Informatics 2D – Reasoning and Agents

Semester 2, 2019–2020

Alex Lascarides
alex@inf.ed.ac.uk

Lecture 17 – State-Space Search and Partial-Order Planning
27th February 2020

Where are we?

Last time . . .

- we defined the planning problem
- discussed problem with using search and logic in planning
- introduced representation languages for planning
- looked at blocks world example

Today . . .

- **State-space search and partial-order planning**

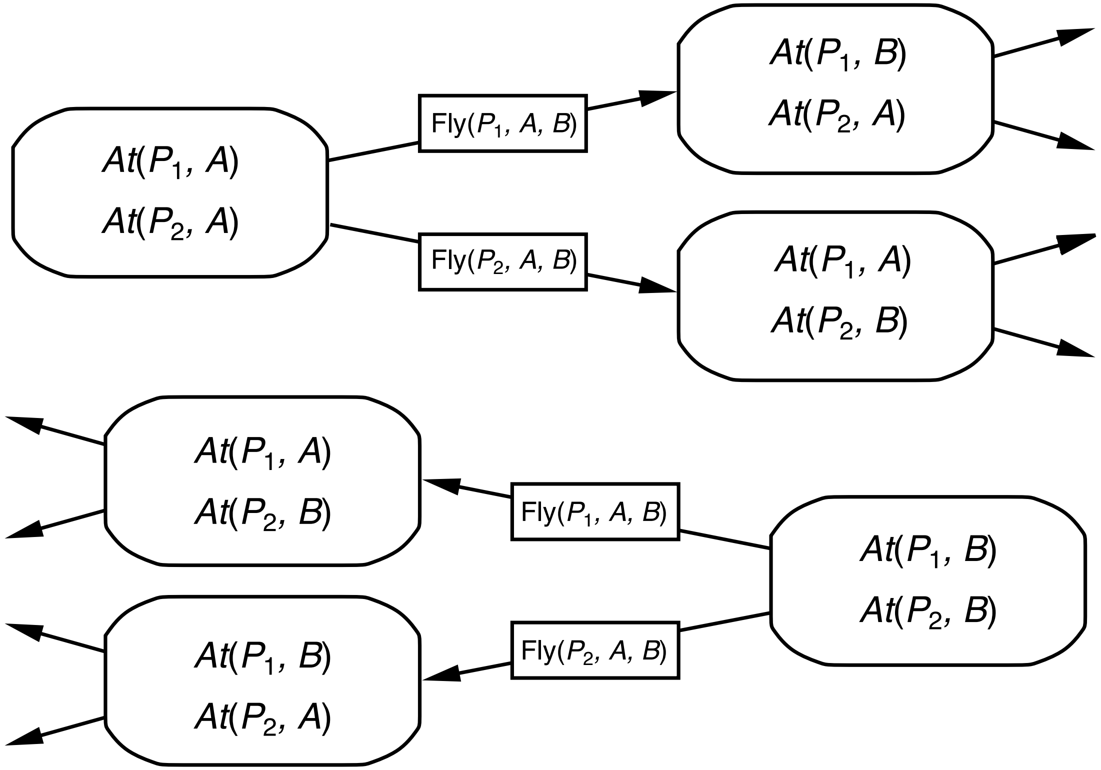
Planning with state-space search

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

Planning with state-space search

(a)

(b)



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
- 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 ($50 \times 9 \times 200$)
  - 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

$$At(C_1, B) \land At(C_2, B) \land \ldots At(C_{20}, B)$$

- To achieve $At(C_1, B)$ there is only one action, $Unload(C_1, p, B)$ (p unspecified)
- Can do this action only if its preconditions are satisfied.
- So the predecessor to the goal state must include $In(C_1, p) \land At(p, B)$, and should not include $At(C_1, B)$ (otherwise irrelevant action)
- Full predecessor:

$$In(C_1, p) \land At(p, B) \land \ldots \land At(C_{20}, B)$$

- $Load(C_1, p)$ would be inconsistent (negates $At(C_1, B)$)

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

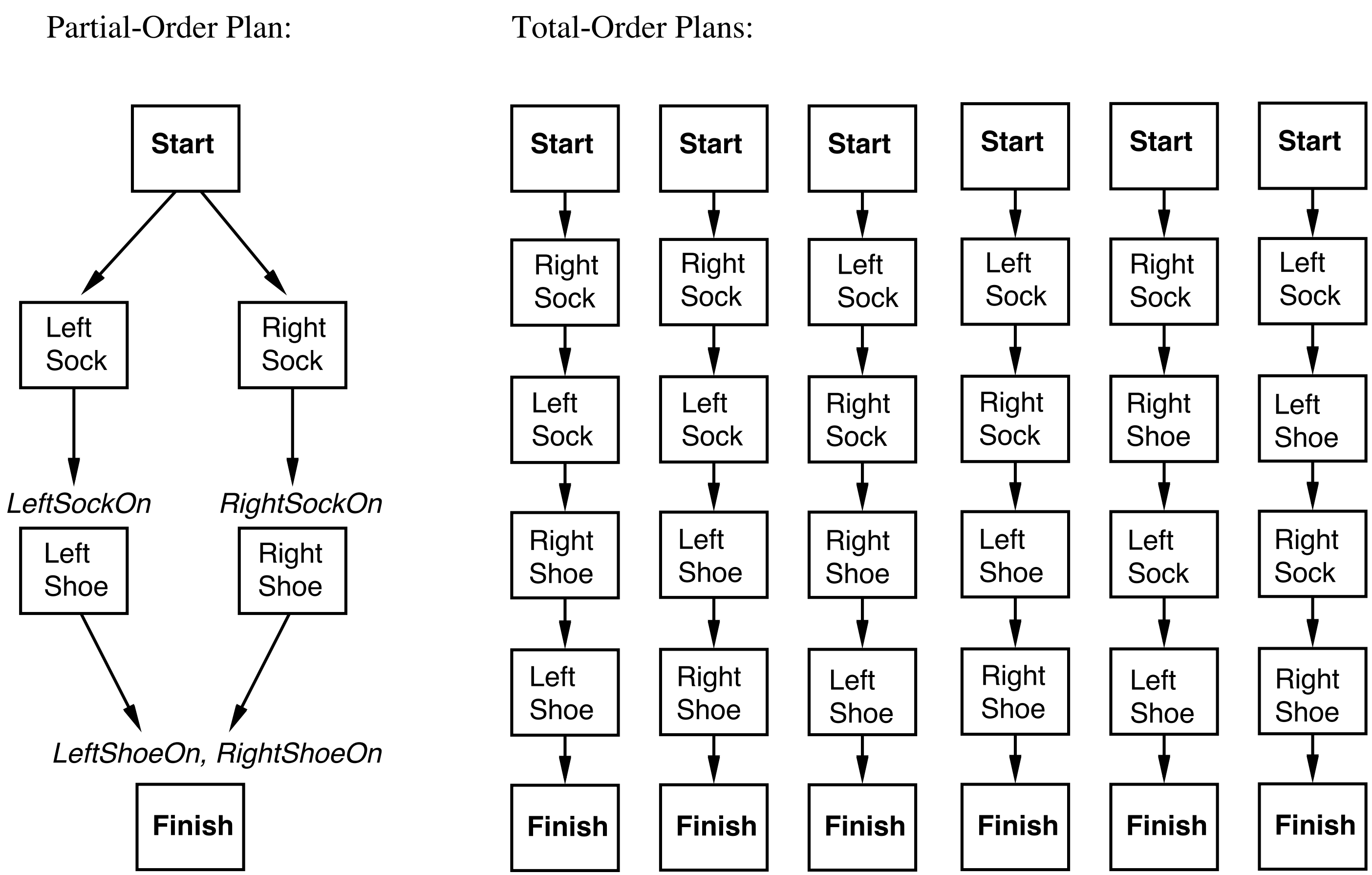
Heuristics for state-space search

- Two possibilities:
  1. Divide and Conquer (**subgoal decomposition**)
  2. 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:**
  1. Add actions to a plan without specifying which comes first unless necessary
  2. 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):
  Actions: $\{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}$
  Orderings: $\{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}$
  Links: $\{RightSock \stackrel{RightSockOn}{\rightarrow} RightShoe,$
  $LeftSock \stackrel{LeftSockOn}{\rightarrow} LeftShoe,$
  $RightShoe \stackrel{RightShoeOn}{\rightarrow} Finish,$
  $LeftShoe \stackrel{LeftShoeOn}{\rightarrow} Finish\}$
  Open preconditions: $\{\}$
- **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:
  1. 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
  2. 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)

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

$Action(Remove(Spare, Trunk),$

$\quad$ PRECOND: $At(Spare, Trunk)$

$\quad$ EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$

$Action(Remove(Flat, Axle),$

$\quad$ PRECOND: $At(Flat, Axle)$

$\quad$ EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$

$Action(PutOn(Spare, Axle),$

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

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

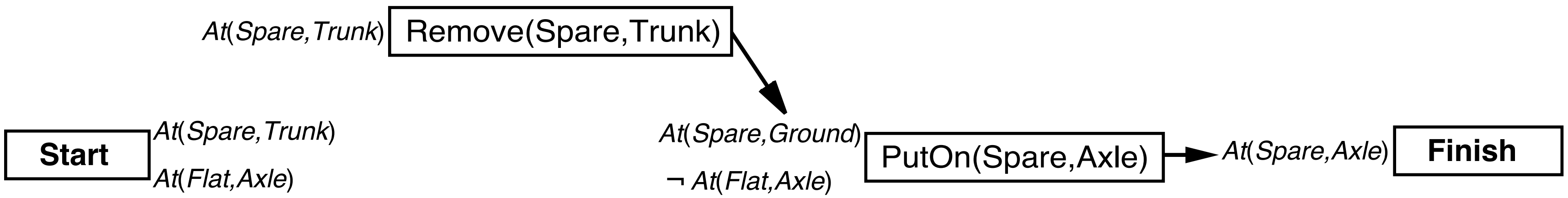
$Action(LeaveOvernight,$ PRECOND:

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

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

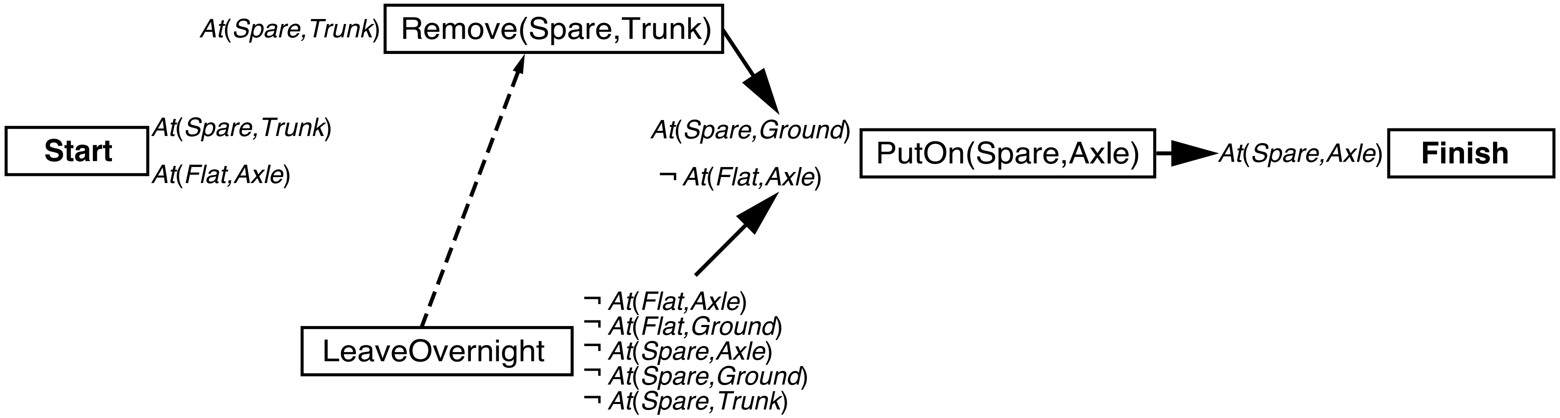
Partial-order planning example (2)

- 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:



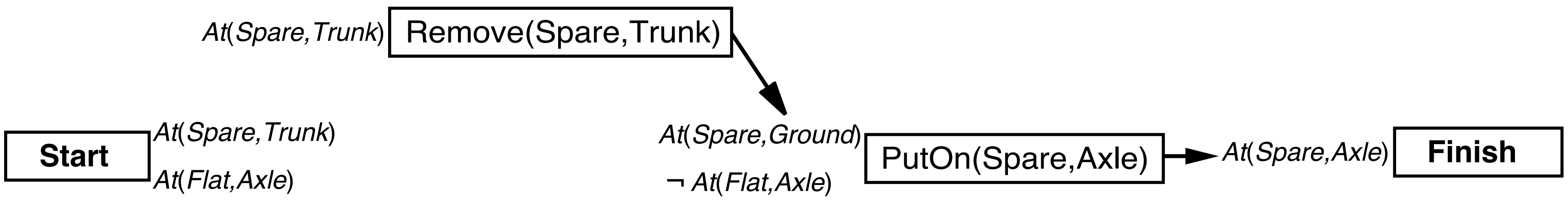
Partial-order planning example (3)

- Pick $\neg At(Flat, Axle)$ precondition of $PutOn(Spare, Axle)$
  Choose $LeaveOvernight$, effect $\neg At(Spare, Ground)$
- Conflict with link
  $Remove(Spare, Trunk) \stackrel{At(Spare,Ground)}{\longrightarrow} PutOn(Spare, Axle)$
- Resolve by adding $LeaveOvernight \prec Remove(Spare, Trunk)$
  Why is this the only solution?



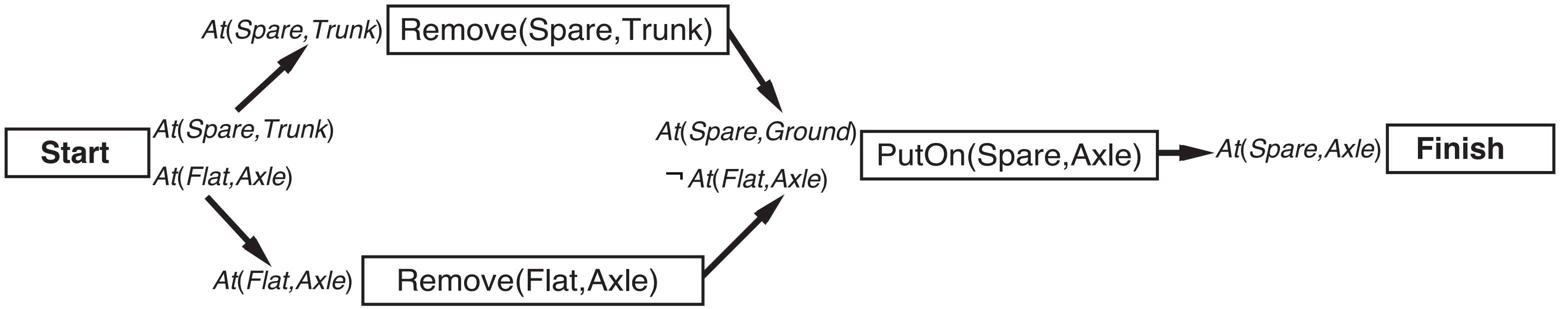
Partial-order planning example (4)

- Remaining open precondition $At(Spare, Trunk)$, but conflict between $Start$ and $\neg At(Spare, Trunk)$ 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:

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

achieves $On(A, B)$ under substitution $\{b/A, y/B\}$

- Applying this substitution yields

$$Action(Move(A, x, B),$$
$$\text{PRECOND:} On(A, x) \wedge Clear(A) \wedge Clear(B)$$
$$\text{EFFECT:} On(A, B) \wedge Clear(x) \wedge \neg On(A, x) \wedge \neg Clear(B))$$

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 $Move(A, x, B) \stackrel{On(A,B)}{\rightarrow} Finish$ would be added
- If another action has effect $\neg On(A, z)$ then this is only a conflict if $z = B$
- Solution: insert **inequality constraints** (in example: $z \neq B$) 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
- Next time: **Planning and Acting in the Real World I**