Planning Graph

“have cake and eat cake too” problem

Init(Have(Cake) $\wedge$ $\neg$Eaten(Cake))

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

Action(Eat(Cake)

PRECOND: Have(Cake)

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

Action(Bake(Cake)

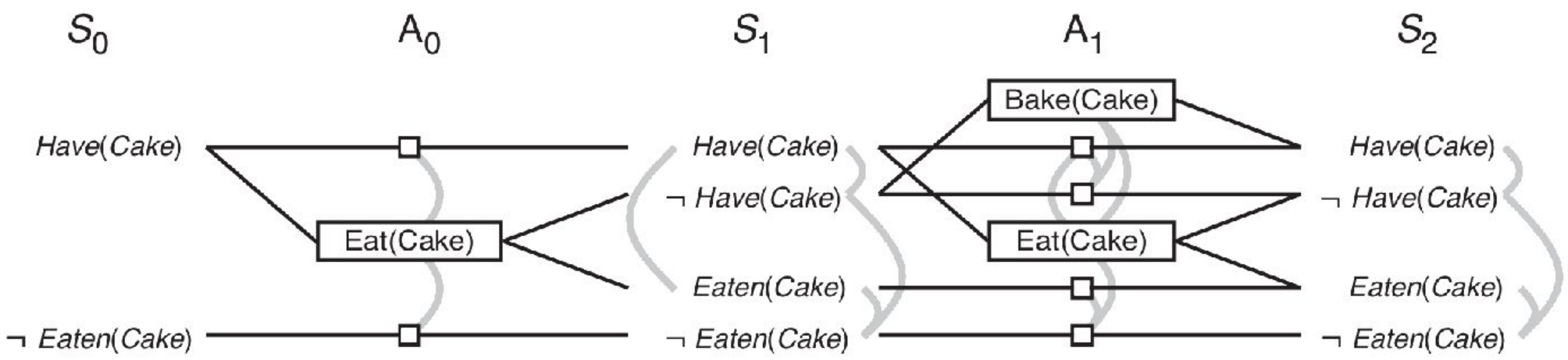
PRECOND: $\neg$Have(Cake)

EFFECT: Have(Cake)

Planning graph: Example

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

You would like to eat your cake and still have a cake. Fortunately, you can bake a new one.



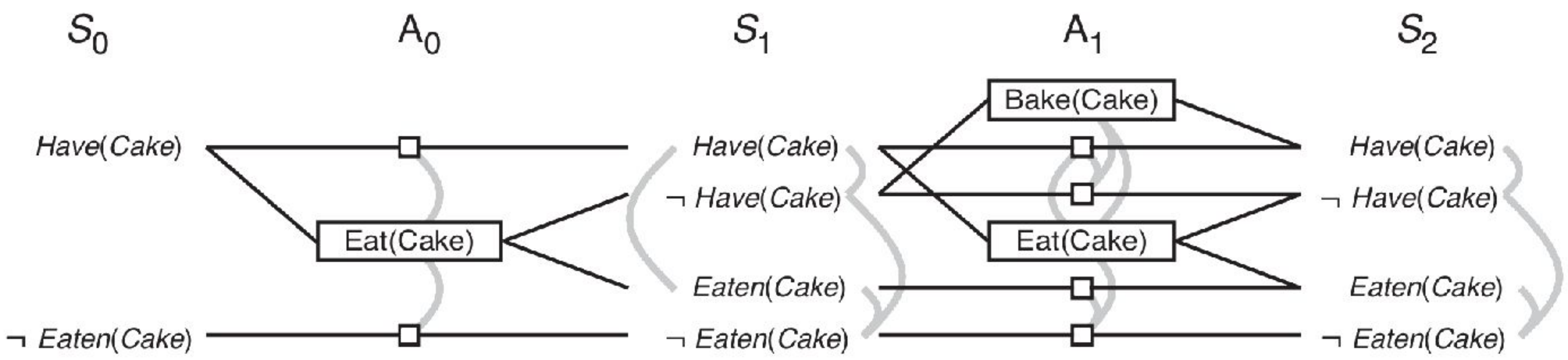
Planning graph: Example

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

You would like to eat your cake and still have a cake. Fortunately, you can bake a new one.

level S_0: contain each ground fluent that holds in the initial state
level A_0: contains each ground action applicable in S_0
...
level A_i: contains all ground actions with preconditions in S_{i-1}
level S_{i+1}: all the effects of all the actions in A_i

- each S_i may contain both P_j and $\neg P_j$



Planning Graph

- Convert the planning problem structure into planning graph called as GRAPHPLAN, in the increment nature.

Planning Graph

- Convert the planning problem structure into planning graph called as GRAPHPLAN, in the increment nature.
- It gives the relation between action and states, the precondition must be satisfy the action.

Planning Graph

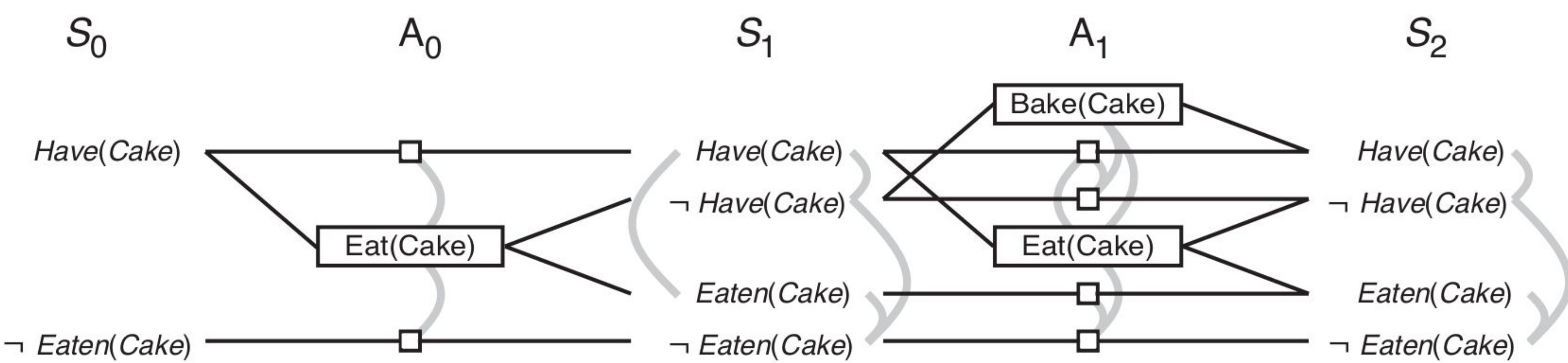
- Convert the planning problem structure into planning graph called as GRAPHPLAN, in the increment nature.
- It gives the relation between action and states, the precondition must be satisfy the action.
- Planning graph is directed, leveled graph with a sequence of layers (corresponds to time steps) of **propositions (states)** and **actions**.

Planning Graph

- Convert the planning problem structure into planning graph called as GRAPHPLAN, in the increment nature.
- It gives the relation between action and states, the precondition must be satisfy the action.
- Planning graph is directed, leveled graph with a sequence of layers (corresponds to time steps) of **propositions (states)** and **actions**.
- Planning graphs aims to solve prepositional planning problems.

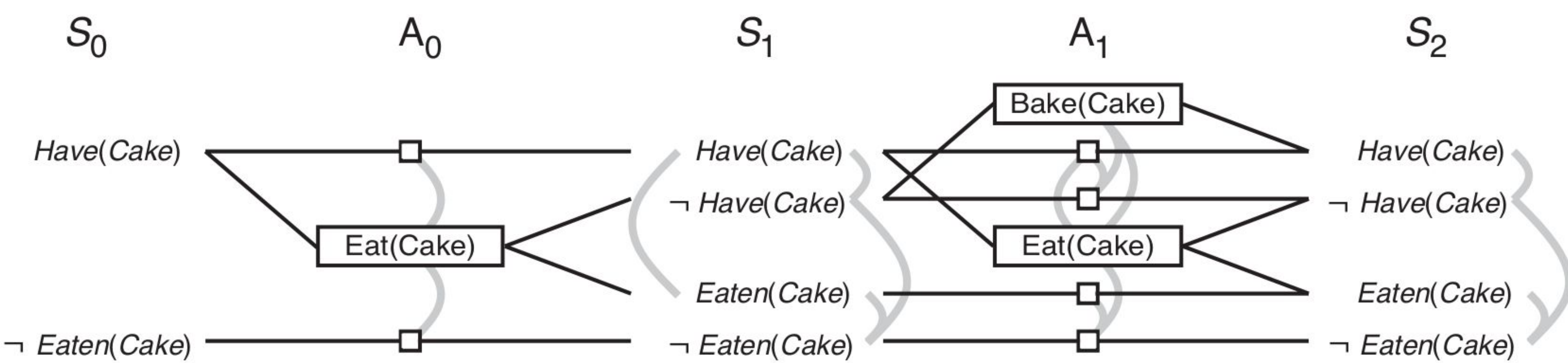
Planning graphs

- Each level consists of
  - Literals = all those that could be true at that time step, depending upon the actions executed at preceding time steps.
  - Actions = all those actions have their preconditions, that satisfied at that time step, depending on which of the literals actually hold.



Planning Graph

- 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



Planning Graph

The "have cake and eat cake too" problem.

S_0

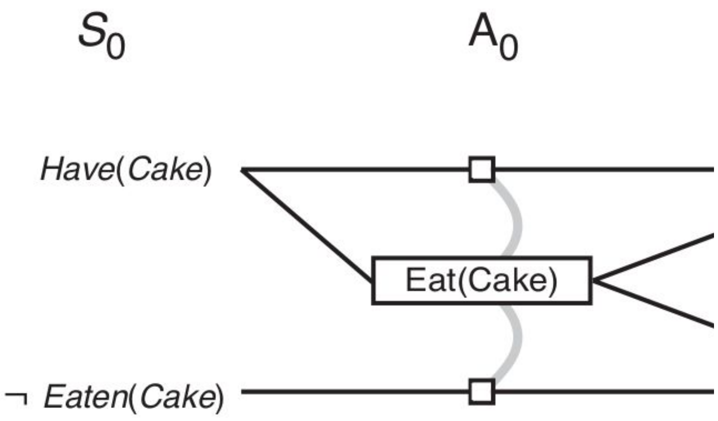
$Have(Cake)$

$\neg\ Eaten(Cake)$

- Level S_0 has all literals from initial state

Planning Graph

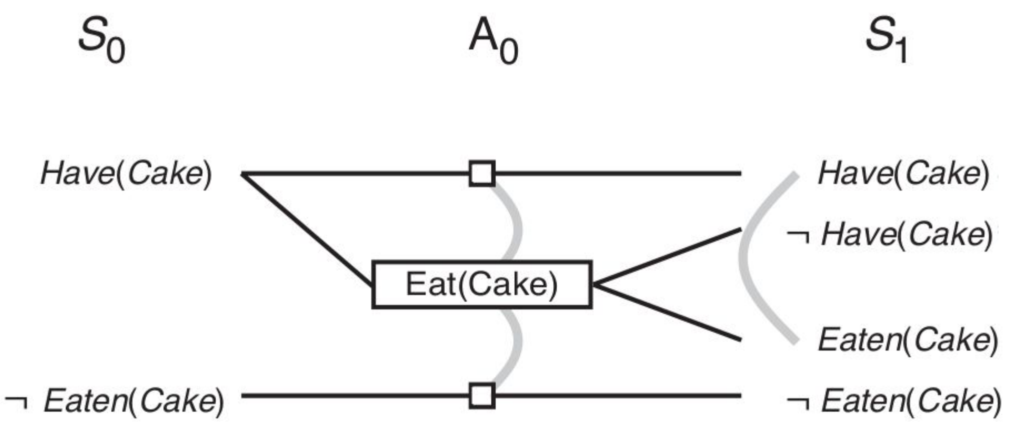
The "have cake and eat cake too" problem.



- Level A_0 has all actions whose preconditions are satisfied in S0
- Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects.

Planning Graph

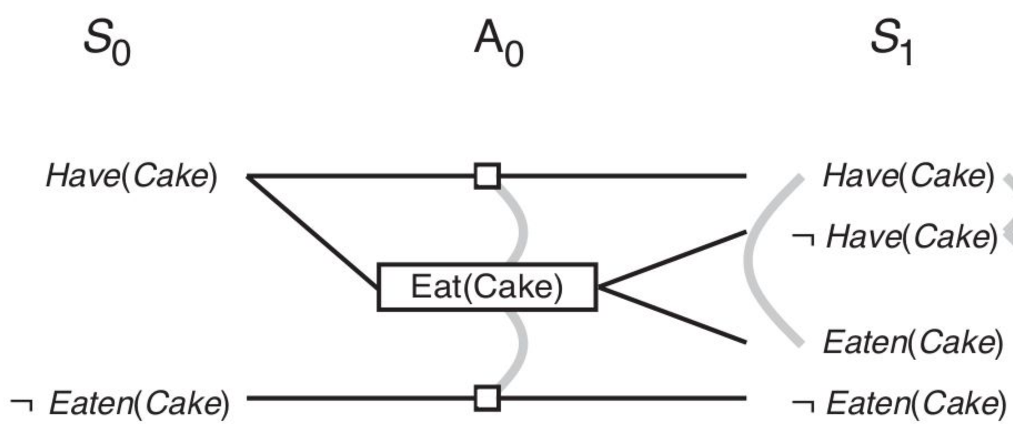
The "have cake and eat cake too" problem.



- Gray arcs connect propositions that are mutex (mutually exclusive) & actions that are mutex

Mutex Arcs

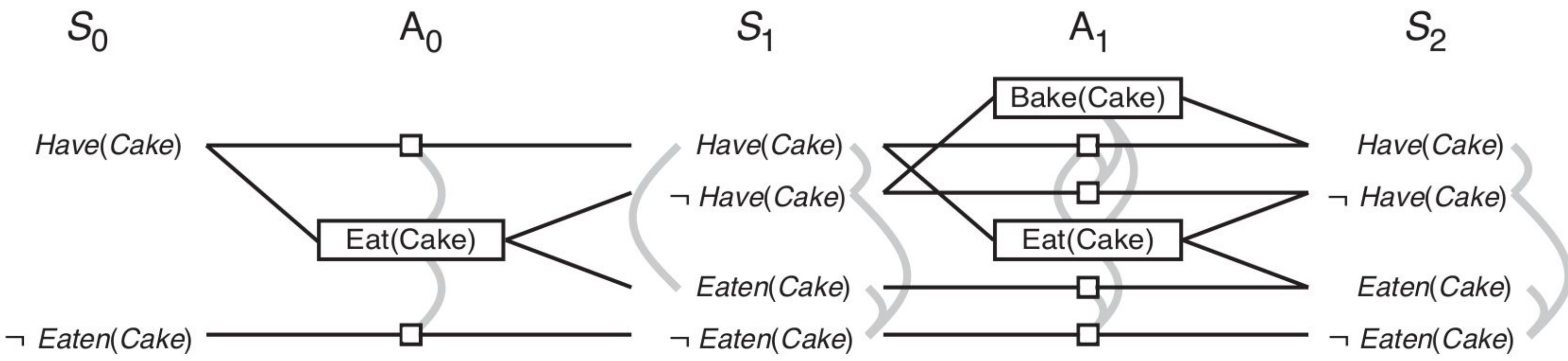
- Mutex arc between two actions indicates that it is ***impossible*** to perform the actions in parallel
- Mutex arc between two literals indicates that it is impossible to have these both true at this stage



Computing mutexes

Mutex actions

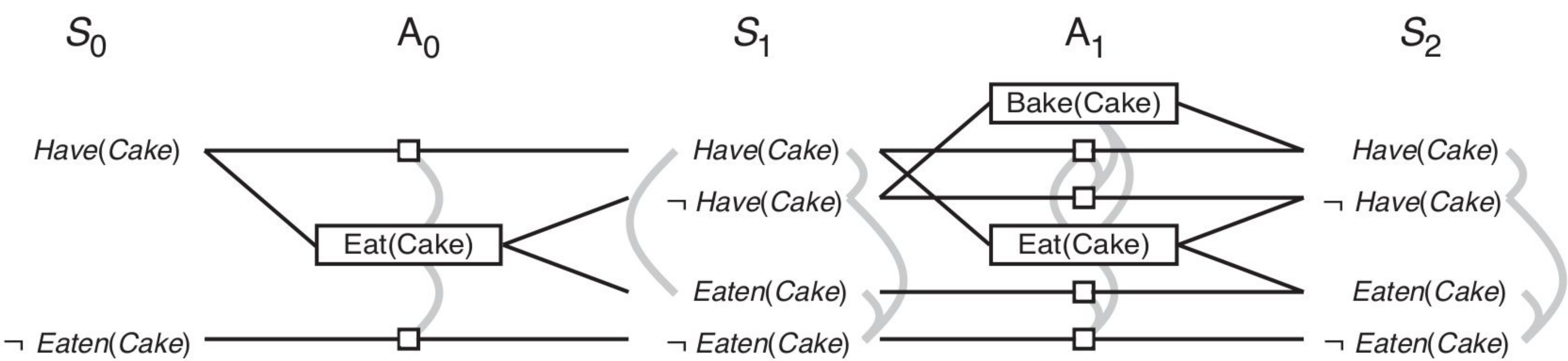
- Inconsistent effects: two actions that lead to inconsistent effects
- Interference: an effect of first action negates precondition of other action
- Competing needs: a precondition of first action is mutex with a precondition of second action



Computing mutexes

Mutex actions

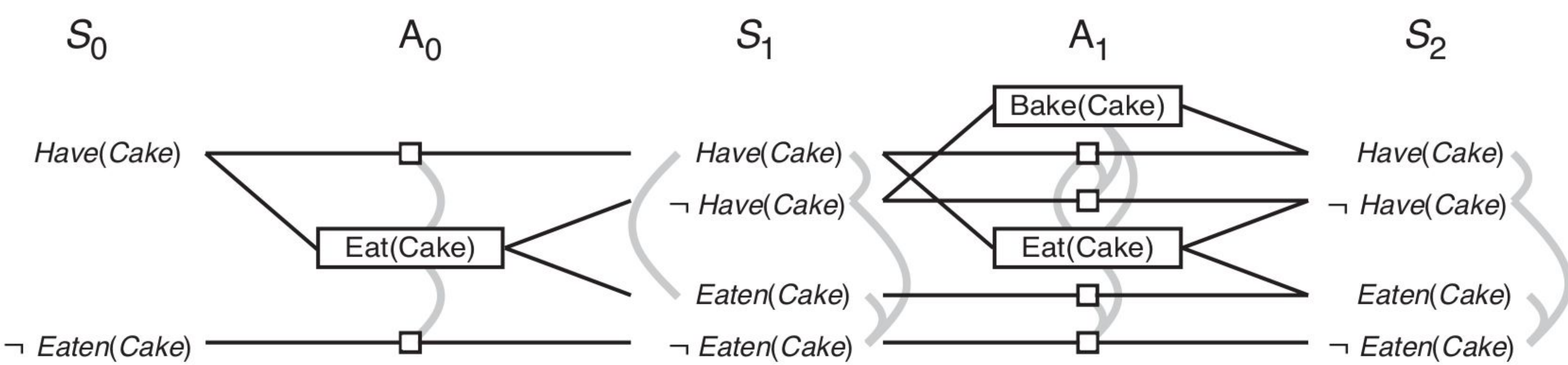
Inconsistent effects: an effect of one negates an effect of the other
ex: persistence of *Have*(*Cake*), *Eat*(*Cake*) have competing effects
ex: *Bake*(*Cake*), *Eat*(*Cake*) have competing effects
Interference: one deletes a precondition of the other
ex: *Eat*(*Cake*) interferes with the persistence of *Have*(*Cake*)
Competing needs: they have mutually exclusive preconditions
ex: *Bake*(*Cake*) and *Eat*(*Cake*)



Computing mutexes

Mutex literals

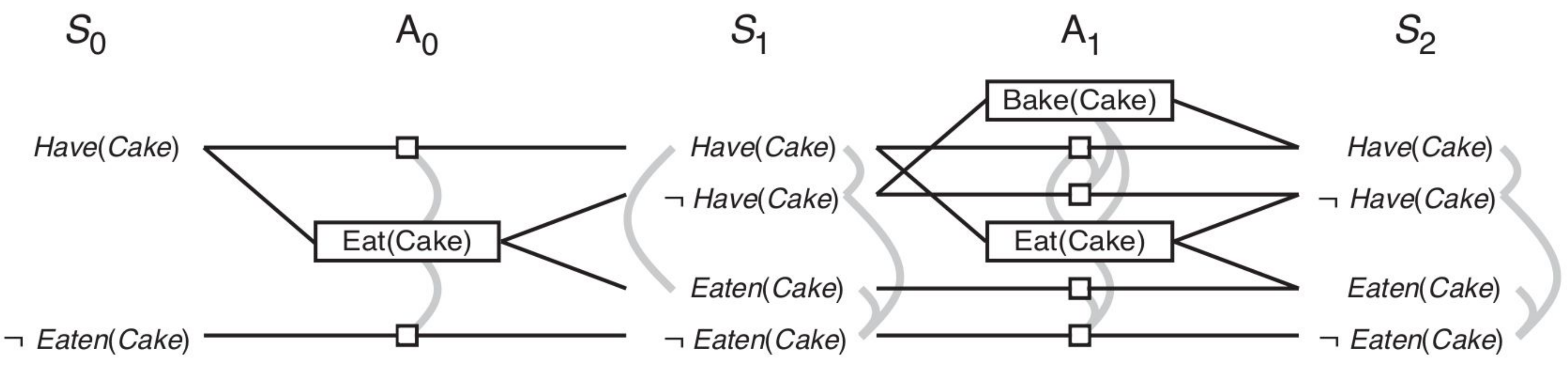
- One literal is negation of the other one
- Inconsistency support: each pair of actions achieving the two literals are mutually exclusive



Computing mutexes

Mutex literals

inconsistent support: one is the negation of the other
ex.: *Have*(*Cake*), ¬*Have*(*Cake*)
all ways of achieving them are pairwise mutex
ex.: (S_1): *Have*(*Cake*) in mutex with *Eaten*(*Cake*) because persist. of *Have*(*Cake*), *Eat*(*Cake*) are mutex



Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, \ldots$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, ...$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, ...$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, \ldots$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

Create state layer S_{i+1}:

1. for each action node a in A_i,
    - add to S_{i+1} the fluents in his Add list, linking them to a
    - add to S_{i+1} the negated fluents in his Del list, linking them to a

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, \ldots$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

Create state layer S_{i+1}:

1. for each action node a in A_i,
   - add to S_{i+1} the fluents in his Add list, linking them to a
   - add to S_{i+1} the negated fluents in his Del list, linking them to a
2. for every "no-op" action node a in A_i,
   - add the corresponding literal to S_{i+1}
   - link it to a

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, \ldots$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

Create state layer S_{i+1}:

1. for each action node a in A_i,
   - add to S_{i+1} the fluents in his Add list, linking them to a
   - add to S_{i+1} the negated fluents in his Del list, linking them to a
2. for every "no-op" action node a in A_i,
   - add the corresponding literal to S_{i+1}
   - link it to a
3. add mutexes between literal nodes in S_{i+1}

Building of the planning graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, ...$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

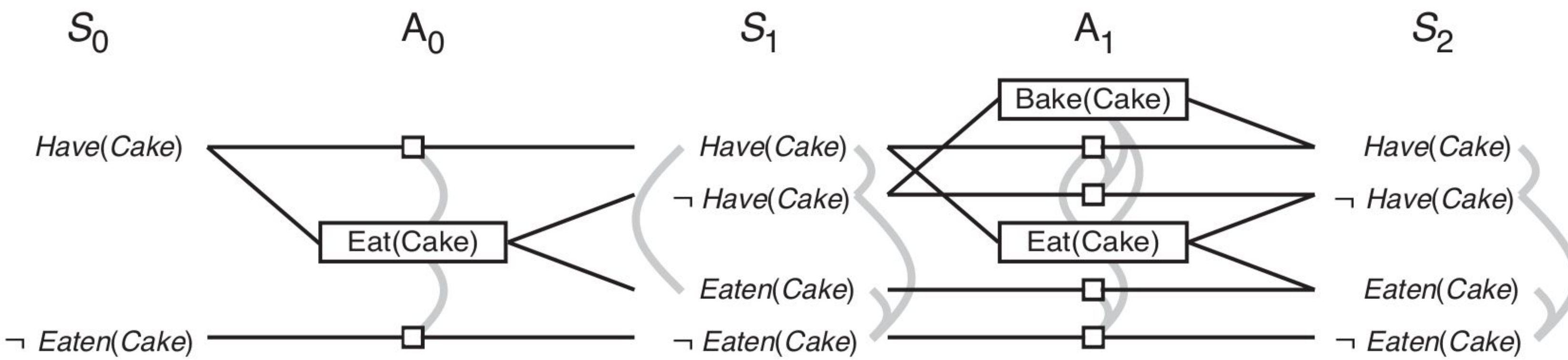
Create state layer S_{i+1}:

1. for each action node a in A_i,
   - add to S_{i+1} the fluents in his Add list, linking them to a
   - add to S_{i+1} the negated fluents in his Del list, linking them to a
2. for every "no-op" action node a in A_i,
   - add the corresponding literal to S_{i+1}
   - link it to a
3. add mutexes between literal nodes in S_{i+1}

Until $S_{i+1} = S_i$ (aka “graph leveled off”) or bound reached (if any)

Planning Graph

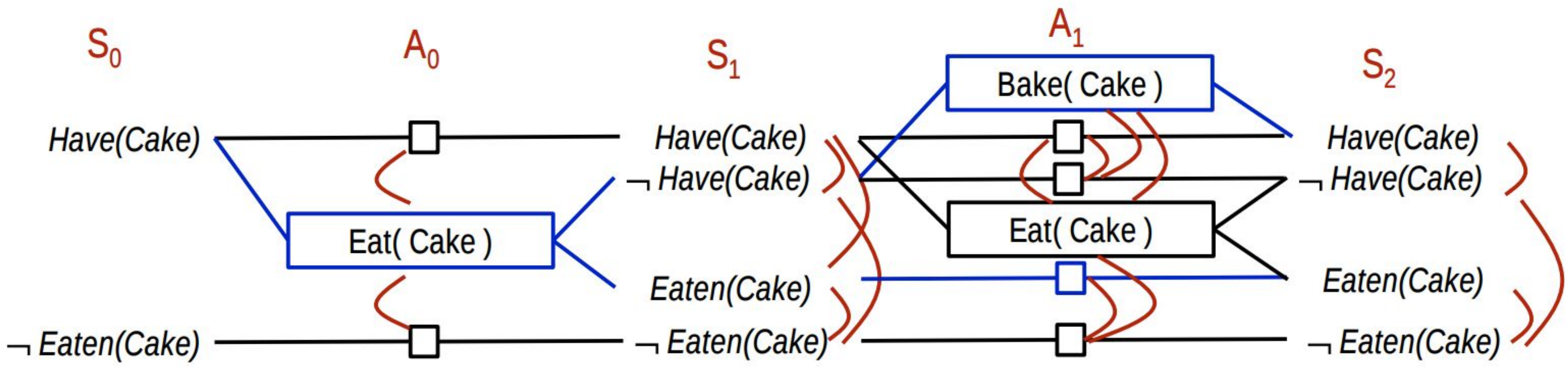
The “have cake and eat cake too” problem.



- Planning graph up to level S2
- Stop when the set of literals and mutex links does has not changed

Planning Graph

The plan is shown in blue below



- AIM is to extract a solution directly from the planning graph, using a specialized algorithm such as GRAPHPLAN
- A planning graph consists of a sequence of levels that correspond to time steps in the plan, where level 0 is the initial state.