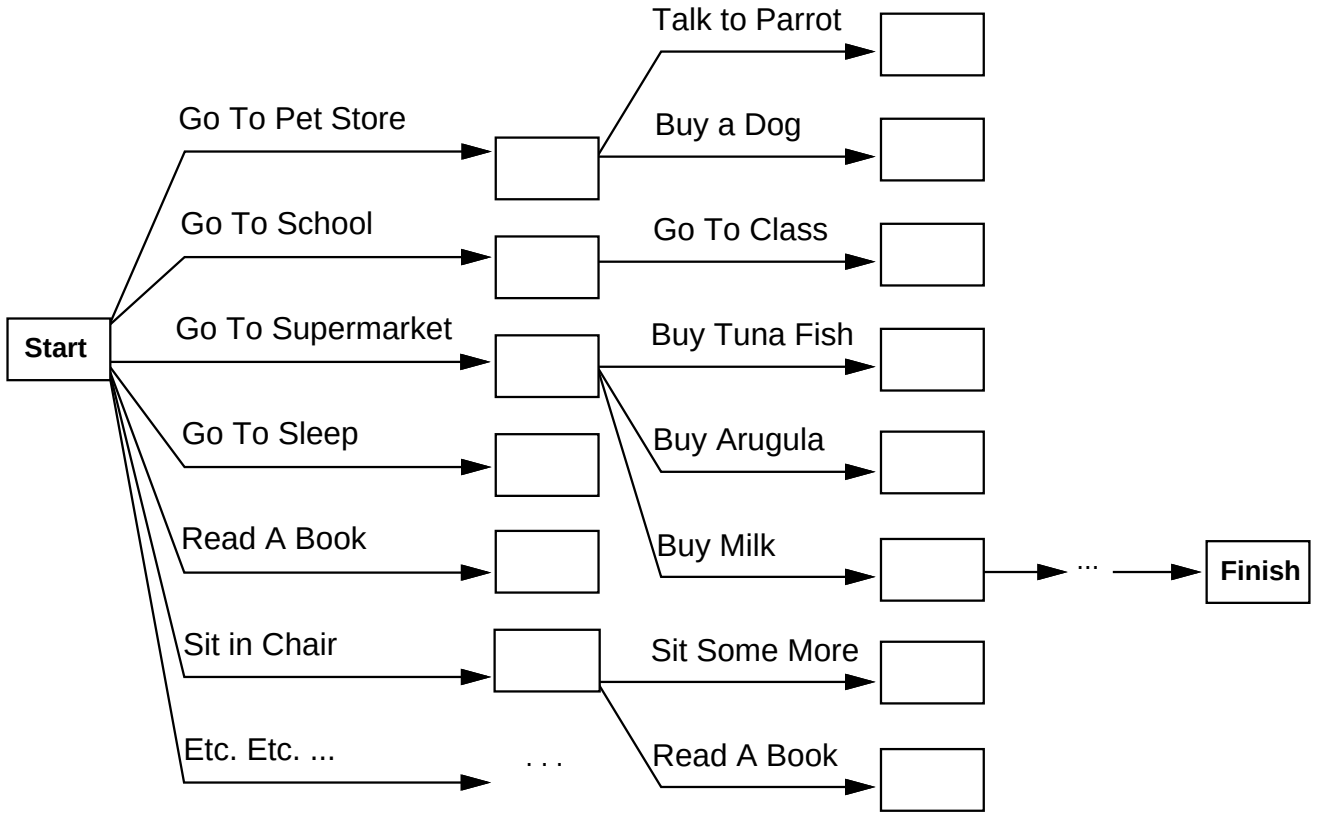
Outline of Planning

- ♢ Search vs. planning
- ♢ STRIPS operators
- ♢ Partial-order planning

Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*
Standard search algorithms seem to fail miserably:



After-the-fact heuristic/goal test inadequate

Search vs. planning contd.

Planning systems do the following:

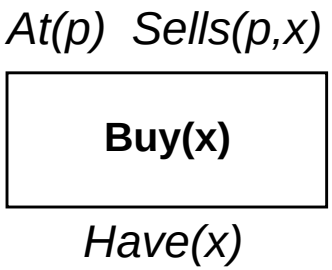
1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

	Search	**Planning**
States	Lisp data structures	Logical sentences
Actions	Lisp code	Preconditions/outcomes
Goal	Lisp code	Logical sentence (conjunction)
Plan	Sequence from S_0	Constraints on actions

STRIPS operators

Tidily arranged actions descriptions, restricted language

ACTION: $Buy(x)$
PRECONDITION: $At(p), Sells(p, x)$
EFFECT: $Have(x)$

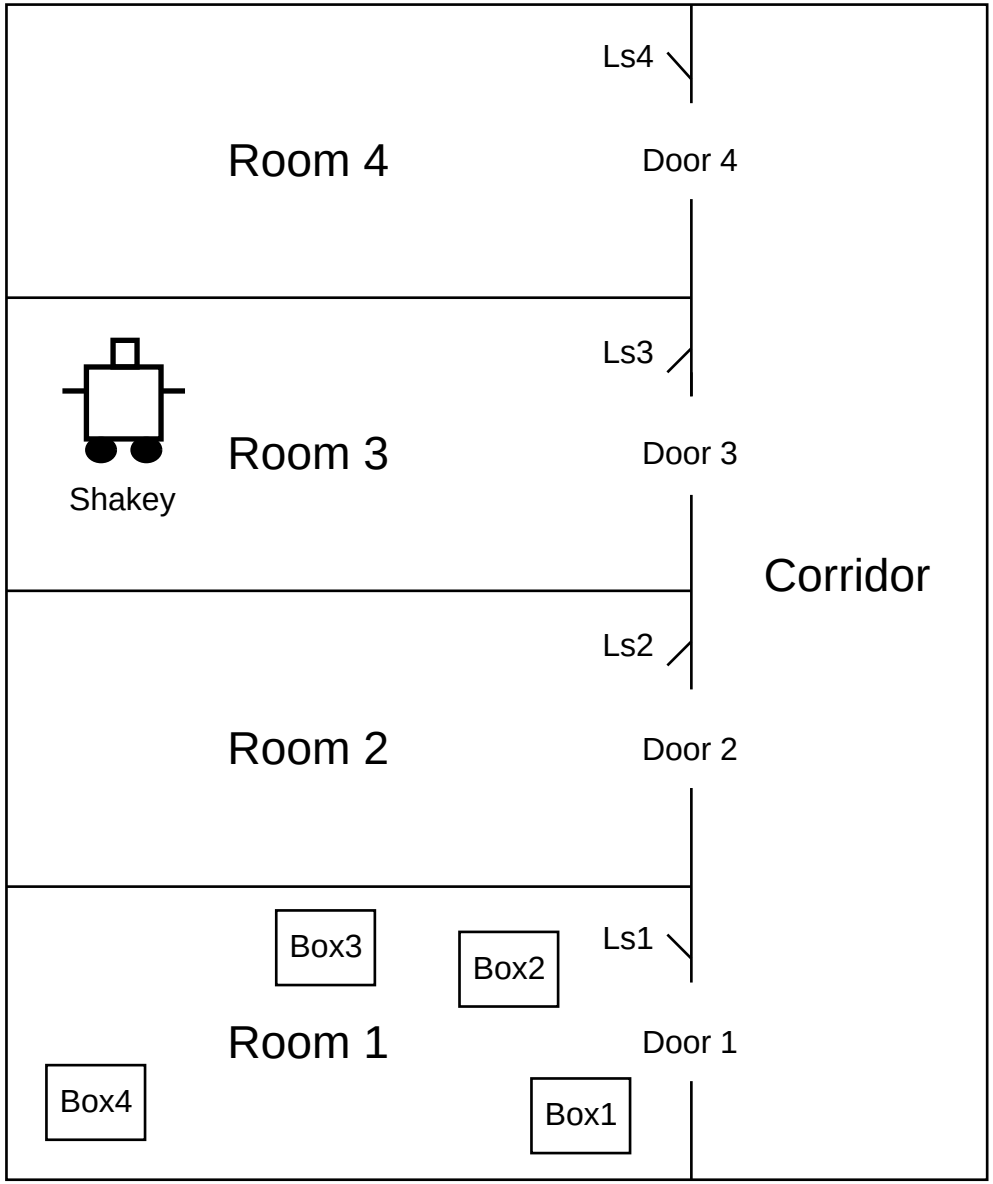


[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm
Precondition: conjunction of positive literals
Effect: conjunction of literals

A complete set of STRIPS operators can be translated into a set of successor-state axioms

Shakey Example



Shakey Example, con't.

ACTION: Go(x,y):
PRECOND:
EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):

PRECOND:

EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):

PRECOND: At(Shakey,x) $\wedge$ Pushable(b)

EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):

PRECOND: At(Shakey,x) $\wedge$ Pushable(b)

EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

Shakey Example, con't.

ACTION: Go(x,y):
PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)
EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):
PRECOND: At(Shakey,x) $\wedge$ Pushable(b)
EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):
PRECOND:
EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):

PRECOND: At(Shakey,x) $\wedge$ Pushable(b)

EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):

PRECOND: At(Shakey,x) $\wedge$ At(b,x) $\wedge$ Climbable(b)

EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):

PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)

EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):

PRECOND: At(Shakey,x) $\wedge$ Pushable(b)

EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):

PRECOND: At(Shakey,x) $\wedge$ At(b,x) $\wedge$ Climbable(b)

EFFECT: On(Shakey,b) $\wedge\neg$On(Shakey,Floor)

Shakey Example, con't.

ACTION: Go(x,y):
PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)
EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):
PRECOND: At(Shakey,x) $\wedge$ Pushable(b)
EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):
PRECOND: At(Shakey,x) $\wedge$ At(b,x) $\wedge$ Climbable(b)
EFFECT: On(Shakey,b) $\wedge\neg$On(Shakey,Floor)

ACTION: ClimbDown(b):
PRECOND:
EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):
PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)
EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):
PRECOND: At(Shakey,x) $\wedge$ Pushable(b)
EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):
PRECOND: At(Shakey,x) $\wedge$ At(b,x) $\wedge$ Climbable(b)
EFFECT: On(Shakey,b) $\wedge\neg$On(Shakey,Floor)

ACTION: ClimbDown(b):
PRECOND: On(Shakey,b)
EFFECT:

Shakey Example, con't.

ACTION: Go(x,y):
PRECOND: At(Shakey,x) $\wedge$ In(x,r) $\wedge$ In(y,r)
EFFECT: At(Shakey,y) $\wedge\neg$(At(Shakey,x))

ACTION: Push(b,x,y):
PRECOND: At(Shakey,x) $\wedge$ Pushable(b)
EFFECT: At(b,y) $\wedge$ At(Shakey,y) $\wedge\neg$At(b,x) $\wedge\neg$At(Shakey,x)

ACTION: ClimbUp(b):
PRECOND: At(Shakey,x) $\wedge$ At(b,x) $\wedge$ Climbable(b)
EFFECT: On(Shakey,b) $\wedge\neg$On(Shakey,Floor)

ACTION: ClimbDown(b):
PRECOND: On(Shakey,b)
EFFECT: On(Shakey,Floor) $\wedge\neg$ On(Shakey,b)

Shakey Example, con't.

ACTION: TurnOn(I):
PRECOND:
EFFECT:

Shakey Example, con't.

ACTION: TurnOn(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT:

Shakey Example, con't.

ACTION: TurnOn(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT: TurnedOn(I)

Shakey Example, con't.

ACTION: TurnOn(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT: TurnedOn(I)

ACTION: TurnOff(I):

PRECOND:

EFFECT:

Shakey Example, con't.

ACTION: TurnOn(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT: TurnedOn(I)

ACTION: TurnOff(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT:

Shakey Example, con't.

ACTION: TurnOn(I):

PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT: TurnedOn(I)

ACTION: TurnOff(I):

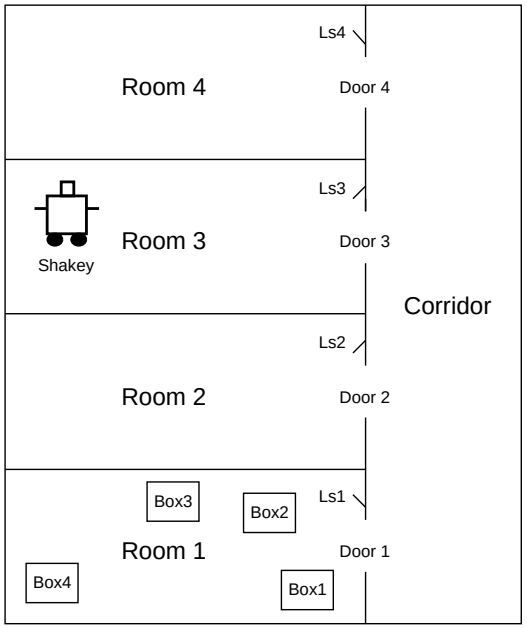
PRECOND: On(Shakey,b) $\wedge$ At(Shakey,x) $\wedge$ At(I,x)

EFFECT: $\neg$TurnedOn(I)

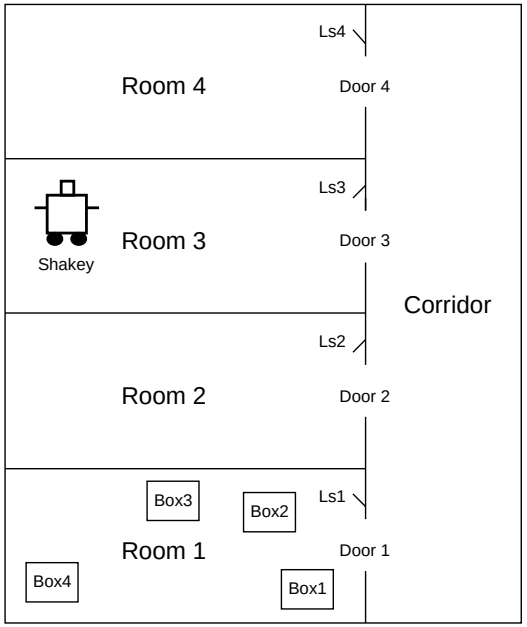
Shakey Example, con't.

INITIAL STATE:

In(...) Climbable(...) Pushable(...) At(...) TurnedOn(...)



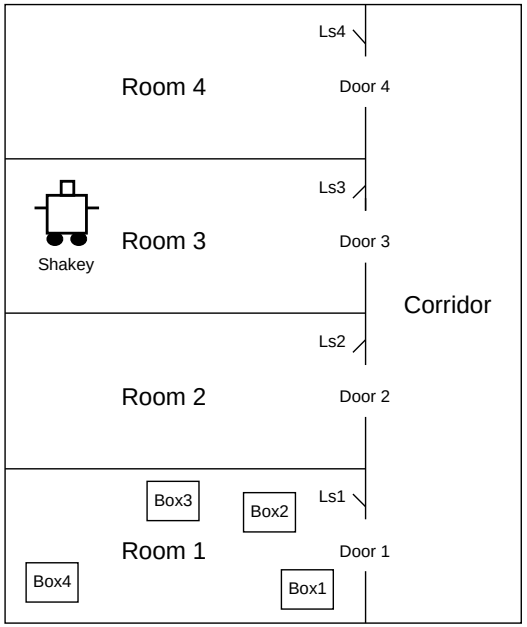
Shakey Example, con't.



INITIAL STATE:

$$In(Switch1,Room1) \wedge In(Door1,Room1) \wedge In(Door1,Corridor)$$
$$In(Switch1,Room2) \wedge In(Door2,Room2) \wedge In(Door2,Corridor)$$
$$In(Switch1,Room3) \wedge In(Door3,Room3) \wedge In(Door3,Corridor)$$
$$In(Switch1,Room4) \wedge In(Door4,Room4) \wedge In(Door4,Corridor)$$
$$In(Shakey,Room3) \wedge At(Shakey,XS)$$
$$In(Box1,Room1) \wedge In(Box2,Room1) \wedge In(Box3,Room1) \wedge In(Box4,Room1)$$

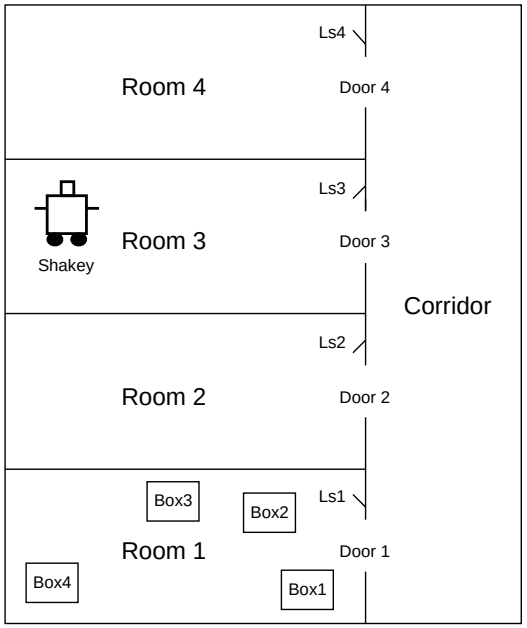
Shakey Example, con't.



INITIAL STATE (con't.):

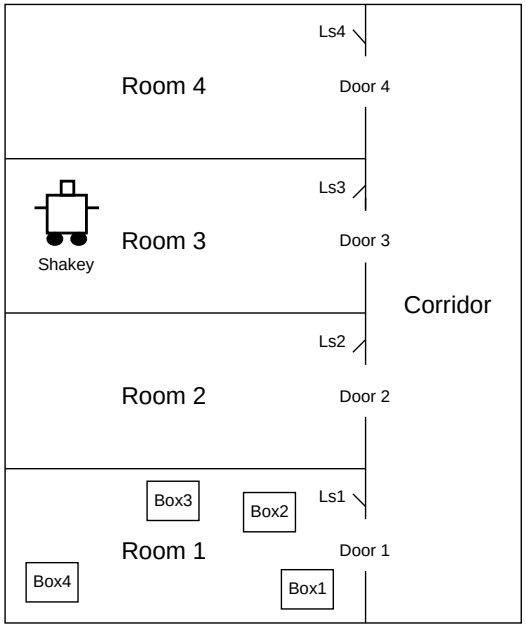
Climbable(Box1) $\wedge$ Climbable(Box2) $\wedge$ Climbable(Box3) $\wedge$ Climbable(Box4)
Pushable(Box1) $\wedge$ Pushable(Box2) $\wedge$ Pushable(Box3) $\wedge$ Pushable(Box4)
At(Box1, X1) $\wedge$ At(Box2, X2) $\wedge$ At(Box3, X3) $\wedge$ At(Box4, X4)
TurnedOn(Switch1) $\wedge$ TurnedOn(Switch4)

Shakey Example, con't.



Plan to achieve goal of getting Box2 into Room2:

Shakey Example, con't.



Plan to achieve goal of getting Box2 into Room2:

Go(XS,Door3)
Go(Door3,Door1)
Go(Door1,X2)
Push(Box2, X2, Door1)
Push(Box2, Door1, Door2)
Push(Box2, Door2, Switch2)

Partially ordered plans

Partially ordered collection of steps with

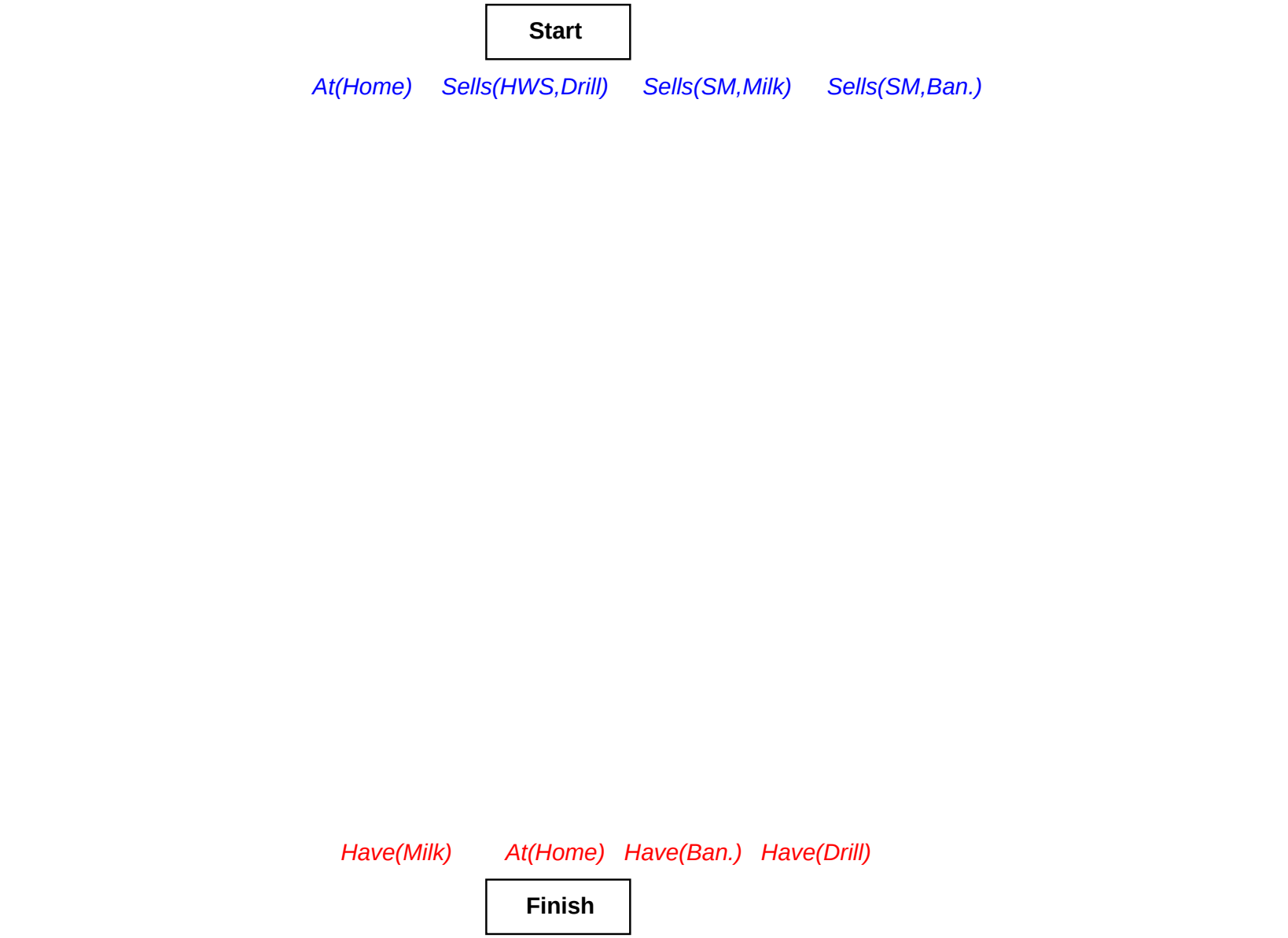
- $Start$ step has the initial state description as its effect
- $Finish$ step has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

Open condition = precondition of a step not yet causally linked

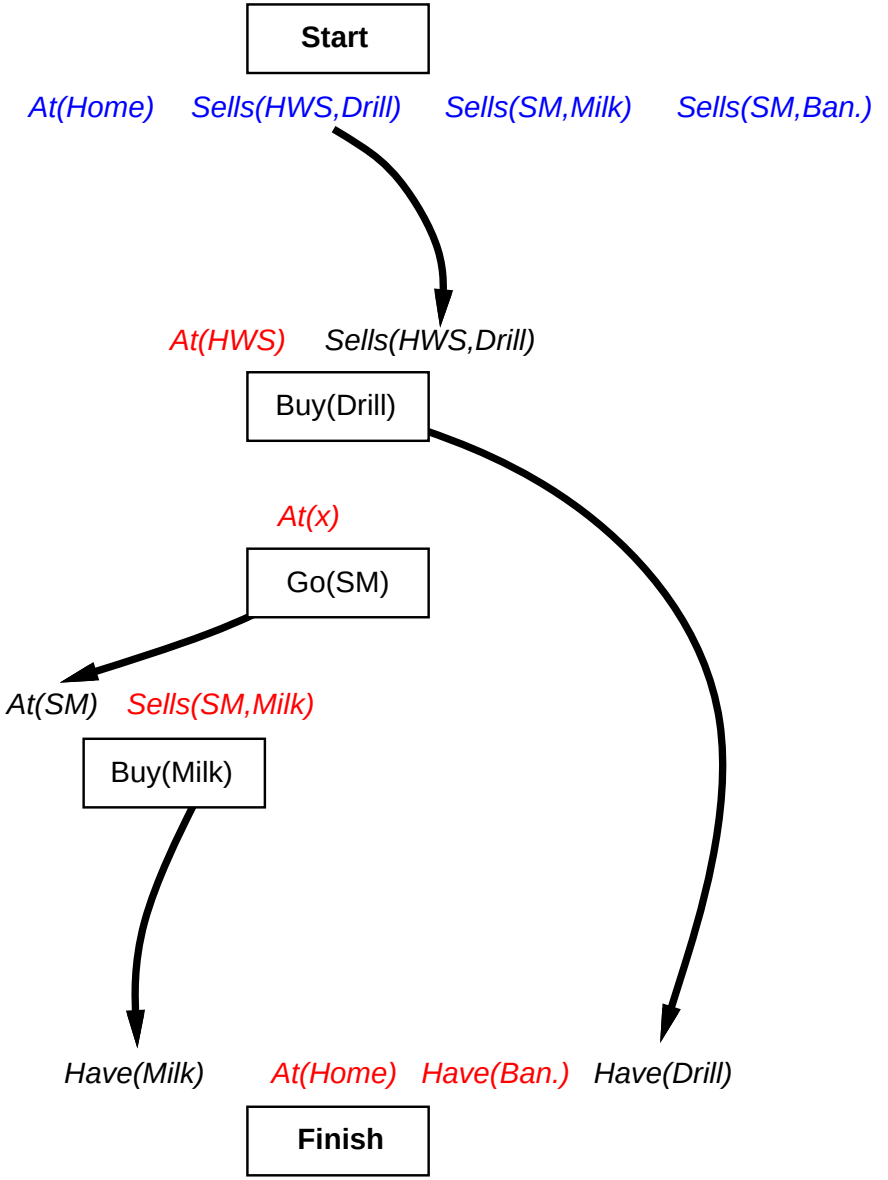
A plan is complete iff every precondition is achieved

A precondition is achieved iff it is the effect of an earlier step
and no possibly intervening step undoes it

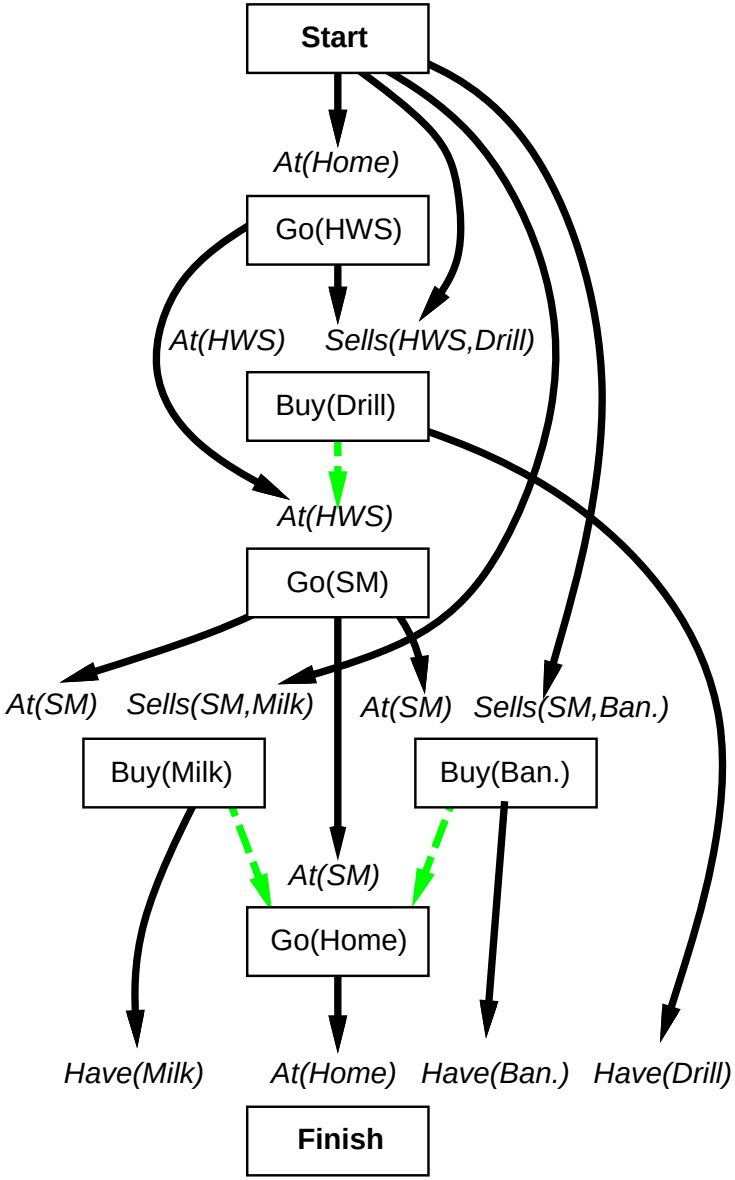
Example


Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Example


Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(x)
Go(SM)
At(SM)
Sells(SM,Milk)
Buy(Milk)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Example


Start
At(Home)
Go(HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Planning process

Operators on partial plans:

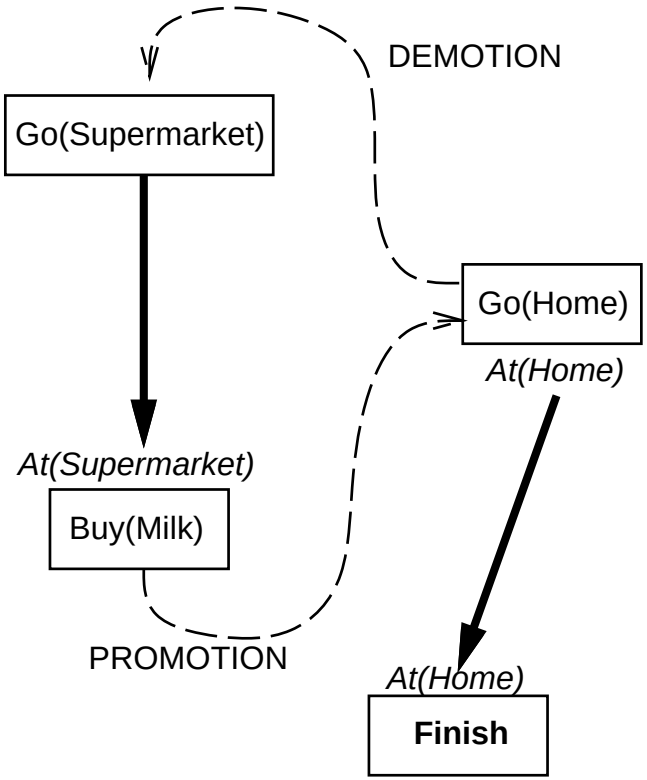
- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

Backtrack if an open condition is unachievable or
if a conflict is unresolvable

Clobbering and promotion/demotion

A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(Supermarket)$:



Demotion: put before $Go(Supermarket)$

Promotion: put after $Buy(Milk)$

Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:

- choice of S_{add} to achieve S_{need}
- choice of demotion or promotion for clobberer
- selection of S_{need} is irrevocable

POP is sound, complete, and systematic (no repetition)

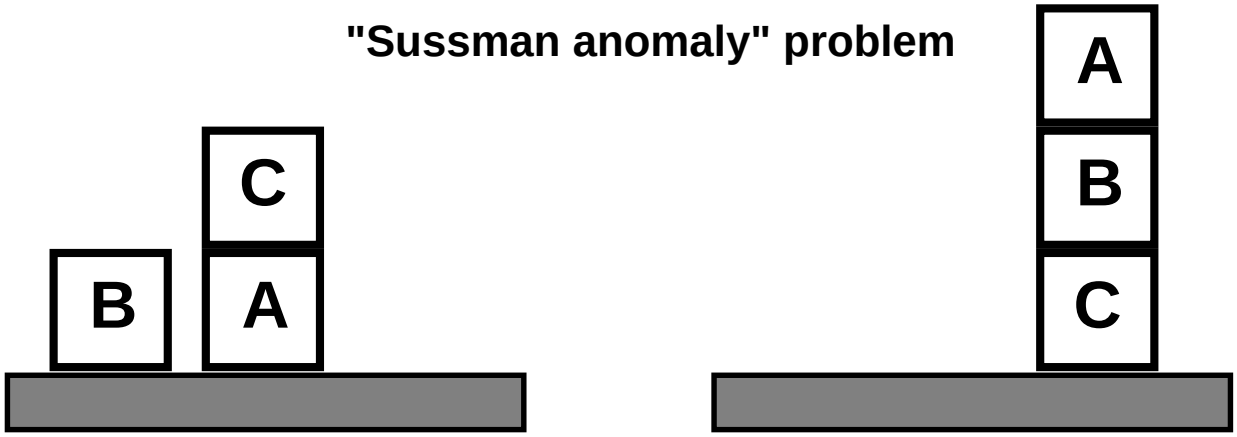
Extensions for disjunction, universals, negation, conditionals

Can be made efficient with good heuristics derived from problem description

Particularly good for problems with many loosely related subgoals

Example: Blocks world

"Sussman anomaly" problem



Start State

Goal State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

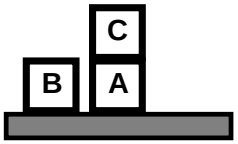
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example contd.

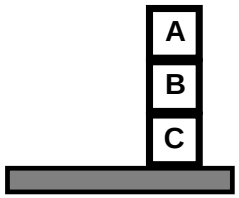
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

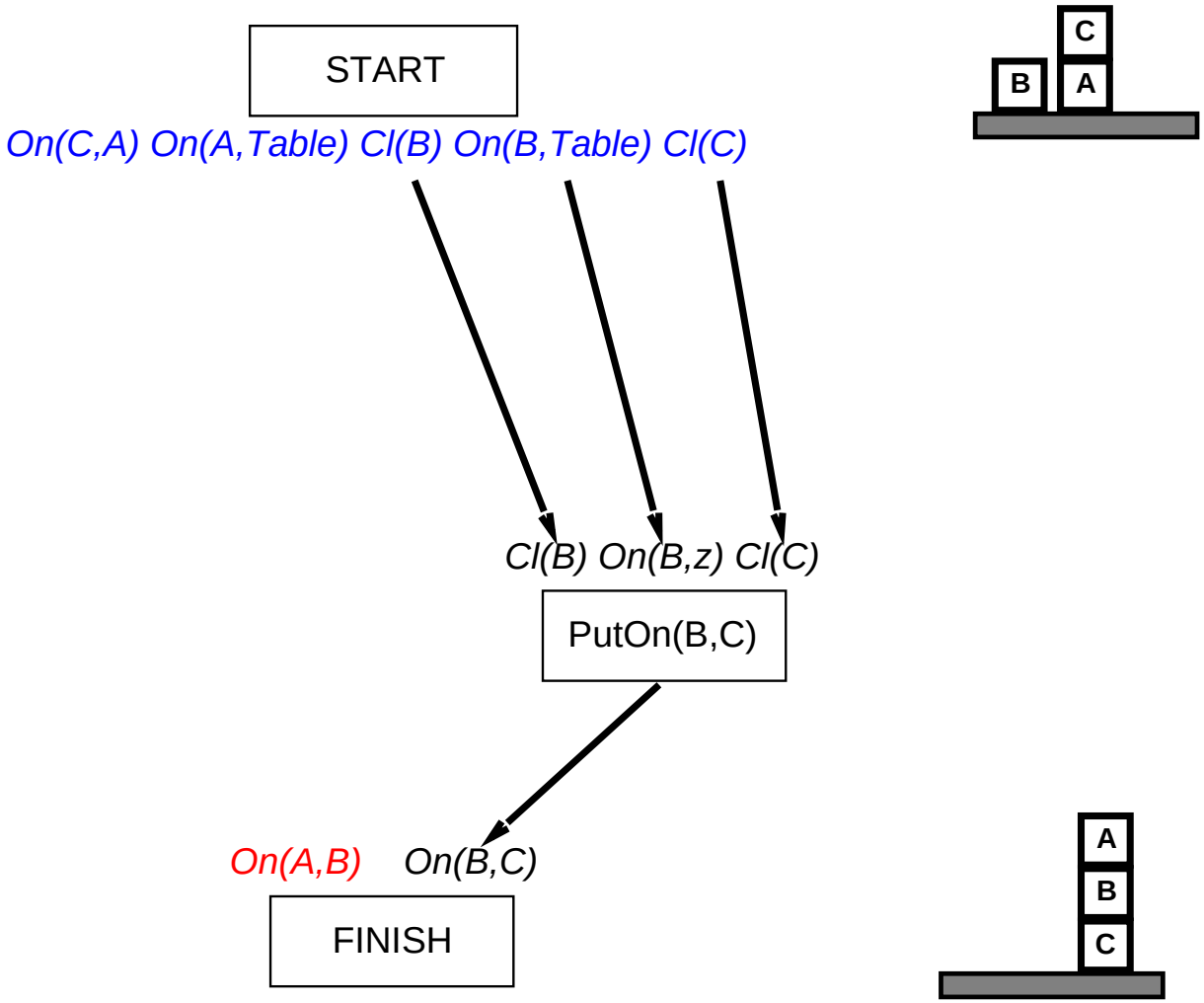


On(A,B) On(B,C)

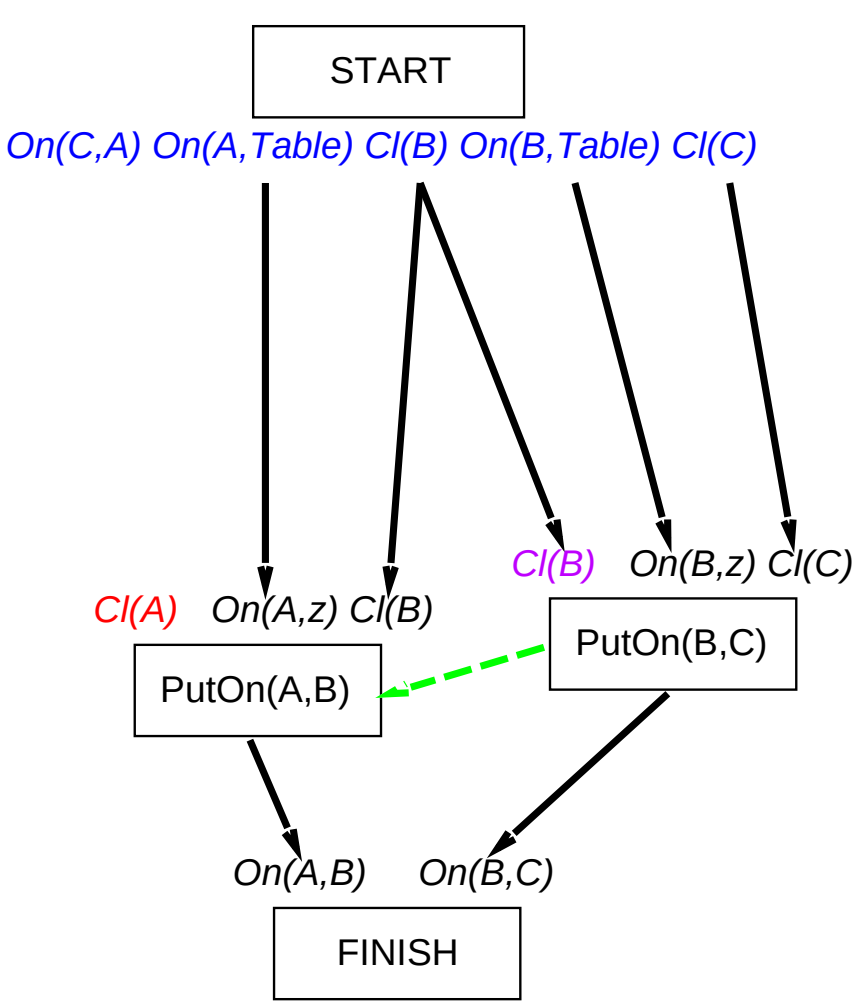
FINISH

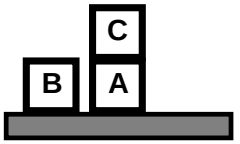


Example contd.

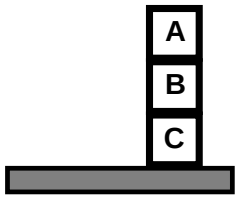


Example contd.

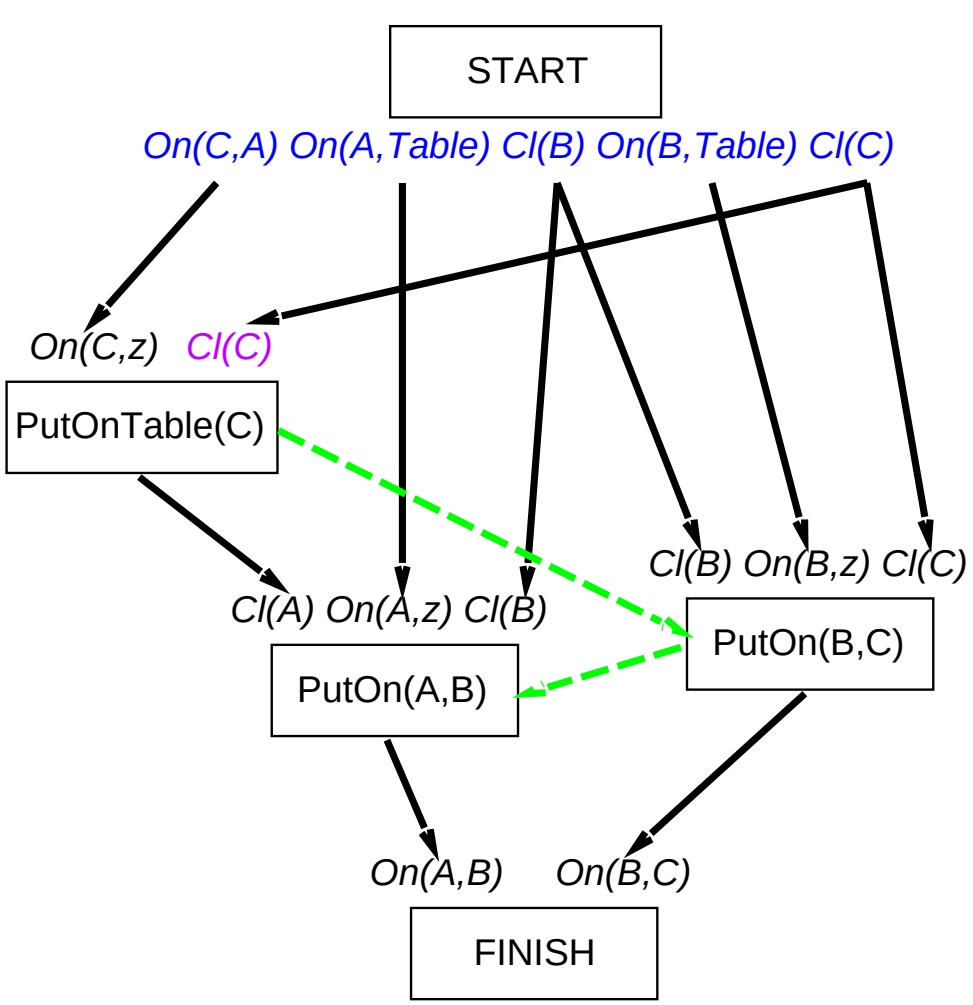


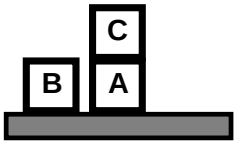


PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)



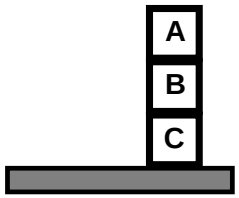
Example contd.





PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)



Heuristics for Planning

Most obvious Heuristic: Number of distinct open preconditions.

Overestimates: When actions achieve multiple goals

Underestimates: When negative interactions between plan steps

Better way: Use planning graph for generating better heuristic estimates.

Planning Graphs

Levels: Correspond to time steps in the plan (0 = initial state)

Each level contains literals + actions: those that *could* be true or executed

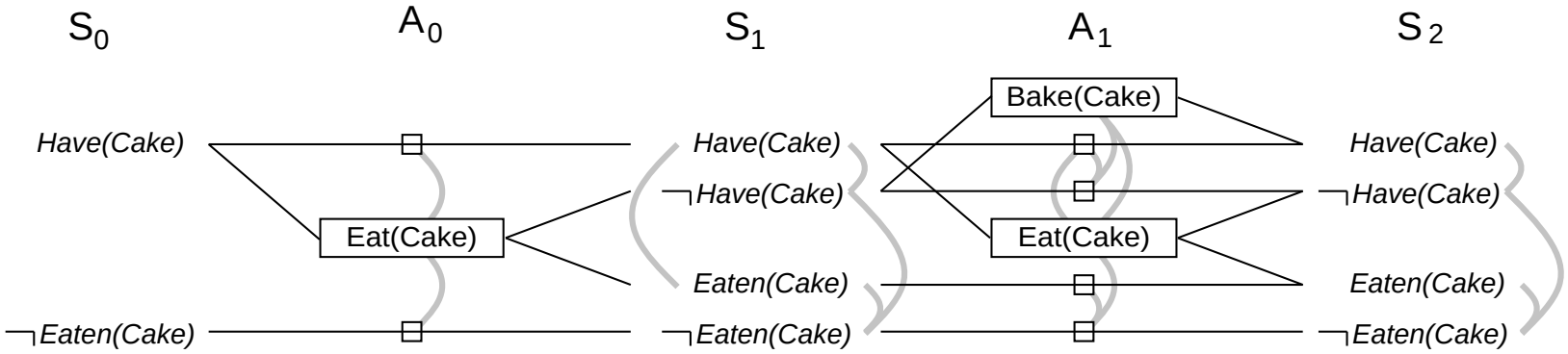
Number of planning steps in planning graph is good estimate of how difficult it is to acheive a given literal from initial state

Can be constructed very efficiently

Works only for *propositionalized problems*

Planning Graph – Have Cake

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
Precond: $Have(Cake)$
Effect: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
Precond: $\neg Have(Cake)$
Effect: $Have(Cake))$



Persistence actions

Mutual exclusion (mutex) links

Mutex Links

A mutex relation holds between two actions at a given level if any of the following is true:

♢ Inconsistent effects: one action negates another.

♢ Interference: one of effects of action is negation of precondition of another action.

♢ Competing needs: one of preconditions of action is mutually exclusive with precondition of other.

A mutex relation holds between two literals at a given level if:

♢ One is negation of other.

♢ Each possible pair of actions that could achieve the literals is mutex.

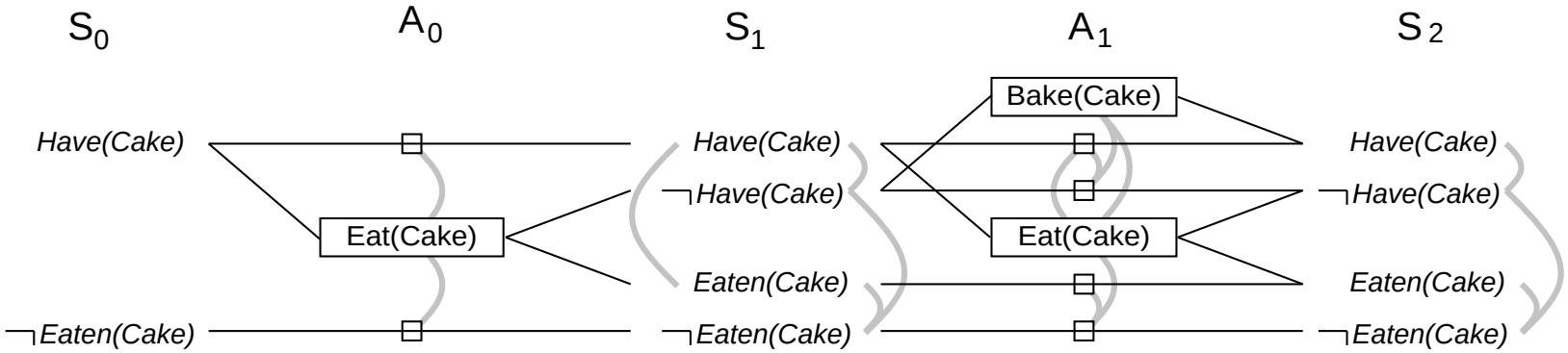
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



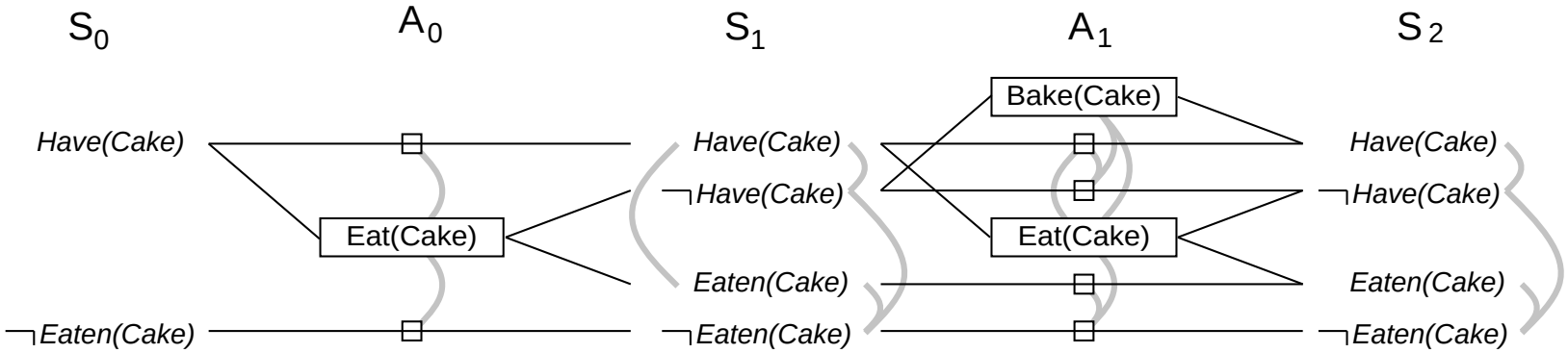
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



Max-level cost?

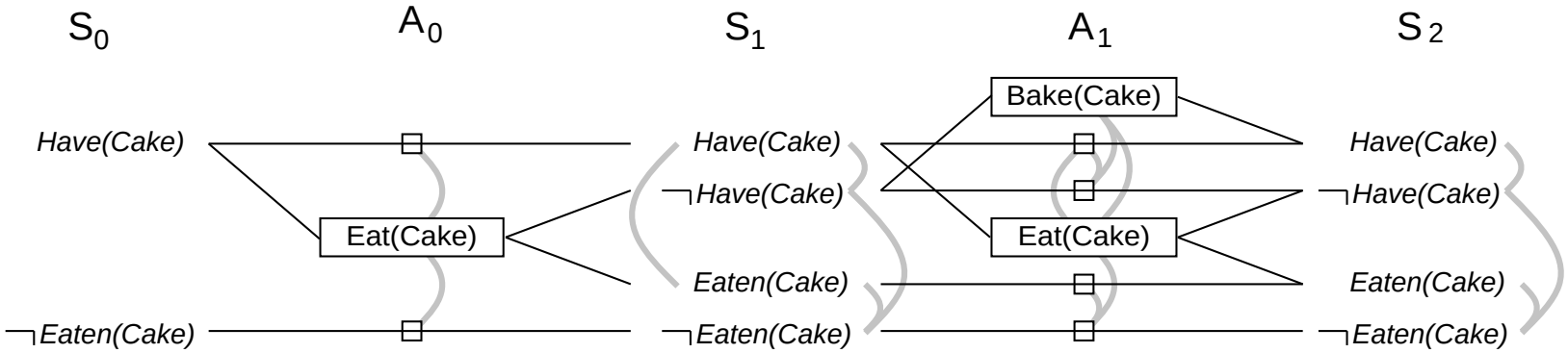
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



Max-level cost? 1

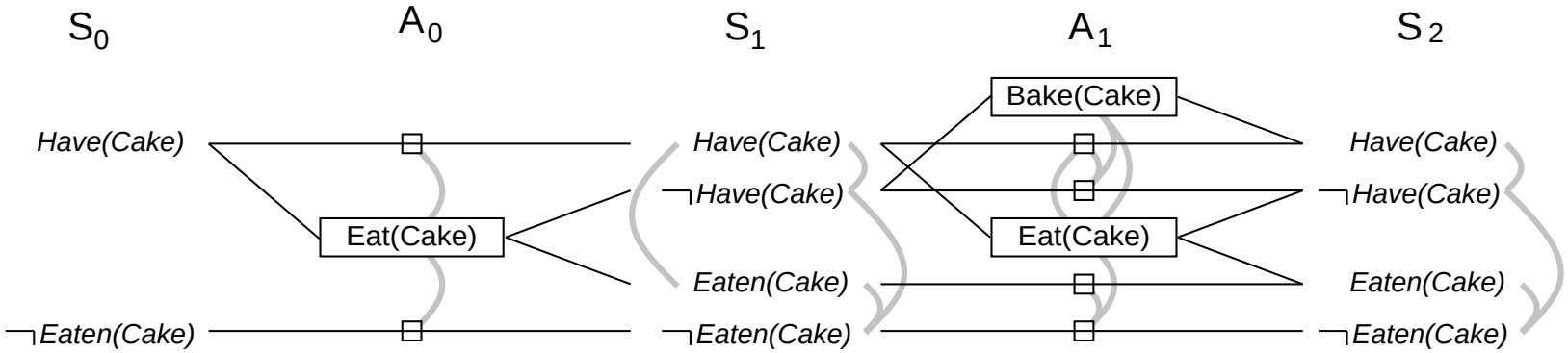
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



Max-level cost? 1 Level sum cost?

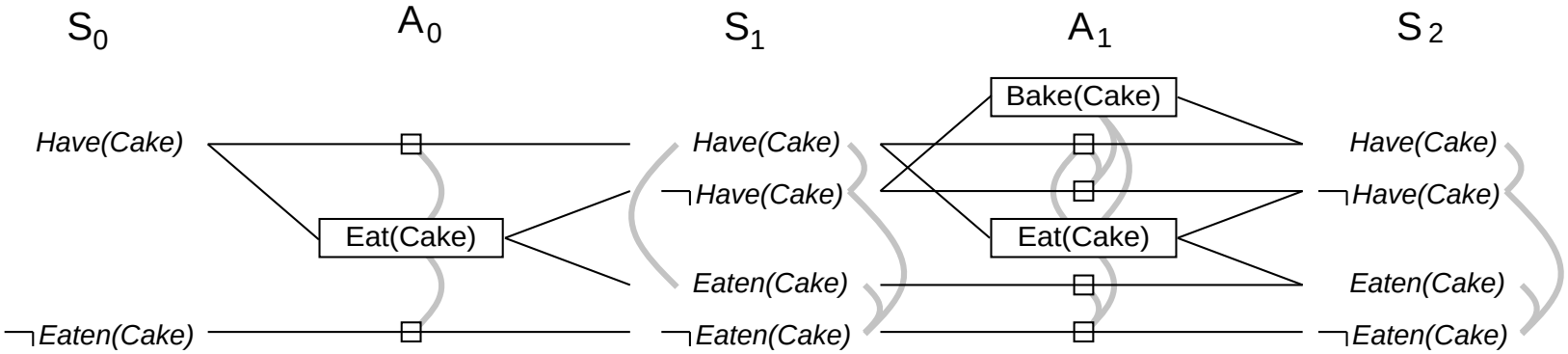
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



Max-level cost? 1 Level sum cost? 1

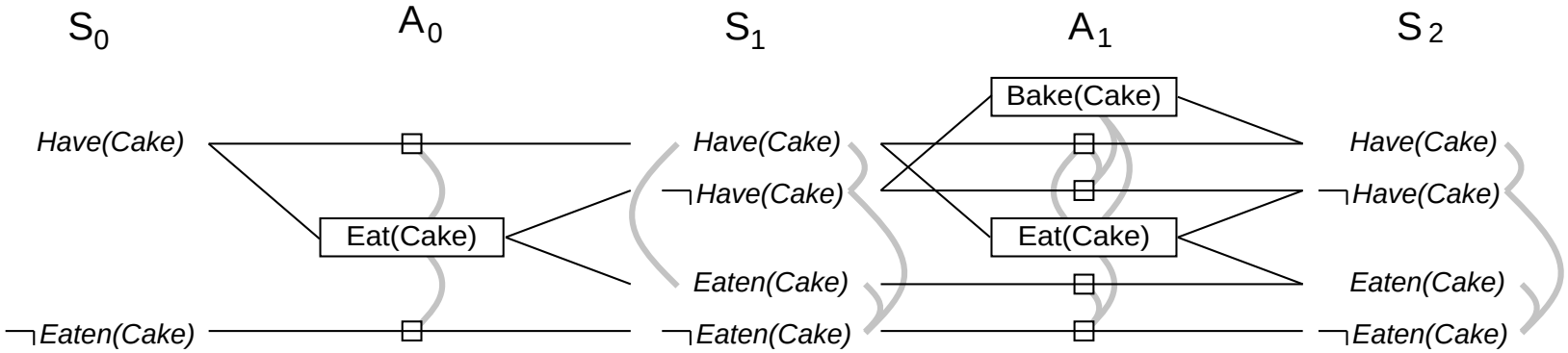
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



Max-level cost? 1 Level sum cost? 1 Set-level Cost?

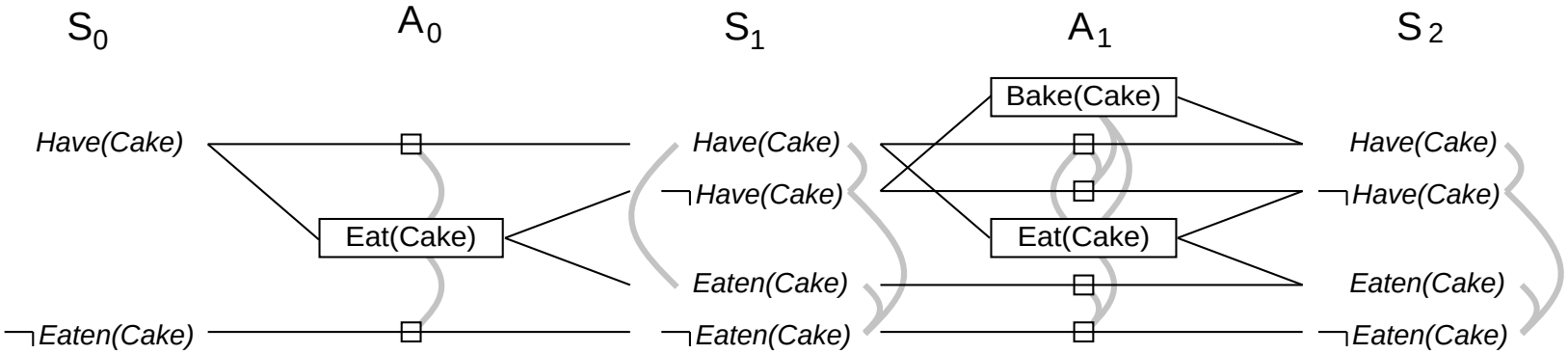
Heuristics from Planning Graphs

Estimate cost of goal literal = level it first appears = Level Cost

Use serial planning graphs to allow only one action at a time.

Cost of conjunction of goals:

- ♢ Max-level: Maximum level cost of any goal
- ♢ Level sum: Sum of level costs of goals (note: inadmissible)
- ♢ Set-level: Level at which all literals appear without mutex



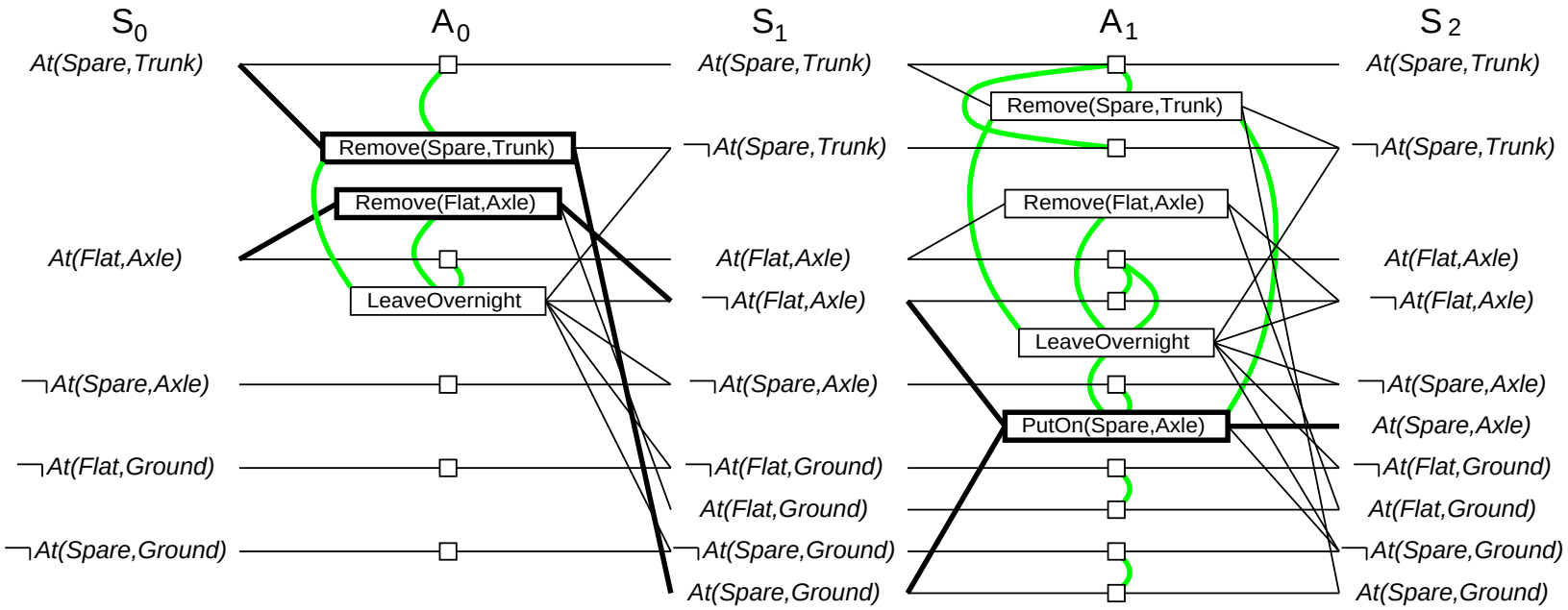
Max-level cost? 1 Level sum cost? 1 Set-level Cost? 2

Spare 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: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

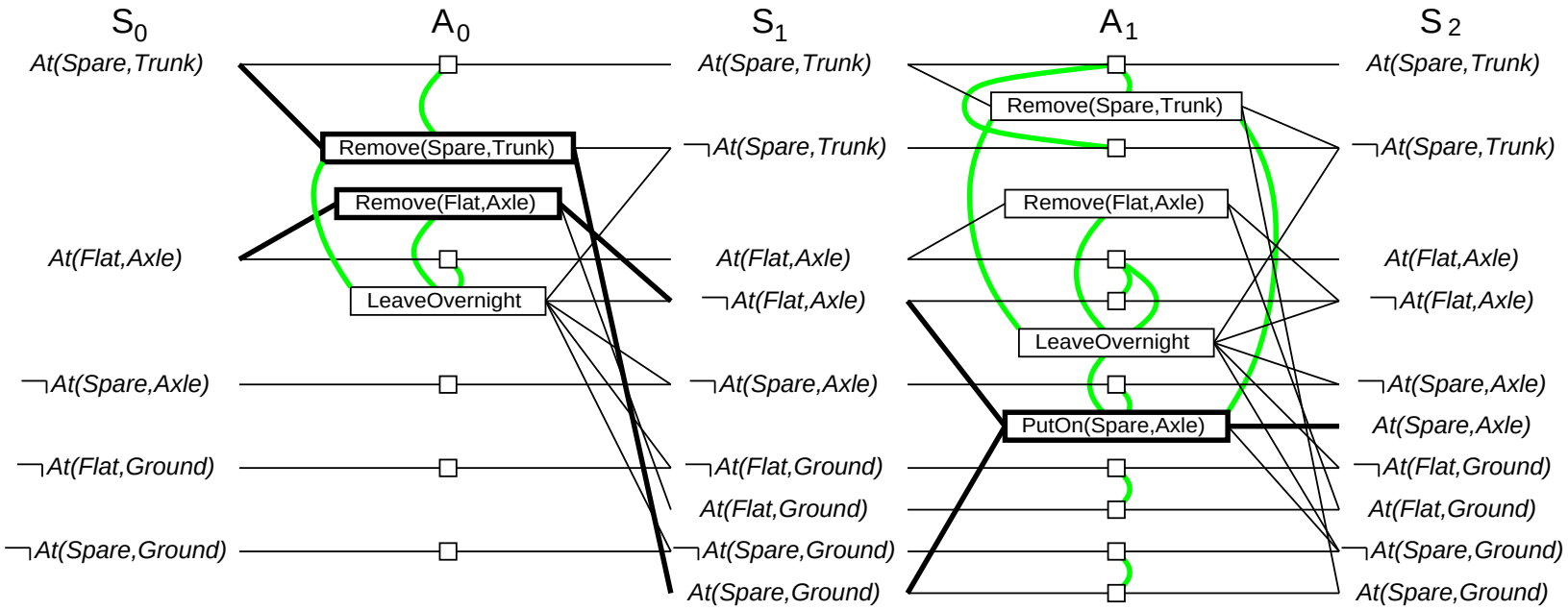
Planning Graph – Spare Tire

(Not all mutex's shown.)



Planning Graph – Spare Tire

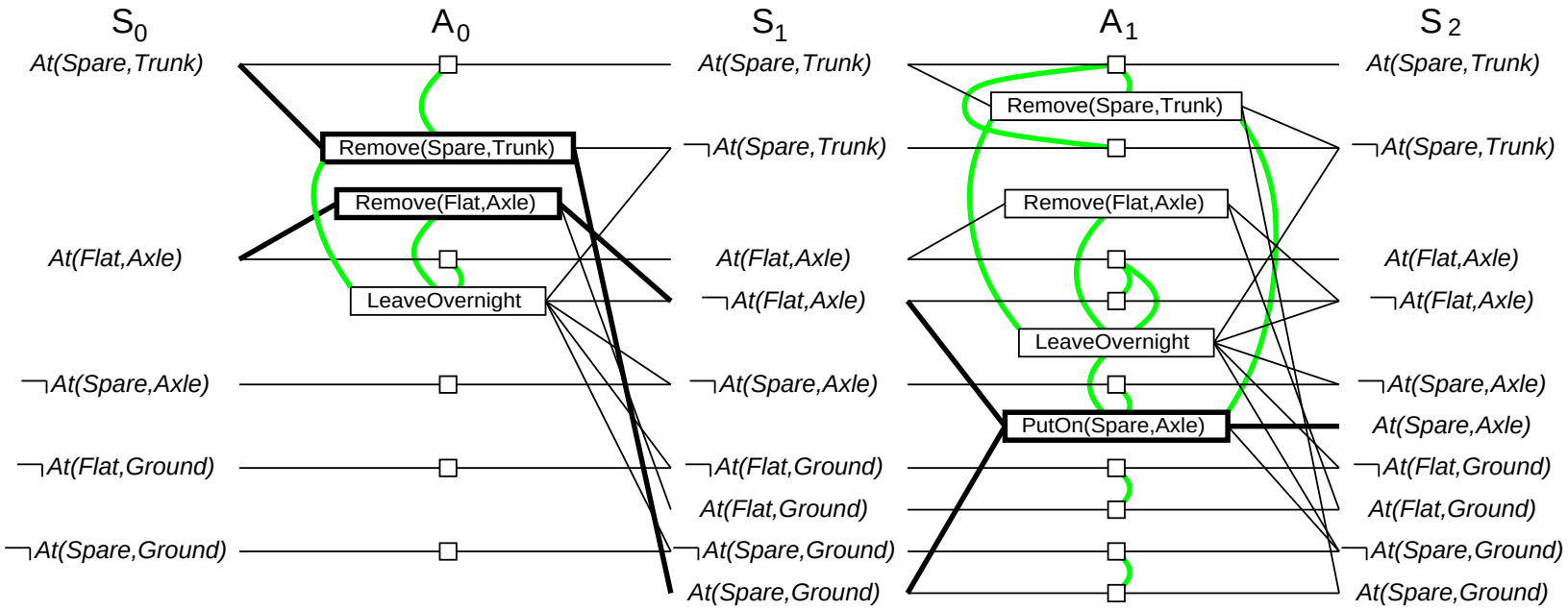
(Not all mutex's shown.)



Example of Inconsistent Effects?

Planning Graph – Spare Tire

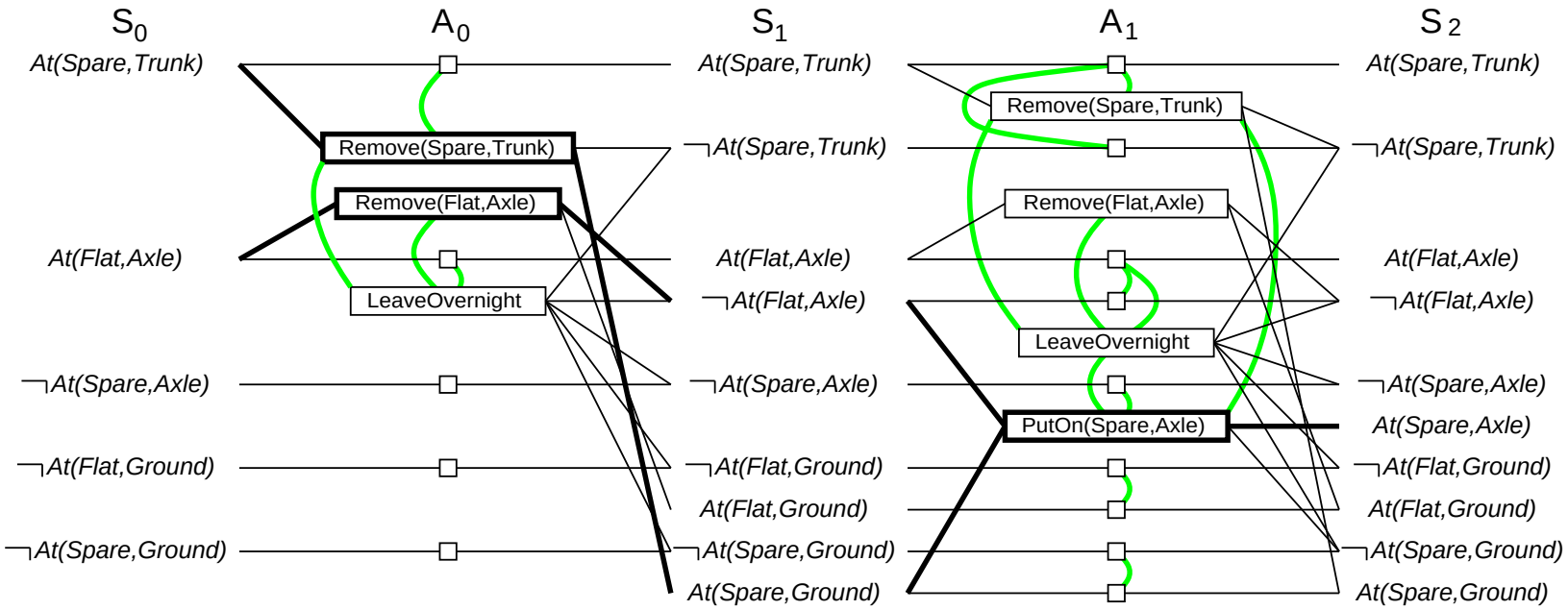
(Not all mutex's shown.)



Example of Inconsistent Effects? Remove(Spare, Trunk) and LeaveOvernight

Planning Graph – Spare Tire

(Not all mutex's shown.)

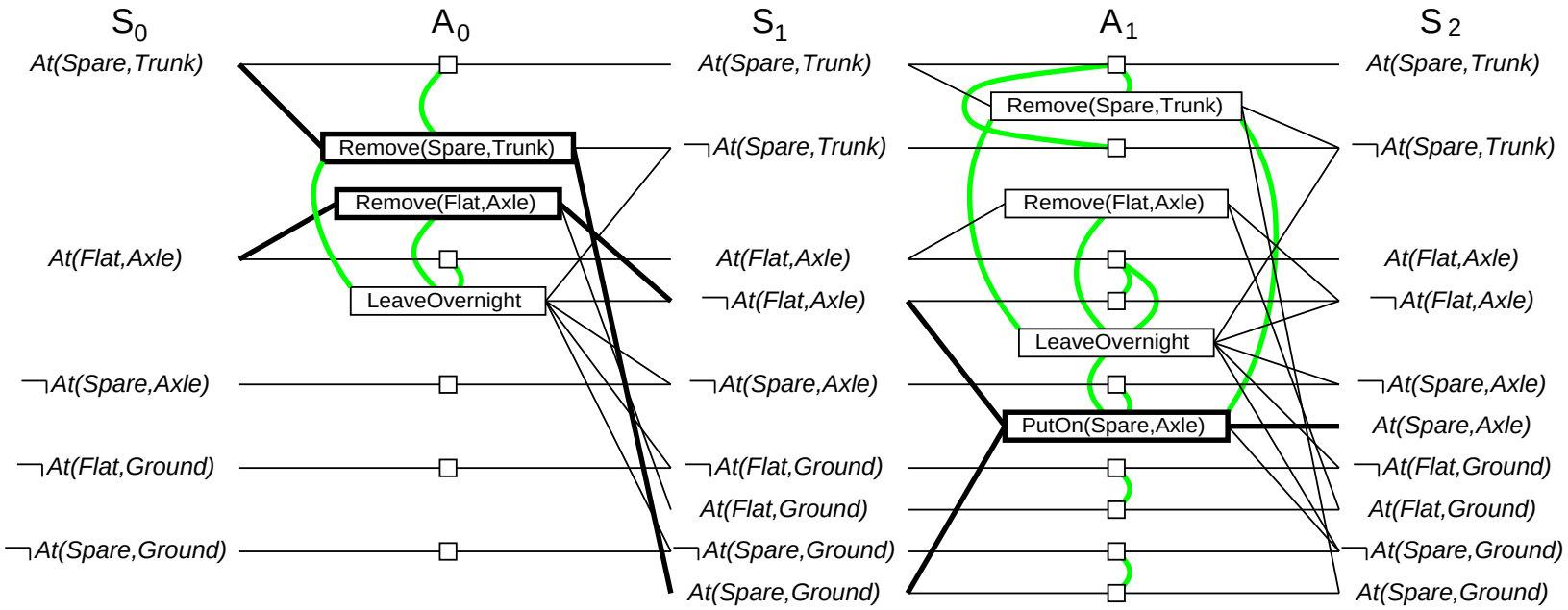


Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

Example of Interference?

Planning Graph – Spare Tire

(Not all mutex's shown.)

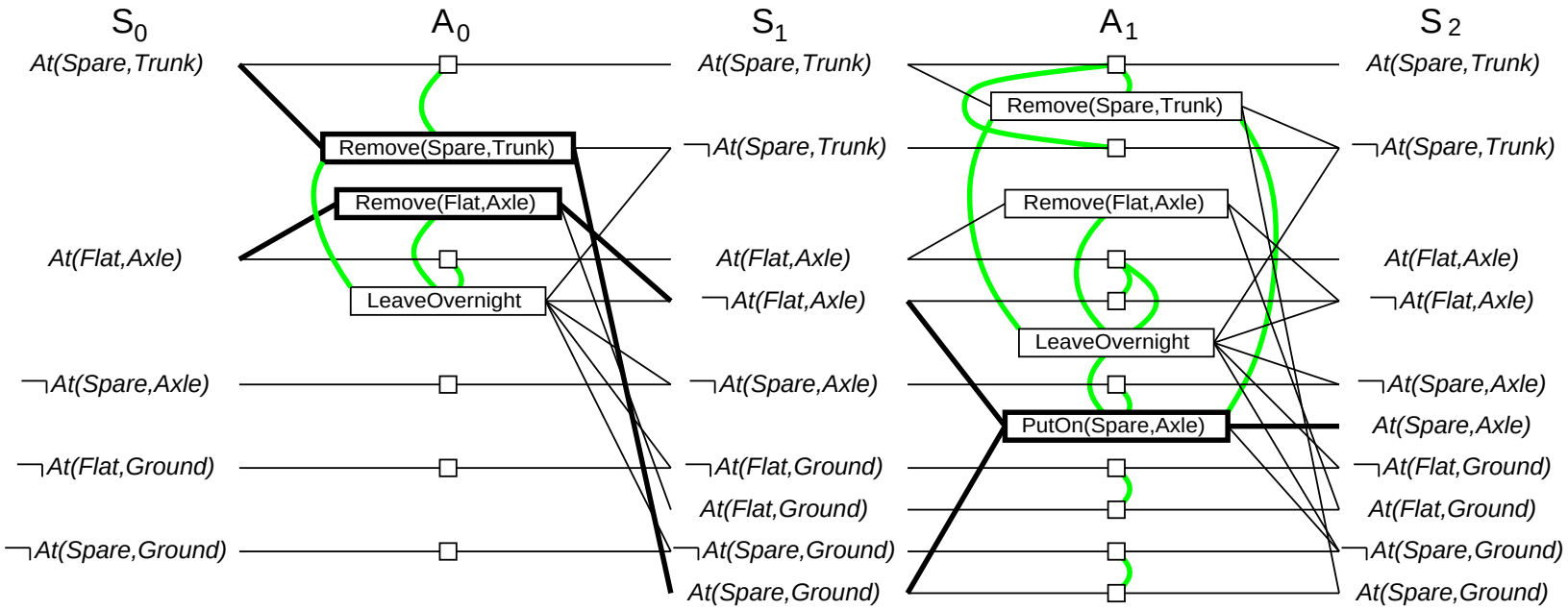


Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

Example of Interference? Remove(Flat,Axle) LeaveOvernight

Planning Graph – Spare Tire

(Not all mutex's shown.)



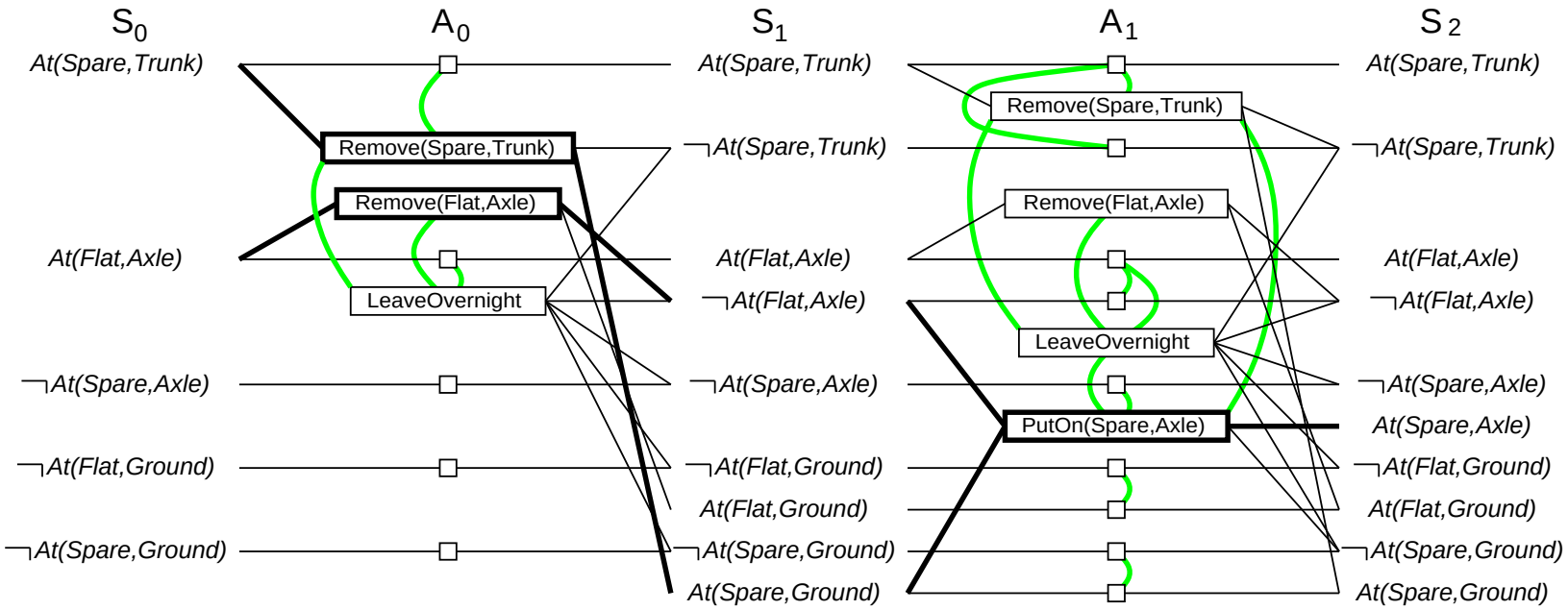
Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

Example of Interference? Remove(Flat,Axle) LeaveOvernight

Example of Competing Needs?

Planning Graph – Spare Tire

(Not all mutex's shown.)



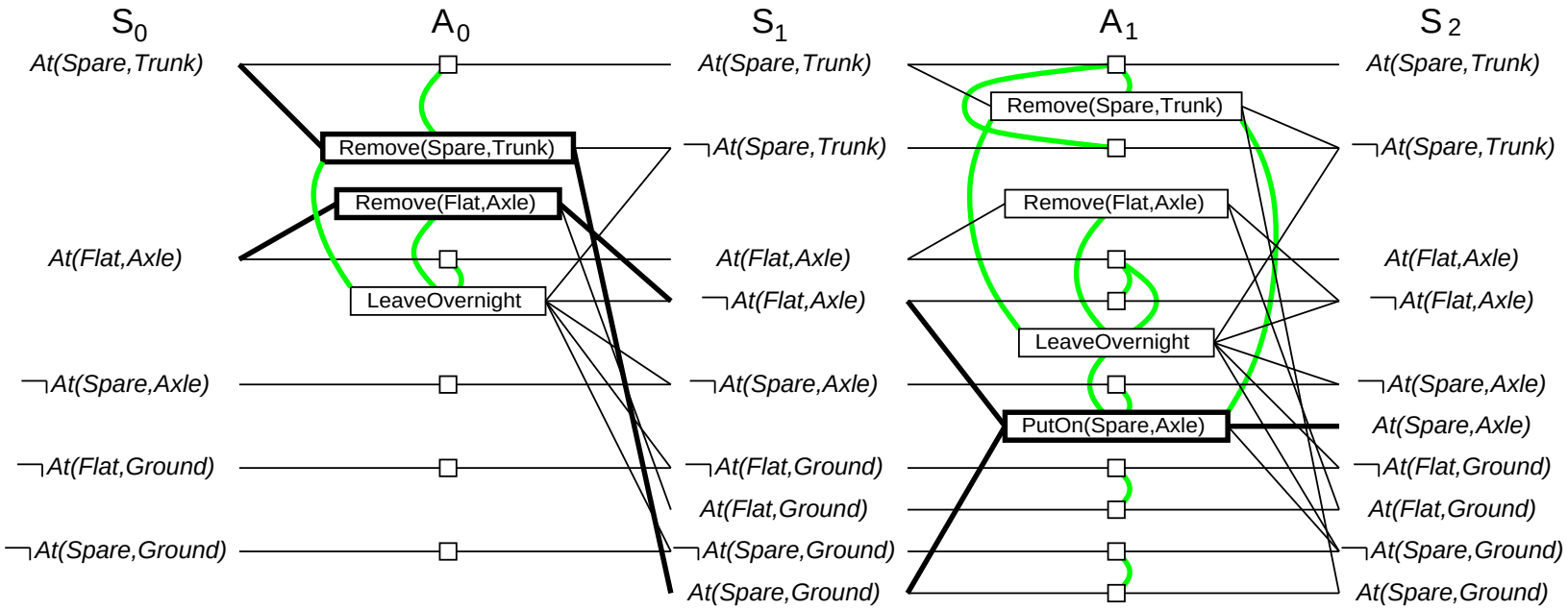
Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

Example of Interference? Remove(Flat,Axle) and LeaveOvernight

Example of Competing Needs? PutOn(Spare,Axle) and Remove(Flat,Axle)

Planning Graph – Spare Tire

(Not all mutex's shown.)



Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

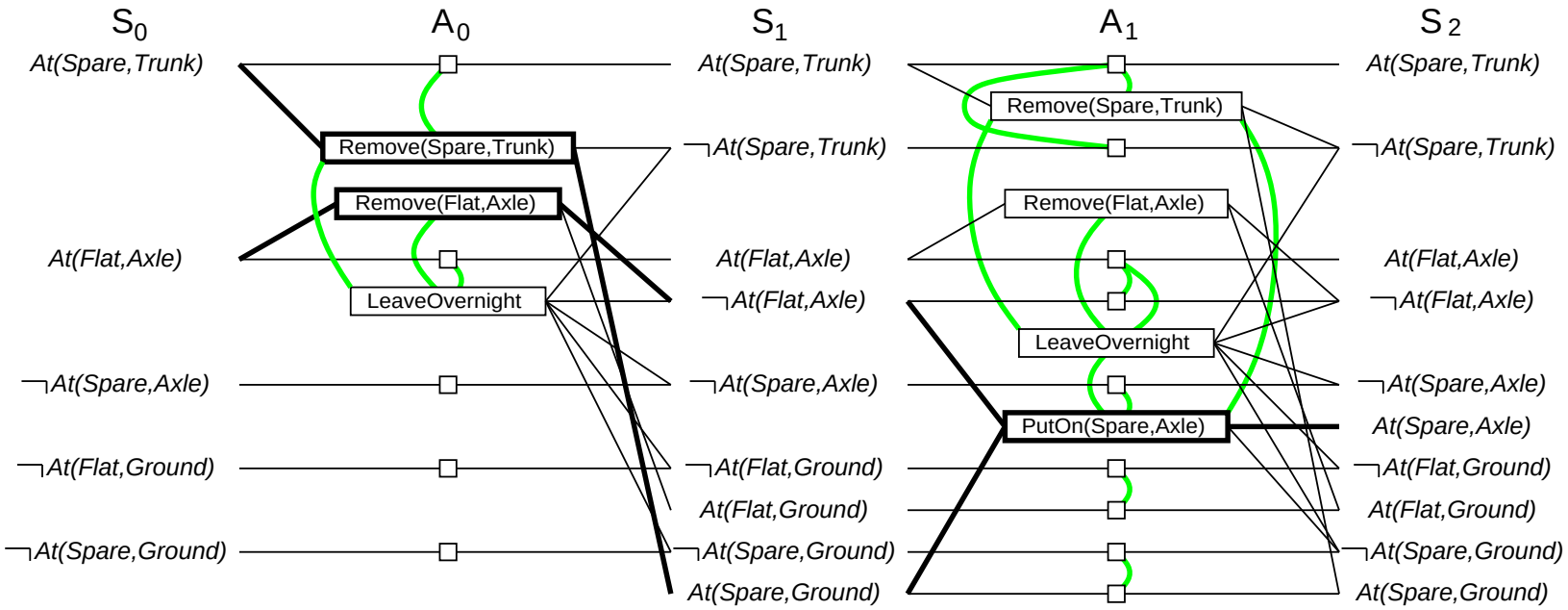
Example of Interference? Remove(Flat,Axle) and LeaveOvernight

Example of Competing Needs? PutOn(Spare,Axle) and Remove(Flat,Axle)

Example of Inconsistent Support?

Planning Graph – Spare Tire

(Not all mutex's shown.)



Example of Inconsistent Effects? Remove(Spare,Trunk) and LeaveOvernight

Example of Interference? Remove(Flat,Axle) and LeaveOvernight

Example of Competing Needs? PutOn(Spare,Axle) and Remove(Flat,Axle)

Example of Inconsistent Support? At(Spare,Axle) and At(Flat,Axle)

Review: Planning Graph

- Is special data structure used for
  1. Deriving better heuristic estimates
  2. Extract a solution to the planning problem: GRAPHPLAN algorithm
- Is a sequence $\langle S_0, A_0, S_1, A_1, \ldots, S_i \rangle$ of levels
  - Alternating state levels & action levels
  - Levels correspond to time stamps
  - Starting at initial state
  - State level is a set of (propositional) literals
    - All those literals that could be true at that level
  - Action level is a set of (propositionalized) actions
    - All those actions whose preconditions appear in the state level (ignoring all negative interactions, etc.)
- Propositionalization may yield combinatorial explosion in the presence of a large number of objects

Example of a Planning Graph (1)

Init(Have(Cake))

Goal(Have(Cake)∧Eaten(Cake))

Action(Eat(Cake)

Precond: Have(Cake)

Effect: ¬Have(Cake)∧Eaten(Cake))

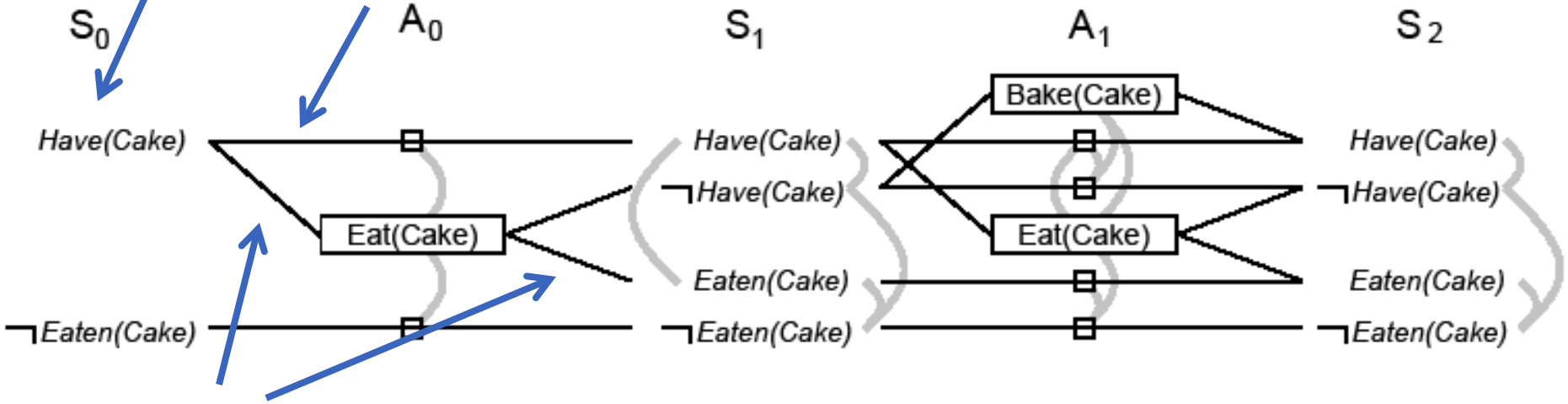
Action(Bake(Cake)

Precond: ¬Have(Cake)

Effect: Have(Cake))

Propositions true at the initial state

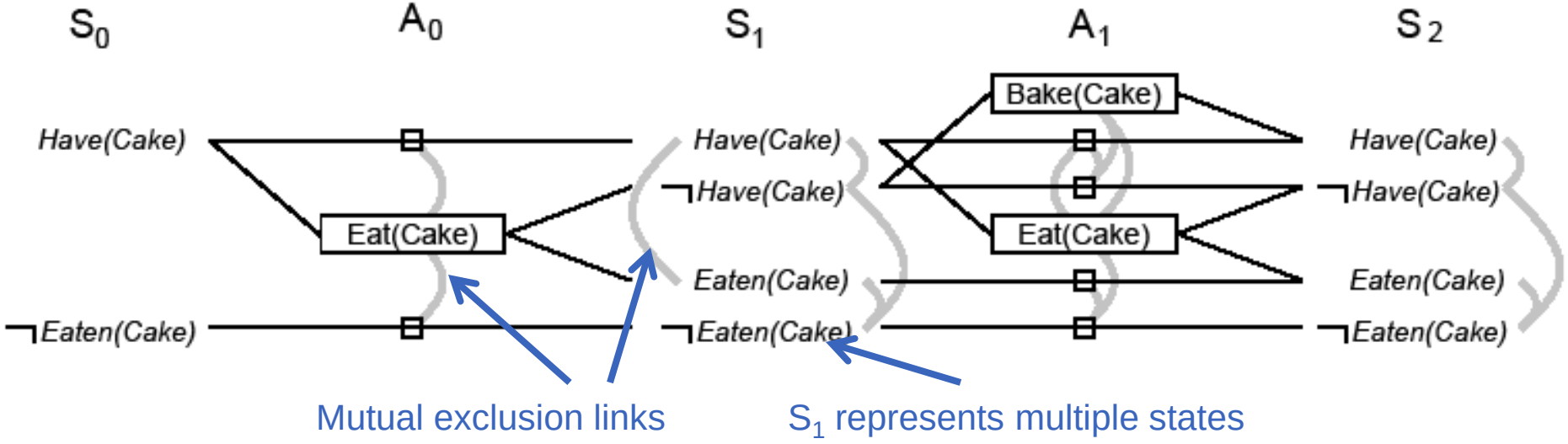
Persistence Actions (noop)



Action is connected to its preconds & effects

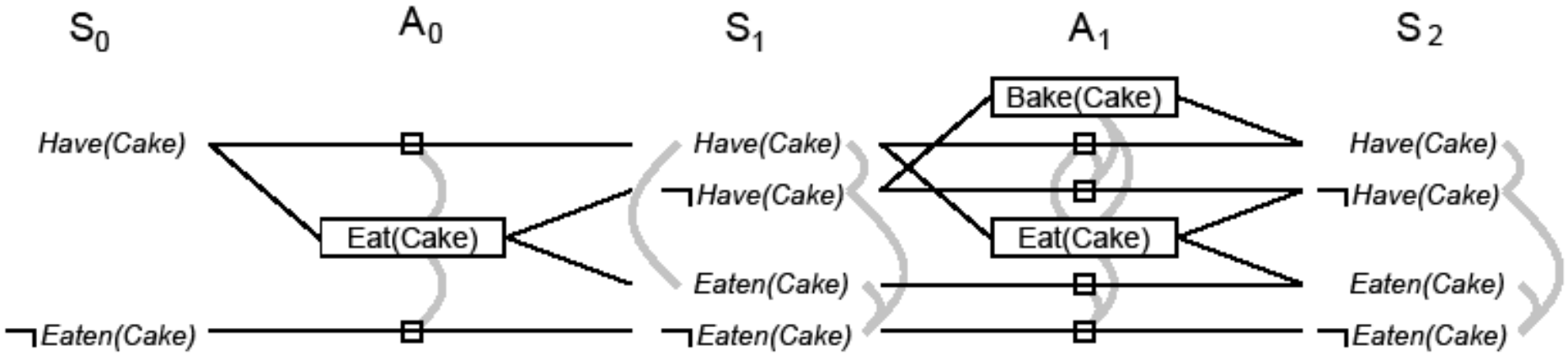
Example of a Planning Graph (2)

- At each state level, list all literals that may hold at that level
- At each action level, list all noops & all actions whose preconditions may hold at previous levels
- Repeat until plan 'levels off,' no new literals appears (S_i=S_{i+1})
- Building the Planning Graph is a polynomial process
- Add (binary) mutual exclusion (mutex) links between conflicting actions and between conflicting literals



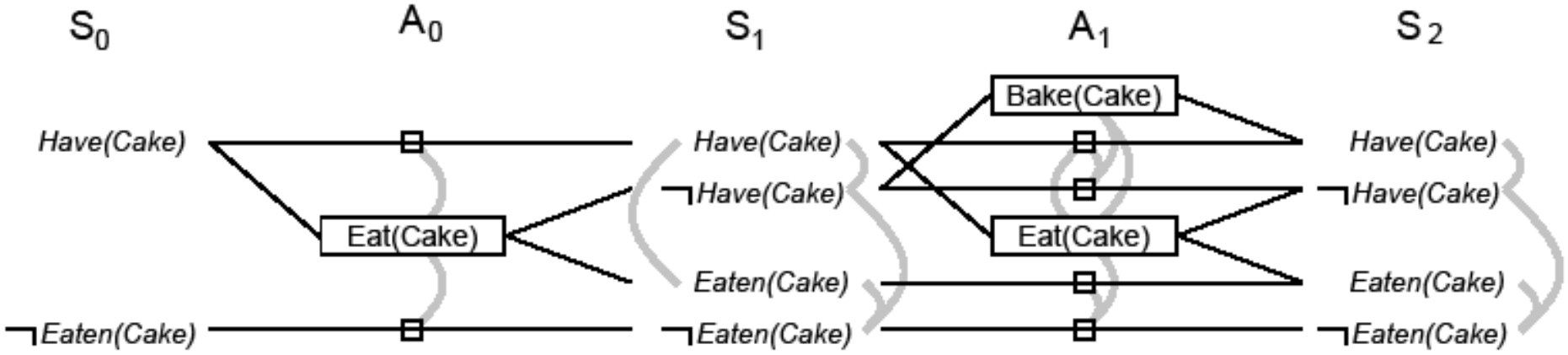
Mutex Links between Actions

1. **Inconsistent effects**: one action negates an effect of another
   - Eat(Cake) & noop of Have(Cake) disagree on effect Have(Cake)
2. **Interference**: An action effect negates the precondition of another
   - Eat(Cake) negates precondition of the noop of Have(Cake):
3. **Competing needs**: A precondition on an action is mutex with the precondition of another
   - Bake(Cake) & Eat(Cake): compete on Have(Cake) precondition



Mutex Links between Literals

1. Two literals are negation of each other
2. **Inconsistent support**: Each pair of actions that can achieve the two literals is mutex. Examples:
   - In S1, Have(Cake) & Eaten(Cake) are mutex
   - In S2, they are not because Bake(Cake) & the noop of Eaten(Cake) are not mutex



Planning Graph for Heuristic Estimation

- A literal that does not appear in the final level cannot be achieved by any plan
  - State-space search: Any state containing an unachievable literal has cost h(n)=∞
  - POP: Any plan with an unachievable open condition has cost h(n)=∞
- The estimate cost of any goal literal is the first level at which it appears
  - Estimate is admissible for individual literals
  - Estimate can be improved by serializing the graph (serial planning graph: one action per level) by adding mutex between all actions in a given level
- The estimate of a conjunction of goal literals
  - Three heuristics: max level, level sum, set level

Estimate of Conjunction of Goal Literals

- Max-level
  - The largest level of a literal in the conjunction
  - Admissible, not very accurate
- Level sum
  - Under subgoal independence assumption, sums the level costs of the literals
  - Inadmissible, works well for largely decomposable problems
- Set level
  - Finds the level at which all literals appear w/o any pair of them being mutex
  - Dominates max-level, works extremely well on problems where there is a great deal of interaction among subplans

GRAPHPLAN algorithm

GRAPHPLAN(*problem*) **returns** *solution* or *failure*
graph ← INITIALPLANNINGGRAPH(*problem*)
goals ← GOALS[*problem*]
nogoods ← an empty hash table // (*level, goals*) pair that can't be achieved
loop do
  if *goals* all non-mutex in last level of graph **then do**
    solution ← EXTRACTSOLUTION(*graph,goals,*NUMLEVELS(*graph),nogoods)*
    if *solution* ≠ *failure* **then return** *solution*
  if *graph* and nogoods have both leveld off **then return** *failure*
  graph ← EXPANDGRAPH (*graph*,*problem*)

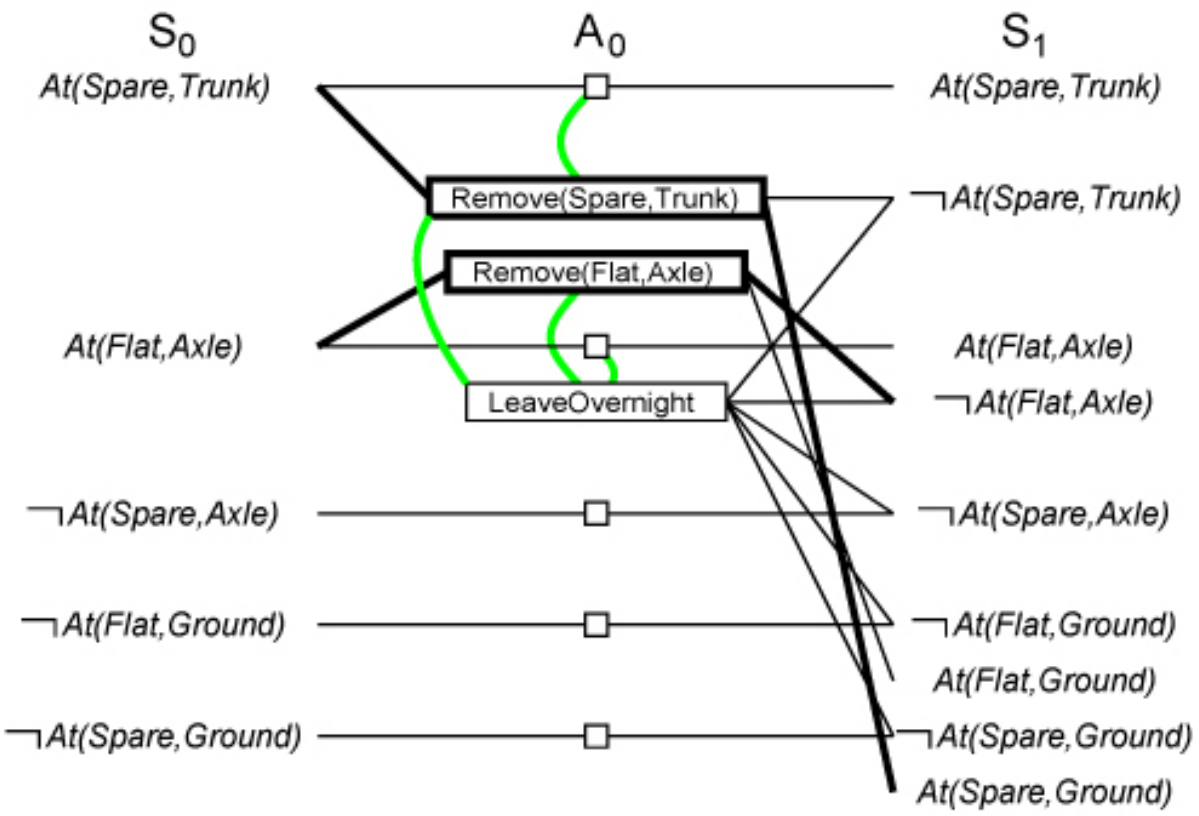
- Two main stages
  1. Extract solution
  2. Expand the graph

Example of GRAPHPLAN Execution (1)

- At(Spare,Axle) is not in S_0
- No need to extract solution
- Expand the plan

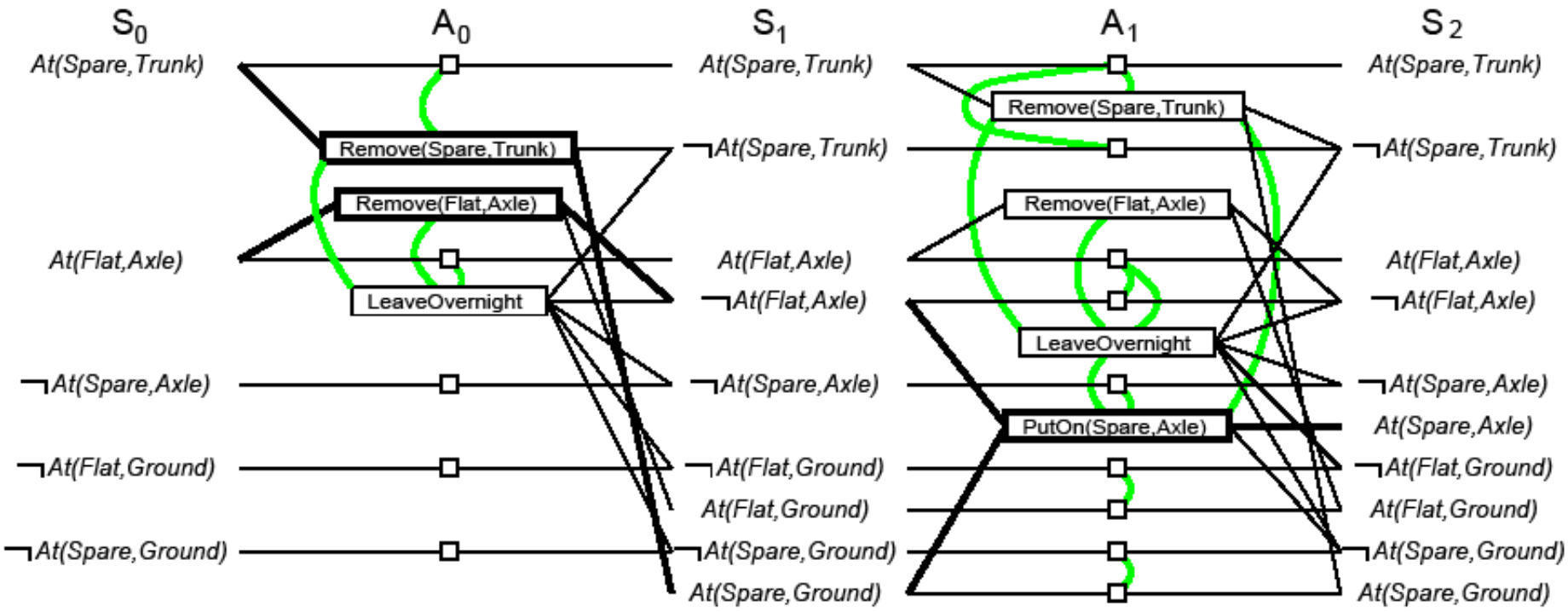
Example of GRAPHPLAN Execution (2)

- Three actions are applicable
- 3 actions and 5 noops are added
- Mutex links are added
- At(Spare,Axle) still not in S_1
- Plan is expanded



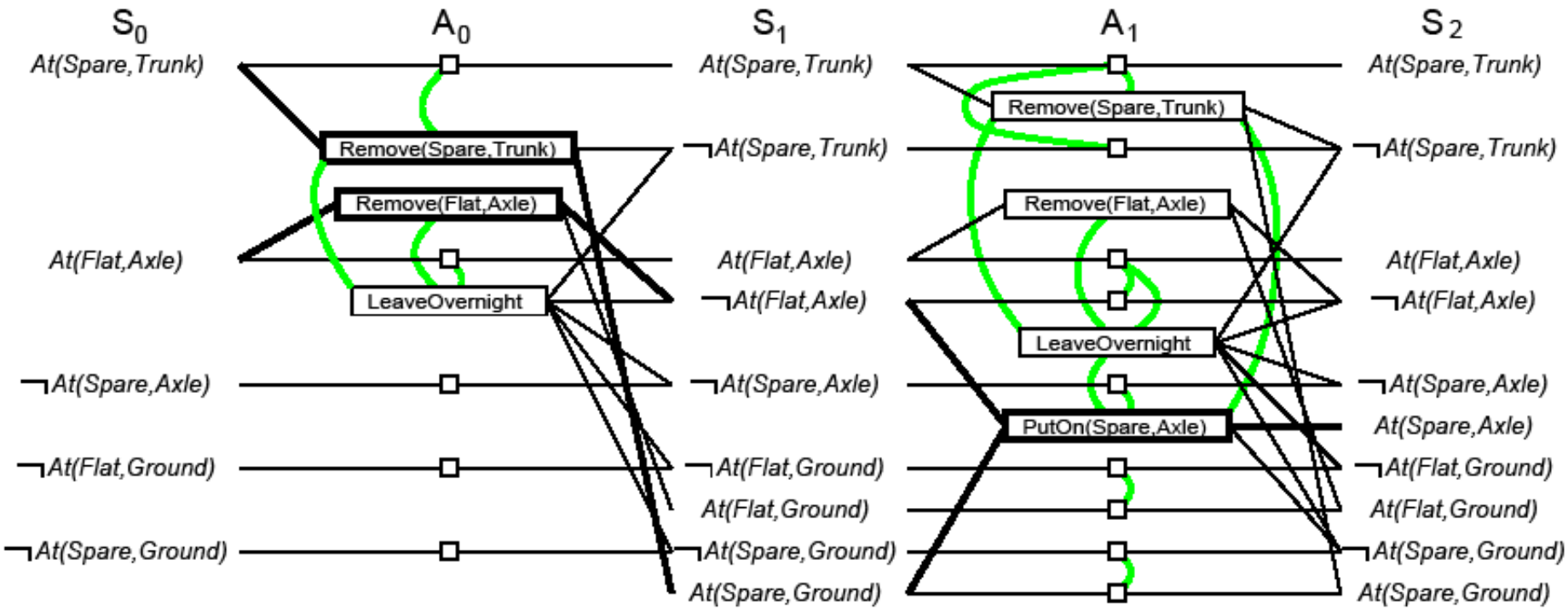
Example of GRAPHPLAN Execution (3)

- Illustrates well mutex links: inconsistent effects, interference, competing needs, inconsistent support



Solution Extraction (Backward)

Search problem from last level backward



Backtrack Search for Solution Extraction

- Starting at the highest fact level
  - Each goal is put in a goal list for the current fact layer
  - Search iterates thru each fact in the goal list trying to find an action to support it which is not mutex with any other chosen action
  - When an action is chosen, its preconditions are added to the goal list of the lower level
  - When all facts in the goal list of the current level have a consistent assignment of actions, the search moves to the next level
- Search backtracks to the previous level when it fails to assign an action to each fact in the goal list at a given level
- Search succeeds when the first level is reached.

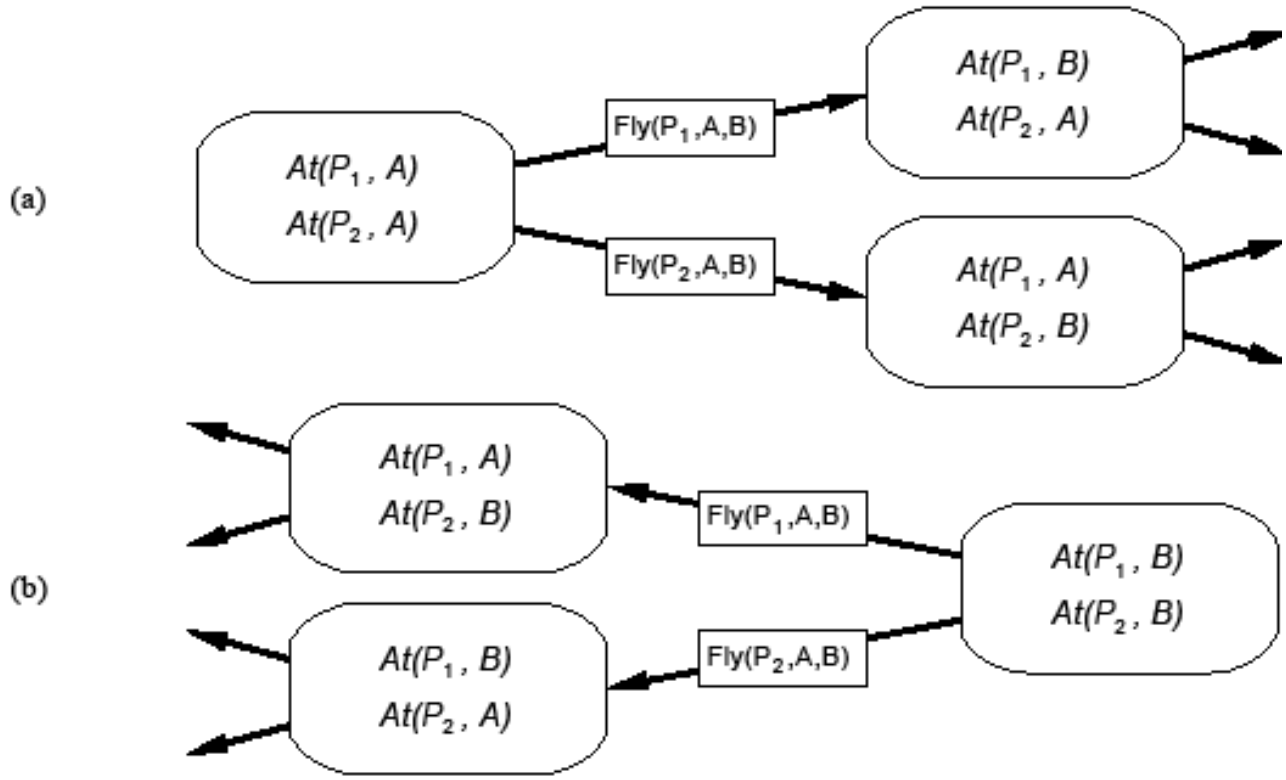
Termination of GRAPHPLAN

- GRAPHPLAN is guaranteed to terminate
  - Literal increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotinically
- A solution is guaranteed not to exist when
  - The graph levels off with all goals present & non-mutex, and
  - EXTRACTSOLUTION fails to find solution

Optimality of GRAPHPLAN

- The plans generated by GRAPHPLAN
  - Are optimal in the number of steps needed to execute the plan
  - Not necessarily optimal in the number of actions in the plan (GRAPHPLAN produces partially ordered plans)

State-Space Search

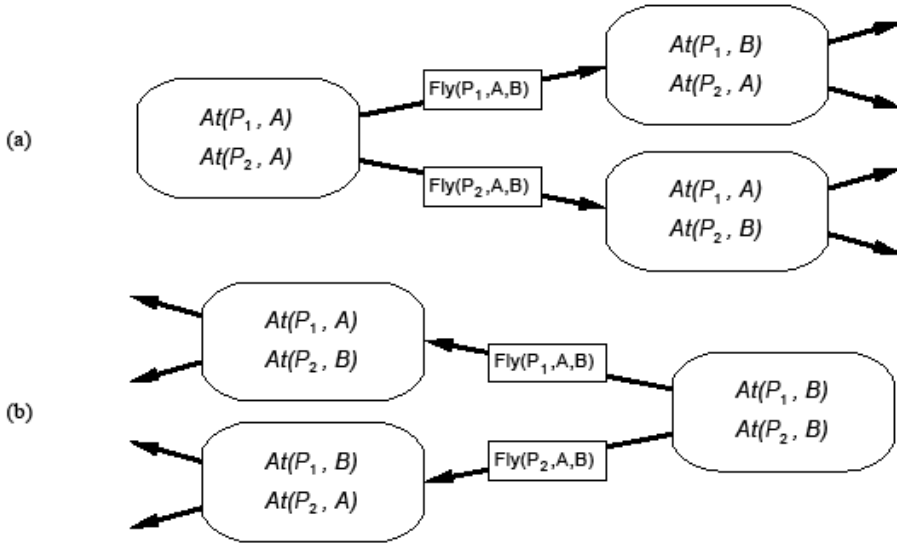


State-Space Search (2)

- Remember that the language has no functions symbols
- Thus number of states is finite
- And we can use any complete search algorithm (e.g., A*)
  - We need an admissible heuristic
  - The solution is a path, a sequence of actions: total-order planning
- Problem: Space and time complexity
  - STRIPS-style planning is PSPACE-complete unless actions have
    - only positive preconditions and
    - only one literal effect

STRIPS in State-Space Search

- STRIPS representation makes it easy to focus on 'relevant' propositions and
  - Work backward from goal (using EFFECTS)
  - Work forward from initial state (using PRECONDITIONS)
  - Facilitating bidirectional search



Heuristics for Planning

- We can use A*, but we need an admissible heuristic
  1. Divide-and-conquer: sub-goal independence assumption
  – Problem relaxation by removing
  2. … all preconditions
  3. … all preconditions and negative effects
  4. … negative effects only: Empty-Delete-List

1. Subgoal Independence Assumption

- The cost of solving a conjunction of subgoals is the sum of the costs of solving each subgoal independently
- Optimistic
  - Where subplans interact negatively
  - Example: one action in a subplan delete goal achieved by an action in another subplan
- Pessimistic (not admissible)
  - Redundant actions in subplans can be replaced by a single action in merged plan

2. Problem Relaxation: Removing Preconditions

- Remove preconditions from action descriptions
  - All actions are applicable
  - Every literal in the goal is achievable in one step
- Number of steps to achieve the conjunction of literals in the goal is equal to the number of unsatisfied literals
- Alert
  - Some actions may achieve several literals
  - Some action may remove the effect of another action

3. Remove Preconditions & Negative Effects

- Considers only positive interactions among actions to achieve multiple subgoals
- The minimum number of actions required is the sum of the union of the actions' positive effects that satisfy the goal
- The problem is reduced to a set cover problem, which is NP-hard
  - Approximation by a greedy algorithm cannot guarantee an admissible heuristic

4. Removing Negative Effects (Only)

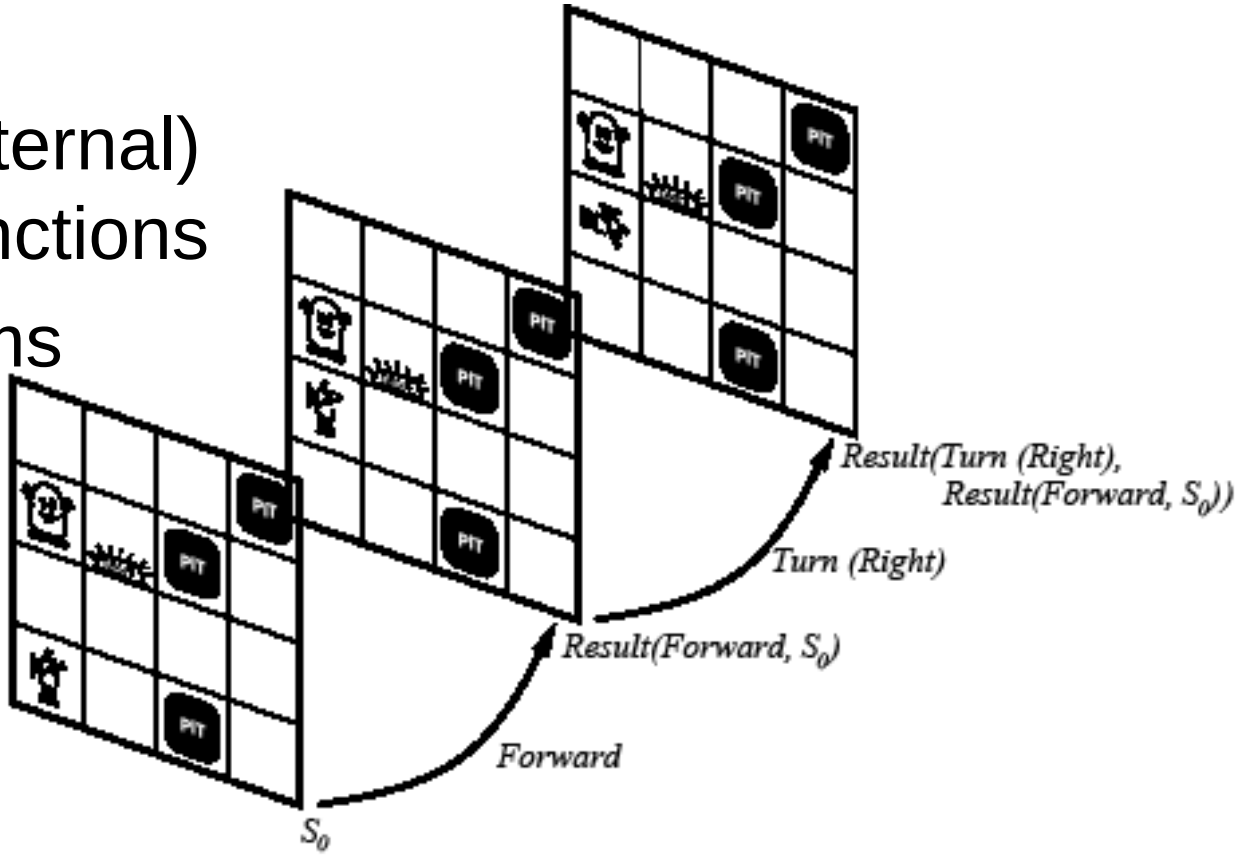
- Remove all negative effects of actions (no action may destroy the effects of another)
- Known as the Empty-Delete-List heuristic
- Requires running a simple planning algorithm
- Quick & effective
- Usable in progression or regression planning

Actions, events, and change

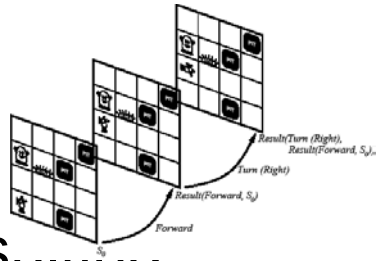
- Planning requires a representation of time
  - to express & reason about sequences of actions
  - to express the effects of actions on the world
- Propositional Logic
  - does not offer a representation for time
  - Each action description needs to be repeated for each step
- Situation Calculus (AIMA Section 10.4.2)
  - Is based on FOL
  - Each time step is a ‘situation’
  - Allows to represent plans and reason about actions & change

Situation Calculus: Ontology

- Situations
- Fluents
- Atemporal (or eternal) predicates & functions
- Possibility axioms



Situation Calculus: Ontology



- Situations
  - Initial state: S_0
  - A function *Result*(*a.s*) gives the situation resulting from applying action *a* in situation *s*
- Fluents
  - Functions & predicates whose truth values can change from one situation to the other
  - Example: $\neg Holding(G_1,S_0)$
- Atemporal (or eternal) predicates and functions
  - Example: $Gold(G_1)$, $LeftLegOf(Wumpus)$
- Possibility axiom
  - Says when an action can be taken: $\Phi(s) \Rightarrow Poss(a,s)$

Situation Calculus

- Sequence of actions
  - Result([],s)=s
  - Result([a|seq],s)=Result(seq,Result(a,s))
- Projection task
  - Deducing the outcome of a sequence of actions
- Planning task
  - Find a sequence of actions that achieves a desired effect

Example: Wumpus World

- Fluents
  - At(o,p,s), Holding(o,s)
- Agent is in [1,1], gold is in [1,2]
  - $At(Agent,[1,1],S_0) \wedge At(G_1,[1,2],S_0)$
- In S_0, we also need to have:
  - $At(o,x,S_0) \Leftrightarrow [(o=Agent) \wedge x=[1,1]] \vee [(o=G_1) \wedge x=[1,2]]$
  - $\neg Holding(o,S_0)$
  - $Gold(G_1) \wedge Adjacent([1,1],[1,2]) \wedge Adjacent([1,2],[1,1])$
- The query is:
  - $\exists$ seq $At(G_1,[1,1],Result(seq,S_0))$
- The answer is
  - $At(G1,[1,1],Result(Go([1,1],[1,2]),Grab(G_1),Go([1,2],[1,1]),S_0))$

Importance of Situation Calculus

- Historical note
  - Situation Calculus was the first attempt to formalizing planning in FOL
  - Other formalisms include Event Calculus
  - The area of using logic for planning is informally called in the literature “Reasoning About Action & Change”
- Highlighted three important problems
  1. Frame problem
  2. Qualification problem
  3. Ramification problem

‘Famous’ Problems

- Frame problem
  - Representing all things that stay the same from one situation to the next
  - Inferential and representational
- Qualification problem
  - Defining the circumstances under which an action is guaranteed to work
  - Example: what if the gold is slippery or nailed down, etc.
- Ramification problem
  - Proliferation of implicit consequences of actions as actions may have secondary consequences
  - Examples: How about the dust on the gold?

Summary of Planning Graphs

♢ Yield useful heuristics of state-space and partial order planners

♢ Consists of multiple layers of literals and actions that can occur at each time step

♢ Includes mutex relations to exclude co-occurrences

♢ Plan can be extracted directly from graph

Summary

♢ Planning systems operate on explicit representations of states and actions

♢ STRIPS language describes actions in terms of preconditions and effects.

♢ Partial-order planning (POP) algorithms explore space of plans without committing to a totally ordered sequence of actions.

♢ POP algorithms work backwards from goal, and are particularly effective on problems amenable to divide-and-conquer.

♢ No consensus on any specific planning approach being the best.