Planning

Lecture 2

Outline

♢ Planning as constraint satisfaction

♢ Planning using Logic

- Planning as propositional satisfiability
- Planning in situation calculus

♢ GraphPlan: forward planning + Heuristics (RN3rd 10.3)

♢ Partial-Order Planning (RN2nd 11.3)

Planning as CSP

Works for bounded length of the plan k

1. consider one variable for each plan step k.
2. instantiate action schemas to get the variable domain.
3. preconditions and effects become constraints on subsequent plan steps.
4. the goals are instantiated and one of them must be satisfied.
5. a new state is generated after each action is instantiated and used to compute the value of the next variable in the sequence.

Planning in Logic

♢ Transforming a planning problem in propositional logic (similar to CSP).

♢ Full logical model: **Situation Calculus**

Initial State, Goal, Preconditions and Effects are expressed as logical formulae

Planning problem: proving $\exists s. G(s) \wedge executable(s)$

GraphPlan

Data structure representing the plan search process:

- to find heuristics for other planning techniques
- to generate plans (**GRAPHPLAN**)

Applicable to propositional planning problems.

Some facts about planning graphs

Structured in levels (˜ plan steps).

State levels represent sets of possible states, each containing all the *literals* that could be true after the corresponding number of steps

Action levels follow each state level and represent actions whose preconditions are satisfied by each state level (including persistence actions for each literal)

The conflicts caused by actions are denoted by mutex links.

Planning graphs contd.

♢ Alternate states and actions, starting form S_0

$$S_0 \to A_0 \to S_1 \to A_1 \to S_2 \cdots$$

S_0: problem initial state
A_0: all the possible actions that could occur in S_0
S_1: all the literals that could result from picking any subset of the actions in A_0 (multiple states are represented)

Continue until two consecutive levels are identical, i.e. the graph has leveled off.

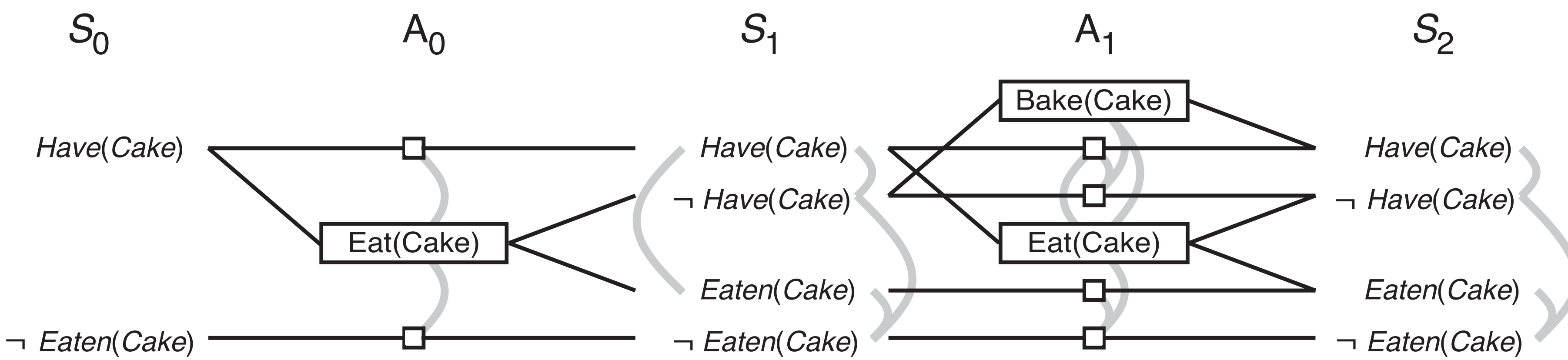
Example

INIT: $Have(Cake)$
GOAL: $Have(Cake), Eaten(Cake)$

ACTION: $Eat(Cake)$
PRECONDITION: $Have(Cake)$
EFFECT: $\neg Have(Cake), Eaten(Cake)$

ACTION: $Bake(Cake)$
PRECONDITION: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

Cake Planning graph



Mutex links

Conflicts between actions:

♢ Inconsistency: effects of one action conflict with effects of another one

♢ Interference: effects of one action conflict with preconditions of another one

♢ Competing needs: the precondition of one action is mutex with the precondition of another one

Conflicts between literals:

♢ negated literals

♢ inconsistent support: two literals are mutually exclusive if each pair of actions that can achieve them is mutex

Heuristic estimation

A literal that does not appear in the final level of the graph cannot be reached by any plan.

The cost of achieving any goal literal is given by the level where it occurs in the planning graph (admissible heuristic).

The planning graph is **polynomial** wrt actions and literals, while the search space is exponential.

Graphplan

A Planning algorithm from Planning graphs

Alternates graph expansion and solution extraction steps

Builds the planning graph and, at each step, tries to extract a solution when the goal literals appear in the last computed level without mutex links among them.

If there is no solution, then the graph is further expanded until either a solution is found or the graph is leveled-off.

POP: Partial Order Planning

Principle of **least committtment**:

- ♢ partial ordering (instead of total)
- ♢ not fully instantiated plan (in the first-order case)

Change of **problem representation**:

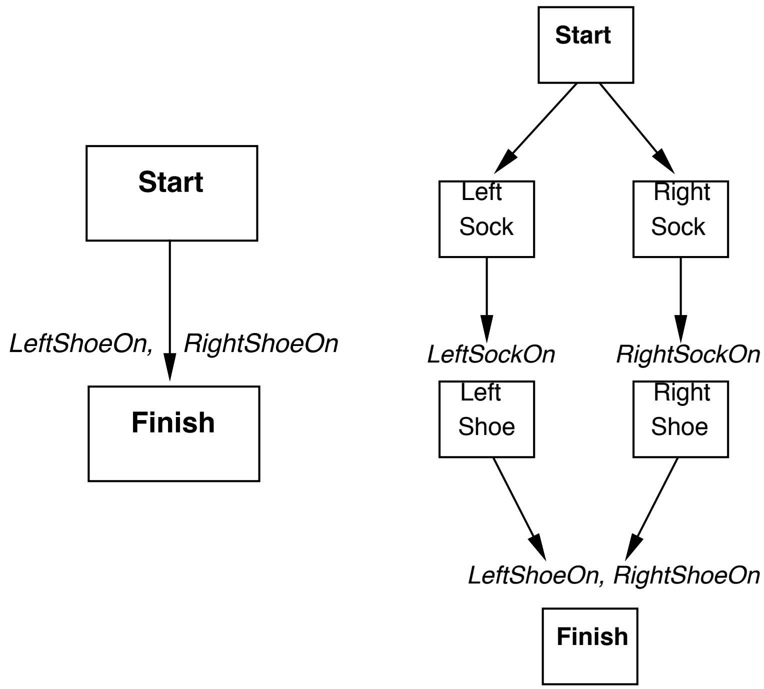
- ♢ state space: node = state in the world
- ♢ plan space: node = partial plan

Dressing up

GOAL: $\{\}$
GOAL: $\{RightShoeOn, LeftShoeOn\}$

ACTION: $RightSock$, EFFECT: $RightSockOn$)
ACTION: $LeftSock$, EFFECT: $LeftSockOn$
ACTION: $RightShoe$, PRECONDITION: $RightSockOn$,
EFFECT: $RightShoeOn$
ACTION: $LeftShoe$, PRECONDITION: $LeftSockOn$,
EFFECT: $LeftShoeOn$

Example


Start
LeftShoeOn, RightShoeOn
Finish
Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish

Partially ordered plans

Partially ordered collection of actions with

- ♢ $Start$ action has the initial state description as its effect
- ♢ $Finish$ action has the goal description as its precondition
- ♢ temporal ordering between pairs of actions

Two *additional elements* are needed to characterize the planning process:

- ♢ Open precondition = precondition of an action not yet causally linked
- ♢ Causal links from outcome of one action to precondition of another

Plan Representation

- ♢ set of actions
- ♢ set of ordering constraints $A \prec B$
- ♢ set of causal links $A \xrightarrow{p} B$

  A achieves p for B
- ♢ set of open preconditions

Initial State:

$$
\begin{aligned}
Plan(&\textsc{Actions}{:}\{Start, Finish\},\\
&\textsc{Orderings}{:}\,\{Start \prec Finish\},\\
&\textsc{Links}{:}\,\{\},\\
&\textsc{Open Preconditions}{:}\,\{RightShoeOn, LeftShoeOn\})
\end{aligned}
$$

Solutions in the plan space

A plan is complete iff every precondition is achieved

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

Plan Representation: solution

$$
\begin{aligned}
&Plan(\textsc{Actions}{:}\{RightSock, RightShoe, LeftSock, LeftShoe,\\
&\qquad Start, Finish\},\\
&\quad \textsc{Orderings}{:}\,\{Start \prec Finish, Start \prec RightSock,\\
&\qquad RightSock \prec RightShoe, RightShoe \prec Finish,\\
&\qquad Start \prec LeftSock, LeftSock \prec LeftShoe,\\
&\qquad LeftShoe \prec Finish\},\\
&\quad \textsc{Links}{:}\,\{RightSock \overset{RightSockOn}{\longrightarrow} RightShoe,\\
&\qquad RightShoe \overset{RightShoeOn}{\longrightarrow} Finish,\\
&\qquad LeftSock \overset{LeftSockOn}{\longrightarrow} LeftShoe,\\
&\qquad LeftShoe \overset{LeftShoeOn}{\longrightarrow} Finish\},\\
&\quad \textsc{Open Preconditions}{:}\,\{\})
\end{aligned}
$$

Planning process as plan refinement

Refinements of partial plans:

- add a link from an existing action to an open condition
- add a action to fulfill an open condition
- order one action 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

The Search Procedure

1. The initial plan includes the constraints for $Start$ and $Finish$, with ordering $Start \prec Finish$;
2. The successor function
   (a) pick one open precondition p on action B
   (b) pick one action A that achieves p
   (c) add the causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$; if A is new add also $Start \prec A$ and $A \prec Finish$
   (d) resolve conflicts, if possible, otherwise backtrack
3. The goal test succeeds when there are no more open preconditions

Example

Our robot is at home and needs to buy bananas, milk and a cordless drill.

(The robot knows that) the supermarket is selling, among many other items, bananas and milk, but not the cordless drill, which is sold by the hardware store, where there are many other items, but not bananas and milk. Both shops have the requested items always available.

The shopping can not be done via internet (nor by phone)! i.e. the robot must go to the shops.

Example

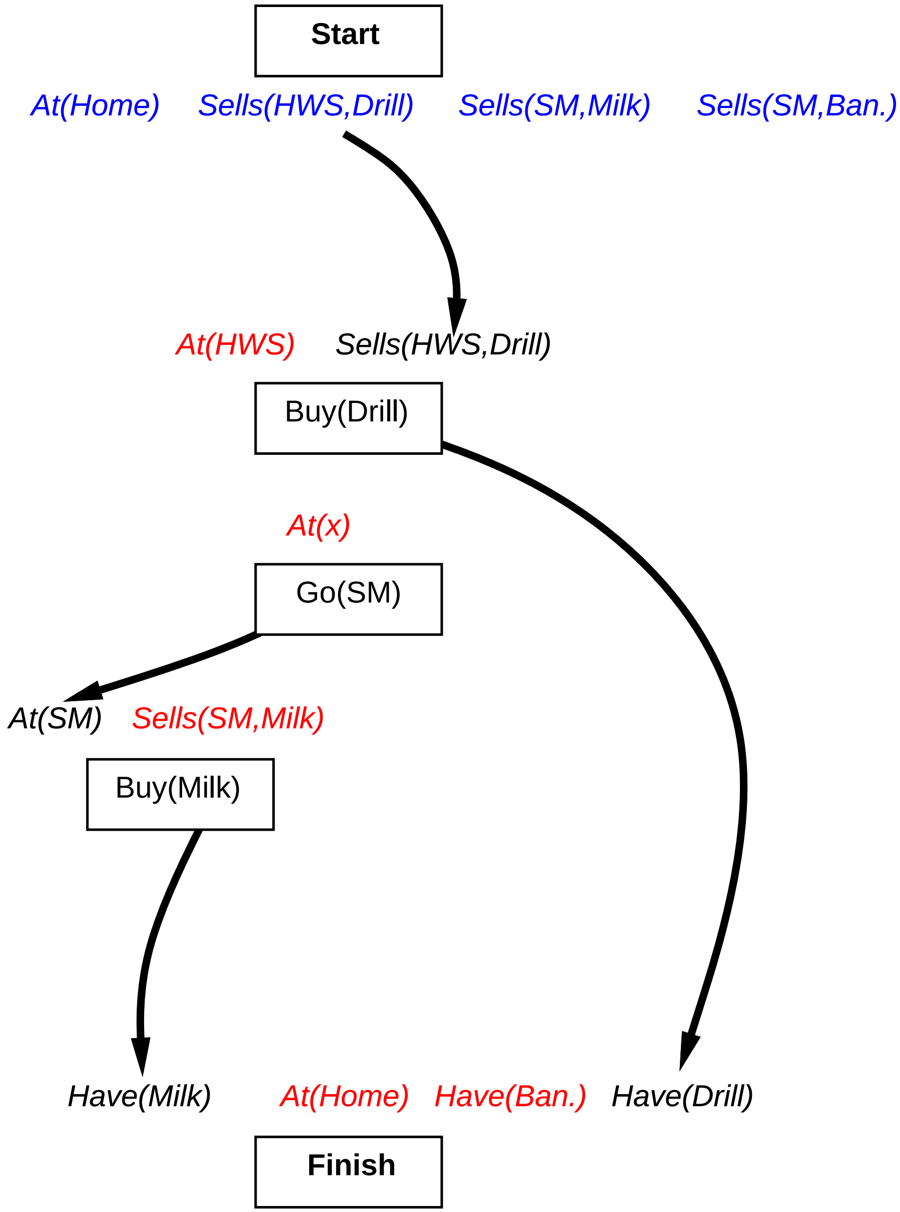

Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

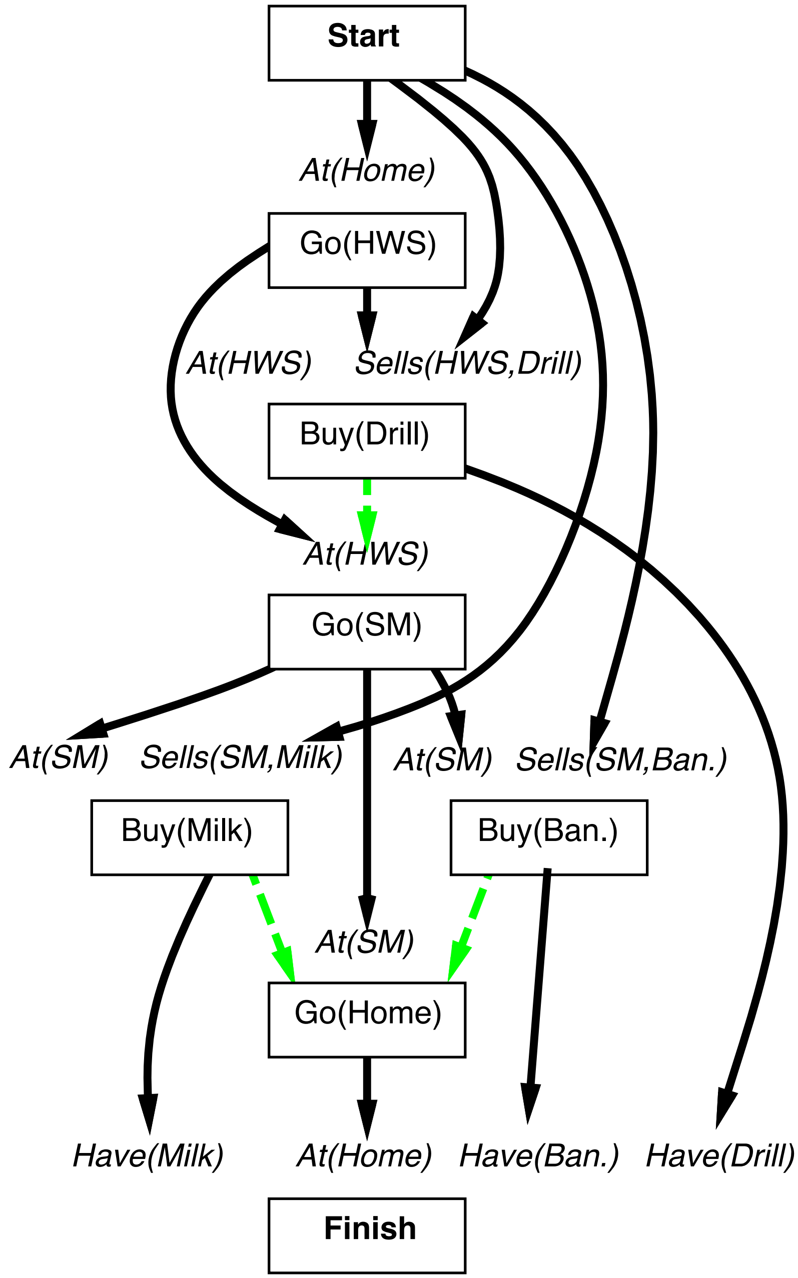
Actions for the example

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

ACTION: $Go(x)$
PRECONDITION: $At(y)$
EFFECT: $At(x) \wedge \neg At(y)$

Objects: $Milk, Bananas, Drill, \ldots$
Places: $Home, SM, HWS, \ldots$


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


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

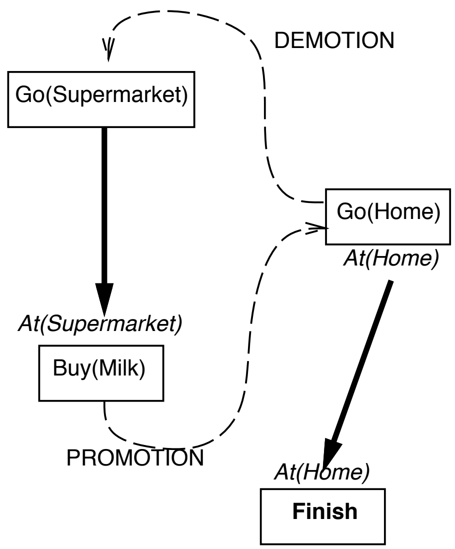
Clobbering and conflicts

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

More specifically, a **conflict** between the causal link $A \xrightarrow{p} B$ and the action C holds when C has effect $\neg p$.

A conflict can be solved by adding:
$\diamondsuit$ $C \prec A$ (**demotion**) or
$\diamondsuit$ $B \prec C$ (**promotion**)

Promotion/demotion



Demotion: put before $Go(SM)$

Promotion: put after $Buy(Milk)$

POP algorithm sketch

function POP(*initial*, *goal*, *operators*) **returns** *plan*

  plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
  loop do
    if SOLUTION?(*plan*) **then return** *plan*
    S_{need}, c ← SELECT-OPENPRECONDITION(*plan*)
    CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
    RESOLVE-THREATS(*plan*)
  end

---

function SELECT-OPENPRECONDITION(*plan*) **returns** S_{need}, c

  pick a plan step S_{need} from ACTIONS(*plan*)
    with a precondition c that has not been achieved
  return S_{need}, c

POP algorithm contd.

procedure CHOOSE-OPERATOR($plan, operators, S_{need}, c$)

choose a step S_{add} from $operators$ or ACTIONS($plan$) that has c as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS($plan$)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS($plan$)
if S_{add} is a newly added step from $operators$ **then**
  add S_{add} to ACTIONS($plan$)
  add $Start \prec S_{add} \prec Finish$ to ORDERINGS($plan$)

Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:

– choice of action (S_{add}) to achieve open precondition (S_{need})
– choice of demotion or promotion for clobberer

Selection of open precondition (S_{need}) is irrevocable: the existence of a plan does not depend on the choice of the open preconditions.

POP is sound, and complete,

Termination? The plan space is infinite ...

Flat tire

Consider the problem of changing a flat tire. The goal is to have a good spare tire (which is in the trunk) properly mounted onto the car's axle, where initially there is the flat tire.

Consider four actions:

remove the spare tire from the trunk;
put the flat tire on the axle;
remove the flat tire from the axle;
leave the car unattended overnight;

Overnight the tires will disappear ...

Flat tire: actions

ACTION: $Remove(Spare, Trunk)$
PRECONDITION: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground)$

ACTION: $Remove(Flat, Axle)$
PRECONDITION: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground)$

ACTION: $PutOn(Spare, Axle)$
PRECONDITION: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle)$

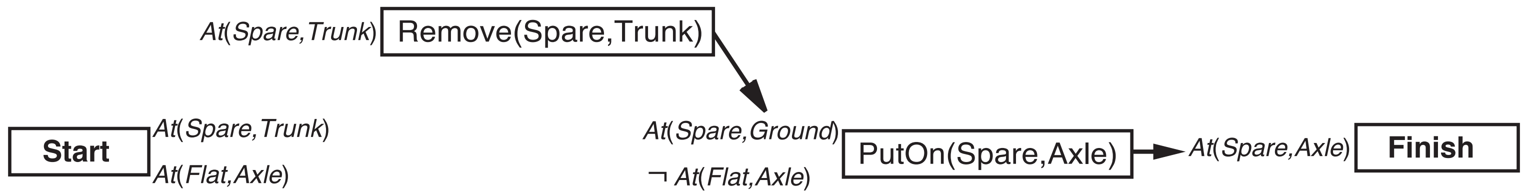
ACTION: $LeaveOvernight$ PRECONDITION:
EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge$
$\neg At(Spare, Trunk) \wedge At(Flat, Ground) \wedge \neg At(Flat, Axle)$

Flat tire

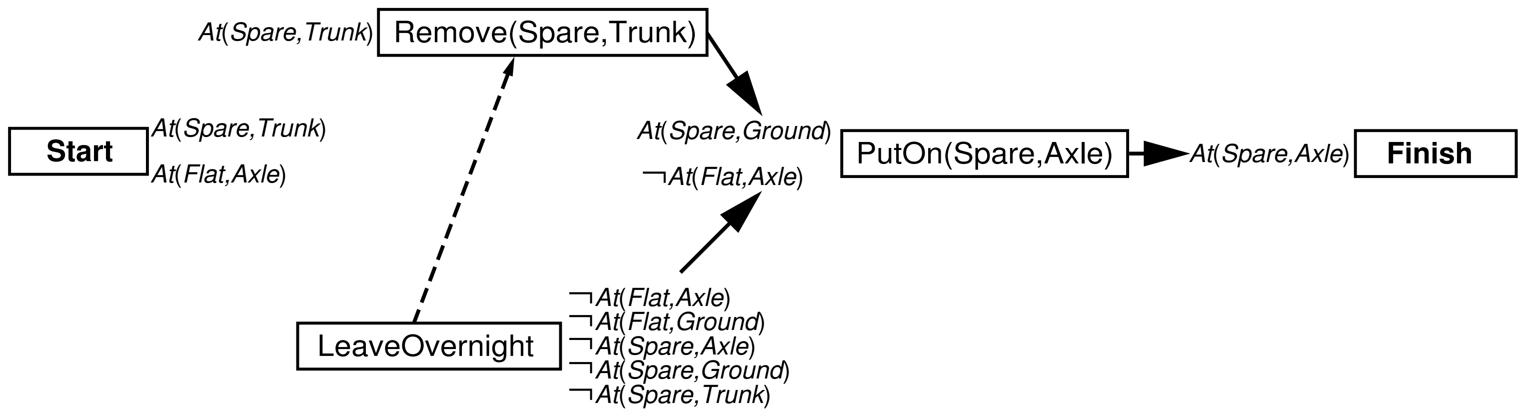
Init: $At(Flat, Axle) \wedge At(Spare, Trunk)$

Goal: $At(Spare, Axle)$

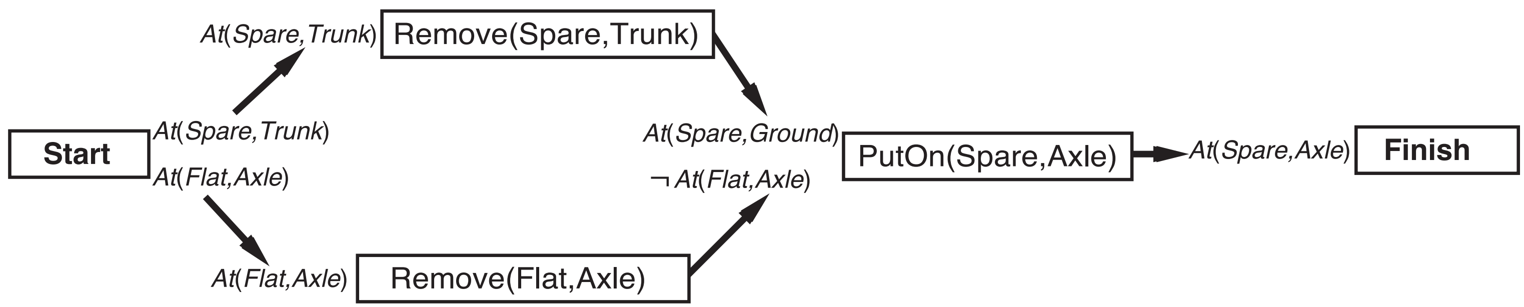
POP: Flat Tire



POP: Flat Tire



POP: Flat Tire



Extensions of POP

Handling variables: again principle of least commitment

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

Achieving $Have(milk)$ leaves as open precondition:
$At(p), Sells(p, milk)$, which can be satisfied by any p

Equality and inequality constraints needed to handle variables

$\diamondsuit$ POP admits also extensions for disjunction, universals, negation, conditionals

Heuristics for POP

General:

- ♢ number of open preconditions
- ♢ most constrained variable
  open precond that are satisfied in fewest ways
- ♢ a special data structure: the **planning graph**

Problem Specific:

Good heuristics can be derived from problem description (by the human operator)

POP is particularly effective on problems with many loosely related subgoals

Summary

Advantages

- least commitment allows for flexible execution
- POP (sound and complete)
- very good for domains that require loose sequential constraints

Disadvantages

- infinite search space
- no representation of states
- planning is complex and difficult to devise heuristics