PLANNING GRAPHS

Planning Graphs

- Planning graphs are an efficient way to create a representation of a planning problem that can be used to
  - Achieve better heuristic estimates
  - Directly construct plans
- Planning graphs only work for propositional problems.

Planning Graphs

- Planning graphs consists of a seq of levels that correspond to time steps in the plan.
  - Level 0 is the initial state.
  - Each level consists of a set of literals and a set of actions that represent what *might be* possible at that step in the plan
  - *Might be* is the key to efficiency
  - Records only a restricted subset of possible negative interactions among actions.

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 that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.

PG Example

Init(Have(Cake))

Goal(Have(Cake) ∧ Eaten(Cake))

Action(Eat(Cake),

PRECOND: Have(Cake)

EFFECT: ¬Have(Cake) ∧ Eaten(Cake))

Action(Bake(Cake),

PRECOND: ¬ Have(Cake)

EFFECT: Have(Cake))

PG Example

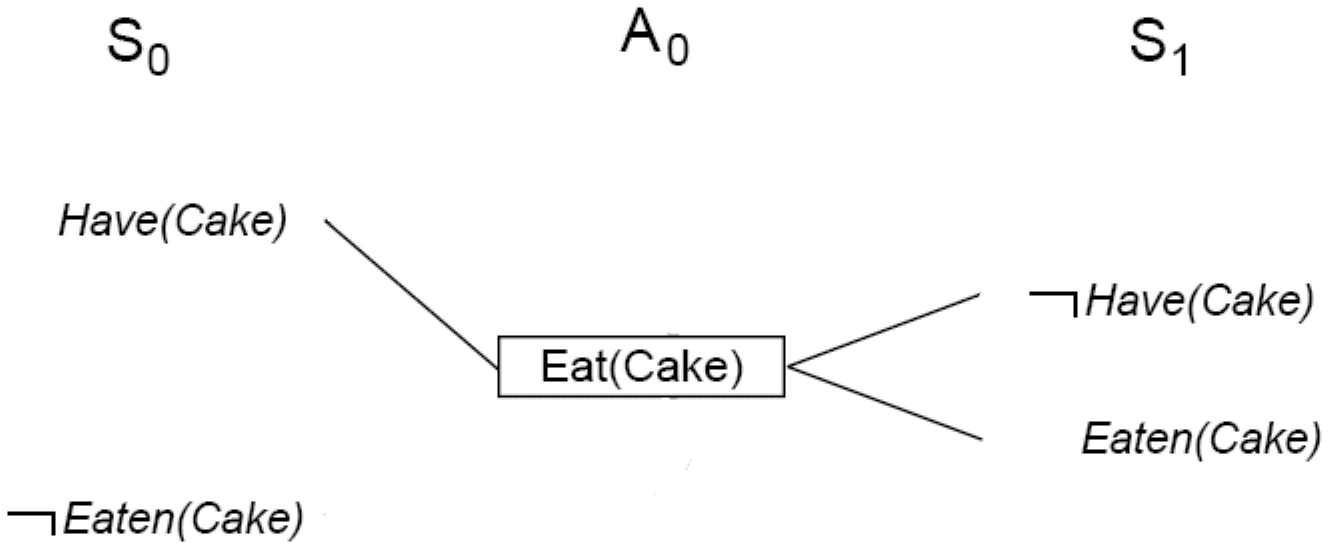
S_0 A_0 S_1

Have(Cake)

¬*Eaten(Cake)*

Create level 0 from initial problem state.

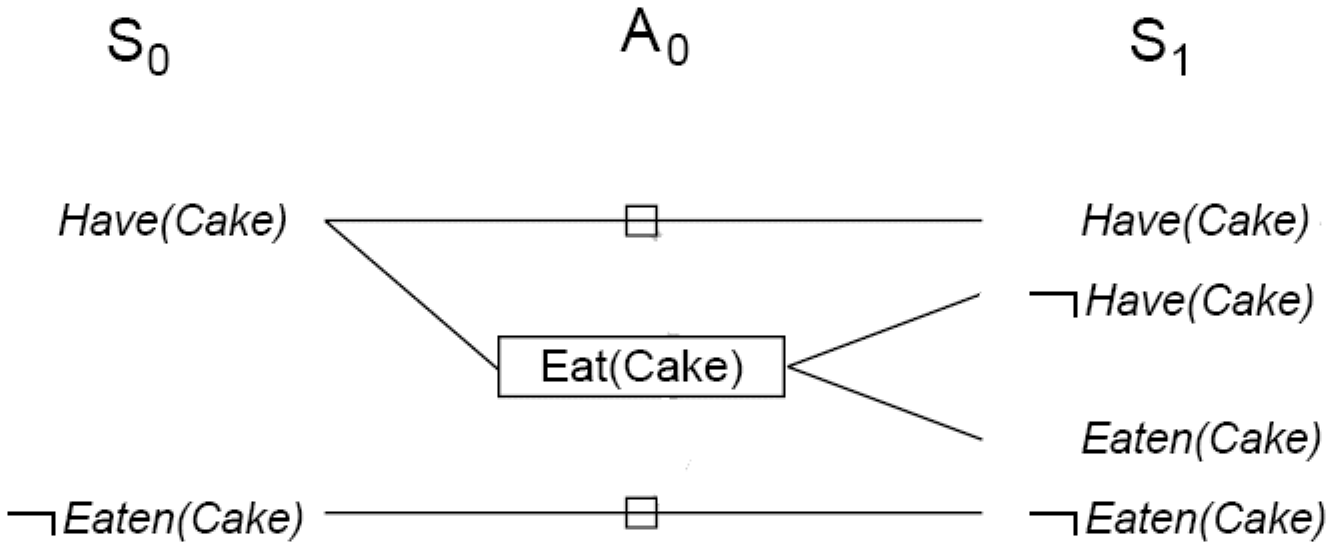
PG Example



Add all applicable actions.

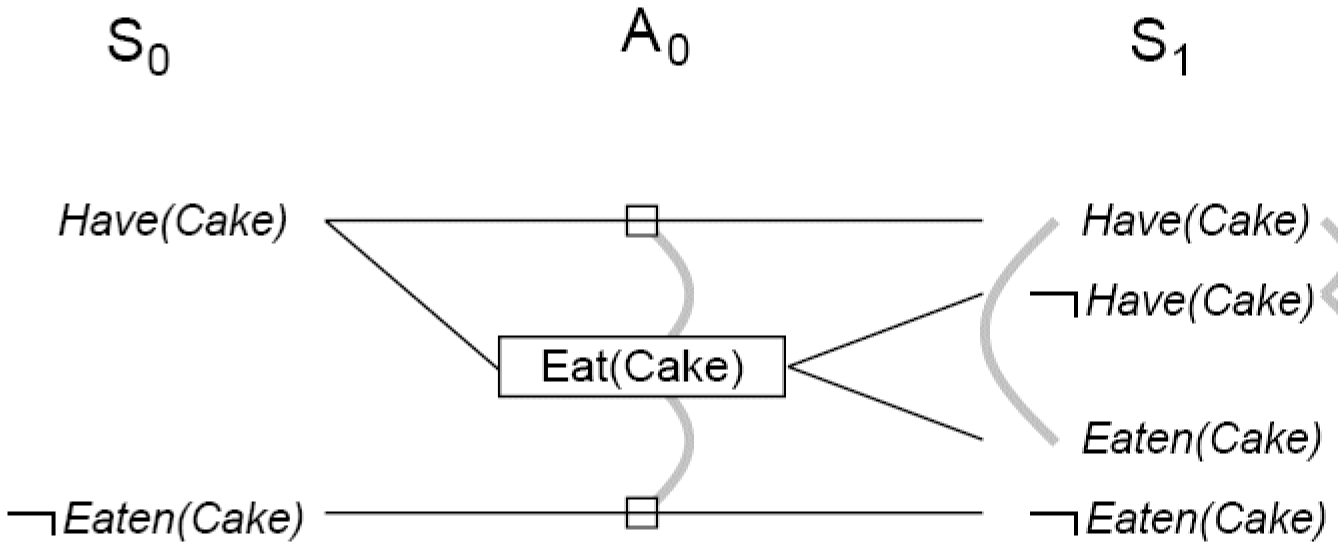
Add all effects to the next state.

PG Example



Add *persistence actions* (inaction = no-ops) to map all literals in state S_i to state S_{i+1}.

PG Example

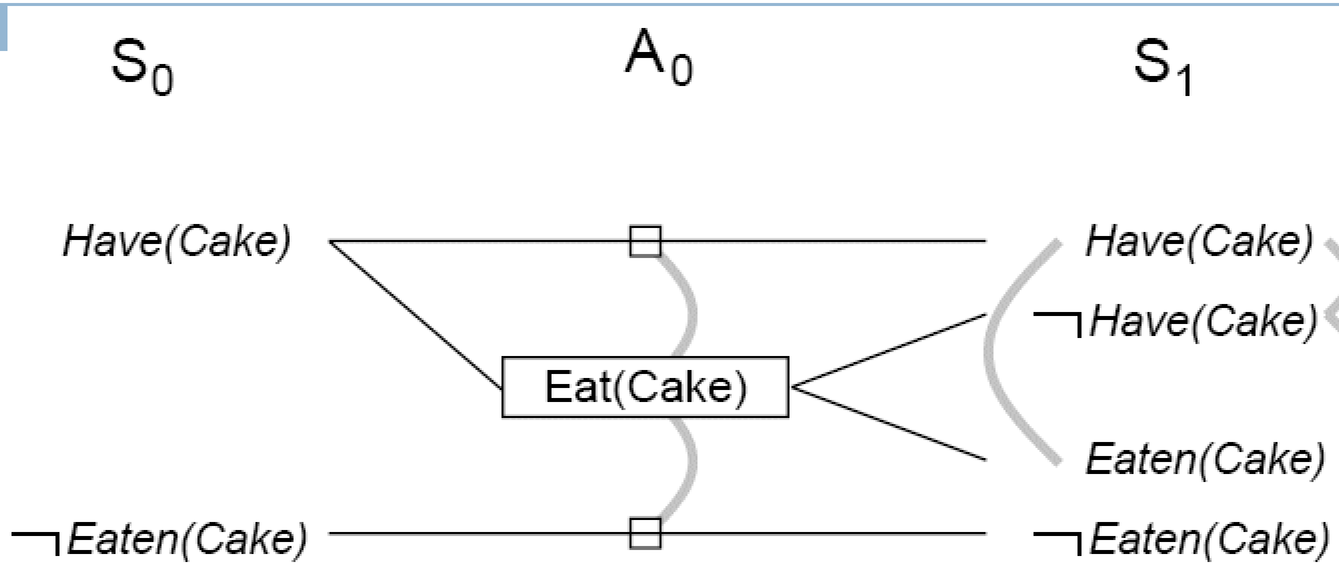


Identify *mutual exclusions* between actions and literals based on potential conflicts.

Mutual exclusion

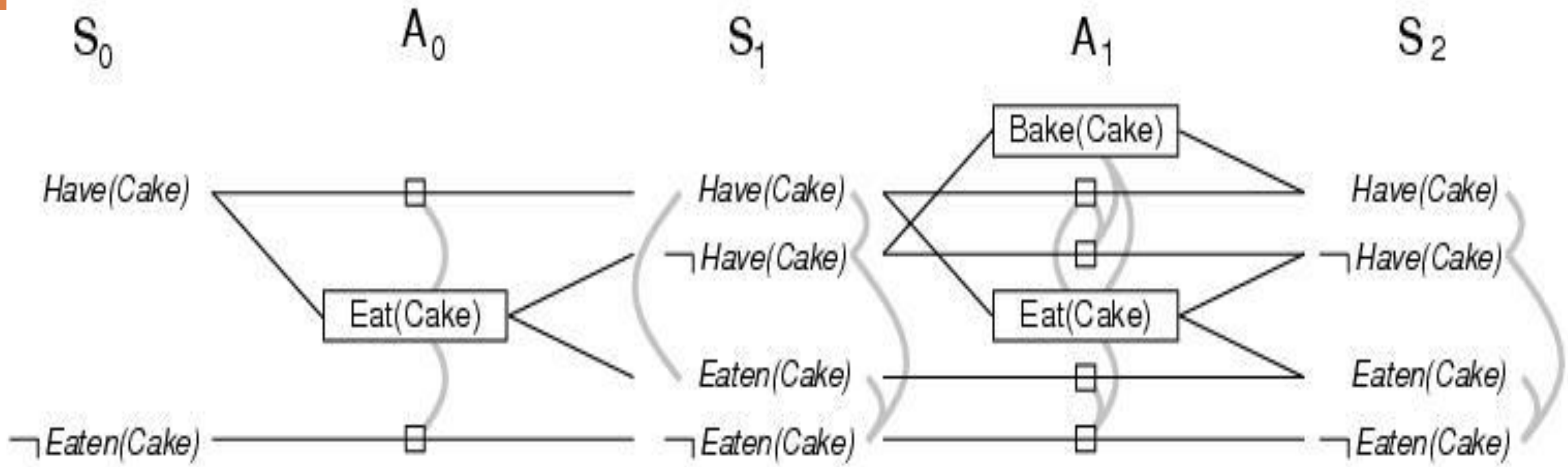
- A mutex relation holds between **two actions** when:
  - *Inconsistent effects*: one action negates the effect of another.
  - *Interference*: one of the effects of one action is the negation of a precondition of the other.
  - *Competing needs*: one of the preconditions of one action is mutually exclusive with the precondition of the other.
- 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).

Cake example



- Level S_1 contains all literals that could result from picking any subset of actions in A_0
  - Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  - S1 defines multiple states and the mutex links are the constraints that define this set of states.

Cake example



- Repeat process until graph levels off:
  - two consecutive levels are identical, or
  - contain the same amount of literals (explanation follows later)

The GRAPHPLAN Algorithm

- Extract a solution directly from the PG

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

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

    goals ← GOALS[*problem*]

    loop do

        if *goals* all non-mutex in last level of graph **then do**

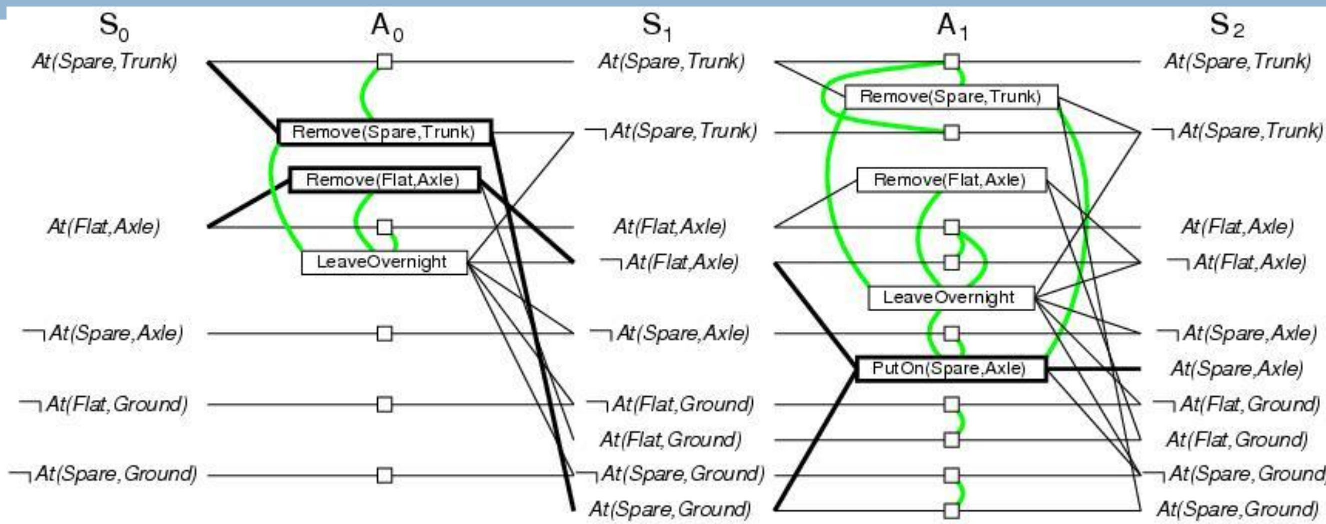
            solution ← EXTRACT-SOLUTION(*graph*, *goals*, LENGTH(*graph*))

            if *solution* ≠ failure **then return** *solution*

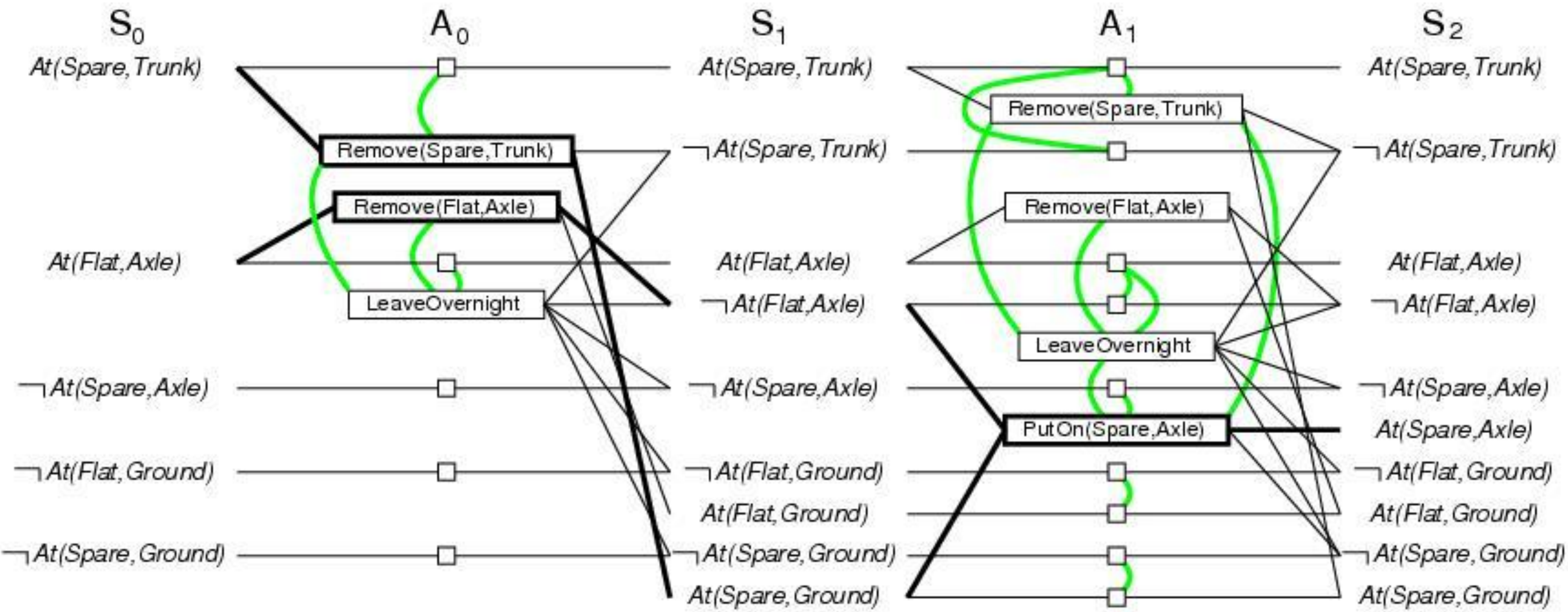
            else if NO-SOLUTION-POSSIBLE(*graph*) **then return** failure

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

GRAPHPLAN example

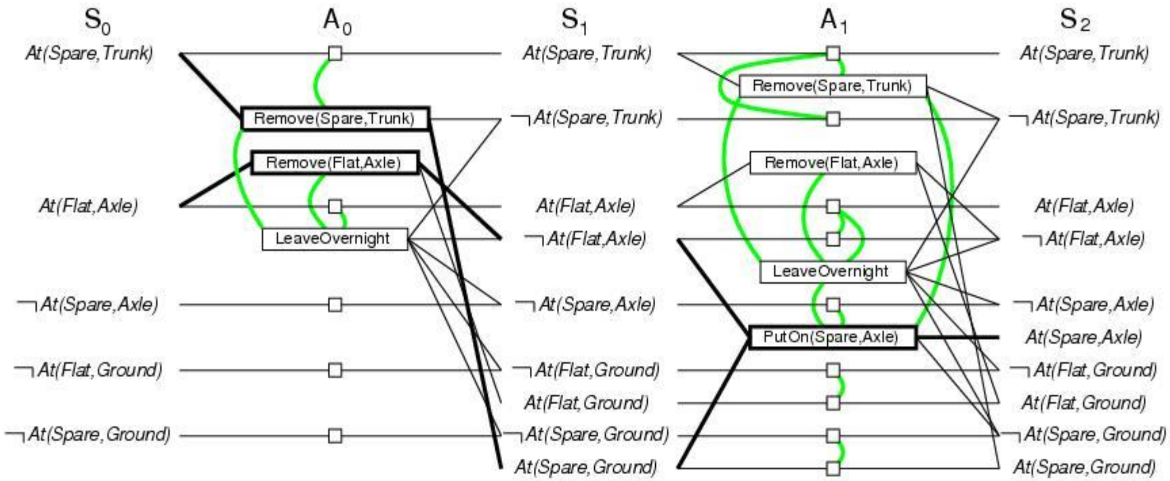


- Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si



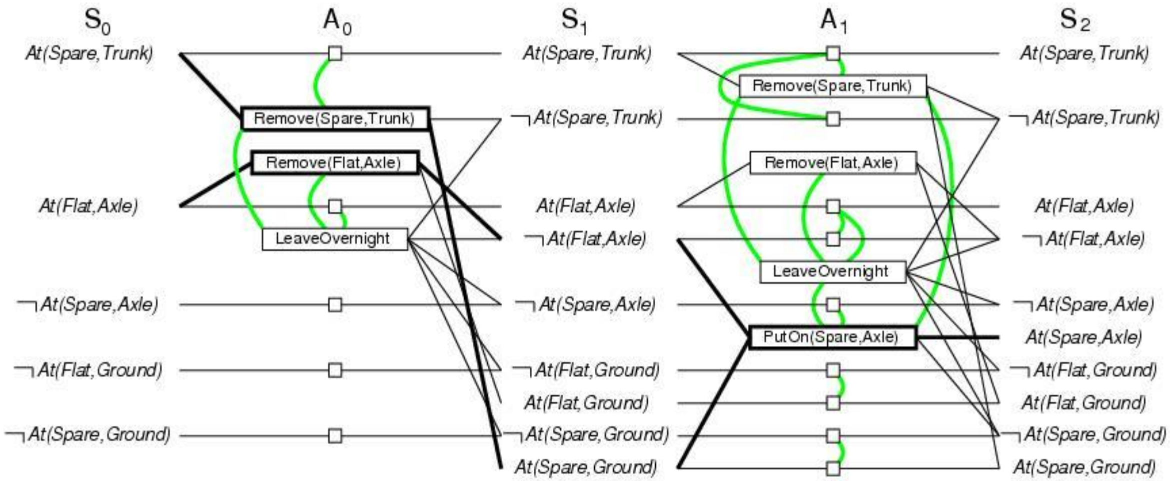
S0
A0
S1
A1
S2
At(Spare,Trunk)
At(Flat,Axle)
¬At(Spare,Axle)
¬At(Flat,Ground)
¬At(Spare,Ground)
Remove(Spare,Trunk)
Remove(Flat,Axle)
LeaveOvernight
¬At(Spare,Trunk)
¬At(Flat,Axle)
At(Flat,Ground)
At(Spare,Ground)
PutOn(Spare,Axle)
At(Spare,Axle)

GRAPHPLAN example



- EXPAND-GRAPH also looks for mutex relations
  - Inconsistent effects
    - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and **not** At(Spare, Ground)
  - Interference
    - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and **not** At(Flat,Axle) as EFFECT
  - Competing needs
    - E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and **not** At(Flat, Axle)
  - Inconsistent support
    - E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

GRAPHPLAN example



- In S2, the goal literals exist and are not mutex with any other
  - Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
  - Initial state = last level of PG and goal goals of planning problem
  - Actions = select any set of non-conflicting actions that cover the goals in the state
  - Goal = reach level S0 such that all goals are satisfied
  - Cost = 1 for each action.

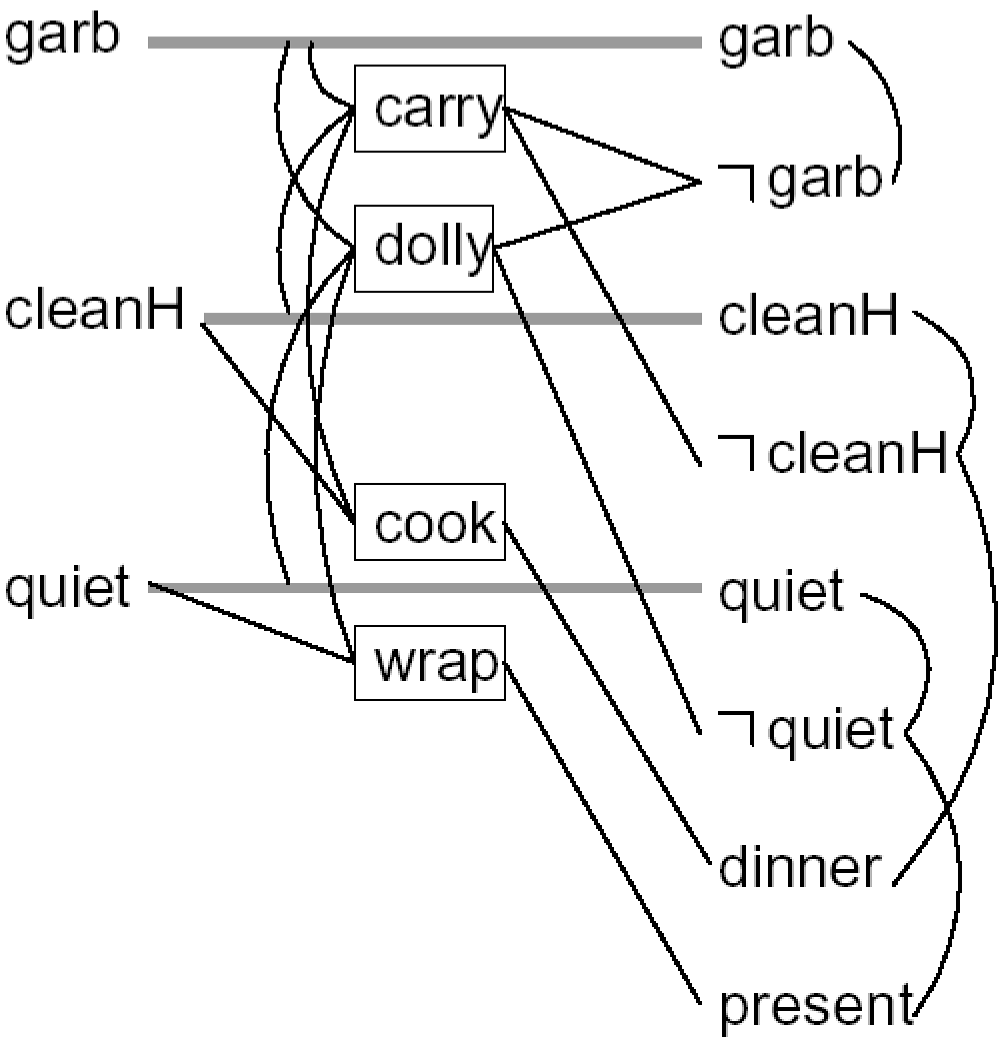
GRAPHPLAN Termination

- Termination? YES
- PG are monotonically increasing or decreasing:
  - Literals increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotonically
- Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off

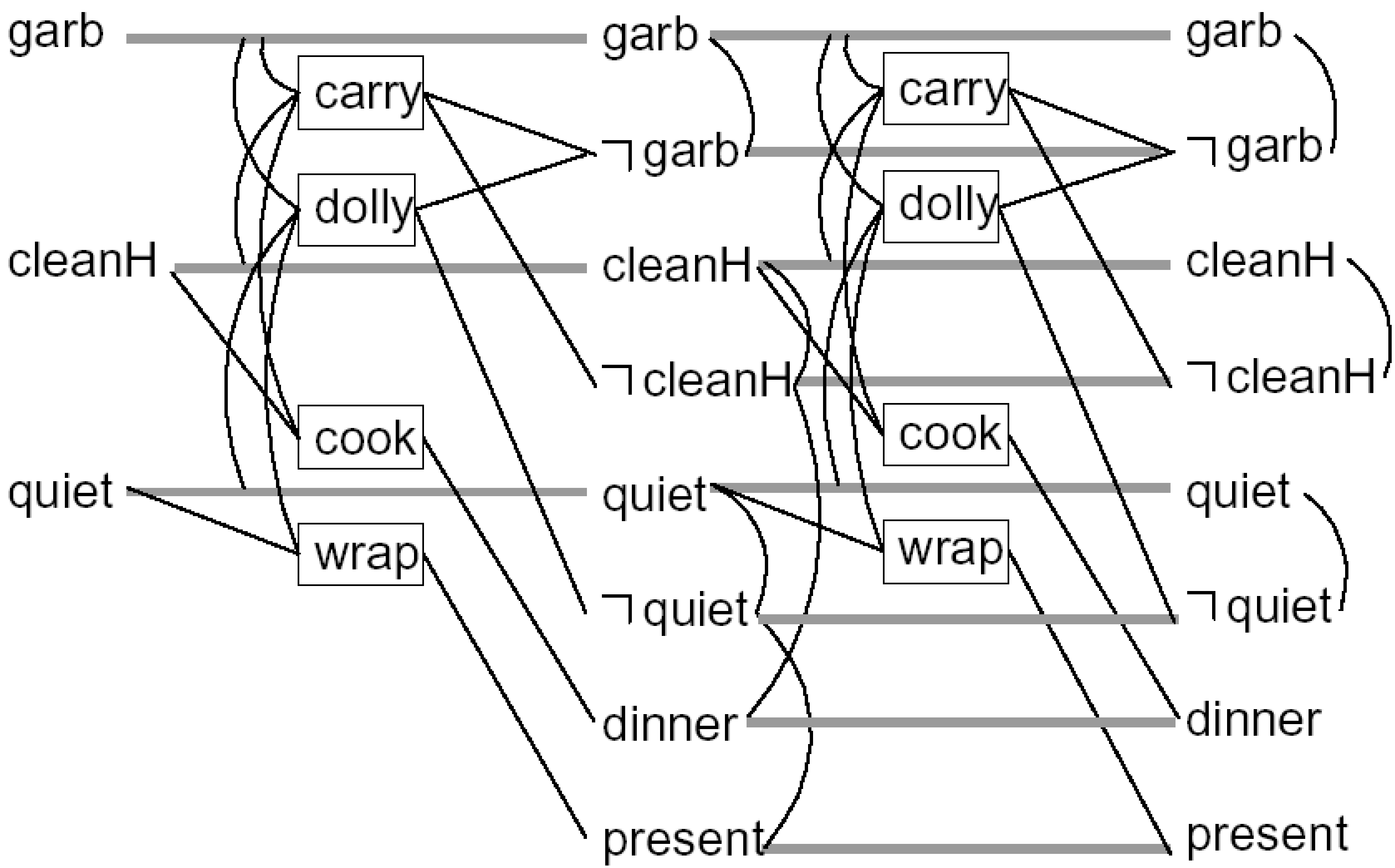
Dinner Date example

- Initial Conditions: (and (garbage) (cleanHands) (quiet))
- Goal: (and (dinner) (present) (not (garbage))
- Actions:
  - Cook :precondition (cleanHands)
    :effect (dinner)
  - Wrap :precondition (quiet)
    :effect (present)
  - Carry :precondition
    :effect (and (not (garbage)) (not (cleanHands))
  - Dolly :precondition
    :effect (and (not (garbage)) (not (quiet)))

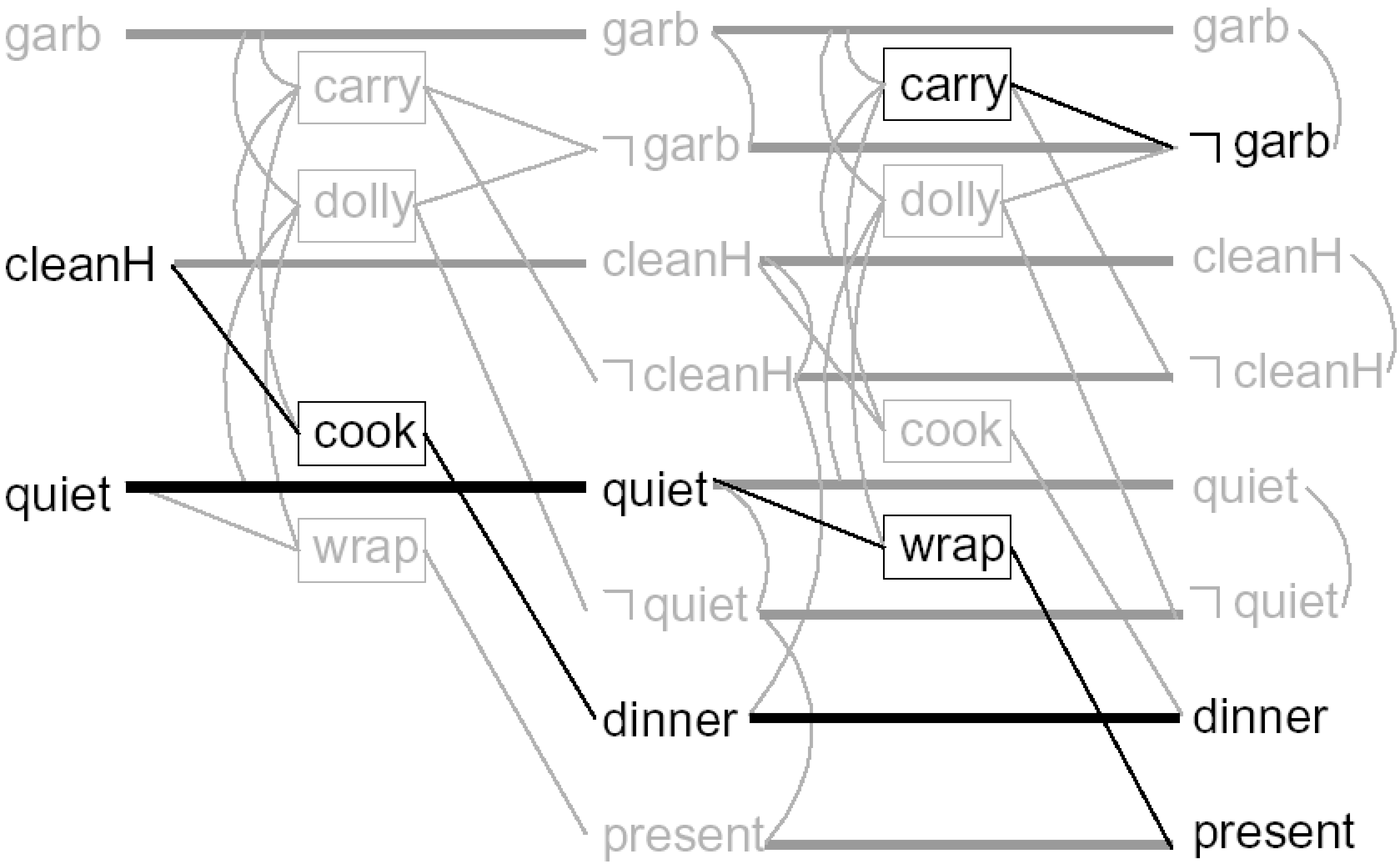
Dinner Date example



Dinner Date example



Dinner Date example



Rocket domain

```
(define (operator move)
    :parameters ((rocket ?r) (place ?from) (place ?to))
    :precondition (:and (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
    :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r))))

(define (operator unload)
    :parameters ((rocket ?r) (place ?p) (cargo ?c))
    :precondition (:and (at ?r ?p) (in ?c ?r))
    :effect (:and (:not (in ?c ?r)) (at ?c ?p)))

(define (operator load)
    :parameters ((rocket ?r) (place ?p) (cargo ?c))
    :precondition (:and (at ?r ?p) (at ?c ?p))
    :effect (:and (:not (at ?c ?p)) (in ?c ?r)))
```

Planning Graph Example
Rocket problem