Planning

Credit: Prof. Alex Lascarides

What is planning?

- Planning is the task of coming up with a sequence of actions
- that will achieve a goal
- We are only considering classical planning in which environments are:
  - fully observable (accessible),
  - deterministic,
  - finite,
  - static (up to agents' actions),
  - discrete (in actions, states, objects and events).

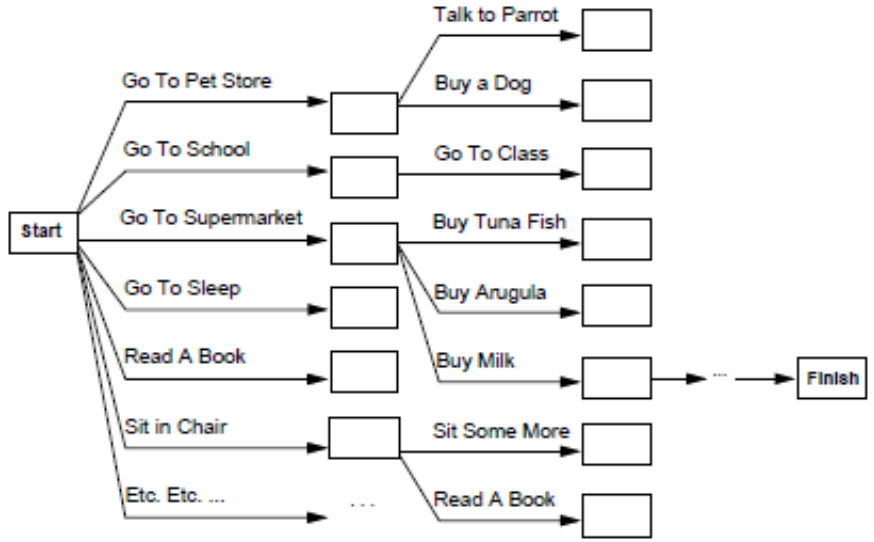
Why planning?

- So far we have dealt with two types of agents:
  - Search-based problem-solving agents
  - Logical planning agents

- Do these techniques work for solving planning problems?

Why planning?

- Consider a search-based problem-solving agent in a robot shopping world
- Task: Go to the supermarket and get milk, bananas and a cordless drill
- What would a search-based agent do?



Problems with search

- No goal-directedness.
- No problem decomposition into sub-goals that build on each other
  - May undo past achievements
  - May go to the store 3 times!
- Simple goal test doesn’t allow for the identification of milestones
- How do we find a good heuristic function?
- How do we model the way humans perceive complex goals and the quality of a plan?

How about logic & deductive inference?

- Generally a good idea, allows for “opening up” representations of states, actions, goals and plans
- If Goal = Have(Bananas)∧Have(Milk) this allows achievement of sub-goals (if independent)
- Current state can be described by properties in a compact way (e.g. Have(Drill) stands for hundreds of states)
- Allows for compact description of actions, for example
- Object(x)⇒Can(a,Grab(x))

- Allows for representing a plan hierarchically, e.g.

GoTo(Supermarket) = Leave(House)∧ ReachLocationOf (Supermarket)∧Enter(Supermarket) then decompose further into sub-plans

How about logic & deductive inference?

- Problems:
  - In its general form either awkward (propositional logic) or tractability problems (first-order logic)
  - If p is a sequence that achieves the goal, then so is $[a,a^{-1}|p]$!
- (Logically independent) subgoals may need to be undone to achieve other goals.
  - Goal: $on(A,B) \wedge on(B,C)$

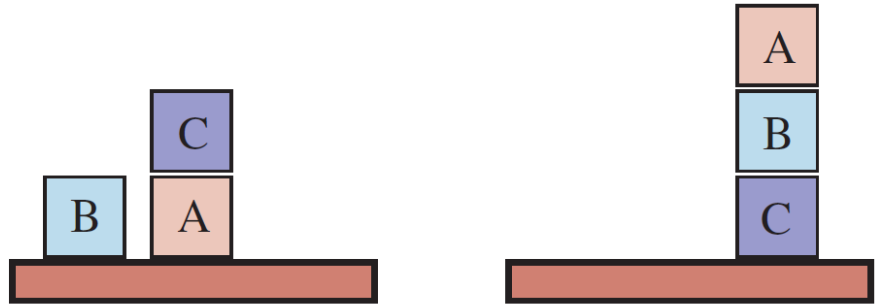


Figure 11.3 Diagram of the blocks-world problem in Figure 11.4.

Solutions

- We need
  - 1 To reduce complexity to allow scaling up.
  - 2 To allow reasoning to be guided by plan ‘quality’/efficiency.

  - Do 1. next, and 2. after that.

Representing planning problems

- Need a language expressive enough to cover interesting problems, restrictive enough to allow efficient algorithms.

- Planning Domain Definition Language or PDDL

- PDDL will allow you to express:
  - states
  - actions: a description of transitions between states
  - and goals: a (partial) description of a state.

Representing States and Goals in PDDL

- States represented as conjunctions of propositional or function-free first order positive literals:
  - Happy ∧Sunshine, At(Plane1,Melbourne)∧ At(Plane2,Sydney)

- So these aren't states:
  - At(x,y) (no variables allowed), Love(Father(Fred),Fred) (no function symbols allowed)
  - ¬Happy (no negation allowed).

  Closed-world assumption!

- A goal is a partial description of a state, and you can use negation, variables etc. to express that description.
  - ¬Happy, At(x,SFO), Love(Father(Fred),Fred) . . .

Actions in PDDL

$$Action(Fly(p, from, to),$$
$$\text{Precond:} At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$$
$$\text{Effect:} \neg At(p, from) \land At(p, to))$$

- Actually action schemata, as they may contain variables
- Action name and parameter list serves to identify the action
- Precondition: defines states in which action is executable:
  - Conjunction of positive and negative literals, where all variables must occur in action name.
- Effect: defines how literals in the input state get changed (anything not mentioned stays the same).
  - Conjunction of positive and negative literals, with all its variables also in the preconditions.
  - Often, effects divided into add list and delete list

The semantics of PDDL: States and their Descriptions

$s \models At(P_1, SFO)$ iff $At(P_1, SFO) \in s$

$s \models \neg At(P_1, SFO)$ iff $At(P_1, SFO) \notin s$

$s \models \phi(x)$ iff there is a ground term d such that $s \models \phi[x/d]$.

$s \models \phi \wedge \psi$ iff $s \models \phi$ and $s \models \psi$

The Semantics of PDDL: Applicable Actions

- Any action is applicable in any state that satisfies the precondition with an appropriate substitution for parameters.

- Example: State
  - At(P1,Melbourne)∧At(P2,Sydney)∧Plane(P1)∧Plane(P2) ∧Airport(Sydney)∧Airport(Melbourne)∧Airport(Heathrow)

  - Satisfies
  - At(p,from)∧Plane(p)∧Airport(from)∧Airport(to)

  - with substitution (among others)
    - {p/P2,from/Sydney, to/Heathrow}

The semantics of PDDL: The Result of an Action

- Result of executing action a in state s is state s' with any positive literal P in a's Effects added to the state and every negative literal ¬P removed from it (under the given substitution) .

- In our example s' would be
  - At(P1,Melbourne)∧At(P2,Heathrow)∧Plane(P1)∧Plane(P2) ∧Airport(Sydney)∧Airport(Melbourne)∧Airport(Heathrow)

- "PDDL assumption": every literal not mentioned in the effect remains unchanged (cf. frame problem)

- Solution = action sequence that leads from the initial state to a state that satisfies the goal.

Blocks world example

- Given: A set of cube-shaped blocks sitting on a table
- Can be stacked, but only one on top of the other
- Robot arm can move around blocks (one at a time)
- Goal: to stack blocks in a certain way
- Formalization in PDDL:
  - On(b,x) to denote that block b is on x (block/table)
  - Move(b,x,y) to indicate action of moving b from x to y
  - Precondition for this action requires Clear (z): nothing stacked on z.

Blocks world example

- Action schema:

$$Action(Move(b,x,y),$$
$$\text{Precond:} On(b,x) \wedge Clear(b) \wedge Clear(y)$$
$$\text{Effect:} On(b,y) \wedge Clear(x) \wedge \neg On(b,x) \wedge \neg Clear(y))$$

- Problem: when x = Table or y = Table we infer that the table is clear when we have moved a block from it (not true) and require that table is clear to move something on it (not true)
- Solution: introduce another action

$$Action(MoveToTable(b,x),$$
$$\text{Precond:} On(b,x) \wedge Clear(b)$$
$$\text{Effect:} On(b, Table) \wedge Clear(x) \wedge \neg On(b,x))$$

Does this Work?

- Interpret Clear (b) as “there is space on b to hold a block” (thus Clear (Table) is always true)
- But without further modification, planner can still use Move(b,x,Table):
  - Needlessly increases search space (not a big problem here, but can be)
- So part of solution is to also add Block(b)∧Block(y) to precondition of Move

Summary

- Defined the planning problem
- Discussed problems with search/logic
- Introduced PDDL: a special representation language for Planning
- Blocks world example as a famous application domain
- Next time: Algorithms for planning!
- **State-Space Search and Partial-Order Planning**

State-Space Search and Partial-Order Planning

Planning with state-space search

- Most straightforward way to think of planning process:
  - search the space of states using action schemata

- Since actions are defined both in terms of preconditions and
- effects we can search in both directions

- Two methods:
  - forward state-space search: Start in initial state; consider action sequences until goal state is reached.
  - backward state-space search: Start from goal state; consider action sequences until initial state is reached

Planning with state-space search

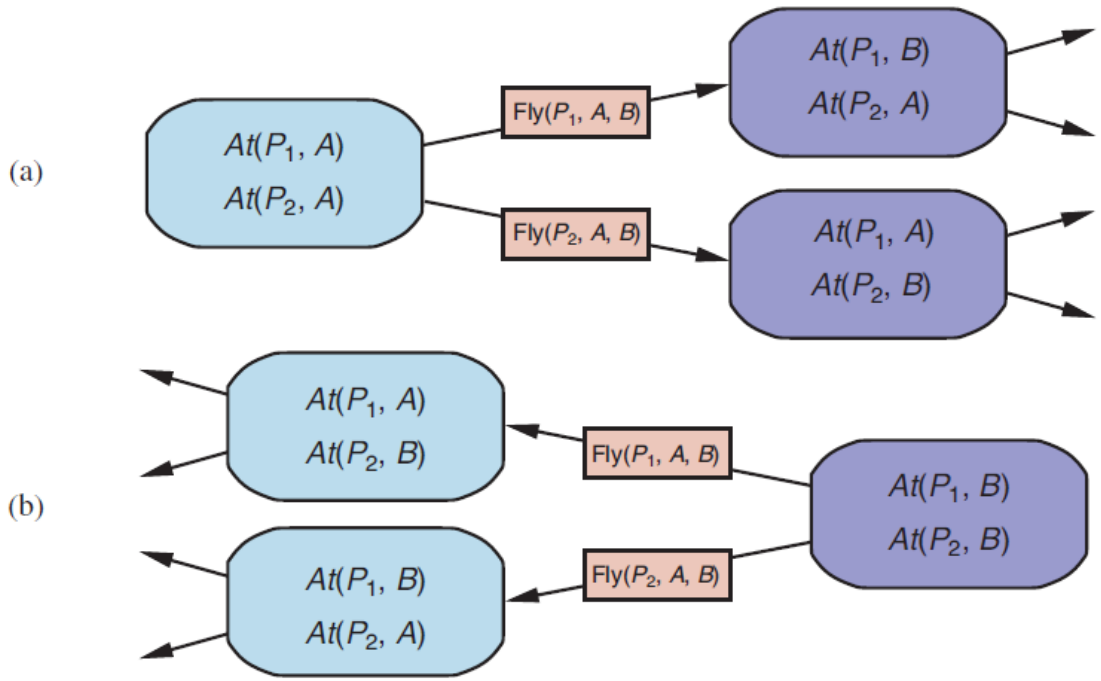


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

Forward state-space search

- Also called progression planning
- Formulation of planning problem:
  - Initial state of search is initial state of planning problem (=set of positive literals)
  - Applicable actions are those whose preconditions are satisfied
  - Single successor function works for all planning problems (consequence of action representation)
  - Goal test = checking whether state satisfies goal of planning problem
  - Step cost usually 1, but different costs can be allowed

Forward state-space search

- Search space is finite in the absence of function symbols
- Any complete graph search algorithm (like A$*$) will be a complete graph planning algorithm
- Forward search does not solve problem of irrelevant actions (all actions considered from each state)
- Efficiency depends largely on quality of heuristics

Forward state-space search

- Example:
  - Air cargo problem, 10 airports with 5 planes each, 20 pieces of cargo
  - Task: move all 20 pieces of cargo at airport A to airport B
  - Each of 50 planes can fly to 9 airports, each of 200 packages can be unloaded or loaded (individually)
  - So approximately 10K executable actions in each state (50×9×200)
  - Lots of irrelevant actions get considered, although solution is trivial!

Backward state-space search

- In normal search, backward approach hard because goal described by a set of constraints (rather than being listed explicitly)
- Problem of how to generate predecessors, but planning representations allow us to consider only relevant actions
- Exclusion of irrelevant actions decreases branching factor
- In example, only about 20 actions working backward from goal
- Regression planning = computing the states from which applying a given action leads to the goal
- Must ensure that actions are consistent, i.e. they don’t undo any desired literals

Air cargo domain example

- Goal can be described as
  - At(C1,B)∧At(C2,B)∧. . .At(C20,B)
- To achieve At(C1,B) there is only one action,
  - Unload(C1,p,B) (p unspecified)
- Can do this action only if its preconditions are satisfied.
- So the predecessor to the goal state must include
  - In(C1,p)∧At(p,B), and should not include At(C1,B) (otherwise irrelevant action)
- Full predecessor:
  - In(C1,p)∧At(p,B)∧. . .∧At(C20,B)
- Load(C1,p) would be inconsistent (negates At(C1,B))

Backward state-space search

- General process of constructing predecessors for backward
- search given goal description G, relevant and consistent action
- A:
  - Any positive effects of A that appear in G are deleted
  - Each precondition of A is added unless it already appears
- Any standard search algorithm can be used, terminates when predecessor description is satisfied by initial (planing) state
- First-order case may require additional substitutions which must be applied to actions leading from state to goal

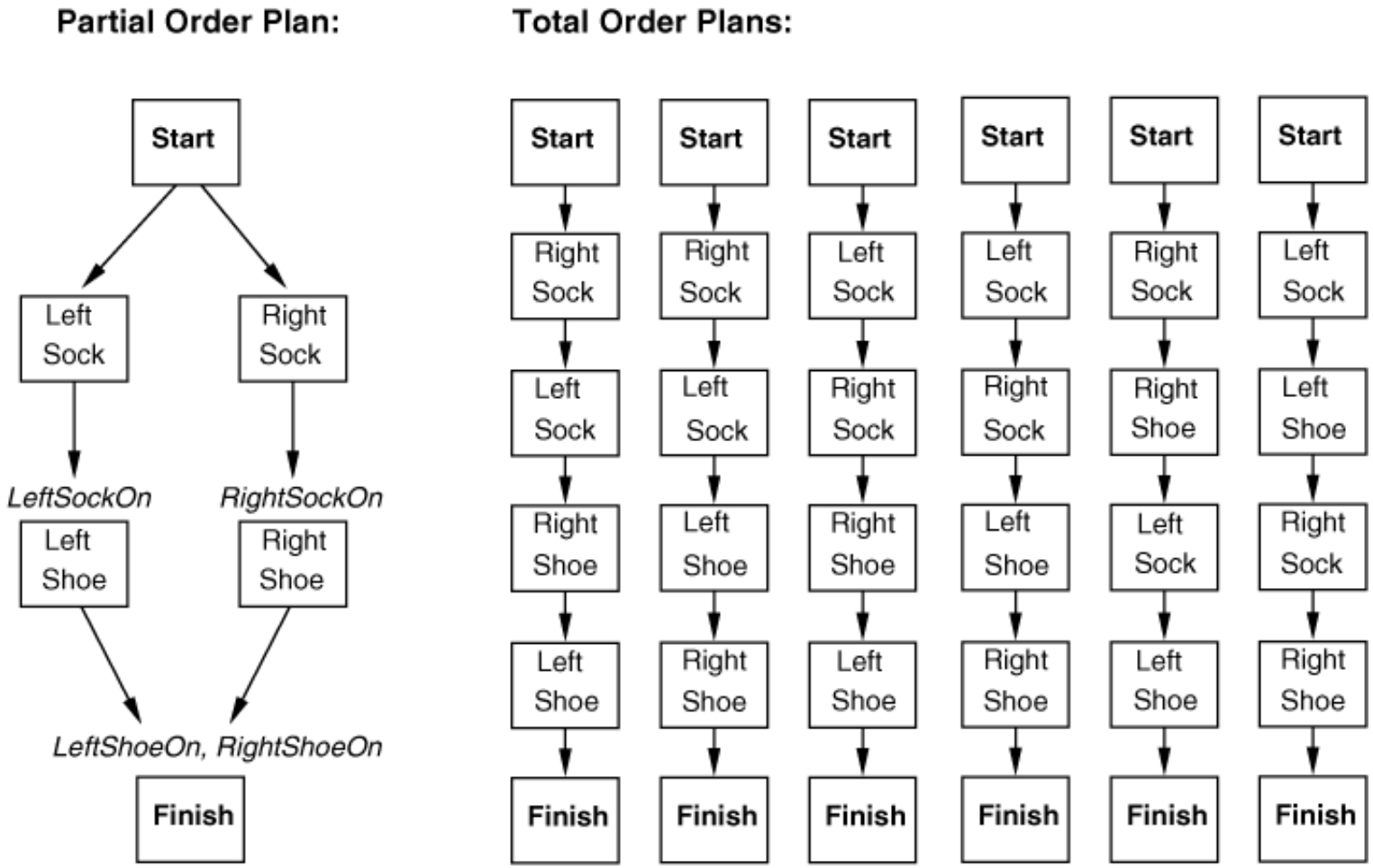
Heuristics for state-space search

- Two possibilities:
  - Divide and Conquer (subgoal decomposition)
  - Derive a Relaxed Problem
- Subgoal decomposition is . . .
  - optimistic (admissible) if negative interactions exist (e.g. subplan deletes goal achieved by other subplan)
  - pessimistic (inadmissible) if positive interactions exist (e.g. subplans contain redundant actions)
- Relaxations:
  - drop all preconditions (all actions always applicable, combined with subgoal independence makes prediction even easier)
  - remove all negative effects (and count minimum number of actions so that union satisfies goals)
  - empty delete lists approach (involves running a simple planning problem to compute heuristic value)

Partial-order planning

- State-space search planning algorithms consider totally ordered sequences of actions
- Better not to commit ourselves to complete chronological ordering of tasks (least commitment strategy)
- Basic idea:
  - Add actions to a plan without specifying which comes first unless necessary
  - Combine ‘independent’ subsequences afterwards
- Partial-order solution will correspond to one or several linearisations of partial-order plan
- Search in plan space rather than state spaces (because your search is over ordering constraints on actions, as well as transitions among states).

Example: Put your socks and shoes on



Partial-order planning (POP) as a search problem

- Define POP as search problem over plans consisting of:
  - **Actions**; initial plan contains dummy actions Start (no preconditions, effect=initial state) and Finish (no effects, precondition=goal literals)
  - **Ordering constraints** on actions A $\prec$ B (A must occur before B); contradictory constraints prohibited

    - **Causal links** between actions $A \xrightarrow{p} B$ express A achieves p for B (p precondition of B, effect of A, must remain true between A and B); inserting action C with effect $\neg p$ ($A \prec C$ and $C \prec B$) would lead to **conflict**

  - **Open preconditions**: set of conditions not yet achieved by the plan (planners try to make open precondition set empty without introducing contradictions)

The POP algorithm

- Final plan for socks and shoes example (without trivial ordering constraints):

$$\begin{aligned}
&\text{Actions: } \{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}\\
&\text{Orderings: } \{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}\\
&\text{Links: } \{RightSock \overset{RightSockOn}{\rightarrow} RightShoe,\\
&\qquad LeftSock \overset{LeftSockOn}{\rightarrow} LeftShoe,\\
&\qquad RightShoe \overset{RightShoeOn}{\rightarrow} Finish,\\
&\qquad LeftShoe \overset{LeftShoeOn}{\rightarrow} Finish\}\\
&\text{Open preconditions: } \{\}
\end{aligned}$$

- Consistent plan = plan without cycles in orderings and conflicts with links
- Solution = consistent plan without open preconditions
- Every linearisation of a partial-order solution is a total-order solution (implications for execution!)

The POP algorithm

- Initial plan:

  Actions: $\{Start, Finish\}$, Orderings: $\{Start \prec Finish\}$,
  Links: $\{\}$, Open preconditions: Preconditions of *Finish*

- Pick p from open preconditions on some action B, generate a consistent successor plan for every A that achieves p
- Ensuring consistency:

  Add $A \xrightarrow{p} B$ and $A \prec B$ to plan. If A new, add A and $Start \prec A$
  and $A \prec Finish$ to plan
  Resolve conflicts between the new link and all actions and
  between A (if new) and all links as follows:
  If conflict between $A \xrightarrow{p} B$ and C, add $B \prec C$ or $C \prec A$

- Goal test: check whether there are open preconditions (only consistent plans are generated)

Partial-order planning example (1)

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

$Action(Remove(Spare, Trunk),$

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

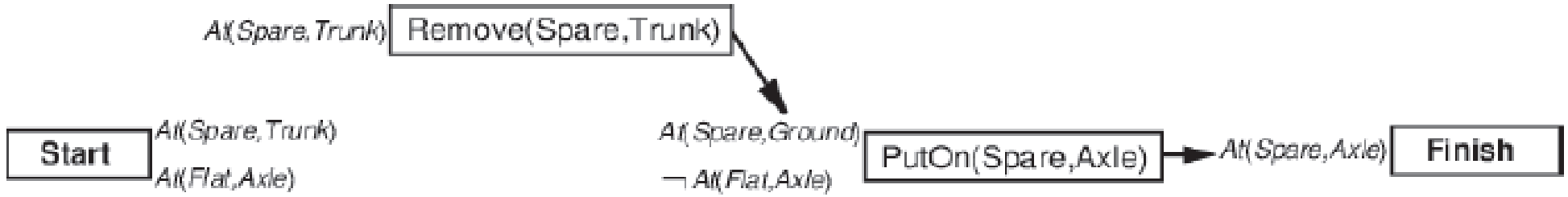
$Action(LeaveOvernight,$ Precond:

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

$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Partial-order planning example (2)

- Pick (only) open precondition At(Spare,Axle) of Finish Only applicable action = PutOn(Spare,Axle)
- Pick At(Spare,Ground) from PutOn(Spare,Axle) Only applicable action = Remove(Spare,Trunk)
- Situation after two steps:

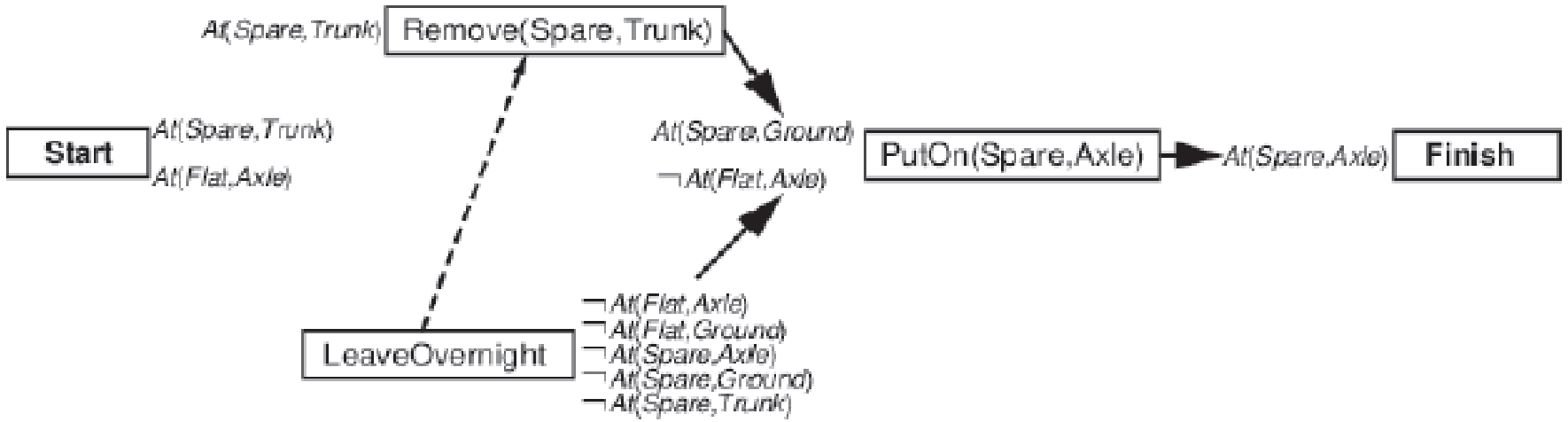


Partial-order planning example (3)

- Pick ¬At(Flat,Axle) precondition of PutOn(Spare,Axle) Choose LeaveOvernight, effect ¬At(Spare,Ground)
- Conflict with link

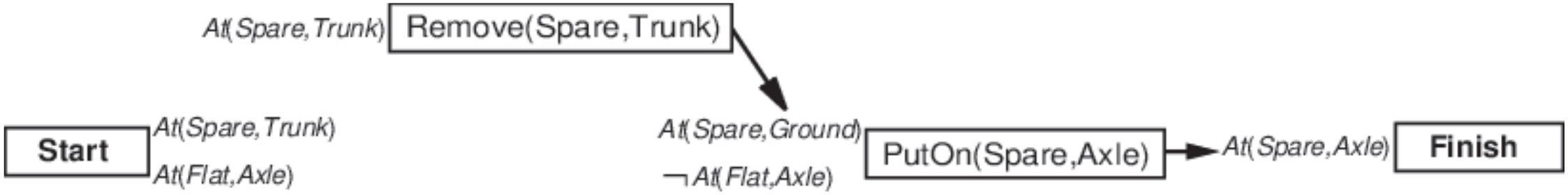
$$Remove(Spare, Trunk) \overset{At(Spare,Ground)}{\rightarrow} PutOn(Spare, Axle)$$

- Resolve by adding LeaveOvernight $\prec$ Remove(Spare,Trunk) Why is this the only solution?



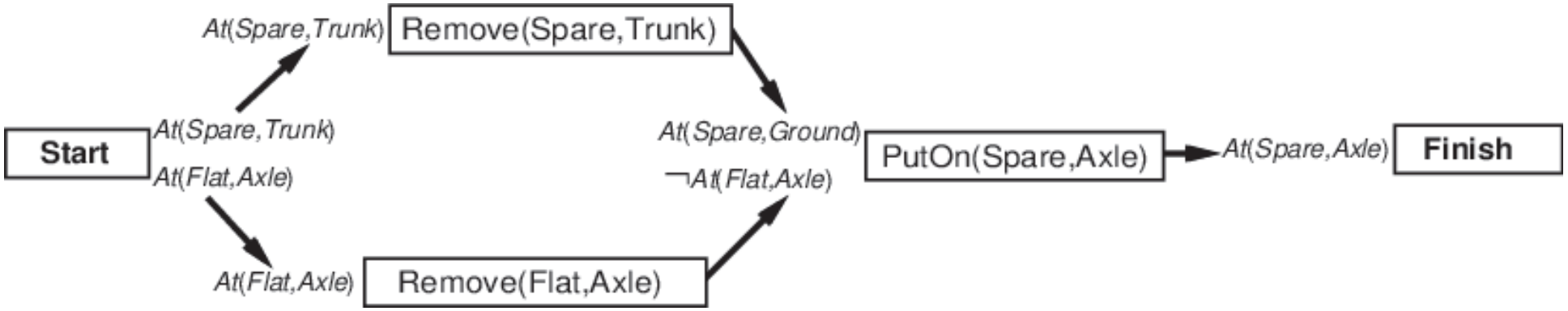
Partial-order planning example (4)

- Remaining open precondition At(Spare,Trunk), but conflict between Start and ¬At(Spare,Trunk) effect of LeaveOvernight
- No ordering before Start possible or after Remove(Spare,Trunk) possible
- No successor state, backtrack to previous state and remove LeaveOvernight, resulting in this situation:



Partial-order planning example (5)

- Now choose Remove(Flat,Axle) instead of LeaveOvernight
- Next, choose At(Spark,Trunk) precondition of Remove(Spare,Trunk)
  - Choose Start to achieve this
- Pick At(Flat,Axle) precondition of Remove(Flat,Axle), choose Start to achieve it
- Final, complete, consistent plan:



Dealing with unbound variables

- In first-order case, unbound variables may occur during planning process
- Example:
  - Action(Move(b,x,y),

  Precond:On(b,x)∧Clear (b)∧Clear (y)
  Effect:On(b,y)∧Clear (x)∧¬On(b,x)∧¬Clear (y))

  achieves On(A,B) under substitution {b/A,y/B}
- Applying this substitution yields

  Action(Move(A,x,B),
  Precond:On(A,x)∧Clear (A)∧Clear (B)
  Effect:On(A,B)∧Clear (x)∧¬On(A,x)∧¬Clear (B))

  and x is still unbound (another side of the least commitment approach)

Dealing with unbound variables

- Also has an effect on links, e.g. in example above

$$Move(A, x, B) \overset{On(A,B)}{\rightarrow} Finish \text{ would be added}$$

- If another action has effect ¬On(A, z) then this is only a conflict if z = B
- Solution: insert inequality constraints (in example: z != B) and check these constraints whenever applying substitutions
- Remark on heuristics: Even harder than in total-order planning, e.g. adapt most-constrained-variable approach from CSPs

Summary

- State-space search approaches (forward/backward)
- Heuristics for state-space search planning
- Partial-order planning
- The POP algorithms
- POP as search in planning space
- POP example
- POP with unbound variables