Planning Graphs

Chapter 10
Partial Order Planning

• Least Commitment Strategy:
  As much as possible, while planning do not commit to specific actions or action sequences (Keep your options open).

• Partial Order Planning:
  Construct a plan as a Directed Acyclic Graph (DAG) to be linearized later.
Put On Shoes:

\[
\text{Goal}(\text{RightShoeOn} \land \text{LeftShoeOn})
\]

\[
\text{Init}()
\]

\[
\text{Action}(\text{RightShoe}, \text{Precond} : \text{RightSockOn}, \text{Effect} : \text{RightShoeOn})
\]

\[
\text{Action}(\text{RightSock}, \text{Effect} : \text{RightSockOn})
\]

\[
\text{Action}(\text{LeftShoe}, \text{Precond} : \text{LeftSockOn}, \text{Effect} : \text{LeftShoeOn})
\]

\[
\text{Action}(\text{LeftSock}, \text{Effect} : \text{LeftSockOn})
\]

(Q) List 2 valid optimal plans for this world.
Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.
(Q) Suppose that POP creates 3 partial plans of size 4. How many ways can this be linearized?
11.3: POP Algorithm

- **Actions**
  - Start, Finish added to “empty” plan
  - Start - no preconditions, effect=world initial state
  - Finish - preconditions=goal, no effect

- **Ordering constraints**
  - $A \prec B$: “A before B” (no cycles allowed)

- **Causal Links**
  - $A \xrightarrow{p} B$: “A achieves p for B”
  - $\text{RightSock} \xrightarrow{\text{RightSockOn}} \text{RightShoe}$ asserts that
    - $\text{RightSockOn}$ is an effect of the $\text{RightSock}$ action and a precondition of $\text{RightShoe}$. It also asserts that $\text{RightSockOn}$ must remain true from the time of action $\text{RightSock}$ to the time of action $\text{RightShoe}$.

- **Open Preconditions**
  - Open = not achieved by some action in the plan (including Start)
(Q) Build partial order plan for shoe world.

- We need the following:
  - Actions
  - Orderings
  - Links
(Q) Write partial order plan for shoe world.

Actions: \{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}

Orderings: \{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}

Links: \{RightSock \xrightarrow{RightSockOn} RightShoe, 
LeftSock \xrightarrow{LeftSockOn} LeftShoe, 
RightShoe \xrightarrow{RightShoeOn} Finish, 
LeftShoe \xrightarrow{LeftShoeOne} Finish\}

Open Preconditions: \{\}
Consistent Plans

- Consistent Plan
  - No cycles in constraint ordering
  - No conflicts in causal links

- Solution
  - Consistent plan with no open preconditions

- Constructing a total order from a partial order:
  - Any linearization of a partial order solution!
Partial-order planning, unbound variables

- Action specification:
  \[ \text{Action}(\text{Move}(b, x, y)) \]

- Completely bound action:
  \[ \text{Move}(A, C, B) \]

- Partly bound action:
  \[ \text{Move}(A, x, B) \]

- Allows more flexibility in the plan later.

- Inequality constraints narrow possibilities.
Planning Graphs

A Planning Graph is an data structure that can be generated in polynomial time, from which plans can be extracted. Extracting a plan can take exponential time, but it can be orders of magnitude faster than forward or backward state space search.

A planning graph consists of:

- Sequence of levels corresponding to time steps
  - State (S) levels
  - Action (A) levels

  \[ S_0 \quad A_0 \quad S_1 \quad A_1 \quad S_2 \quad A_2 \quad \ldots \]

- Persistence actions
  - Dummy actions that allow literals to remain the same from one S to another.

- Leveling off
  - When two successive S-A pairs are identical
Planning Graphs

Mutual exclusion (mutex) links

- **Mutex between actions**
  - Inconsistent effects: one action negates result of other.
    - Ex: $Eat(cake)$ and persistence action $Have(cake)$
  - Interference: one of the effects of an action is the negation of the precondition of another.
    - Ex: $Eat(cake)$ interferes with $Have(cake)$ by negating precond.
  - Competing needs: a precondition of one action is mutex with the precondition of another.

- **Mutex between literals**
  - If one is the negation of the other.
    - Each pair of actions that could achieve the two literals is mutex.
Cake Example

\[\text{Init(Have(Cake))}\]
\[\text{Goal(Have(Cake) \land Eaten(Cake))}\]
\[\text{Action(Eat(Cake))}\]
\[\begin{align*}
\text{PRECOND: } & \text{Have(Cake)} \\
\text{EFFECT: } & \neg \text{Have(Cake)} \land \text{Eaten(Cake)}
\end{align*}\]
\[\text{Action(Bake(Cake))}\]
\[\begin{align*}
\text{PRECOND: } & \neg \text{Have(Cake)} \\
\text{EFFECT: } & \text{Have(Cake)}
\end{align*}\]

**Figure 11.11** The “have cake and eat cake too” problem.

**Figure 11.12** The planning graph for the “have cake and eat cake too” problem up to level \(S_2\). Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.
Graph-Plan algorithm basics

• Expand planning graph until
  All goals represented and no goals mutex, OR
  Graph levels off (in which case return failure)

• Try to extract a solution
  If successful, return the solution

• Expand the graph one more level and try again
  If graph and nogoods (list of levels and goals that can’t be achieve on them) both level off, we can return failure.

(Q) At which level of the cake graph can we attempt to extract a solution?

(Q) Give a solution to the cake world.