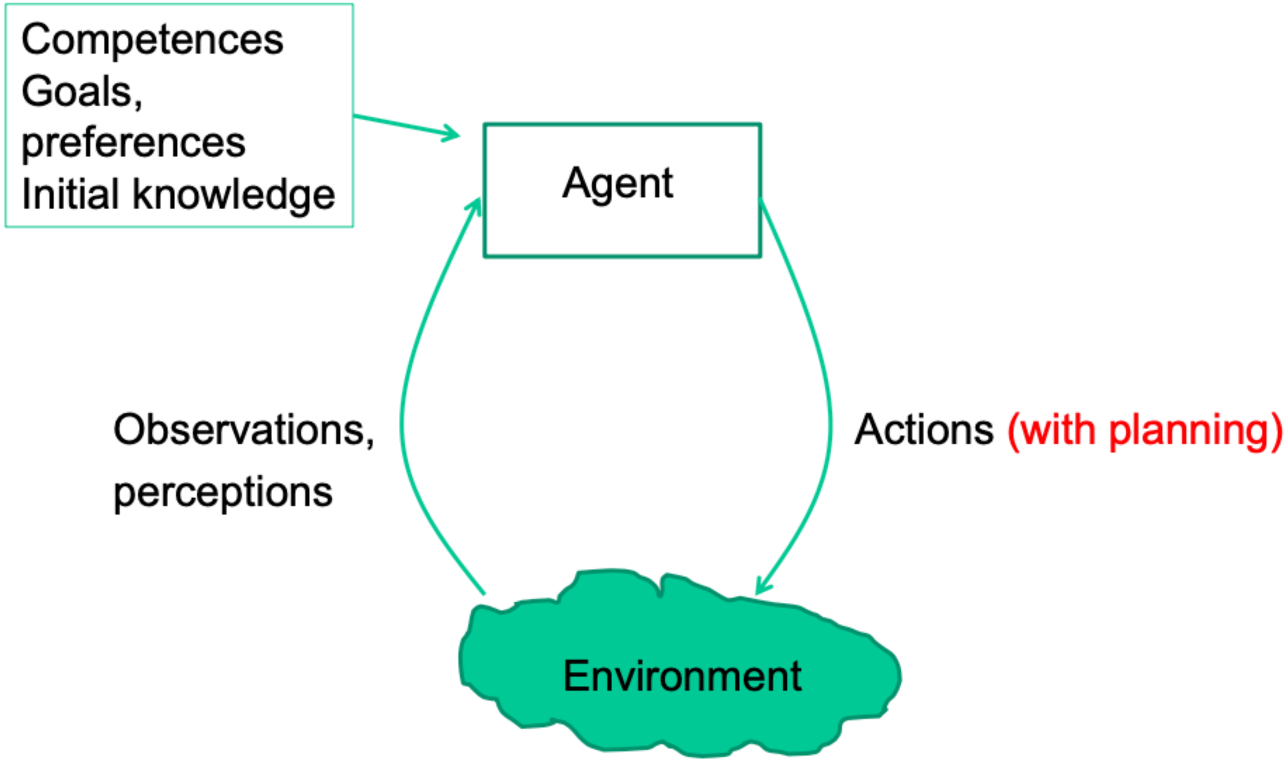
Planning

COMP3411/9814: Artificial Intelligence

Lecture Overview

- Reasoning About Action
- STRIPS Planner
- Forward planning
- Regression Planning
- Partial Order Planning
- GraphPlan
- Planning as Constraint Satisfaction

Agent acting in its environment



Competences
Goals,
preferences
Initial knowledge

Agent

Observations,
perceptions

Actions (with planning)

Environment

Planning

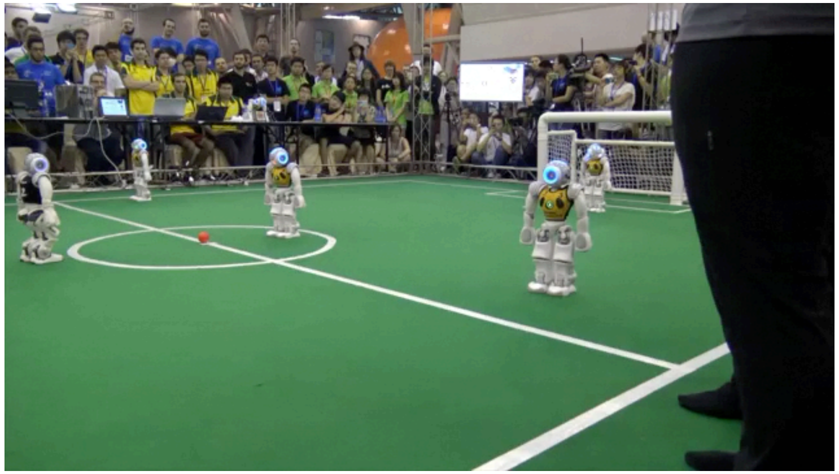
- Planning is deciding what to do based on an agent’s ability, its goals. and the state of the world.

- Planning is finding a sequence of actions to solve a goal.

- Assumptions:
  - World is deterministic.
  - No exogenous events outside of control of robot change state of world.
  - The agent knows what state it is in.
  - Time progresses discretely from one state to the next.
  - Goals are predicates of states that need to be achieved or maintained.

Planning Agent

- The planning agent or goal-based agent is more flexible than a reactive agent because the knowledge that supports its decisions is represented explicitly and can be modified.
- The agent's behaviour can easily be changed.
- Doesn't work when assumptions are violated



Planning Agent

- Environment changes due to the performance of actions
- Planning scenario
  - Agent can control its environment
  - Only atomic actions, not processes with duration
  - Only single agent in the environment (no interference)
  - Only changes due to agent executing actions (no evolution)

- More complex examples
  - RoboCup soccer
  - Delivery robot
  - Self-driving car

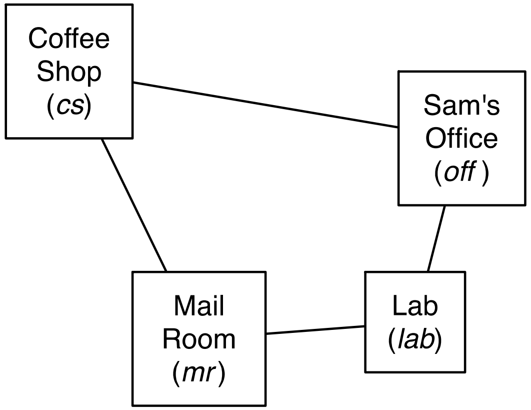
Representation

- How to represent a classical planning problem?
  - States, Actions, and Goals
- Can represent relation between states and actions
  - explicit state space representation
  - action-centric
  - feature-based

Actions

- A deterministic action is a partial function from states to states.
- The preconditions of an action specify when the action can be carried out.
- The effect of an action specifies the resulting state.

Delivery Robot Example



Coffee Shop (*cs*)

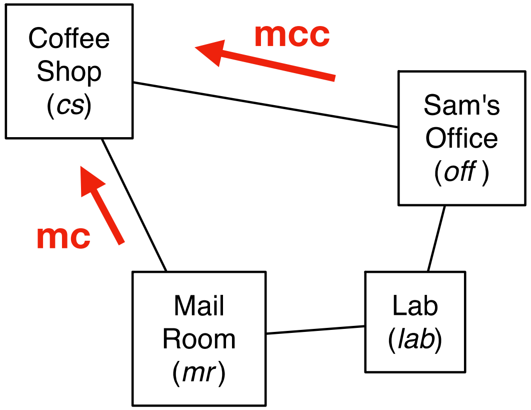
Sam's Office (*off*)

Mail Room (*mr*)

Lab (*lab*)

The robot, called Rob, can buy coffee at the coffee shop, pick up mail in the mail room, move, and deliver coffee and/or mail.

Delivery Robot Example



Features:
RLoc **– Rob’s location**
RHC **– Rob has coffee**
SWC **– Sam wants coffee**
MW **– Mail is waiting**
RHM **– Rob has mail**

Features to describe states

Actions:
mc **– move clockwise**
mcc **– move counterclockwise**
puc **– pickup coffee**
dc **– deliver coffee**
pum **– pickup mail**
dm **– deliver mail**

Robot actions

State Description

The state is described in terms of the following features:

- RLoc - the robot's location,
  - coffee shop (*cs*), Sam's office (*off*), the mail room (*mr)* or laboratory (*lab*)
- SWC - Sam wants coffee.
  - The atom *swc* means Sam wants coffee and ¬ *swc* means Sam does not want coffee.

Robot Actions

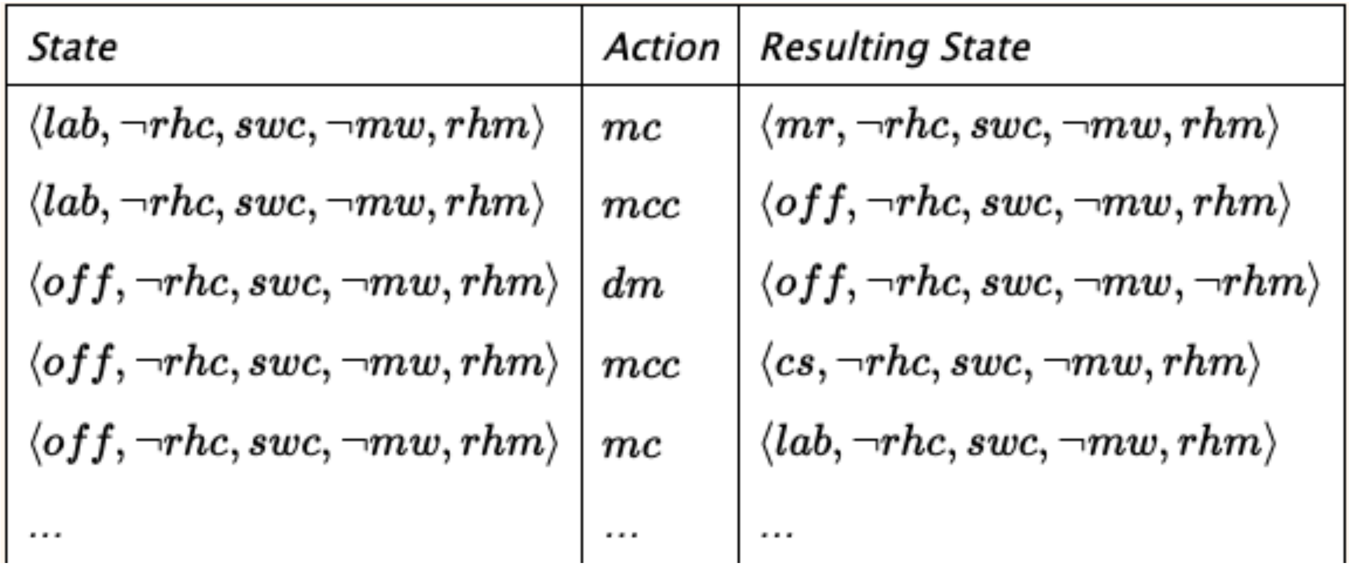
- Rob has six actions
  - Rob can move clockwise (*mc*)
  - Rob can move counterclockwise (*mcc*) or (*mac*), for now we use (*mcc*).
  - Rob can pick up coffee (puc) if Rob is at the coffee shop.
  - Rob can deliver coffee (dc) if Rob has coffee and is in the same location as Sam.
  - Rob can pick up mail (pum) if Rob is in the mail room.
  - Rob can deliver mail (dm) if Rob has mail and is in the same location as Sam.
- Assume that it is only possible for Rob to do one action at a time.

Explicit State-Space Representation

- The states are specifying the following:
  - the robot's location,
  - whether the robot has coffee,
  - whether Sam wants coffee,
  - whether mail is waiting,
  - whether the robot is carrying the mail.

$$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$$

Explicit State-Space Representation



State	*Action*	*Resulting State*
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	mc	$\langle mr, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	mcc	$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	dm	$\langle off, \neg rhc, swc, \neg mw, \neg rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	mcc	$\langle cs, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	mc	$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$
...	...	...

The complete representation includes the transitions for the other 62 states.

Explicit State-Space Representation

This is not a good representation:

- There are usually too many states to represent, to acquire, and to reason with.
- Small changes to the model mean a large change to the representation.
  - Adding another feature means changing the whole representation.
- It does not represent the structure of states;
  - there is much structure and regularity in the effects of actions that is not reflected in the state transitions.

STRIPS Language for Problem Definition

- STRIPS = Stanford Research Institute Problem Solver

- Most planners use a “STRIPS-like representation”
  - i.e. STRIPS with some extensions

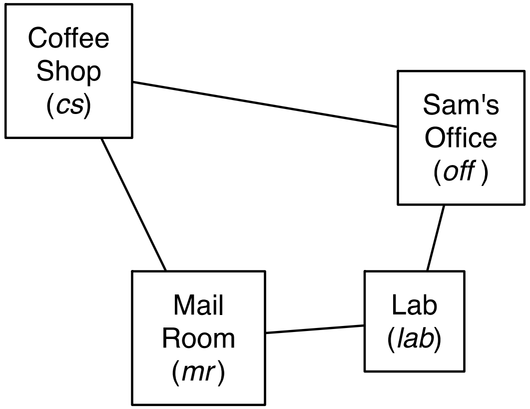
- STRIPS makes some simplifications:
  - no variables in goals
  - positive relations given only
  - unmentioned relations are assumed false (c.w.a. – closed world assumption)
  - effects are conjunctions of relations

STRIPS Representation

- Each action has a:
  - precondition that specifies when the action can be carried out.
  - effect a set of assignments of values to primitive features that are made true by this action.
    - Often split into an ADD list (things that become true after action)
    - and DELETE list (things that become false after action)

Assumption: every primitive feature not mentioned in the effects is unaffected by the action.

Example STRIPS Representation



Pick-up coffee (puc):

- precondition: [cs, ¬rhc]
- effect: [rhc]

Deliver coffee (dc):

- precondition: [off, rhc]
- effect: [¬rhc, ¬swc]

Feature-Based Representation of Actions

- For each action:
  - precondition is a proposition that specifies when the action can be carried out.

- For each feature:
  - causal rules that specify when the feature gets a new value and
  - frame rules that specify when the feature keeps its value.

Example Feature-Based Representation

- Causal rules say how features change
- Frame rules say how features stay the same

- Precondition of pick-up coffee (puc):

  $RLoc=cs \land \neg rhc$

- Rules for “location is cs”:

  causal rules

  $RLoc'=cs \leftarrow Rloc = off \land Act=mcc$

  $RLoc'=cs \leftarrow Rloc = mr \land Act=mc$

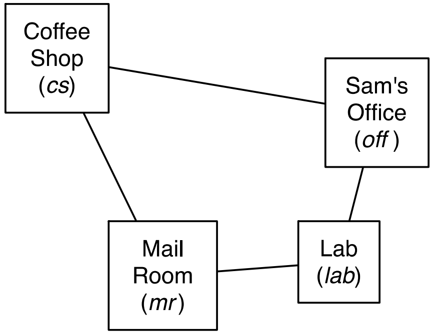
  frame rule

  $RLoc'=cs \leftarrow Rloc = cs \land Act \neq mcc \land Act \neq mc$

- Rules for “robot has coffee”

  $rhc' \leftarrow Act=puc \land \neg rhc$

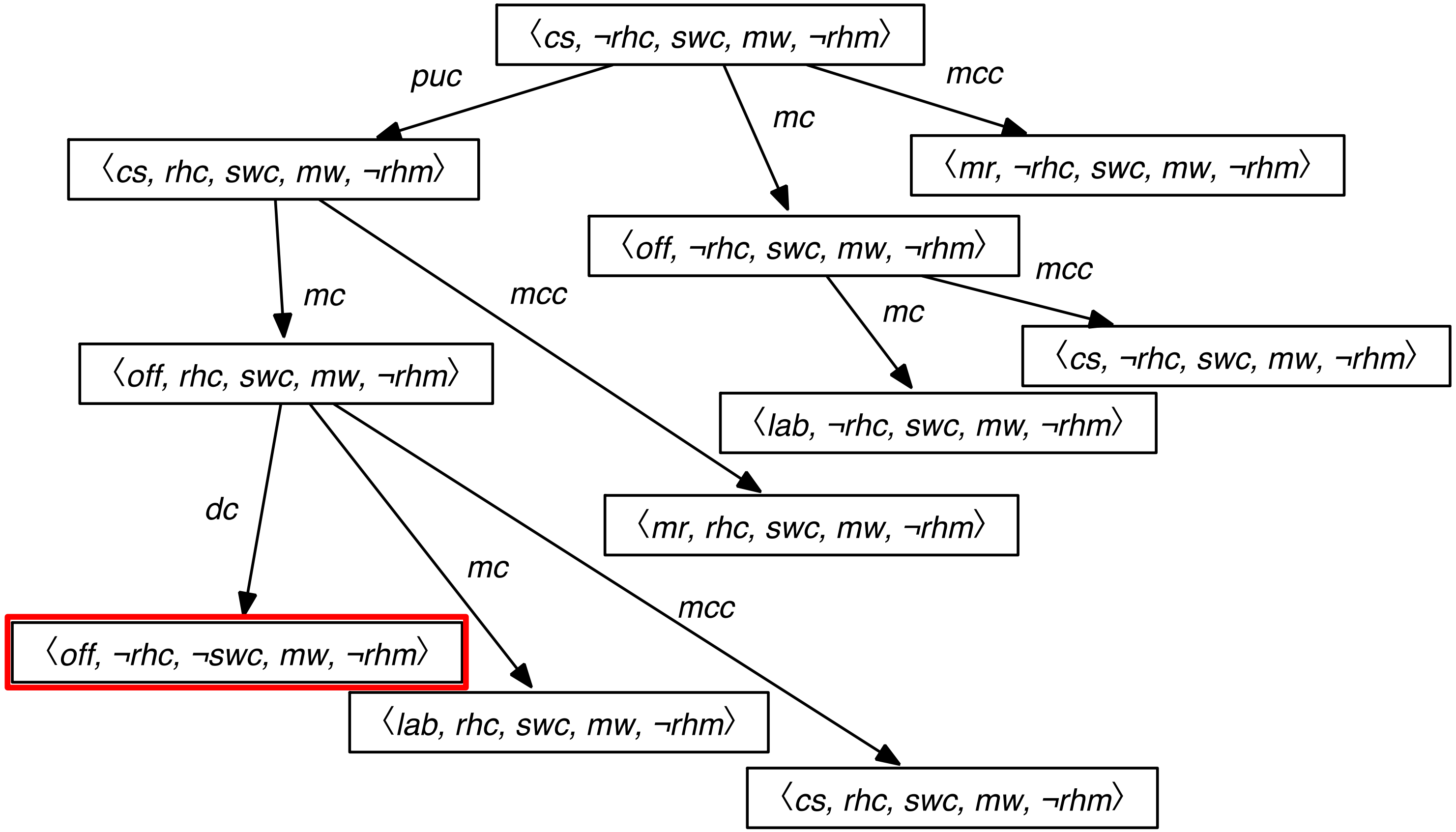
  $rhc' \leftarrow rhc \land Act \neq dc$



Forward Planning

- Nodes are states in the world
- Arcs correspond to actions that transform one state into another
- Start node is the initial state
- If *goal condition* is satisfied, search terminates successfully
- A path corresponds to a plan to achieve goal

Forward Search



⟨cs, ¬rhc, swc, mw, ¬rhm⟩

puc

mc

mcc

⟨cs, rhc, swc, mw, ¬rhm⟩

⟨mr, ¬rhc, swc, mw, ¬rhm⟩

⟨off, ¬rhc, swc, mw, ¬rhm⟩

mc

mcc

mcc

mc

⟨cs, ¬rhc, swc, mw, ¬rhm⟩

⟨off, rhc, swc, mw, ¬rhm⟩

⟨lab, ¬rhc, swc, mw, ¬rhm⟩

dc

⟨mr, rhc, swc, mw, ¬rhm⟩

mc

mcc

⟨off, ¬rhc, ¬swc, mw, ¬rhm⟩

⟨lab, rhc, swc, mw, ¬rhm⟩

⟨cs, rhc, swc, mw, ¬rhm⟩

Actions:
mc **– move clockwise**
mcc **– move counterclockwise**
puc **– pickup coffee**
dc **– deliver coffee**
pum **– pickup mail**
dm **– deliver mail**

Locations:
cs **– coffee shop**
off **– office**
lab **– laboratory**
mr **– mail room**

Features:
RLoc **– Rob's location**
RHC **– Rob has coffee**
SWC **– Sam wants coffee**
MW **– Mail is waiting**
RHM **– Rob has mail**

Initial:
SWC **– Sam wants coffee**

Goal:
¬SWC **– Sam wants coffee**

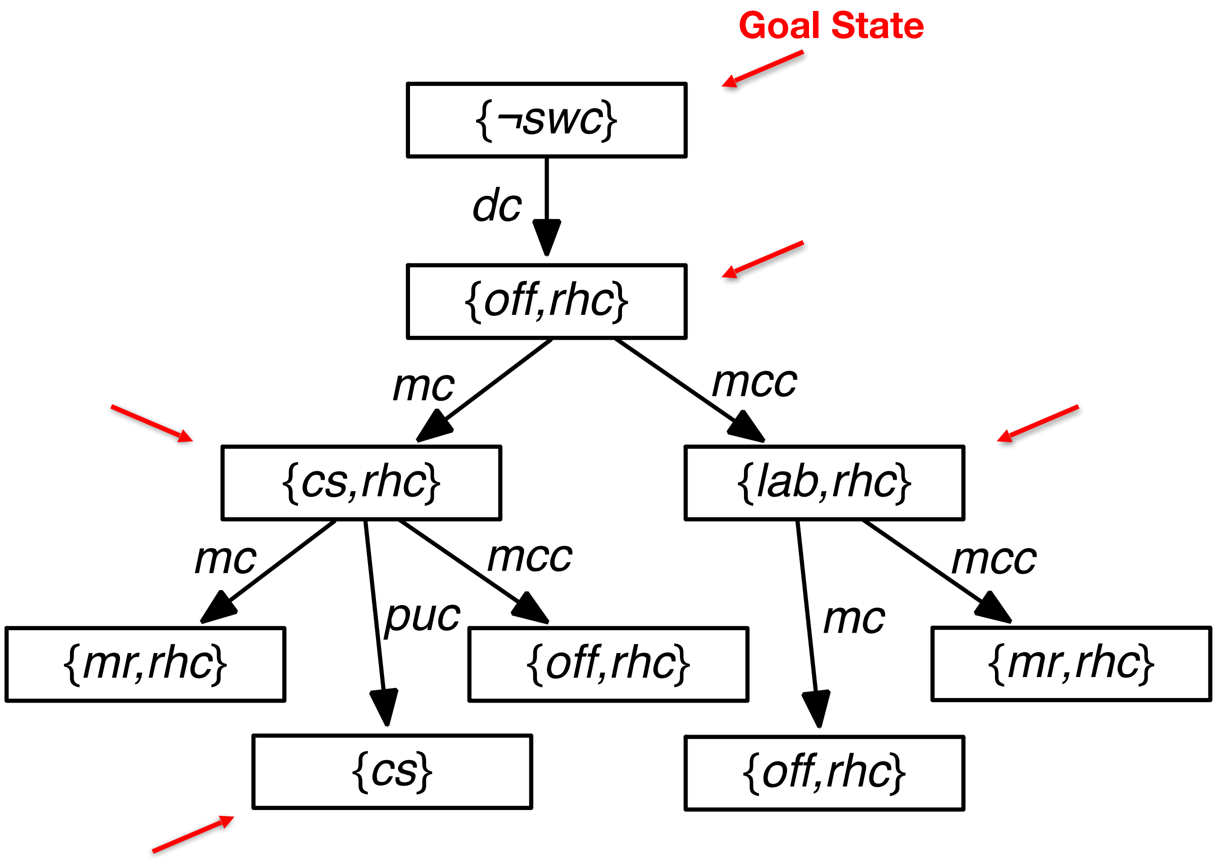
Recall our Prolog Planner

```
% State of the robot's world = state(RobotLocation, BasketLocation, RubbishLocation)
% action(Action, State, NewState): Action in State produces NewState
% We assume robot never drops rubbish on floor and never pushes rubbish around

action(pickup,                                          % Pick up rubbish from floor
       state(Pos1, Pos2, floor(Pos1)),                  % Before action, robot and rubbish both at Pos1
       state(Pos1, Pos2, held)).                        % After action, rubbish held by robot

action(drop,                                            % Drop rubbish into basket
       state(Pos, Pos, held),                           % Before action, robot and basket both at Pos
       state(Pos, Pos, in_basket)).                     % After action, rubbish in basket

action(push(Pos, NewPos),                               % Push basket from Pos to NewPos
       state(Pos, Pos, Loc),                            % Before action, robot and basket both at Pos
       state(NewPos, NewPos, Loc)).                     % After action, robot and basket at NewPos

action(go(Pos1, NewPos1),                               % Go from Pos1 to NewPos1
       state(Pos1, Pos2, Loc),                          % Before action, robot at Pos1
       state(NewPos1, Pos2, Loc)).                      % After action, robot at Pos2

% plan(StartState, FinalState, Plan)

plan(State, State, []).                                 % to achieve State from State itself, do nothing
plan(State1, GoalState, [Action1 | RestofPlan]) :-
       action(Action1, State1, State2),                 % Make first action resulting in State2
       plan(State2, GoalState, RestofPlan).             % Find rest of plan
```

Regression Planning
(Backward Search)

- Nodes are subgoals.
- Arcs correspond to actions. An arc from node *g* to *g′*, labelled with action *act*, means
  - *act* is the last action that is carried out before subgoal *g* is achieved, and
  - node *g′* is a subgoal that must be true immediately before *act* so that *g* is true immediately after *act*.
- The start node is the planning goal to be achieved.
- The goal condition for the search, goal(g), is true if g is true of the initial state.

Regression Planning



Actions:
mc **– move clockwise**
mcc **– move counterclockwise**
puc **– pickup coffee**
dc **– deliver coffee**
pum **– pickup mail**
dm **– deliver mail**

Locations:
cs **– coffee shop**
off **– office**
lab **– laboratory**
mr **– mail room**

Features:
RLoc **– Rob's location**
RHC **– Rob has coffee**
SWC **– Sam wants coffee**
MW **– Mail is waiting**
RHM **– Rob has mail**

Backward Regression

$$g' = (g - Add(a)) \cup Precond(a)$$

- g' is the regression from goal g over action a
- I.e. going backwards from g, we look for an action, a, that has preconditions and effects that satisfy g'

Backward Chaining

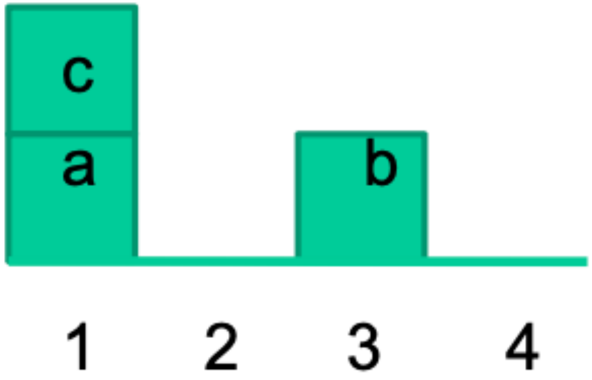
```
% State of the robot's world = state(RobotLocation, BasketLocation, RubbishLocation)
% action(Action, State, NewState): Action in State produces NewState
% We assume robot never drops rubbish on floor and never pushes rubbish around

plan_backwards(State, State, []).                          % To achieve State from State itself, do nothing
plan_backwards(State1, GoalState, [Action | RestofPlan]) :-    % Actions will be in reverse order
      action(Action, PreviousState, GoalState),            % Find an action tat achieves GoalState
      plan_backwards(State1, PreviousState, RestofPlan).    % Make PreviousState the new goal


id_plan_backwards(Start, Goal, Plan) :-
      append(RevPlan, _, _),
      plan_backwards(Start, Goal, RevPlan),
      reverse(RevPlan, Plan).

?- id_plan_backwards(state(door, corner, floor(middle)), state(_, _, in_basket), Plan).
X = [go(door, corner), push(corner, middle), pickup, drop]
```

Relational State Representation

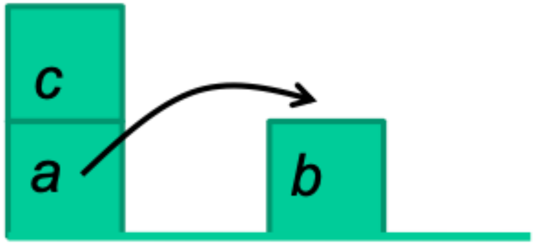
First-order representations are more flexible

- e.g. states in blocks world can be represented by set of relations
- 1, 2, 3, 4 represent positions on a table:

on(c, a).
on(a, 1).
on(b, 3).
clear(2).
clear(4).
clear(b).
clear(c).



Defining Goals and Possible Actions



- Example of goals:

  on(a, b), on(b, c)

- Example of action:

  move(a, 1, b)
  (Move block a from 1 to b)

- Action preconditions:

  clear(a), on(a, 1), clear(b)

- Action effects:

  on(a, b), clear(1), ¬ on(a, 1), ¬ clear(b)

“add” (true after action)

“delete” (no longer true after action)

STRIPS Action Schema

Action schema represents a set of actions using variables
(variable names start with capital letter, like Prolog)

Action:

move(Block, From, To)

Precondition:

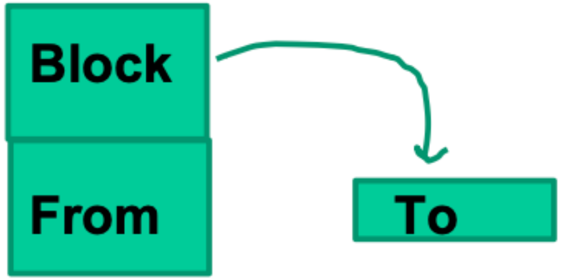
clear(Block), clear(To), on(Block, From)

Adds:

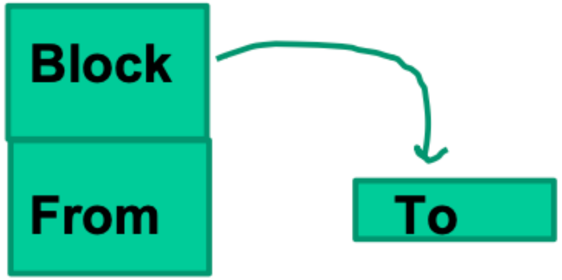
on(Block, To), clear(From)

Deletes:

on(Block, From), clear(To)



Better with Additional Constraints



Action:

move(Block, From, To)

Precondition for Action:

clear(Block), clear(To), on(Block, From)

Additional constraints:

block(Block),	% Object Block to be moved must be a block
object(To),	% "To" is an object, i.e. a block or a place
To $\neq$ Block,	% Block cannot be moved to itself
object(From),	% "From" is a block or a place
From $\neq$ To,	% Move to new position
Block $\neq$ From	

PDDL
Planning Domain Description Language

- Extension of STRIPS representation
- Invented for planning competitions to provide an implementation independent language for describing action schema and domain knowledge
- There are several variants to cover different planning domains
  - e.g. continuous domains, continuous actions, probabilities, etc.

Example

Init: airport(mel) ∧ airport(syd) ∧ plane(p1) ∧ plane(p2) ∧ cargo(c1) ∧ cargo(c2) ∧ at(c1, syd) ∧ at(c2, mel) ∧ at(p1, syd) ∧ at(p2, mel)

Goal: at(c1, mel) ∧ at(c2, syd)

Action load(C, P, A)

PRECOND: cargo(C) ∧ plane(P) ∧ airport(A) ∧ at(C, A) ∧ at(P, A)

EFFECT: ¬ at(C, A) ∧ in(C, P)

Action unload(C, P, A)

PRECOND: cargo(C) ∧ plane(P) ∧ airport(a) ∧ In(C, P) ∧ at(P, A)

EFFECT: at(C, A) ∧ ¬ in(C, P)

Action fly(P, From, To)

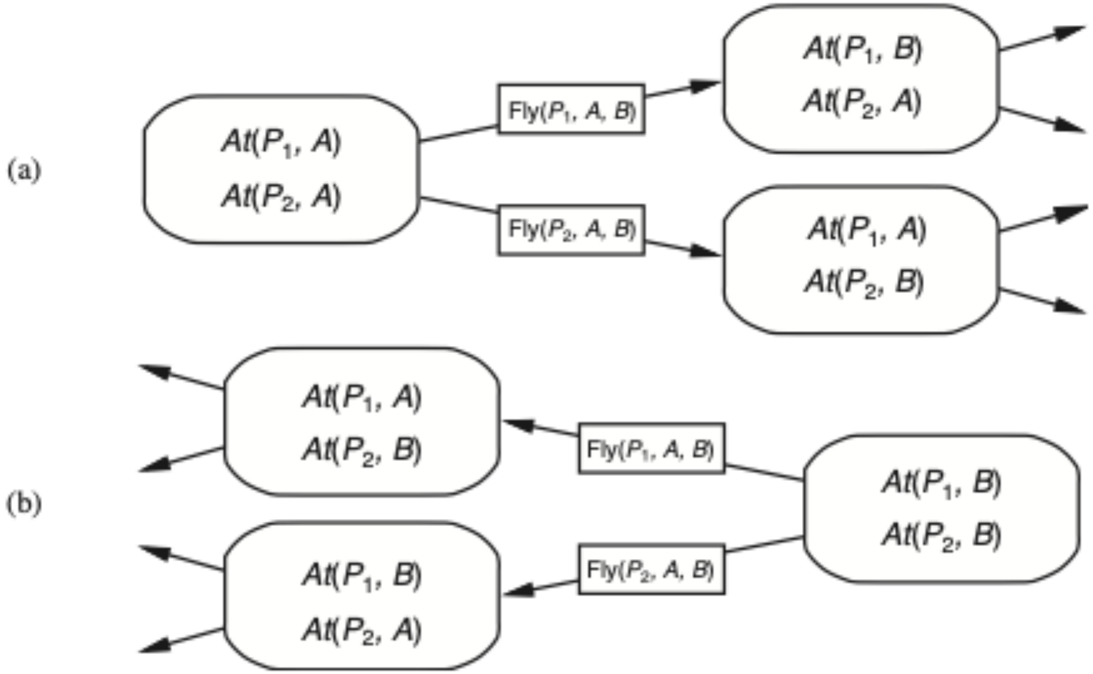
PRECOND: plane(P) ∧ airport(From) ∧ airport(To) ∧ at(P, From)

EFFECT: ¬ at(P, From) ∧ at(P, To)

load(c1, p1, syd)
fly(p1, syd, mel)
unload(c1, p1, mel)
load(c2, p2, mel)
fly(p2, mel, syd)
unload(c2, p2, syd)

Simple Planning Algorithms

Forward search and goal regression



Sussman’s Anomaly

- Goal: on(a, b) ∧ on(b, c)
- Try achieving on(a, b) first

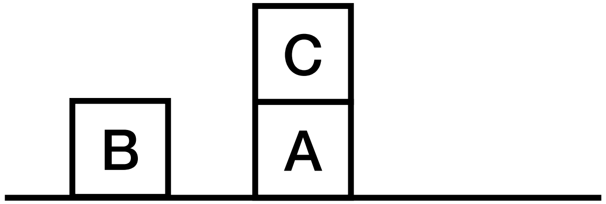
  [move(c,a,floor), move(a,floor,b),
  move(a,b,floor), move(b,floor,c)]

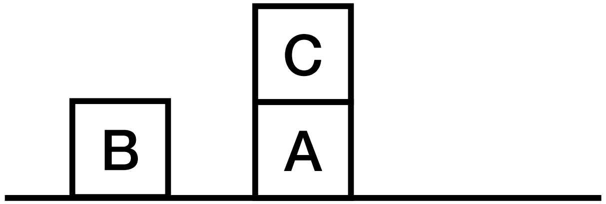
- Trying on(b, c) first

  [move(b,floor,c), **move(b,c,floor)**,
  move(c,a,floor), move(a,floor,b)]

- Should be:

  [move(c,a,floor), move(b,floor,c), move(a,floor,b)]





WARPLAN

Warren, D. H. D. (1974). *Warplan: A system for generating plans.*
Memo No. 76, Department of Computational Logic, University of Edinburgh.

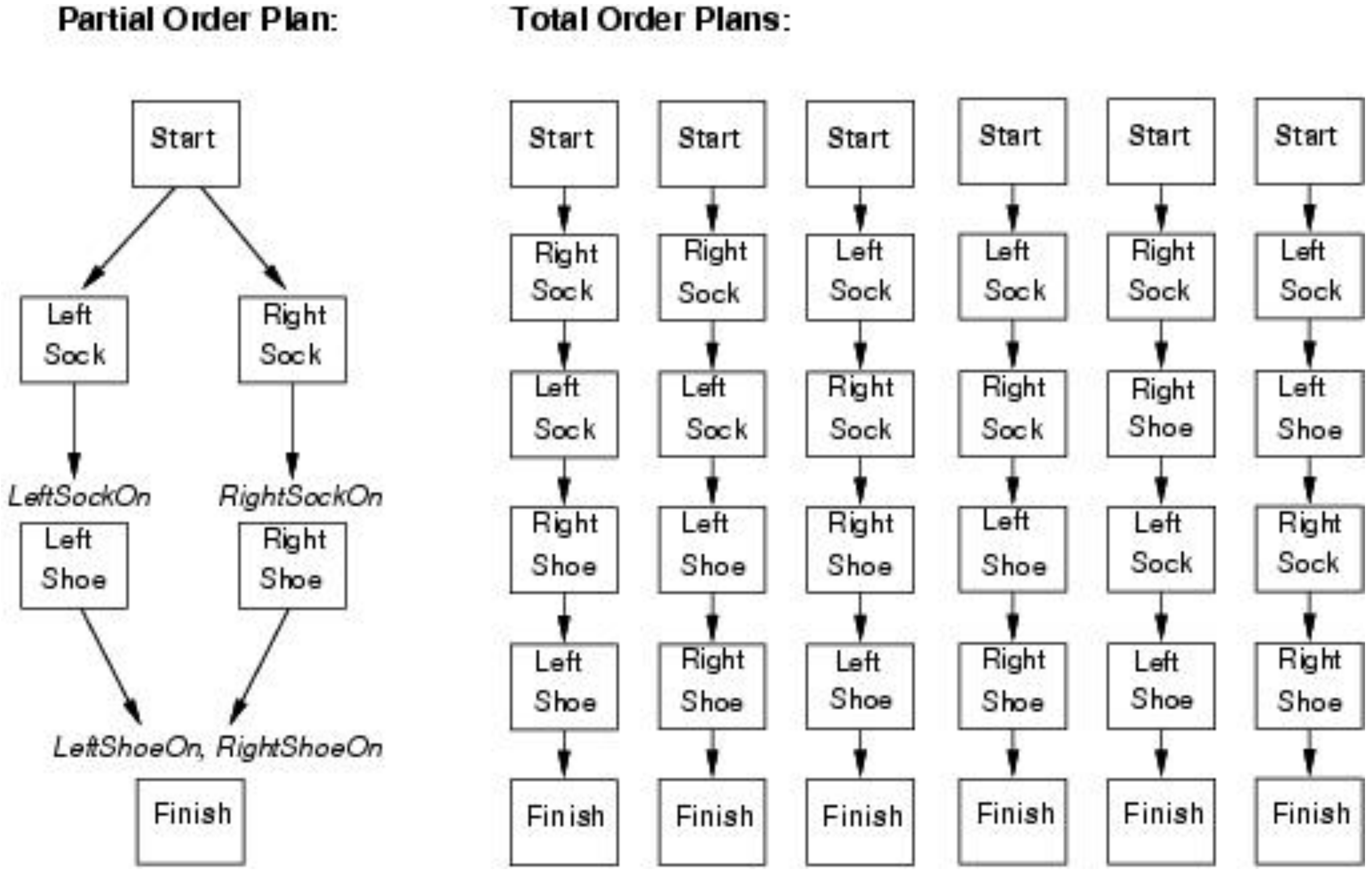
- Protect goals once achieved
- If an action undoes a goal
  - try moving new action backwards through plan before the action that achieved first goal
  - check that goals before and after are preserved

[move(c,a,floor), move(a,floor,b), **move(a,b,floor)**, ..]

[move(c,a,floor), .., move(a,floor,b)]

Try inserting plan for on(b,c) here

Partially Ordered Plans



Partial-Order Planning

Init: Tire(Flat) ∧ Tire(Spare) ∧ at(Flat, Axle) ∧ at(Spare, Boot)

Goal: at (Spare, Axle)

Action Remove(obj, loc)

PRECOND: at(obj, loc)

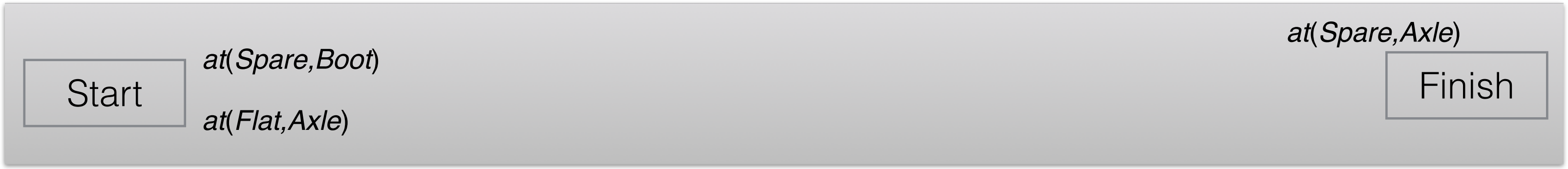
EFFECT: ¬ at(obj, loc) ∧ at(obj, Ground)

Action PutOn(t, Axle)

PRECOND: Tire(t) ∧ at(t, Ground) ∧ ¬ at(Flat, Axle)

EFFECT: ¬ at(t, Ground) ∧ at(t, Axle)

Partial-Order Planning



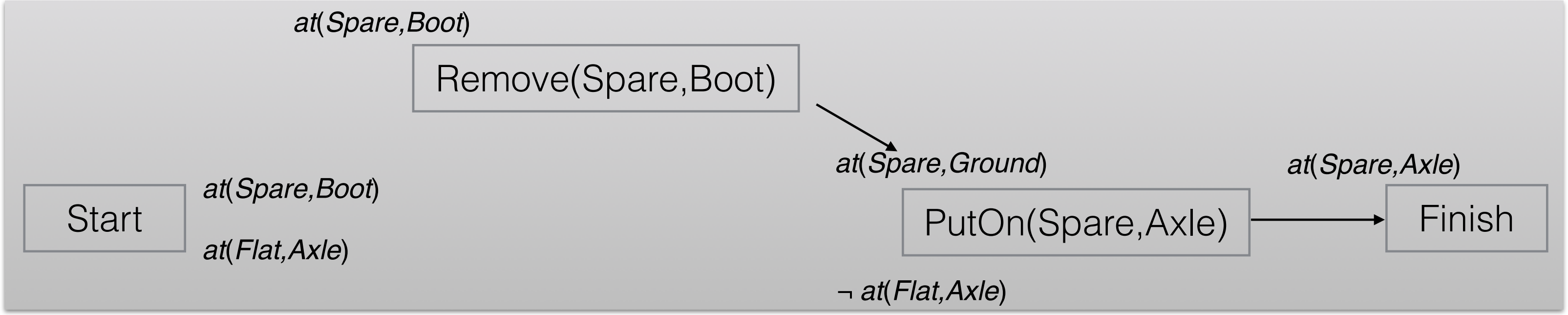
Start

at(*Spare*,*Boot*)

at(*Flat*,*Axle*)

at(*Spare*,*Axle*)

Finish



at(*Spare*,*Boot*)

Remove(Spare,Boot)

at(*Spare*,*Ground*)

at(*Spare*,*Axle*)

Start

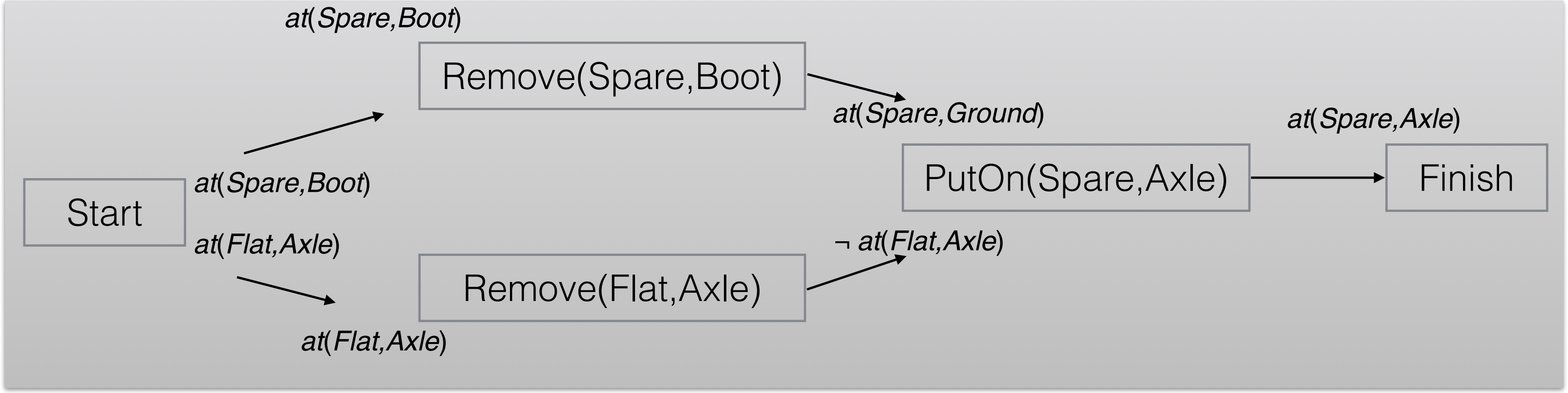
at(*Spare*,*Boot*)

at(*Flat*,*Axle*)

PutOn(Spare,Axle)

Finish

¬ *at*(*Flat*,*Axle*)



at(*Spare*,*Boot*)

Remove(Spare,Boot)

at(*Spare*,*Ground*)

at(*Spare*,*Axle*)

Start

at(*Spare*,*Boot*)

at(*Flat*,*Axle*)

PutOn(Spare,Axle)

Finish

¬ *at*(*Flat*,*Axle*)

Remove(Flat,Axle)

at(*Flat*,*Axle*)

Forward Planning

- Forward planners are now among the best.
- Use heuristics to estimate costs
- Possible to use heuristic search, like A*, to reduce search

Planning Graphs

- Use constraint solving to achieve better heuristic estimates
- Only for propositional problems
- Like consistency checking in CSP
  - preprocess constraints to create a planning graph
  - planning graph constrains possible states and actions
- Planning graph is NOT a plan
  - It constrains the range of possible plans

Planning Graph

- A sequence of levels that correspond to time steps in plan
    - Level 0 is initial state
- Each level consists of:
    - Set of all literals that could be true at that time step
        - depending on actions executed in preceding time step
    - Set of all actions that could have their preconditions satisfied at that time step
        - depending on which literals are true

Mutual Exclusion

- Actions
  - Inconsistent effects: One action negates an effect of the other
  - Competing needs: Precondition of one action is mutually exclusive with a precondition of the other

- Literals
  - One literal is the negation of the other
  - Inconsistent support: Each possible pair of actions that could achieve the two literals is mutually exclusive

Example

Init: Have (Cake)

Goal: Have(Cake) ∧ Eaten(Cake)

Action: Eat (Cake)

PRECOND: Have(Cake)

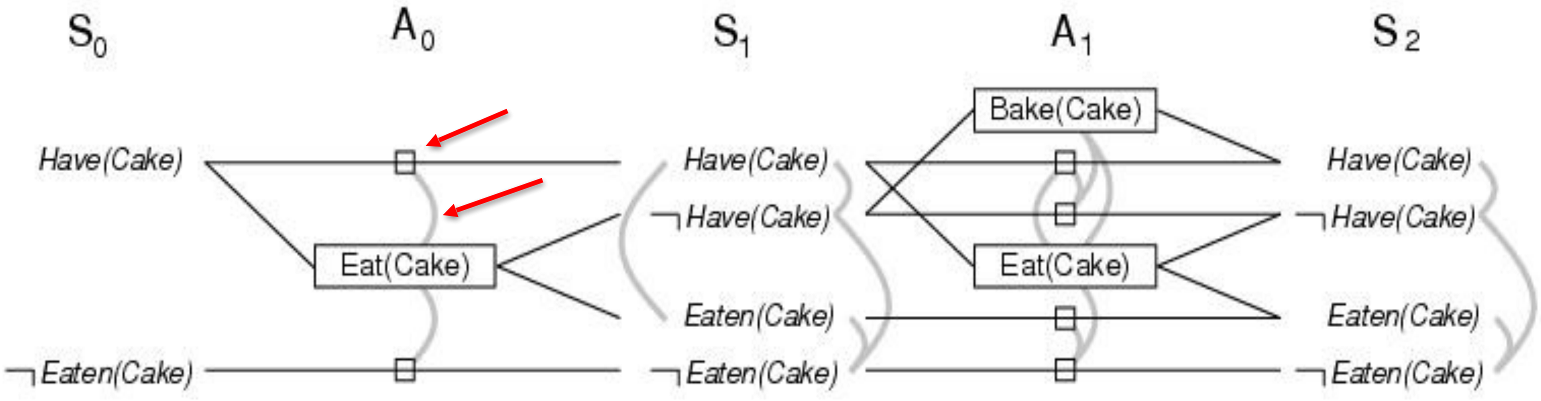
EFFECT: ¬ Have(Cake) ∧ Eaten(Cake)

Action: Bake (Cake)

PRECOND: ¬ Have(Cake)

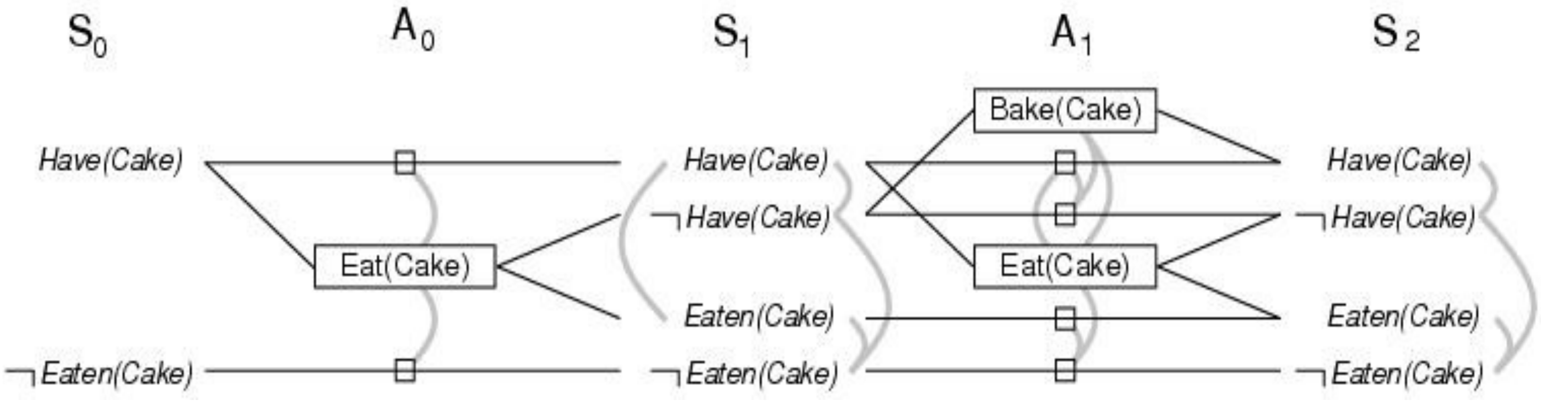
EFFECT: Have(Cake)

Cake Example



- Start at level S_0 and determine action level A_0 and next level S_1.
  - A_0 – all actions whose preconditions are satisfied in the previous level.
  - Connect precondition and effect of actions $S_0 \rightarrow S_1$
  - Inaction is represented by persistent actions
- Level A_0 contains the actions that could occur
  - Conflicts between actions are represented by mutex links

Cake Example



- Level S_1 contains all literals that could result from picking any subset of actions in A_0
    - Conflicts between literals that cannot occur together are represented by mutex links.
    - S_1 defines multiple states and the mutex links are the constraints that define this set of states.
- Continue until goal is satisfied in level S_i, or no change in consecutive levels: levelled off

Planning Graphs and Heuristic Estimation

- Planning Graphs provide information about the problem
  - A literal that does not appear in final level of graph cannot be achieved by any plan.
    - Useful for backward search (cost = inf).
  - Level of appearance can be cost estimate of achieving goal literals = level cost.
  - Cost of a conjunction of goals:
    - max-level, sum-level and set-level heuristics.

Example: Spare tire problem

Init: at(Flat, Axle) ∧ at(Spare,Trunk)

Goal: at(Spare,Axle)

Action: Remove(Spare,Trunk)

PRECOND: at(Spare,Trunk)

EFFECT: ¬at(Spare,Trunk) ∧ at(Spare,Ground))

Action: Remove(Flat,Axle)

PRECOND: at(Flat,Axle)

EFFECT: ¬at(Flat,Axle) ∧ at(Flat,Ground))

Action: PutOn(Spare,Axle)

PRECOND: at(Spare,Ground) ∧¬at(Flat,Axle)

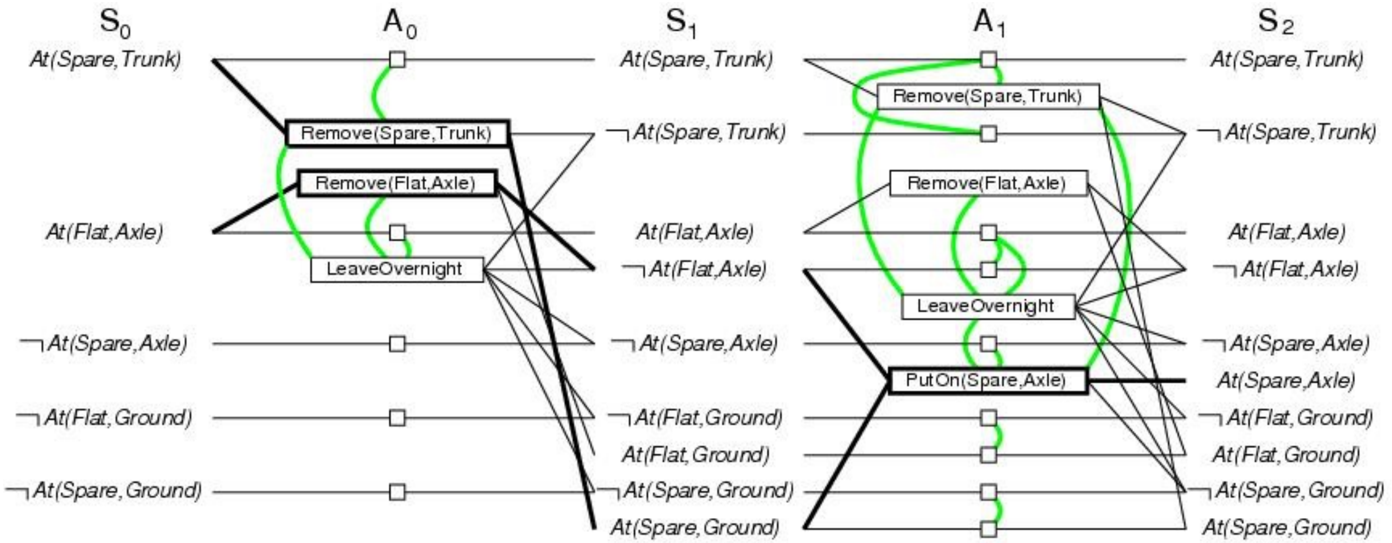
EFFECT: at(Spare,Axle) ∧ ¬at(Spare,Ground))

Action: LeaveOvernight

PRECOND:

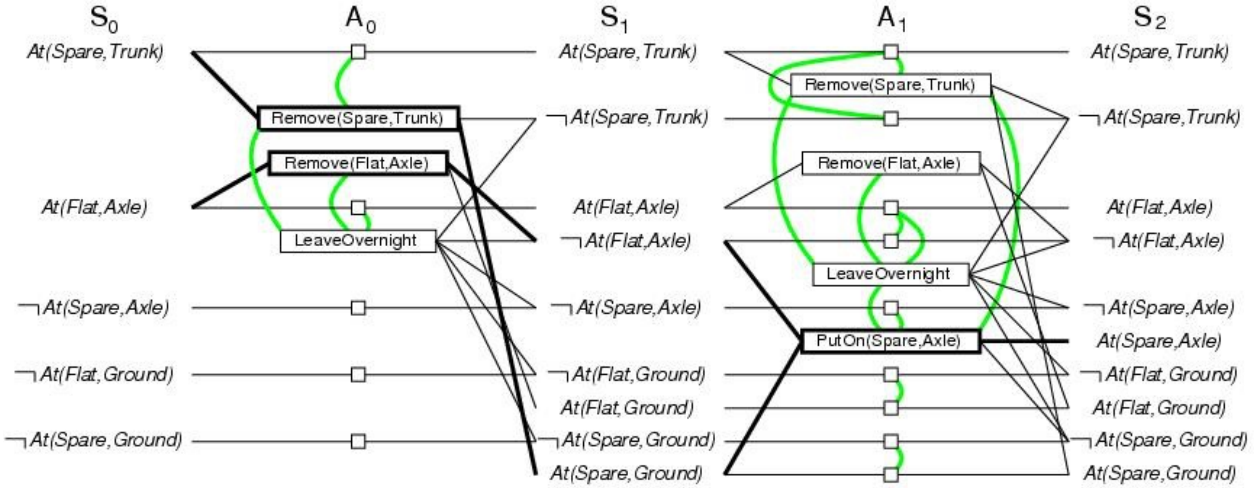
EFFECT: at(Flat,Axle) ∧ at(Spare,trunk) ∧ ¬ at(Spare,Ground) ∧ ¬ at(Spare,Axle) ∧ ¬ at(Flat,Ground)

GRAPHPLAN Example



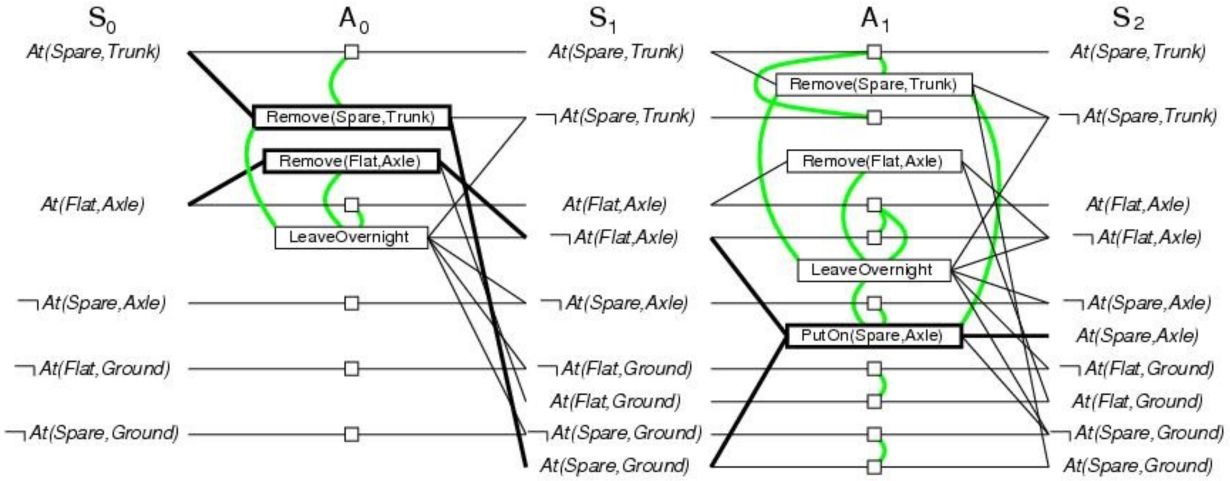
- Initially graph consist of literals from initial state and literals from closed world assumption (S_0).
- Add actions whose preconditions are satisfied by *expanding* A_0 (next slide)
- Also add persistence actions and mutex relations
- Add the effects at level S_1
- Repeat until goal is in level S_i

GRAPHPLAN Example



- Expanding graph looks for mutex relations
  - Inconsistent effects
    - E.g. $Remove(Spare, Trunk) \wedge LeaveOverNight$ inconsistent because $at(Spare, Ground) \wedge \neg at(Spare, Ground)$
  - Interference
    - E.g. $Remove(Flat, Axle) \wedge LeaveOverNight$ but can't have $at(Flat, Axle)$ as PRECOND and $\neg at(Flat, Axle)$ as EFFECT
  - Competing needs
    - E.g. $PutOn(Spare, Axle) \wedge Remove(Flat, Axle)$ due to $at(Flat, Axle) \wedge \neg at(Flat, Axle)$
  - Inconsistent support
    - E.g. in S_2, $at(Spare, Axle) \wedge at(Flat, Axle)$

GRAPHPLAN Example



- If goal literals exist in S_2 and are not mutex with any other → solution might exist
- To extract solution use Boolean CSP to solve the problem or backwards search:
  - Initial state: last level of graph + goal literals of planning problem
  - Actions: select any set of non-conflicting actions that cover the goals in the state
  - Try to reach level S_0 such that all goals are satisfied
  - Cost = 1 for each action.

Extracting the Plan

- Heuristic forward search using A* can also find path from start to goal
  - Cost is based on level in graph

Summary

- Reasoning About Action
- STRIPS Planner
- Forward planning
- Regression Planning
- Partial Order Planning
- GraphPlan
- Planning as Constraint Satisfaction