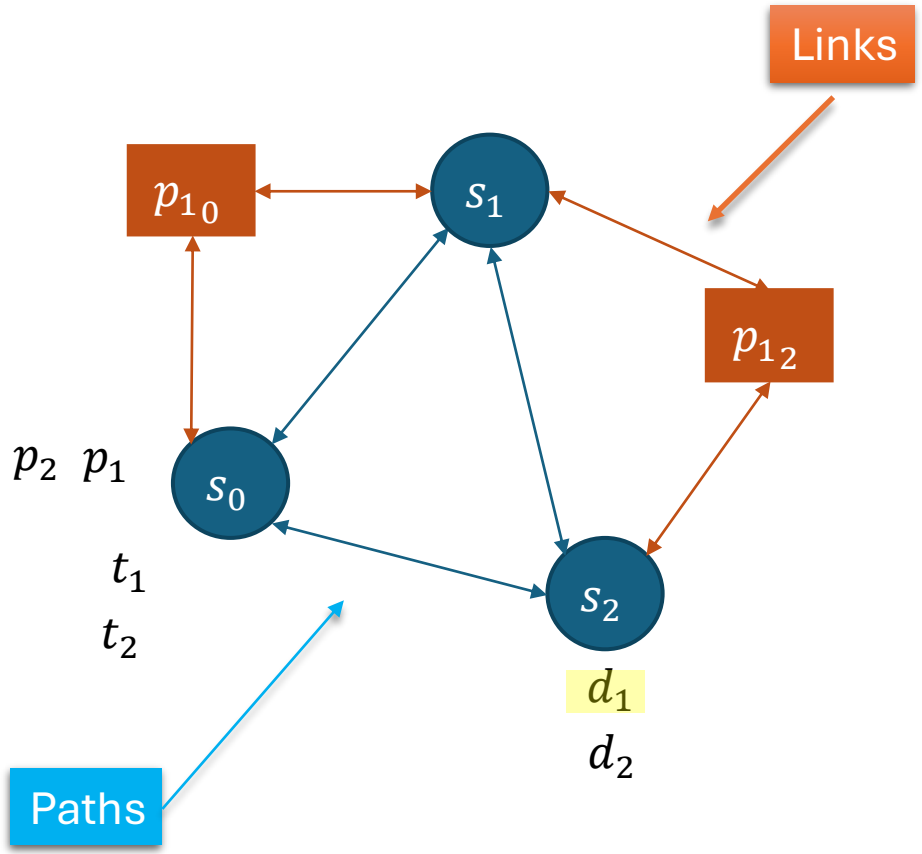
```
(define (problem DLOG-2-2-2)
    (:domain driverlog)
    (:objects
    driver1
    driver2
    truck1
    truck2
    package1
    package2
    s0
    s1
    s2
    p1-0
    p1-2
    )
    (:init
    (at driver1 s2)
    (DRIVER driver1)
    (at driver2 s2)
    (DRIVER driver2)
    (at truck1 s0)
    (empty truck1)
    (TRUCK truck1)
    (at truck2 s0)
    (empty truck2)
    (TRUCK truck2)
    (at package1 s0)
    (OBJ package1)
    (at package2 s0)
    (OBJ package2)
    (LOCATION s0)
    (LOCATION s1)
    (LOCATION s2)
    (LOCATION p1-0)
    (LOCATION p1-2)
    (path s1 p1-0)
    (path p1-0 s1)
    (path s0 p1-0)
    (path p1-0 s0)
    (path s1 p1-2)
    (path p1-2 s1)
    (path s2 p1-2)
    (path p1-2 s2)
    (link s0 s1)
    (link s1 s0)
    (link s0 s2)
    (link s2 s0)
    (link s2 s1)
    (link s1 s2)
)

    (:goal (and
    (at driver1 s1)
    (at truck1 s1)
    (at package1 s0)
    (at package2 s0)
    ))


)
```

```
(:action DRIVE-TRUCK
  :parameters
   (?truck
    ?loc-from
    ?loc-to
    ?driver)
  :precondition
   (and (TRUCK ?truck) (LOCATION ?loc-from) (LOCATION ?loc-to) (DRIVER ?driver)
   (at ?truck ?loc-from)
   (driving ?driver ?truck) (link ?loc-from ?loc-to))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
```

```
(:action WALK
  :parameters
   (?driver
    ?loc-from
    ?loc-to)
  :precondition
   (and (DRIVER ?driver) (LOCATION ?loc-from) (LOCATION ?loc-to)
    (at ?driver ?loc-from) (path ?loc-from ?loc-to))
  :effect
   (and (not (at ?driver ?loc-from)) (at ?driver ?loc-to)))
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

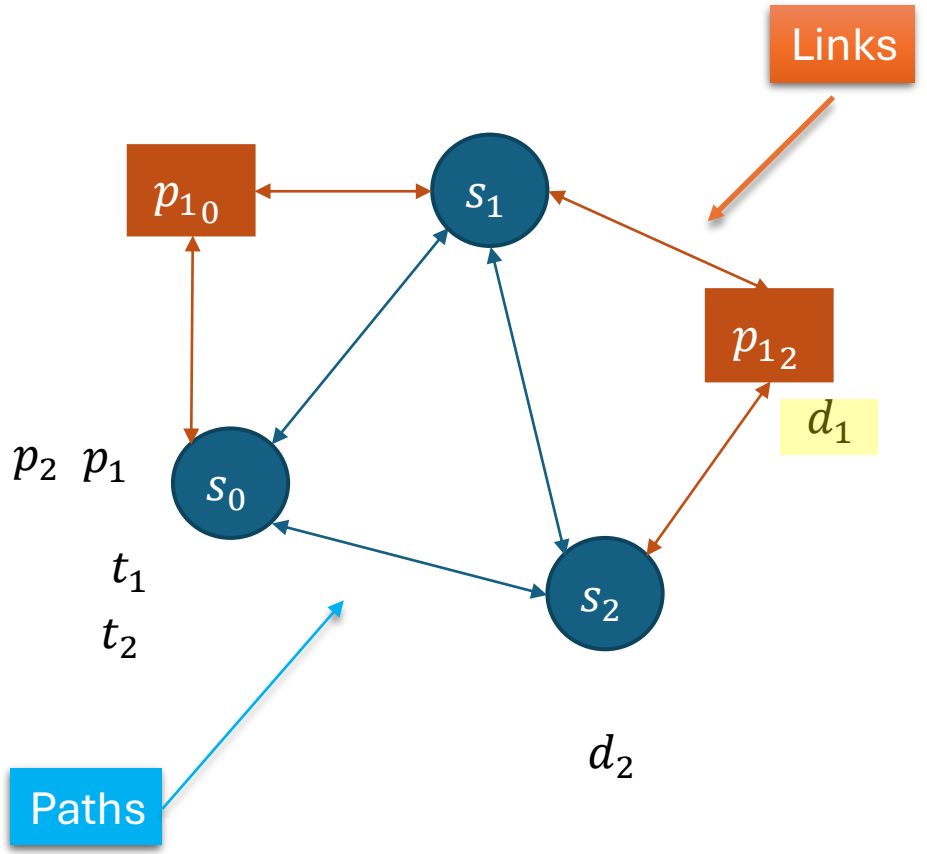
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver1 s2 p1-2)
  :precondition
    (and
      (driver driver1)
      (location s2)
      (location p1-2)
      (at driver1 s2)
      (path s2 p1-2)
    )
  :effect
    (and
      (not
        (at driver1 s2)
      )
      (at driver1 p1-2)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

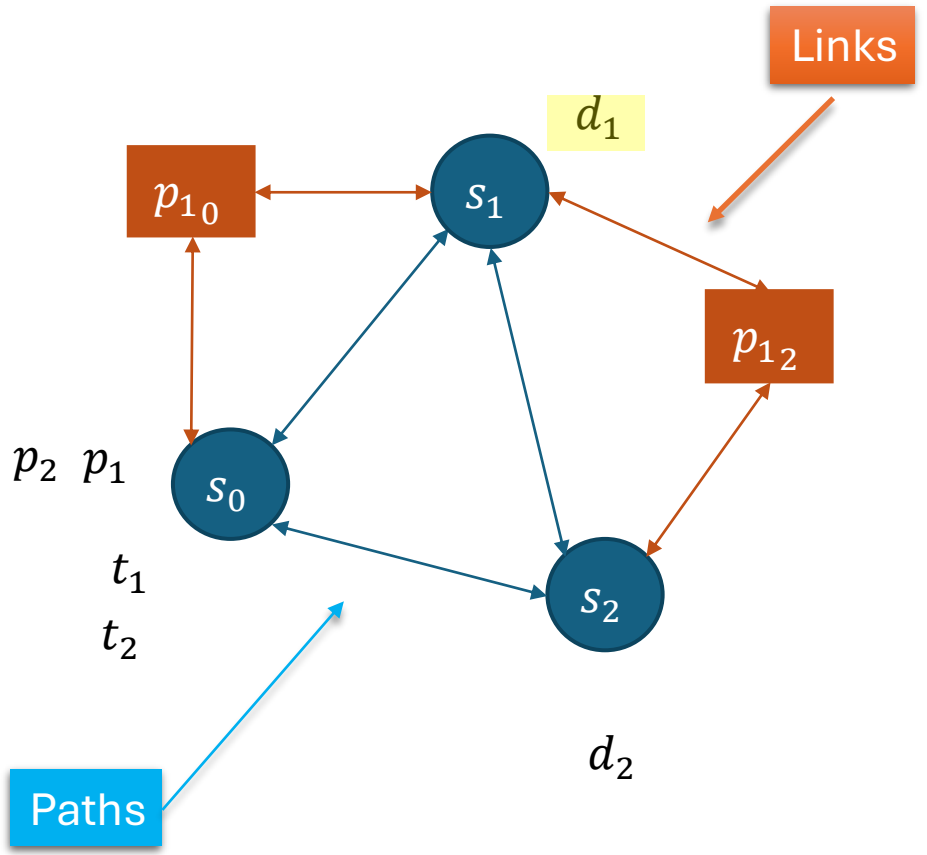
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver1 p1-2 s1)
  :precondition
    (and
      (driver driver1)
      (location p1-2)
      (location s1)
      (at driver1 p1-2)
      (path p1-2 s1)
    )
  :effect
    (and
      (not
        (at driver1 p1-2)
      )
      (at driver1 s1)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

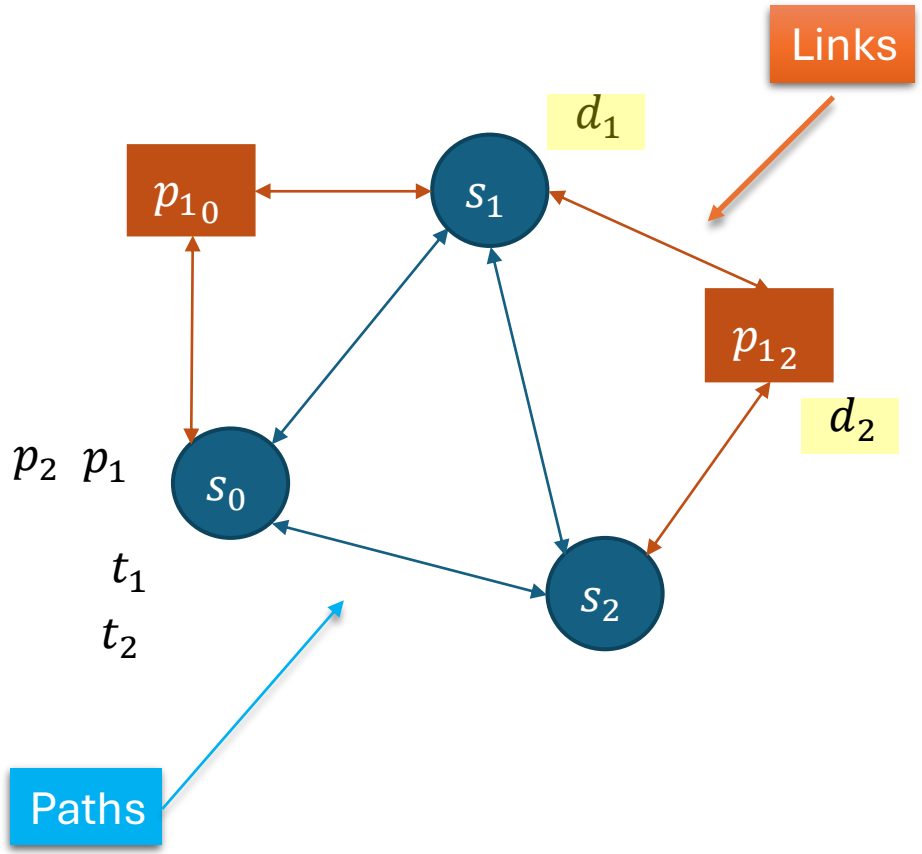
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver2 s2 p1-2)
  :precondition
    (and
      (driver driver2)
      (location s2)
      (location p1-2)
      (at driver2 s2)
      (path s2 p1-2)
    )
  :effect
    (and
      (not
        (at driver2 s2)
      )
      (at driver2 p1-2)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

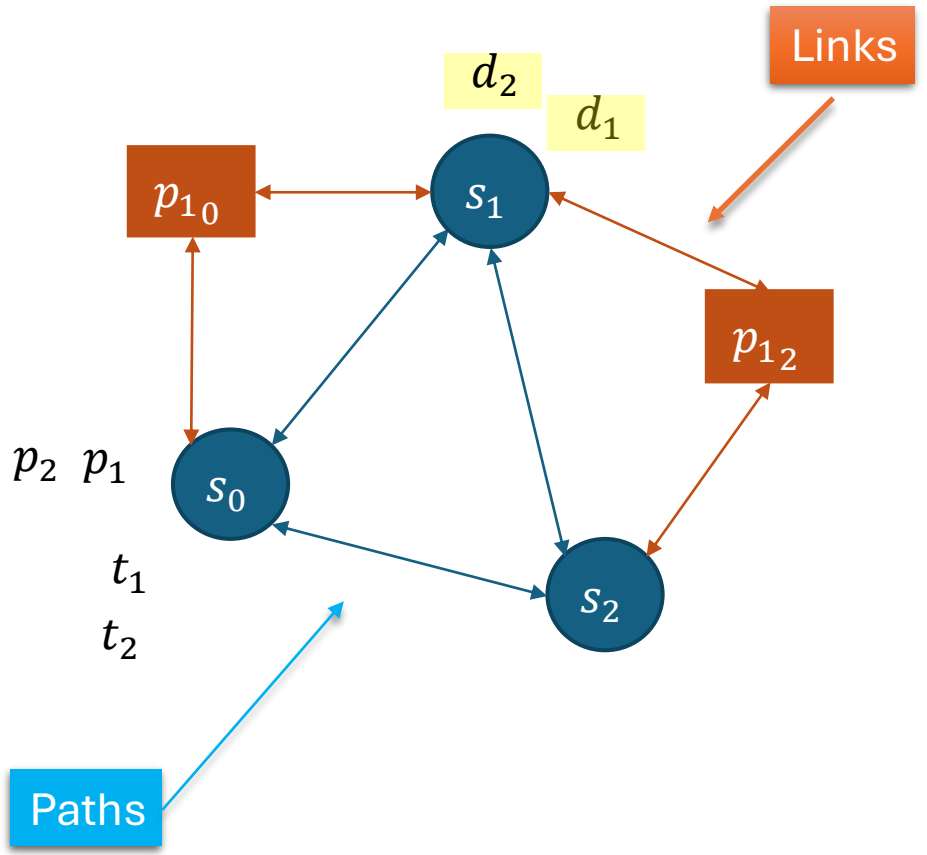
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver2 p1-2 s1)
  :precondition
    (and
      (driver driver2)
      (location p1-2)
      (location s1)
      (at driver2 p1-2)
      (path p1-2 s1)
    )
  :effect
    (and
      (not
        (at driver2 p1-2)
      )
      (at driver2 s1)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

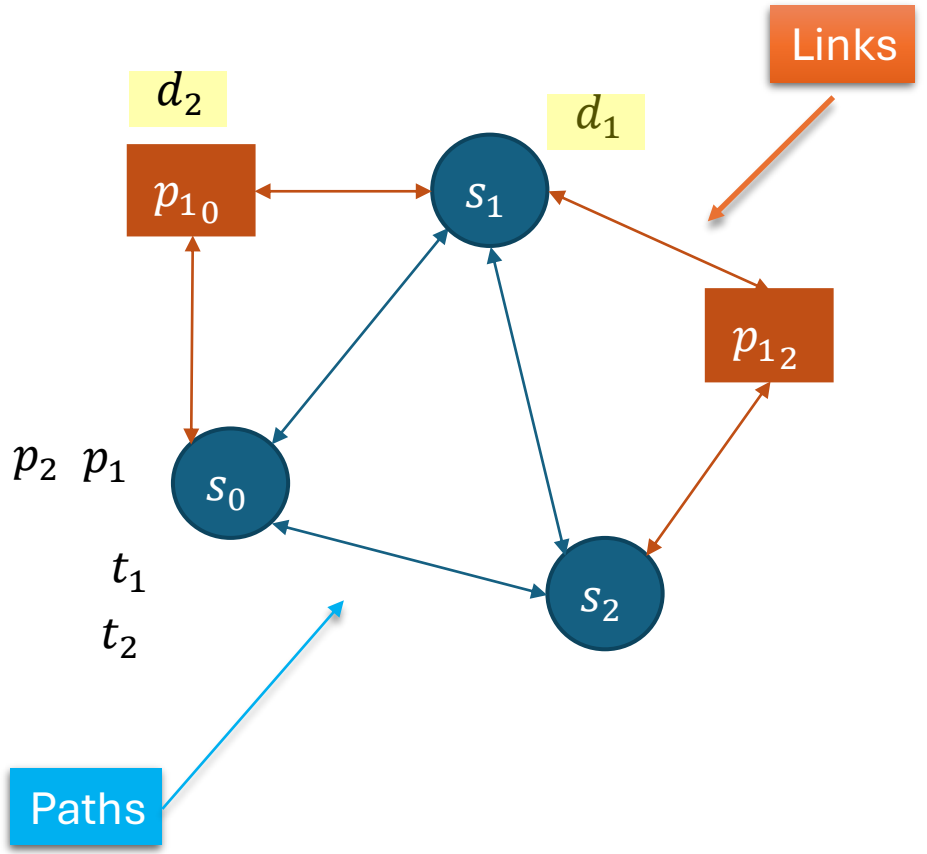
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver2 s1 p1-0)
  :precondition
    (and
      (driver driver2)
      (location s1)
      (location p1-0)
      (at driver2 s1)
      (path s1 p1-0)
    )
  :effect
    (and
      (not
        (at driver2 s1)
      )
      (at driver2 p1-0)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

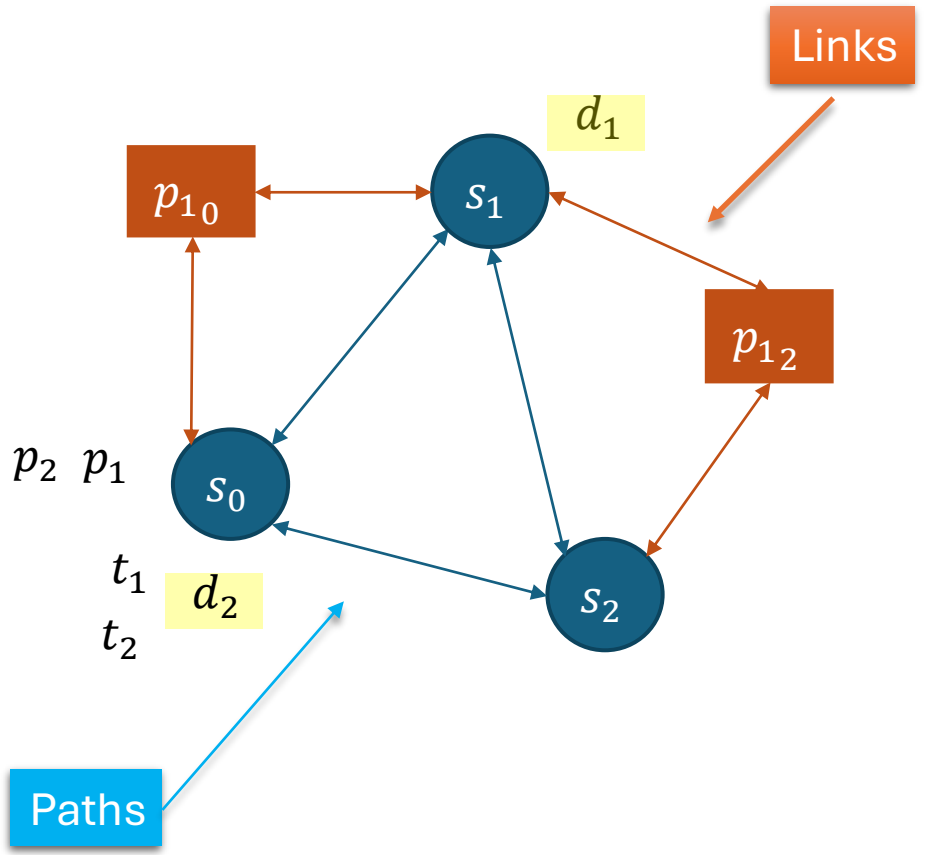
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action walk
  :parameters (driver2 p1-0 s0)
  :precondition
    (and
      (driver driver2)
      (location p1-0)
      (location s0)
      (at driver2 p1-0)
      (path p1-0 s0)
    )
  :effect
    (and
      (not
        (at driver2 p1-0)
      )
      (at driver2 s0)
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

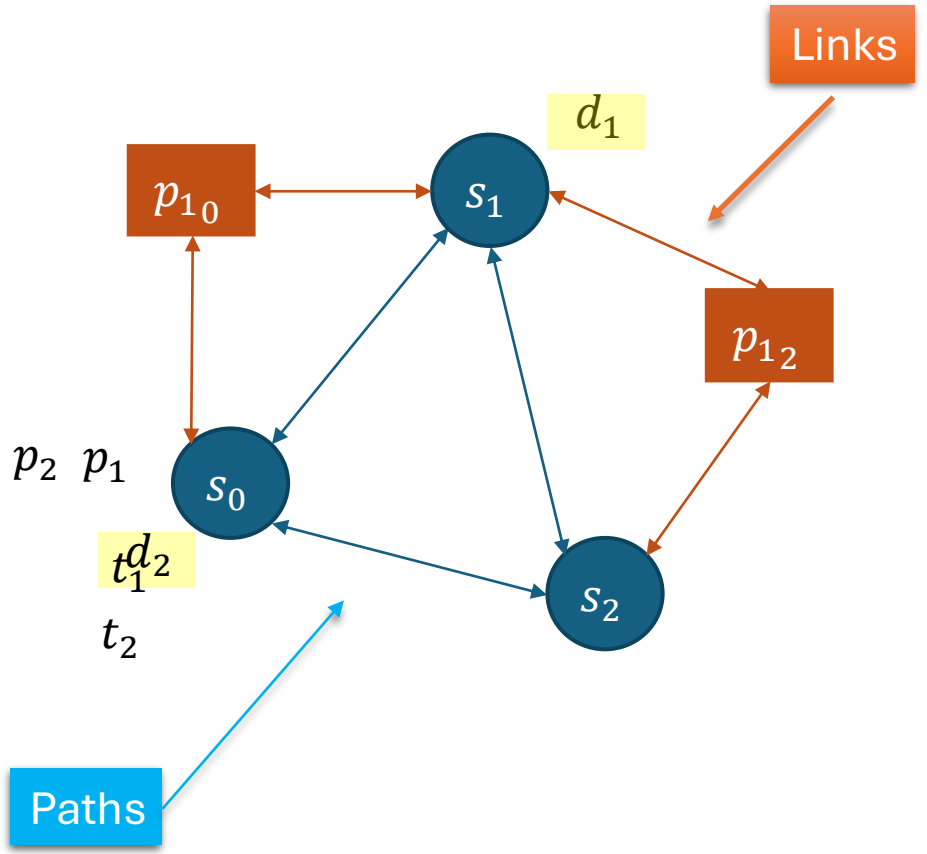
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action board-truck
  :parameters (driver2 truck1 s0)
  :precondition
    (and
      (driver driver2)
      (truck truck1)
      (location s0)
      (at truck1 s0)
      (at driver2 s0)
      (empty truck1)
    )
  :effect
    (and
      (not
        (at driver2 s0)
      )
      (driving driver2 truck1)
      (not
        (empty truck1)
      )
    )
)
```



(walk driver1 s2 p1-2)

(walk driver1 p1-2 s1)

(walk driver2 s2 p1-2)

(walk driver2 p1-2 s1)

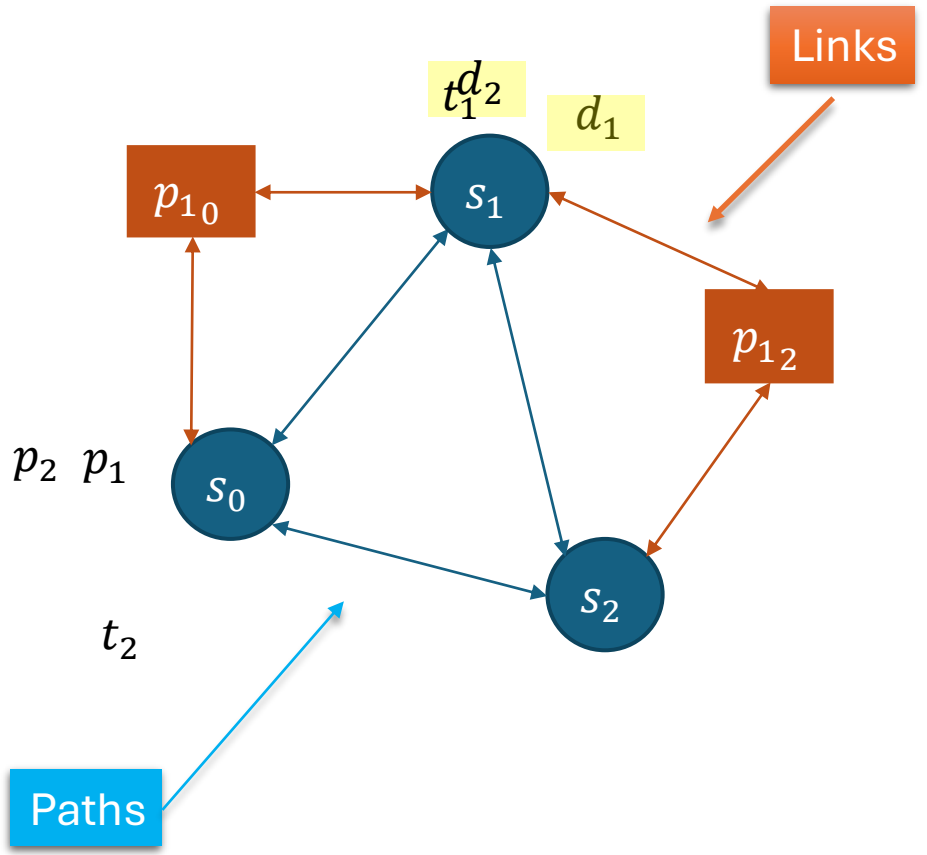
(walk driver2 s1 p1-0)

(walk driver2 p1-0 s0)

(board-truck driver2 truck1 s0)

(drive-truck truck1 s0 s1 driver2)

```
(:action drive-truck
  :parameters (truck1 s0 s1 driver2)
  :precondition
    (and
      (truck truck1)
      (location s0)
      (location s1)
      (driver driver2)
      (at truck1 s0)
      (driving driver2 truck1)
      (link s0 s1)
    )
  :effect
    (and
      (not
        (at truck1 s0)
      )
      (at truck1 s1)
    )
)
```



Is the plan optimal?

Satisficing vs Optimal Plans

- Optimal Plans:
  - Least cost to reach the goal from the initial state
  - Relatively difficult to generate
  - Can be more than one

- Satisficing Plans:
  - Solve the problem (reaches the goal), but not optimal
  - Relatively easier to generate
  - Number of satisficing plans $\geq$ Number of optimal plans

Heuristics leveraging PDDL

Automated Search Control

- PDDL Domain + PDDL Problem
  = Search problem definition
- Most popular solution approach:
  - Informed search without hand-coded heuristic
  - *Pseudo-informed* search?
- Heuristics, pruning strategies derived automatically
  - Using domain independent algorithms

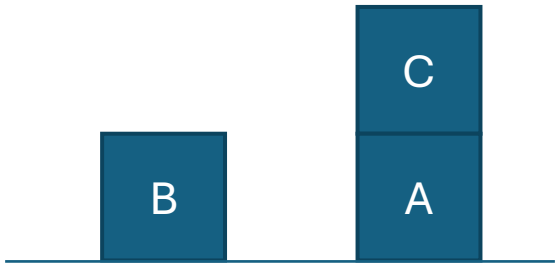
- A search problem is defined by
  - S set of possible states
    - Defined, but not enumerated a priori
  - $s_0 \in S$ start state (~vertex)
  - A set of actions
  - $N: S \rightarrow S$ successor function
    - $T: S \times A \rightarrow S$ deterministic transition function
    - $N(s) = \{s' : \exists a \, T(s,a) = s'\}$
  - $G: S \rightarrow \{True, False\}$ goal test
  - $C: S \times A \times S \rightarrow \mathbb{R}^+$ path cost

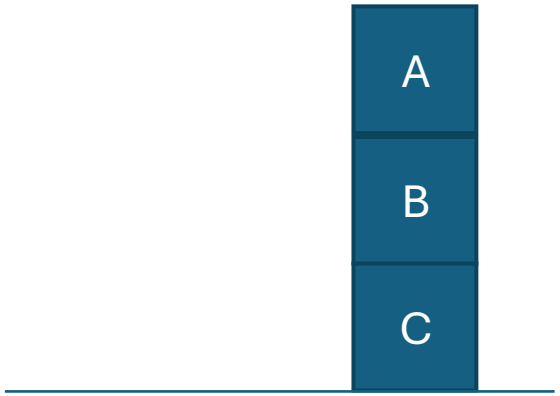
Ideas for Generating Heuristics and Search Control

- Goal decomposition (not quite)
- Planning graph
- Landmarks

Goal Decomposition

- Early approach motivated by the divide-and-conquer principle
- If the goal requires multiple properties, achieve each one independently
- What could go wrong?
- Sussman's anomaly

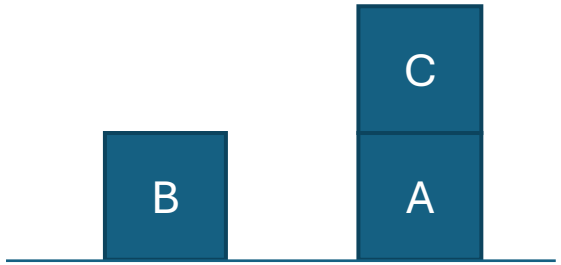




Goal requires: on(A, B); on(B,C)

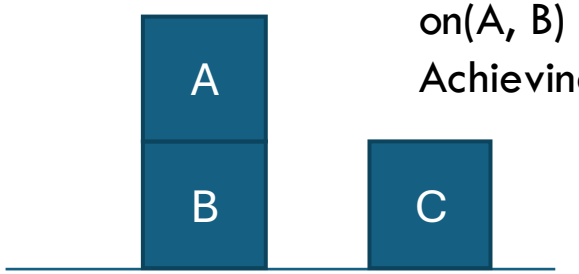
Goal Decomposition: Sussman's Anomaly

Goal requires: on(A, B); on(B,C)
What happens if we try to achieve them independently?

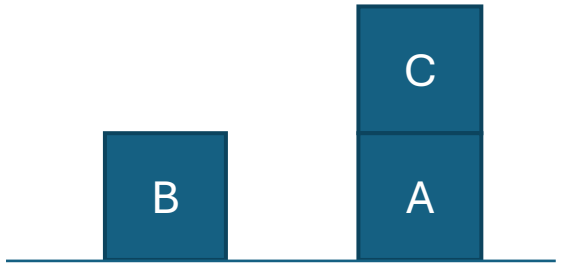


Try to achieve: on(A, B)

Move(C, table); Move(A, B)

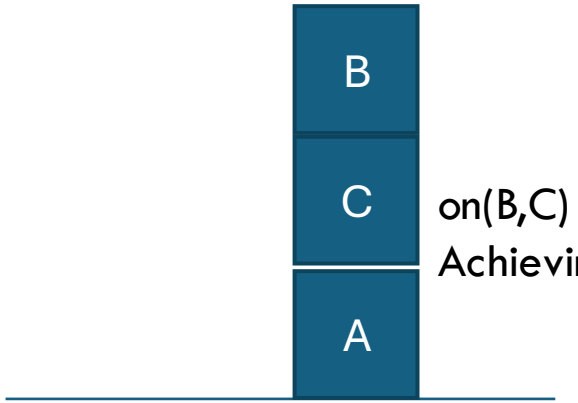


on(A, B) achieved ☺
Achieving on(B,C) will undo it ☹



Try to achieve: on(B, C)

Move(B, C)



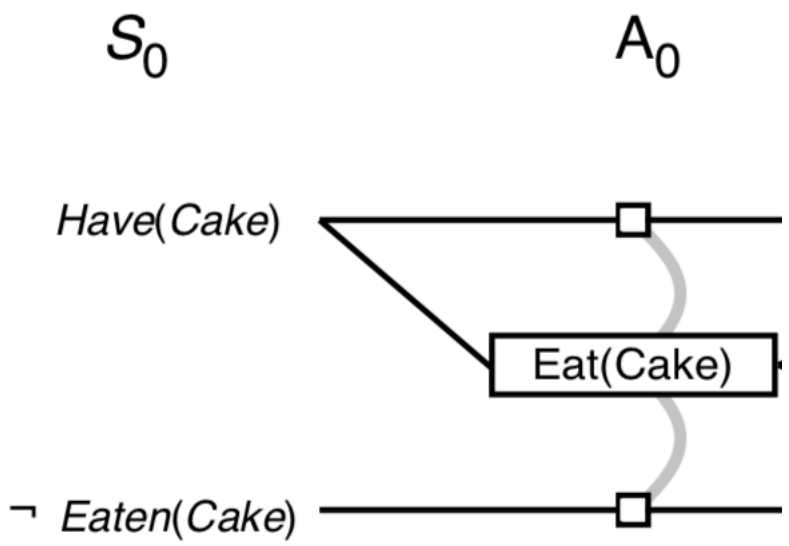
on(B,C) achieved ☺
Achieving on(A,B) will undo it ☹

Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg\, Have(Cake)$
  EFFECT: $Have(Cake)$

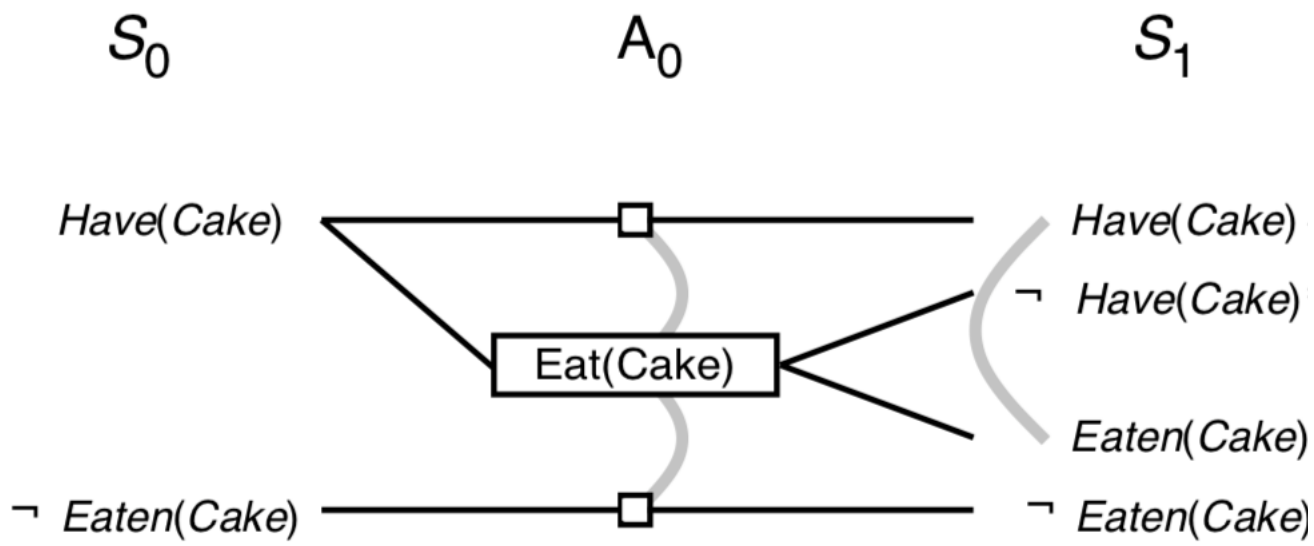
Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg\, Have(Cake)$
  EFFECT: $Have(Cake)$



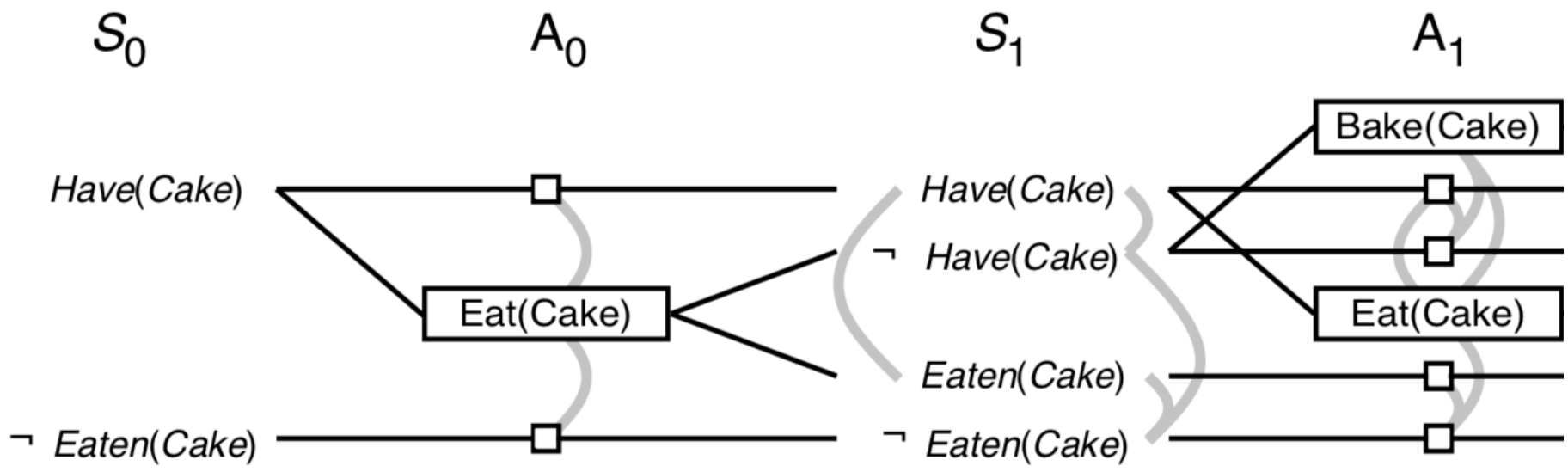
Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg\, Have(Cake)$
  EFFECT: $Have(Cake)$



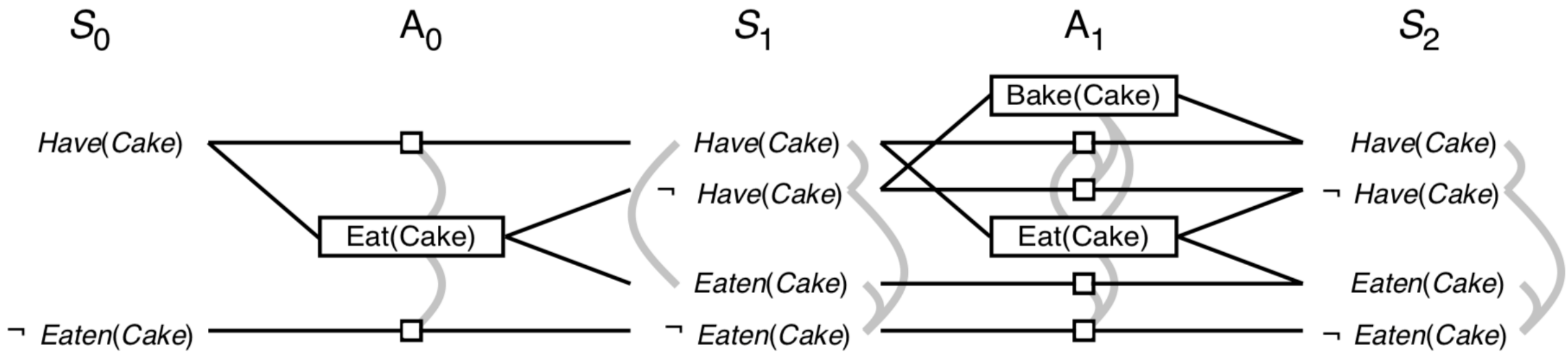
Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$



Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$



Planning Graph

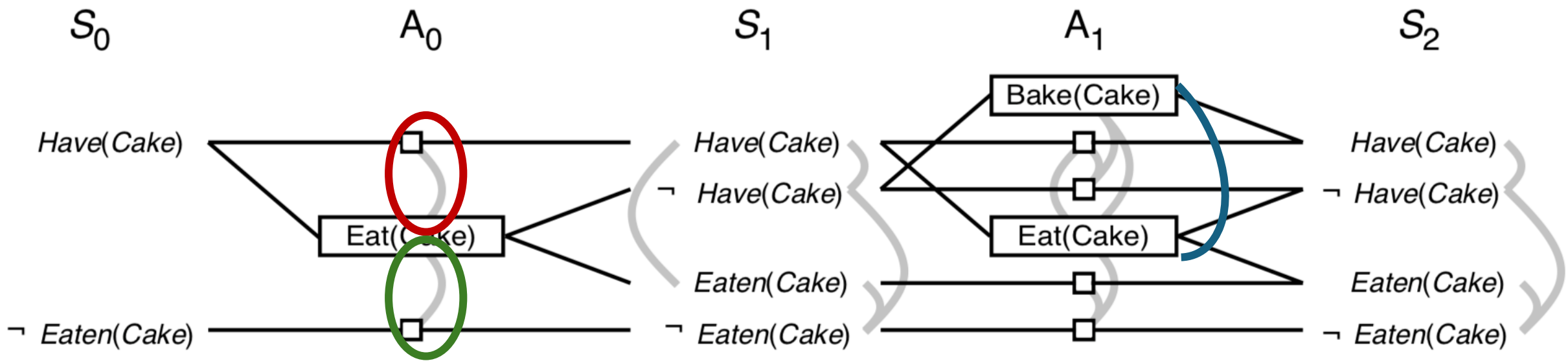
$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

Actions in a layer are *mutex* if:

Inconsistent Effects: one action's effect negates another's effect

Interference: one action negates another's precondition

Competing needs: one action requires negation of another's precondition



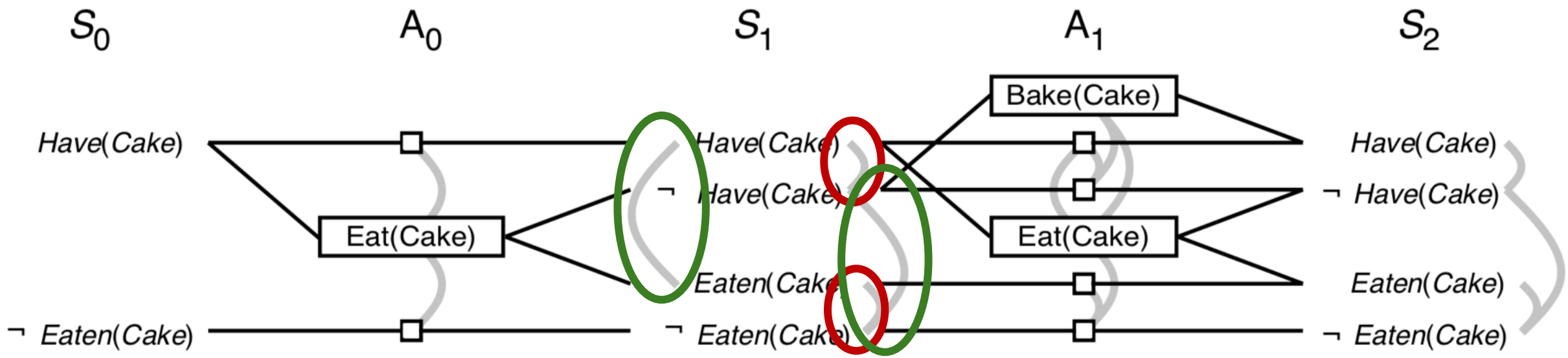
Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

Propositions in a layer are *mutex* if:

Negation: one negates the other

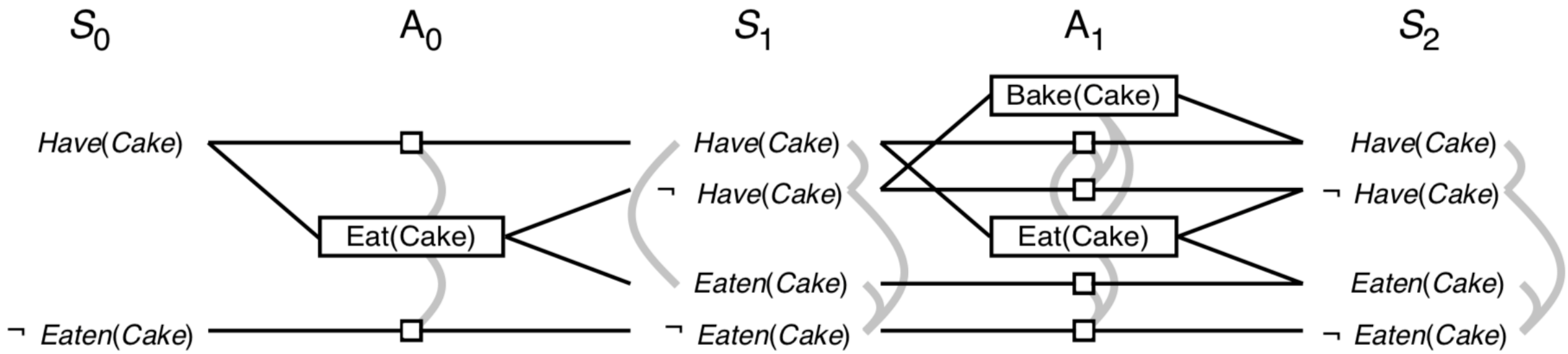
Inconsistent Support: all ways of achieving them are pairwise mutex



Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

- Mutexes are calculated for all pairs of actions/propositions at a level.

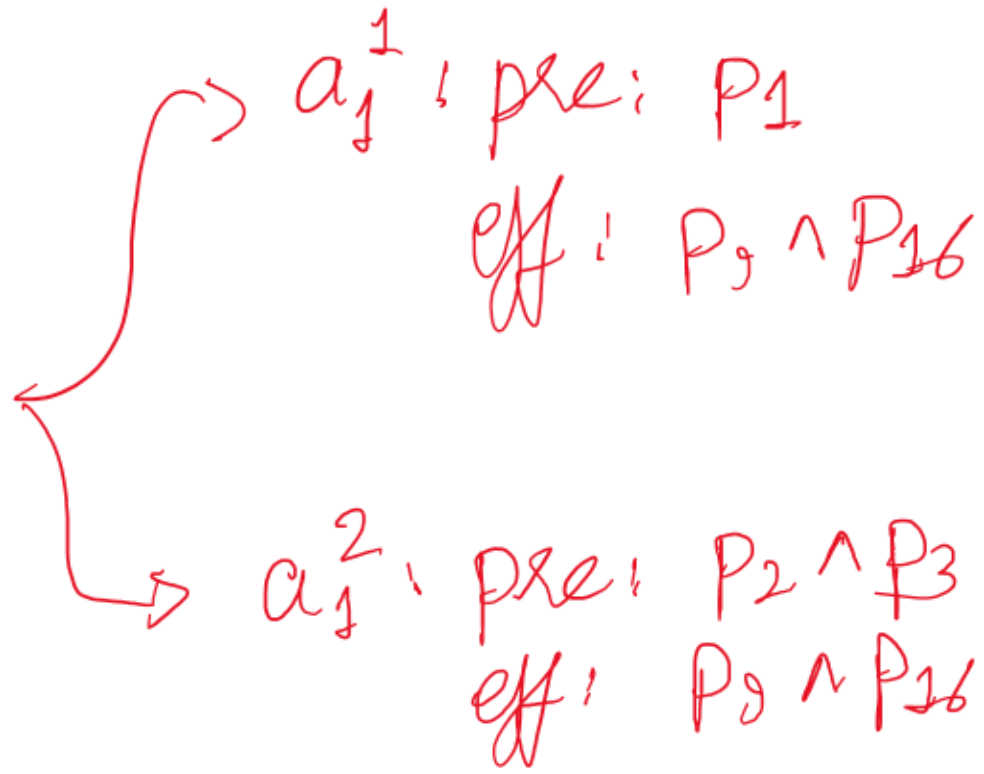


Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

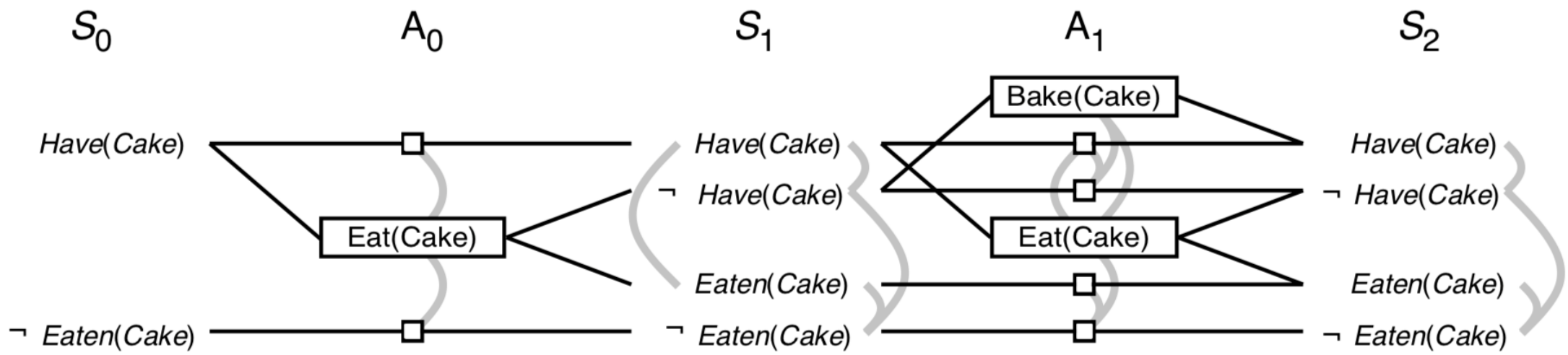
- If disjunction of propositions in an action's precondition?
  - Equivalent to 2 actions, each with a different part of the disjunction as its precondition.

a_1 : pre: $P_1 \vee (P_2 \wedge P_3)$
eff: $P_9 \wedge P_{16}$



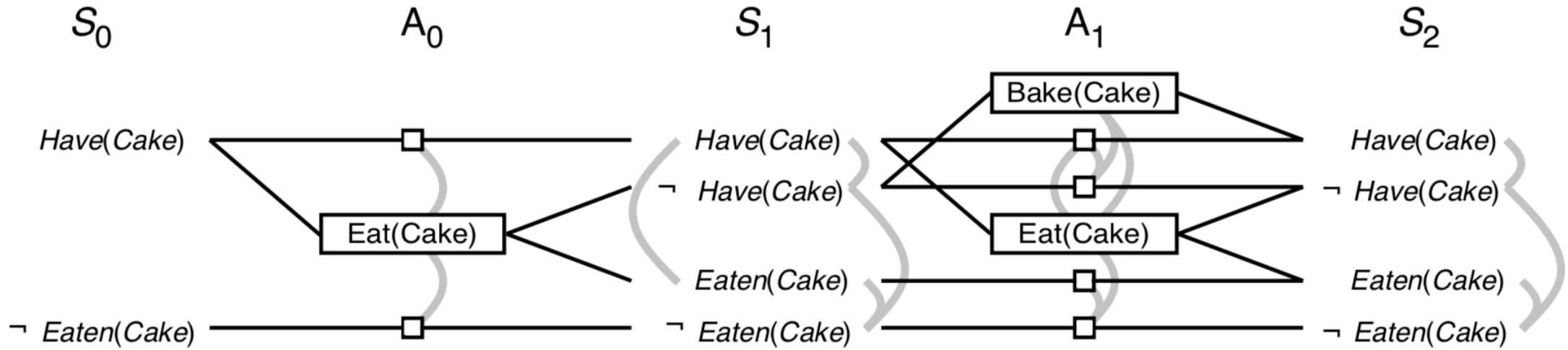
Extracting Heuristics From the Planning Graph

- Mutex relations can be used to try to extract plans that achieve the goal
- It is possible that a plan is not feasible within a given expansion of graphplan
  - Add another layer, try again
  - Until the graph levels off
  - If all the goal literals do not appear in the final level, goal is unreachable



Extracting Heuristics From the Planning Graph

- Expand the graph until you reach a fixed point
- If a literal doesn't occur in the last layer, it cannot be achieved
- Level-cost heuristic
  - Estimated cost of achieving a goal literal = level at which it first appears
  - Is this admissible?
- What about goals that include multiple literals?
  - Max-level heuristic
  - Level-sum heuristic
    - Is this optimistic (admissible) or pessimistic (non-admissible)?



Heuristic Search Planner (HSP)

- Consider only positive effects of actions (ignore negated literals in effects)
  - This creates a relaxed problem: it allows more actions to be applicable in a state than truly possible
  - Recall: To generate a heuristic, use cost of solution in relaxed problem…
  - But computing optimal solutions is too hard even for such relaxed problems
- Compute estimated cost of achieving an atom as

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s, \\ \min\limits_{op \in O(p)} [1 + g_s(Prec(op))] & \text{otherwise,} \end{cases}$$

Operators that add p

Estimated cost of achieving preconditions of op

Heuristic Search Planner (HSP)

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s, \\ \min\limits_{op \in O(p)} [1 + g_s(Prec(op))] & \text{otherwise,} \end{cases}$$

Ignore mutexes, persistence actions

For atoms not already achieved!