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) \[
\begin{align*}
&\text{At}(P_1, A) \\
&\text{At}(P_2, A) \\
\text{Fly}(P_1, A, B) \\
&\text{At}(P_1, B) \\
&\text{At}(P_2, A) \\
\text{Fly}(P_2, A, B) \\
&\text{At}(P_1, A) \\
&\text{At}(P_2, B) \\
\end{align*}
\]

(b) \[
\begin{align*}
&\text{At}(P_1, A) \\
&\text{At}(P_2, B) \\
\text{Fly}(P_1, A, B) \\
&\text{At}(P_1, B) \\
&\text{At}(P_2, B) \\
\text{Fly}(P_2, A, B) \\
&\text{At}(P_1, B) \\
&\text{At}(P_2, A) \\
\end{align*}
\]
Forward state-space search

- Also called *progression* planning
- Formulation of planning problem:
  - Initial state of search is initial state of planning problem
    \[=\text{set of positive literals} \]
  - Applicable actions are those whose preconditions are satisfied
  - Single successor function works for all planning problems
    \[(\text{consequence of action representation})\]
  - 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 (50×9×200)
  - 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

\[ \text{At}(C_1, B) \land \text{At}(C_2, B) \land \ldots \land \text{At}(C_{20}, B) \]

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

\[ \text{In}(C_1, p) \land \text{At}(p, B) \land \ldots \land \text{At}(C_{20}, B) \]

- \( \text{Load}(C_1, p) \) would be inconsistent (negates \( \text{At}(C_1, B) \))
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 (planning) 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 Plan:

Start

- Left Sock
  - LeftSockOn
  - Left Shoe
    - LeftShoeOn, RightShoeOn
    - Finish

- Right Sock
  - RightSockOn
  - Right Shoe
    - RightShoeOn, LeftShoeOn
    - Finish

Total-Order Plans:

1. Start
   - Right Sock
     - RightSockOn
     - Right Shoe
       - RightShoeOn, LeftShoeOn
       - Finish

2. Start
   - Right Sock
     - RightSockOn
     - Left Sock
       - LeftShoeOn, RightShoeOn
       - Finish

3. Start
   - Left Sock
     - LeftSockOn
     - Left Shoe
       - LeftShoeOn, RightShoeOn
       - Finish

4. Start
   - Left Sock
     - LeftSockOn
     - Right Sock
       - RightShoeOn, LeftShoeOn
       - Finish

5. Start
   - Left Sock
     - LeftSockOn
     - Left Shoe
       - LeftShoeOn, RightShoeOn
       - Finish

6. Start
   - Left Sock
     - LeftSockOn
     - Right Shoe
       - RightShoeOn, LeftShoeOn
       - Finish

7. Start
   - Left Sock
     - LeftSockOn
     - Left Shoe
       - LeftShoeOn, RightShoeOn
       - Finish

8. Start
   - Right Sock
     - RightSockOn
     - Left Sock
       - LeftShoeOn, RightShoeOn
       - Finish

9. Start
   - Right Sock
     - RightSockOn
     - Right Shoe
       - RightShoeOn, LeftShoeOn
       - Finish

10. Start
    - Right Sock
      - RightSockOn
      - Left Sock
        - LeftShoeOn, RightShoeOn
        - Finish

11. Start
    - Right Sock
      - RightSockOn
      - Right Shoe
        - RightShoeOn, LeftShoeOn
        - Finish

12. Start
    - Right Sock
      - RightSockOn
      - Left Sock
        - LeftShoeOn, RightShoeOn
        - Finish

13. Start
    - Right Sock
      - RightSockOn
      - Right Shoe
        - RightShoeOn, LeftShoeOn
        - Finish
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^{RightSockOn} \rightarrow \text{RightShoe}, \newline \text{LeftSock}^{LeftSockOn} \rightarrow \text{LeftShoe}, \newline \text{RightShoe}^{RightShoeOn} \rightarrow \text{Finish}, \newline \text{LeftShoe}^{LeftShoeOn} \rightarrow \text{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) \land At(Špare, Trunk)). Goal(At(Špare, Axle)).

Action(Remove(Špare, Trunk),
  \textbf{Precond:} At(Špare, Trunk)
  \textbf{Effect:} \neg At(Špare, Trunk) \land At(Špare, Ground))

Action(Remove(Flat, Axle),
  \textbf{Precond:} At(Flat, Axle)
  \textbf{Effect:} \neg At(Flat, Axle) \land At(Flat, Ground))

Action(PutOn(Špare, Axle),
  \textbf{Precond:} At(Špare, Ground) \land \neg At(Flat, Axle)
  \textbf{Effect:} \neg At(Špare, Ground) \land At(Špare, Axle))

Action(LeaveOvernight,
  \textbf{Precond:}
  \textbf{Effect:} \neg At(Špare, Ground) \land \neg At(Špare, Axle) \land \neg At(Špare, Trunk)
  \land \neg At(Flat, Ground) \land \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:

![Diagram showing the situation after two steps]
Partial-order planning example (3)

- Pick \( \neg \text{At}(Flat, Axle) \) precondition of \( \text{PutOn}(Spare, Axle) \)
- Choose \( \text{LeaveOvernight} \), effect \( \neg \text{At}(Spare, Ground) \)
- Conflict with link
  \[
  \text{Remove}(Spare, Trunk) \quad \rightarrow \quad \text{PutOn}(Spare, Axle)
  \]
- Resolve by adding \( \text{LeaveOvernight} \prec \text{Remove}(Spare, Trunk) \)

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 \( \text{Remove}(Flat, Axle) \) instead of \( \text{LeaveOvernight} \)
- Next, choose \( \text{At}(Spark, Trunk) \) precondition of \( \text{Remove}(Spare, Trunk) \)
  Choose \( \text{Start} \) to achieve this
- Pick \( \text{At}(Flat, Axle) \) precondition of \( \text{Remove}(Flat, Axle) \), choose \( \text{Start} \) to achieve it
- Final, complete, consistent plan:
Dealing with unbound variables

- In first-order case, unbound variables may occur during planning process
- Example:

  \[
  \text{Action}(\text{Move}(b, x, y),
  \begin{align*}
  \text{Precond: } & \text{On}(b, x) \land \text{Clear}(b) \land \text{Clear}(y) \\
  \text{Effect: } & \text{On}(b, y) \land \text{Clear}(x) \land \neg \text{On}(b, x) \land \neg \text{Clear}(y)
  \end{align*}
  \]
  
  achieves \(\text{On}(A, B)\) under substitution \(\{b/A, y/B\}\)

- Applying this substitution yields

  \[
  \text{Action}(\text{Move}(A, x, B),
  \begin{align*}
  \text{Precond: } & \text{On}(A, x) \land \text{Clear}(A) \land \text{Clear}(B) \\
  \text{Effect: } & \text{On}(A, B) \land \text{Clear}(x) \land \neg \text{On}(A, x) \land \neg \text{Clear}(B)
  \end{align*}
  \]
  
  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
  \[ \text{Move}(A, x, B) \xrightarrow{On(A,B)} \text{Finish} \] would be added
- If another action has effect \( \neg On(A, z) \) then this is only a conflict if \( z = B \)
- Solution: insert \textbf{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**