Chapter 2
Deliberation with Deterministic Models

2.3: Heuristic Functions
2.4: Backward Search
2.5: Plan-Space Search
2.6: Planning and Acting

Dana S. Nau
University of Maryland
Outline

2.1 State-variable representation
   - State = {values of variables}; action = changes to those values

2.2 Forward state-space search
   - Start at initial state, look for sequence of actions that achieve goal

2.3 Heuristic functions
   - How to guide a forward state-space search

2.4 Backward search
   - Start at goal state, go backwards toward initial state

2.5 Plan-space search
   - Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

2.6 Incorporating planning into an actor
   - Online lookahead, unexpected events
2.3 Heuristic Functions

Planning problem $P$ in domain $\Sigma$

- Creating a heuristic function:
  - Weaken some of the constraints that
    - restrict what the states, actions, and plans are
    - restrict when an action or plan is applicable, what goals it achieves
    - increase the costs of actions and plans

- *Relaxed* planning domain $\Sigma' = (S', A', \gamma')$ and problem $P' = (\Sigma', s'_0, g')$
  - for every solution $\pi$ for $P$, $P'$ has a solution $\pi'$ with $\text{cost}'(\pi') \leq \text{cost}(\pi)$

- Suppose we have an algorithm $A$ for solving planning problems in $\Sigma'$
  - Heuristic function $h^A(s)$ for $P$:
    - Find a solution $\pi'$ for $(\Sigma', s, g')$; return $\text{cost}(\pi')$
    - If $A$ runs quickly, then $h^A$ may be a useful heuristic function
    - If $A$ always finds optimal solutions, then $h^A$ is admissible
Example

- Relaxation: let vehicle travel in a straight line between any pair of cities
  - straight-line-distance ≤ distance by road
Domain-independent Heuristics

- Heuristic functions that can be used work in any classical planning problem
  - Additive-cost heuristic
  - Max-cost heuristic
  - Delete-relaxation heuristics
    - Optimal relaxed solution
    - Fast-forward heuristic
  - Landmark heuristics

In the book, but I’ll skip them
2.3.2 Delete-Relaxation

- Relaxation:
  - A state variable can have more than one value at the same time
  - When assigning a new value, keep the old one too

- Suppose state $s$ includes an atom $x=v$, action $a$ has effect $x \leftarrow w$
  - $\gamma^+(s,a)$ is a relaxed state
  - Includes both $x=v$ and $x=w$

$s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$

$\text{move}(r1, d3, d1)$

- pre: $\text{loc}(r1) = d3$
- eff: $\text{loc}(r1) \leftarrow d1$

$\hat{s}_1 = \gamma^+(s_0, \text{move}(r1,d3,d1))$

$= \{\text{loc}(r1) = d3, \text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
Relaxed States

- **Relaxed state** (or r-state):
  - a set $\hat{s}$ of ground atoms that includes at least 1 value for each state variable
  - represents $\{\text{all states that are subsets of } \hat{s}\}$

- Note: every state $s$ is also a relaxed state that represents $\{s\}$

\[
\{\text{loc}(r_1) = d_1, \text{loc}(r_1) = d_3, \text{cargo}(r_1) = \text{nil}, \text{loc}(c_1) = d_1\}
\]

\[
\{\text{loc}(r_1)=d_1, \text{loc}(r_1)=d_3, \text{cargo}(r_1)=\text{nil}, \text{loc}(c_1)=r_1, \text{loc}(c_1)=d_1, \text{cargo}(r_1)=c_1\}
\]
Relaxed States

- **Relaxed state** (or r-state):
  - a set $\hat{s}$ of ground atoms that includes at least 1 value for each state variable
  - represents \{all states that are subsets of $\hat{s}$\}
- Note: every state $s$ is also a relaxed state that represents \{s\}

- Action $a$ is r-applicable in $\hat{s}$ if $\hat{s}$ contains a subset that satisfies $a$’s preconditions
  - If $a$ is r-applicable then $\gamma^+(\hat{s},a) = \hat{s} \cup \gamma(s,a)$

- $\pi = \langle a_1, \ldots, a_n \rangle$ is r-applicable in $\hat{s}_0$ if there are r-states $\hat{s}_1$, $\hat{s}_2$, …, $\hat{s}_n$ such that
  - $a_1$ is r-applicable in $\hat{s}_0$ and $\gamma^+(\hat{s}_0,a_1) = \hat{s}_1$
  - $a_2$ is r-applicable in $\hat{s}_1$ and $\gamma^+(\hat{s}_1,a_2) = \hat{s}_2$
  - …
  - In this case, $\gamma^+(\hat{s},\pi) = \hat{s}_n$

Why a subset, rather than $\hat{s}$ itself?
Example

\[ \hat{s}_0 = s_0 = \{ \text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1 \} \]

move(r1, d3, d1)
   \[ \text{pre: } \text{loc}(r1) = d3 \]
   \[ \text{eff: } \text{loc}(r1) \leftarrow d1 \]

\[ \hat{s}_1 = \gamma^+(s_0, \text{move}(r1,d3,d1)) = \{ \text{loc}(r1) = d1, \text{loc}(r1) = d3, \]
   \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \]

load(r1, c1, d1)
   \[ \text{pre: } \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1, \text{loc}(r1)=d1 \]
   \[ \text{eff: } \text{cargo}(r1) \leftarrow c1, \text{loc}(c1) \leftarrow r1 \]

\[ \hat{s}_2 = \gamma^+(s_1, \text{load}(r1,c1,d1)) = \{ \text{loc}(r1)=d1, \text{loc}(r1)=d3, \]
   \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=r1, \]
   \text{loc}(c1)=d1, \text{cargo}(r1)=c1 \} \]
Relaxed Solution

- Planning problem $P = (\Sigma, s_0, g)$
  - An r-state $\hat{s}$ r-satisfies $g$ if a subset of $\hat{s}$ satisfies $g$
- $\pi$ is a relaxed solution for $P = (\Sigma, s_0, g)$ if $\gamma^+(s_0,\pi)$ r-satisfies $g$
- Example:
  
  \[ s_0 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\} \]
  
  \[ g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\} \]

  \[ \pi = \langle \text{move}(r1,d3,d1), \text{load}(r1,c1,d1) \rangle \]

  \[ \gamma^+(s_0,\pi) = \{\text{loc}(r1)=d1, \text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=r1, \text{loc}(c1)=d1, \text{cargo}(r1)=c1\} \]
Optimal Relaxed Solution Heuristic

- Given a planning problem $P = (\Sigma, s_0, g)$
- *Optimal relaxed solution* heuristic:
  - $h^+(s) = \text{minimum cost of all relaxed solutions for } (\Sigma, s, g)$
- Example:
  - $\pi = \langle \text{move}(r1,d3,d1), \text{load}(r1,c1,d1) \rangle$
    - $\text{cost}(\pi) = 2$
  - No less-costly relaxed solution, so $h^+(s_0) = 2$
- How does this compare with $h^*(s_0)$?

$s_0 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}$

$g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\}$
Example

- $s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
- In $s_0$, two applicable actions
  - $a_1 = \text{move}(r1,d3,d1)$
  - $s_1 = \{\text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
  - $a_2 = \text{move}(r1,d3,d2)$
  - $s_2 = \{\text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
- GBFS evaluates $h^+(s_1)$ and $h^+(s_2)$, and chooses to move to whichever is smaller
- What are $h^+(s_1)$ and $h^+(s_2)$?
- What does GBFS choose?

$g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\}$
Fast-Forward Heuristic

- Every state is also a relaxed state
- Every solution is also a relaxed solution

- $h^+(s) =$ minimum cost of all relaxed solutions
  - Thus $h^+$ is admissible
  - Problem: computing it is NP-hard

- Fast-Forward Heuristic, $h^{FF}$
  - An approximation of $h^+$ that’s easier to compute
    - Upper bound on $h^+$
  - Name comes from a planner called Fast Forward
Preliminaries

- Let $A_1$ be a set of actions that all are r-applicable in $\hat{s}_0$
  - Can apply them in any order and get same result
    - Define $\gamma^+(\hat{s}_0, A_1) = \hat{s} \cup \bigcup \{\text{eff}(a) \mid a \in A_1\}$

- Let $\hat{s}_1 = \gamma^+(\hat{s}_0, A_1)$

- Suppose $A_2$ is a set of actions that are r-applicable in $\hat{s}_1$
  - Define $\gamma^+(\hat{s}, \langle A_1, A_2 \rangle) = \gamma^+(\hat{s}_1, A_2)$

- ...

- Define $\gamma^+(\hat{s}, \langle A_1, A_2, \ldots, A_n \rangle)$ in the obvious way
Fast-Forward Heuristic

\[
\text{HFF}(\Sigma, s, g): \quad \text{\textit{find a minimal relaxed solution, return its cost}}
\]

\[
\begin{align*}
\text{\texttt{// construct a relaxed solution } } & \langle A_1, A_2, \ldots, A_k \rangle: \\
\hat{s}_0 & \leftarrow s \\
\text{for } k = 1 \text{ by 1 until a subset of } \hat{s}_k \text{ r-satisfies } g \\
A_k & = \{\text{all actions r-applicable in } \hat{s}_{k-1} \}; \hat{s}_k = \gamma^+(s_{k-1}, A_k) \\
\text{if } k > 1 \text{ and } \hat{s}_k = \hat{s}_{k-1} \text{ then return } \infty \quad \text{\textit{// there’s no solution}}
\end{align*}
\]

\[
\text{\texttt{// extract minimal relaxed solution } } \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle:
\]

\[
\hat{g}_k = g
\]

\[
\text{for } i = k, k-1, \ldots, 1:
\]

\[
\hat{a}_i = \text{any minimal subset of } A_i \text{ such that } \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \text{ r-satisfies } \hat{g}_i \\
\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)
\]

\[
\text{return } \sum \text{ costs of the actions in } \hat{a}_1, \ldots, \hat{a}_k \quad \text{\textit{// upper bound on } h^+}
\]

- Define \( h^{\text{FF}}(s) = \text{the value returned by HFF}(\Sigma, s, g) \)

\text{ambiguous}
Example

- $s_0 = \{\text{loc}(c1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}\}$

- Two applicable actions
  - $a_1 = \text{move}(r1,d3,d1)$
  - $s_1 = \gamma(s_0,a_1) = \{\text{loc}(c1) = d1, \text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}\}$
  - $a_2 = \text{move}(r1,d3,d2)$
  - $s_2 = \gamma(s_0,a_2) = \{\text{loc}(c1) = d1, \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}\}$

- GBFS using $h^{FF}$
  - Compute $h^{FF}(s_1)$ and $h^{FF}(s_2)$
  - Move to whichever is smaller

- Next several slides: $h^{FF}(s_1)$ and $h^{FF}(s_2)$

$g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\}$
Example

// construct a relaxed solution $\langle A_1, A_2, \ldots, A_k \rangle$:

\[ \hat{s}_0 \leftarrow s \]

for $k = 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies $g$

\[ A_k = \{ \text{all actions r-applicable in } \hat{s}_{k-1} \} ; \hat{s}_k = \gamma^+(s_{k-1}, A_k) \]

if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return $\infty$  // there’s no solution

$s_1 = \{ \text{loc}(r_1) = d_1, \text{cargo}(r_1) = \text{nil}, \text{loc}(c_1) = d_1 \}$

$g = \{ \text{loc}(r_1) = d_3, \text{loc}(c_1) = r_1 \}$

Relaxed Planning Graph (RPG) from $\hat{s}_0 = s_2$ to $g$:

Atoms in $\hat{s}_0 = s_1$:
- $\text{loc}(r_1) = d_1$
- $\text{loc}(c_1) = d_1$
- $\text{cargo}(r_1) = \text{nil}$

Actions in $A_1$:
- $\text{move}(r_1, d_1, d_2)$
- $\text{move}(r_1, d_1, d_3)$
- $\text{load}(r_1, c_1, d_1)$
- $\text{load}(r_1, c_1, d_2)$

Atoms in $\hat{s}_1$:
- $\text{loc}(r_1) = d_1$
- $\text{loc}(c_1) = d_1$
- $\text{cargo}(r_1) = \text{nil}$
- $\text{loc}(r_1) = d_3$
- $\text{load}(r_1, c_1, d_1)$

$\langle A_1 \rangle$ is a relaxed solution

$\gamma^+(s_0, A_1)$ r-satisfies $g$
Example

```
// extract minimal relaxed solution \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle:
\hat{g}_k = g
for i = k, k-1, \ldots, 1:
    \hat{a}_i = any minimal subset of \mathcal{A}_i such that \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \text{ r-satisfies } \text{pre}(\hat{a}_i)
    \hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)
```

Relaxed Planning Graph (RPG) from \( \hat{s}_0 = s_2 \) to \( g \):

- Atoms in \( \hat{s}_0 = s_1 \):
  - \text{loc}(r1) = d1
  - \text{loc}(c1) = d1
  - \text{cargo}(r1) = \text{nil}
- Actions in \( \mathcal{A}_1 \):
  - \text{move}(r1, d1, d2)
  - \text{move}(r1, d1, d3)
  - \text{load}(r1, c1, d1)
- Atoms in \( s_1 \):
  - \text{loc}(r1) = d1
  - \text{loc}(c1) = d1
  - \text{cargo}(r1) = \text{nil}

- \( \langle \hat{a}_1 \rangle \) is a minimal relaxed solution
- each action’s cost is 1, so \( h^{\text{FF}}(s_1) = 2 \)
Example

RPG from \( \hat{s}_0 = s_2 \) to \( g \):

Atoms in \( \hat{s}_0 = s_1 \):
- \( \text{loc}(r1) = d2 \)
- \( \text{loc}(c1) = d1 \)
- \( \text{cargo}(r1) = \text{nil} \)

Actions in \( A_1 \):
- \( \text{move}(r1,d2,d3) \) — \( \text{loc}(r1) = d3 \)
- \( \text{move}(r1,d2,d1) \) — \( \text{loc}(r1) = d1 \)

from \( \hat{s}_0 \):
- \( \text{loc}(r1) = d2 \)
- \( \text{loc}(c1) = d1 \)
- \( \text{cargo}(r1) = \text{nil} \)

Atoms in \( \hat{s}_1 \):
- \( \text{loc}(r1) = d3 \)
- \( \text{loc}(c1) = d1 \)
- \( \text{cargo}(r1) = \text{nil} \)

Actions in \( A_2 \):
- \( \text{move}(r1,d3,d2) \)
- \( \text{move}(r1,d1,d2) \)
- \( \text{move}(r1,d3,d1) \)
- \( \text{move}(r1,d1,d3) \)
- \( \text{load}(r1,c1,d1) \)

Atoms in \( \hat{s}_2 \):
- \( \text{loc}(r1) = d2 \)
- \( \text{loc}(c1) = d1 \)
- \( \text{cargo}(r1) = \text{nil} \)
- \( \text{loc}(r1) = d3 \)
- \( \text{cargo}(r1) = c1 \)
- \( \text{loc}(c1) = r1 \)

\( s_2 = \{ \text{loc}(r1)=d2, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d2 \} \)

\( g = \{ \text{loc}(r1)=d3, \text{loc}(c1)=r1 \} \)

\( \langle A_1, A_2 \rangle \) is a relaxed solution
Example

RPG from $\hat{s}_0 = s_2$ to $g$:

Atoms in $\hat{s}_0 = s_1$:
- $\text{loc}(r1) = d2$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

Atomic in $\hat{s}_0$:
- $\text{move}(r1,d2,d1)$

Atoms in $\hat{s}_1$:
- $\text{loc}(r1) = d1$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

Actions in $A_1$:
- $\text{move}(r1,d2,d3)$
- $\text{move}(r1,d2,d1)$

Actions in $A_2$:
- $\text{move}(r1,d3,d2)$
- $\text{move}(r1,d1,d2)$
- $\text{move}(r1,d1,d3)$

$\hat{a}_1$:
- $\langle \text{move}(r1,d2,d1) \rangle$

$\hat{a}_2$:
- $\langle \text{move}(r1,d1,d3) \rangle$

$\langle \hat{a}_1, \hat{a}_2 \rangle$ is a minimal relaxed solution
- each action’s cost is 1, so $h^{FF}(s_2) = 3$

$s_2 = \{\text{loc}(r1)=d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d2\}$

$g = \{\text{loc}(r1)=d3, \text{loc}(c1) = r1\}$
Fast-Forward Heuristic

- Running time is polynomial in $|A| + \sum_{x \in X} |\text{Range}(x)|$

- $h^{\text{FF}}(s) =$ value returned by $H^{\text{FF}}(\Sigma, s, g)$
  \[ = \sum \text{costs of } \hat{a}_1, \ldots, \hat{a}_k \]

- Value is ambiguous
  - each $\hat{a}_i$ is a minimal set of actions such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies pre($\hat{a}_i$)
  - depends on which minimal subsets we choose

- $\text{minimal}$ doesn’t mean $\text{smallest}$
  - $h^{\text{FF}}(s) \geq h^+(s) =$ smallest cost of any relaxed plan from $s$ to goal
  - $h^{\text{FF}}$ not admissible
Example

- **Poll.** Suppose the goal atoms are $c_7$, $c_8$, $c_9$. How many minimal solutions are there?
2.3.3 Landmark Heuristics

- \( P = (\Sigma, s_0, g) \) be a planning problem
- Let \( \varphi = \varphi_1 \lor \ldots \lor \varphi_m \) be a disjunction of ground atoms
- \( \varphi \) is a \textit{landmark} for \( P \) if \( \varphi \) is true at some point in every solution for \( P \)

**Example Landmarks**
- \( \text{loc}(r1)=d1 \)
- \( \text{loc}(r1)=d3 \lor \text{loc}(r1)=d2 \)
- \( \text{loc}(r1)=d3 \)

\[ s_0 = \{ \text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1 \} \]

\[ g = \{ \text{loc}(r1)=d3, \text{loc}(c1)=r1 \} \]
Why are Landmarks Useful?

- Breaks down a problem into smaller subproblems

  Suppose $m_1, m_2, m_3$ are landmarks
  
  - Every solution to $P$ must achieve $m_1, m_2, m_3$

- Possible strategy:
  
  - find a plan to go from $s_0$ to any state $s_1$ that satisfies $m_1$
  - find a plan to go from $s_1$ to any state $s_2$ that satisfies $m_2$
  - …
Computing Landmarks

- Worst-case complexity:
  - Deciding whether $\varphi$ is a landmark is PSPACE-hard
  - As hard as solving the planning problem itself
- But there are often useful landmarks that can be found more easily
  - polynomial time
  - Going to see one such procedure based on *Relaxed Planning Graphs*
- Why Relaxed Planning Graphs?
  - Solving relaxed planning problems easier
    - Computing landmarks for relaxed planning problems easier
  - A landmark for a relaxed planning problem is a landmark for the original planning problem as well
RPG-based Landmark Computation

- **Main intuition:**
  - if $\varphi$ is a landmark, can get new landmarks from the preconditions of the actions that achieve $\varphi$

- **Example:**
  - goal $g$
  - $\{a_1, a_2\} = \text{all actions that achieve } g$
  - $\text{pre}(a_1) = \{p_1, q\}$
  - $\text{pre}(a_2) = \{q, p_2\}$
  - To achieve $g$, must achieve $(p_1 \land q) \lor (p_2 \land q)$
    - same as $q \land (p_1 \lor p_2)$
  - Landmarks:
    - $q$
    - $p_1 \lor p_2$
RPG-based Landmark Computation

- Suppose goal is $g = \{g_1, g_2, \ldots, g_k\}$
  - Trivially, every $g_i$ is a landmark
- Suppose $g_1 = \text{loc}(r1)=d1$
  - Two actions can achieve $g_1$: move(r1,d3,d1) and move(r1,d2,d1)
- Preconditions $\text{loc}(r1)=d3$ and $\text{loc}(r1)=d2$
- New landmark:
  $$\phi' = \text{loc}(r1)=d3 \lor \text{loc}(r1)=d2$$

$m_0 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}$

- move($r$, $d$, $e$)
  - pre: $\text{loc}(r)=d$
  - eff: $\text{loc}(r) \leftarrow e$
- load($r$, $c$, $l$)
  - pre: $\text{cargo}(r)=\text{nil}$, $\text{loc}(c)=l$, $\text{loc}(r)=l$
  - eff: $\text{cargo}(r) \leftarrow c$, $\text{loc}(c) \leftarrow r$
- unload($r$, $c$, $l$)
  - pre: $\text{loc}(c)=r$, $\text{loc}(r)=l$
  - eff: $\text{cargo}(r) \leftarrow \text{nil}$, $\text{loc}(c) \leftarrow l$
RPG-based Landmark Computation

RPG-Landmarks\( (s_0, g = \{g_1, g_2, \ldots, g_k\}) \)

\[
\begin{align*}
\text{queue} & \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \quad \text{Landmarks} \leftarrow \emptyset \\
\text{while queue} & \neq \emptyset \\
& \quad \text{remove a } g_i \text{ from queue; add it to Landmarks} \\
R & \leftarrow \{\text{actions whose effects include } g_i\} \\
\text{if } s_0 \text{ satisfies } & \text{pre}(a) \text{ for some } a \in R \text{ then return Landmarks} \\
\text{generate RPG from } s_0 & \text{ using } A \setminus R, \text{ stopping when } \hat{s}_k = \hat{s}_{k-1} \\
N & \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\} \\
\text{if } N = \emptyset & \text{ then return failure} \\
Preconds & \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0 \\
\Phi & \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{ every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\} \\
\text{for each } \phi & \in \Phi: \\
& \quad \text{add } \phi \text{ to queue} \\
\text{return Landmarks}
\end{align*}
\]
RPG-based Landmark Computation

\[
\text{RPG-Landmarks}(s_0, g = \{g_1, g_2, \ldots, g_k\})
\]

\[
\begin{align*}
\text{queue} & \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \quad \text{Landmarks} \leftarrow \emptyset \\
\text{while } \text{queue} \neq \emptyset \quad & \\
& \quad \text{remove a } g_i \text{ from } \text{queue}; \text{ add it to } \text{Landmarks} \\
& \quad R \leftarrow \{\text{actions whose effects include } g_i\} \\
& \quad \text{if } s_0 \text{ satisfies } \text{pre}(a) \text{ for some } a \in R \text{ then return } \text{Landmarks} \\
& \quad \text{generate RPG from } s_0 \text{ using } A \setminus R, \text{ stopping when } \hat{s}_k = \hat{s}_{k-1} \\
& \quad N \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\} \\
& \quad \text{if } N = \emptyset \text{ then return failure} \\
& \quad \text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0 \\
& \quad \Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{ every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\} \\
& \quad \text{for each } \phi \in \Phi: \\
& \quad \quad \text{add } \phi \text{ to } \text{queue} \\
& \quad \text{return } \text{Landmarks}
\end{align*}
\]
RPG-based Landmark Computation

**RPG-Landmarks**($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

$$\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \ Landmarks \leftarrow \emptyset$$

while $\text{queue} \neq \emptyset$

\begin{itemize}
  \item remove a $g_i$ from $\text{queue}$; add it to $\text{Landmarks}$
  \item $R \leftarrow \{\text{actions whose effects include } g_i\}$
  \item if $s_0$ satisfies $\text{pre}(a)$ for some $a \in R$ then return $\text{Landmarks}$
  \item generate RPG from $s_0$ using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$
  \item $N \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\}$
  \item if $N = \emptyset$ then return failure
  \item $\text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0$
  \item $\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{ every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\}$
  \item for each $\varphi \in \Phi$:
    \begin{itemize}
      \item add $\varphi$ to $\text{queue}$
    \end{itemize}
\end{itemize}

return $\text{Landmarks}$
RPG-based Landmark Computation

RPG-Landmarks($s_0$, $g = \{g_1, g_2, \ldots, g_k\}$)

\[
\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \quad \text{Landmarks} \leftarrow \emptyset
\]

while queue $\neq \emptyset$

remove a $g_i$ from queue; add it to Landmarks

$R \leftarrow \{\text{actions whose effects include } g_i\}$

if $s_0$ satisfies pre($a$) for some $a \in R$ then return Landmarks

generate RPG from $s_0$ using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow \{\text{all actions in } R \text{ that are r-applicable in } \hat{s}_k\}$

if $N = \emptyset$ then return failure

$Preconds \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4$, every action in $N$ has at least one $p_i$ as a precondition, and every $p_i \in Preconds\}$

for each $\varphi \in \Phi$:

add $\varphi$ to queue

return Landmarks

“necessary” actions: the only ones that can be r-applied and achieve $g_i$
RPG-based Landmark Computation

RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

\[
\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \quad \text{Landmarks} \leftarrow \emptyset
\]

while \(\text{queue} \neq \emptyset\)

remove a \(g_i\) from \(\text{queue}\); add it to \(\text{Landmarks}\)

\(R \leftarrow \{\text{actions whose effects include } g_i\}\)

if \(s_0\) satisfies pre(\(a\)) for some \(a \in R\) then return \(\text{Landmarks}\)

generate RPG from \(s_0\) using \(A \setminus R\), stopping when \(\hat{s}_k = \hat{s}_{k-1}\)

\(N \leftarrow \{\text{all actions in } R\text{ that are } r\text{-applicable in } \hat{s}_k\}\)

if \(N = \emptyset\) then return failure

\(\text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0\)

\(\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{every action in } N\text{ has at least one } p_i\text{ as a precondition, and every } p_i \in \text{Preconds}\}\)

for each \(\phi \in \Phi\):

add \(\phi\) to \(\text{queue}\)

return \(\text{Landmarks}\)
Example

RPG-Landmarks\((s_0, g = \{g_1, g_2, \ldots, g_k\})\)

\[
\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \text{Landmarks} \leftarrow \emptyset
\]

while \(\text{queue} \neq \emptyset\)

remove a \(g_i\) from \(\text{queue}\); add it to \(\text{Landmarks}\)

\(R \leftarrow \{\text{actions whose effects include } g_i\}\)

if \(s_0\) satisfies \(\text{pre}(a)\) for some \(a \in R\) then return \(\text{Landmarks}\)

generate RPG from \(s_0\) using \(A \setminus R\), stopping when \(\hat{s}_k = \hat{s}_{k-1}\)

\(N \leftarrow \{\text{all actions in } R \text{ that are } \text{r-applicable in } \hat{s}_k\}\)

if \(N = \emptyset\) then return failure

\(\text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0\)

\(\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\}\)

for each \(\varphi \in \Phi\):

add \(\varphi\) to \(\text{queue}\)

return \(\text{Landmarks}\)

\(s_0 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}\)

\(g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\}\)

\[
\text{queue} = \{\text{loc}(c1)=r1\}
\]

\(\text{Landmarks} = \emptyset\)

load\((r, c, l)\)

pre: \(\text{cargo}(r)=\text{nil}, \text{loc}(c)=l, \text{loc}(r)=l\)

eff: \(\text{cargo}(r)\leftarrow c, \text{loc}(c)\leftarrow r\)

move\((r, d, e)\)

pre: \(\text{loc}(r)=d\)

eff: \(\text{loc}(r)\leftarrow e\)

unload\((r, c, l)\)

pre: \(\text{loc}(c)=r, \text{loc}(r)=l\)

eff: \(\text{cargo}(r)\leftarrow \text{nil}, \text{loc}(c)\leftarrow l\)

true in \(s_0\)
Example

RPG-Landmarks($s_0, \ g = \{g_1, g_2, \ldots, g_k\}$)

\[
\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \ \text{Landmarks} \leftarrow \emptyset
\]

\[
\text{while } \text{queue} \neq \emptyset
\]

remove a $g_i$ from queue; add it to Landmarks

$\ R \leftarrow \{\text{actions whose effects include } g_i\}$

if $s_0$ satisfies $\text{pre}(a)$ for some $a \in R$ then return Landmarks

generate RPG from $s_0$ using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\}$

if $N = \emptyset$ then return failure

$\text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{ every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\}$

for each $\varphi \in \Phi$:

add $\varphi$ to queue

return Landmarks

\[
s_0 = \{\text{loc}(r1)=d3, \ \text{cargo}(r1)=\text{nil}, \ \text{loc}(c1)=d1\}
\]

\[
\text{g} = \{\text{loc}(r1)=d3, \ \text{loc}(c1)=r1\}
\]

queue = $\emptyset$

Landmarks = $\{\text{loc}(c1)=r1\}$

$R = \{\text{load}(r1,c1,d1), \ \text{load}(r1,c1,d2), \ \text{load}(r1,c1,d3)\}$

\[
\text{load}(r, c, l)
\]

pre: $\text{cargo}(r)=$nil, $\text{loc}(c)=l$, $\text{loc}(r)=l$

eff: $\text{cargo}(r)\leftarrow c$, $\text{loc}(c)\leftarrow r$

\[
\text{move}(r, d, e)
\]

pre: $\text{loc}(r)=d$

eff: $\text{loc}(r)\leftarrow e$

\[
\text{unload}(r, c, l)
\]

pre: $\text{loc}(c)=r, \ \text{loc}(r)=l$

eff: $\text{cargo}(r)\leftarrow \text{nil}, \ \text{loc}(c)\leftarrow l$
Example

RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

\[
\text{queue} \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \text{Landmarks} \leftarrow \emptyset
\]

while queue $\neq \emptyset$

- remove a $g_i$ from queue; add it to Landmarks
- $R \leftarrow \{\text{actions whose effects include } g_i\}$
- if $s_0$ satisfies $\text{pre}(a)$ for some $a \in R$ then return Landmarks

generate RPG from $s_0$ using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\}$

if $N = \emptyset$ then return failure

$\text{Preconds} \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \text{ every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in \text{Preconds}\}$

for each $\varphi \in \Phi$:

- add $\varphi$ to queue

return Landmarks

\[
\text{load}(r, c, l)
\]

pre: cargo($r$)=nil, loc($c$)=l, loc($r$)=l

\[
\text{eff: cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r
\]

\[
\text{move}(r, d, e)
\]

pre: loc($r$)=d

\[
\text{eff: loc}(r) \leftarrow e
\]

\[
\text{unload}(r, c, l)
\]

pre: loc($c$)=r, loc($r$)=l

\[
\text{eff: cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l
\]

queue $= \emptyset$

Landmarks = \{loc(c1)=r1\}

$R = \{\text{load}(r1,c1,d1), \text{load}(r1,c1,d2), \text{load}(r1,c1,d3)\}$

$N = \{\text{load}(r1,c1,d1)\}$

\[
\text{Relaxed planning graph using } A \setminus R
\]

$\hat{s}_0$:

- loc(c1)=d1
- loc(r1)=d3
- cargo(r1)=nil

$A_1$:

- move(r1,d3,d1)
- move(r1,d3,d2)

both $\hat{s}_1$ and $\hat{s}_2$:

- loc(r1)=d1
- loc(r1)=d2
- loc(c1)=d1
- loc(r1)=d3
- cargo(r1)=nil

load(r1,c1,d1)
Example

RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

$$queue \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \ Landmarks \leftarrow \emptyset$$

while $queue \neq \emptyset$

remove a $g_i$ from $queue$; add it to $Landmarks$

$\ R \leftarrow \{\text{actions whose effects include } g_i\}$

if $s_0$ satisfies pre($a$) for some $a \in R$ then return $Landmarks$

generate RPG from $s_0$ using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow \{\text{all actions in } R \text{ that are } r\text{-applicable in } \hat{s}_k\}$

if $N = \emptyset$ then return failure

$Preconds \leftarrow \bigcup \{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4, \text{every action in } N \text{ has at least one } p_i \text{ as a precondition, and every } p_i \in Preconds\}$

for each $\phi \in \Phi$: add $\phi$ to $queue$

return $Landmarks$

$s_0 = \{\text{loc(r1)=d3, cargo(r1)=nil, loc(c1)=d1}\}$

$g = \{\text{loc(r1)=d3, loc(c1)=r1}\}$
Landmark Heuristic

• Every solution to the problem needs to achieve all the computed landmarks

• One possible heuristic: 
  \[ h_{sl}(s) = \text{number of landmarks returned by RPG-Landmarks} \]

• **Poll**: Is this heuristic admissible?
  
  1. Yes  
  2. No
Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  \[ h^{sl}(s) = \text{number of landmarks returned by RPG-Landmarks} \]
- Is this heuristic admissible?
  \[ \text{No} \]

\[ g = \{g_1, g_2\} \]
Two landmarks: \( g_1, g_2 \)
Optimal plan: \( \langle a_1 \rangle \), length = 1

- There are other more-advanced landmark heuristics
  - Some of them are admissible
  - Check textbook for references
Summary

- 2.3 Heuristic Functions
  - Straight-line distance example
  - Delete-relaxation heuristics
    - relaxed states, $\gamma^+$, $h^+$, HFF, $h^{\text{FF}}$
  - Disjunctive landmarks, RPG-Landmark, $h^{\text{sl}}$
    - Get necessary actions by making RPG for all non-relevant actions
2.6 Incorporating Planning into an Actor

- Plans are abstract
  - Need additional refinement
  - (Chapter 3)

The best laid schemes o’ mice an’ men, 
Gang aft agley.
–Robert Burns

- Plans don’t always work
  - What to do about it?
Service Robot

\[ s_0 = \{ \text{loc}(r1)=\text{loc}3, \text{loc}(o7)=\text{loc}1, \text{cargo}(r1)=\text{nil} \} \]
\[ g = \{ \text{loc}(o7)=\text{loc}2 \} \]
\[ \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \]
\[ a_1 = \text{go}(r1,\text{loc}3,\text{hall}) \]
\[ a_2 = \text{navigate}(r1,\text{hall},\text{loc}1) \]
\[ a_3 = \text{take}(r1,\text{loc}1,o7) \]
\[ a_4 = \text{navigate}(r1,\text{loc}1,\text{loc}2) \]
\[ a_5 = \text{put}(r1,\text{loc}2,o7) \]

**go** \((r, l, m)\)
- \text{pre}: adjacent\((l,m)\), \text{loc}(r)=l
- \text{eff}: \text{loc}(r) \leftarrow m

**navigate** \((r, l, m)\)
- \text{pre}: \neg adjacent\((l,m)\), \text{loc}(r)=l
- \text{eff}: \text{loc}(r) \leftarrow m

**take** \((r, l, o)\)
- \text{pre}: \text{loc}(r)=l, \text{loc}(o)=l, \text{cargo}(r)=\text{nil}
- \text{eff}: \text{loc}(o) \leftarrow r, \text{cargo}(r) \leftarrow o

---

**Service Robot**

- respond to user requests
  - bring o7 to loc2
  
  - a1: go to hallway
  - a2: navigate to loc1
  - a3: fetch o7
  - a4: navigate to loc2
  - a5: deliver o7

- move to door
- open door
- get out
- close door

**Tasks**
- identify type of door
- move close to knob
- grasp knob
- turn knob
- maintain
- pull
- monitor
- move back
- pull
- monitor
- ungrasp
- get out
- close door
- respond to user requests

**States**

- Initial state: \( s_0 = \{ \text{loc}(r1)=\text{loc}3, \text{loc}(o7)=\text{loc}1, \text{cargo}(r1)=\text{nil} \} \)
- Goal state: \( g = \{ \text{loc}(o7)=\text{loc}2 \} \)
- Plan: \( \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \)

**Actions**
- \( a_1 = \text{go}(r1,\text{loc}3,\text{hall}) \)
- \( a_2 = \text{navigate}(r1,\text{hall},\text{loc}1) \)
- \( a_3 = \text{take}(r1,\text{loc}1,o7) \)
- \( a_4 = \text{navigate}(r1,\text{loc}1,\text{loc}2) \)
- \( a_5 = \text{put}(r1,\text{loc}2,o7) \)

**Preconditions**
- go: adjacent\((l,m)\), \text{loc}(r)=l
- navigate: \neg adjacent\((l,m)\), \text{loc}(r)=l
- take: \text{loc}(r)=l, \text{loc}(o)=l, \text{cargo}(r)=\text{nil}

**Efficiencies**
- go: \text{loc}(r) \leftarrow m
- navigate: \text{loc}(r) \leftarrow m
- take: \text{loc}(o) \leftarrow r, \text{cargo}(r) \leftarrow o

**Variables**

- \( s_0 = \{ \text{loc}(r1)=\text{loc}3, \text{loc}(o7)=\text{loc}1, \text{cargo}(r1)=\text{nil} \} \)
- \( g = \{ \text{loc}(o7)=\text{loc}2 \} \)
- \( \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \)
- \( a_1 = \text{go}(r1,\text{loc}3,\text{hall}) \)
- \( a_2 = \text{navigate}(r1,\text{hall},\text{loc}1) \)
- \( a_3 = \text{take}(r1,\text{loc}1,o7) \)
- \( a_4 = \text{navigate}(r1,\text{loc}1,\text{loc}2) \)
- \( a_5 = \text{put}(r1,\text{loc}2,o7) \)

---

Nau – Lecture slides for Automated Planning and Acting
Service Robot

\[ s_0 = \{ \text{loc}(r1)=\text{loc}3, \text{loc}(o7)=\text{loc}1, \text{cargo}(r1)=\text{nil} \} \]
\[ g = \{ \text{loc}(o7)=\text{loc}2 \} \]
\[ \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \]
\[ a_1 = \text{go}(r1,\text{loc}3,\text{hall}) \]
\[ a_2 = \text{navigate}(r1,\text{hall},\text{loc}1) \]
\[ a_3 = \text{take}(r1,\text{loc}1,o7) \]
\[ a_4 = \text{navigate}(r1,\text{loc}1,\text{loc}2) \]
\[ a_5 = \text{put}(r1,\text{loc}2,o7) \]

- **Execution failures** – “open door” fails
- **Unexpected events** – someone loads an object onto r1
- **Incorrect info** – navigation error, \( a_2 \) goes to wrong place
- **Partial information** – don’t know \( \text{loc}(o7) \)

respond to user requests

bring o7 to loc2

\[ a_1 \quad \text{go to hallway} \]
\[ a_2 \quad \text{navigate to loc1} \]
\[ a_3 \quad \text{fetch o7} \]
\[ a_4 \quad \text{navigate to loc2} \]
\[ a_5 \quad \text{deliver o7} \]

move to door  open door  get out  close door

identify type of door  move close to knob  grasp knob  turn knob  maintain

ungrasp  pull  monitor

\[ s_0 = \{ \text{loc}(r1)=\text{loc}3, \text{loc}(o7)=\text{loc}1, \text{cargo}(r1)=\text{nil} \} \]
\[ g = \{ \text{loc}(o7)=\text{loc}2 \} \]
\[ \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \]
\[ a_1 = \text{go}(r1,\text{loc}3,\text{hall}) \]
\[ a_2 = \text{navigate}(r1,\text{hall},\text{loc}1) \]
\[ a_3 = \text{take}(r1,\text{loc}1,o7) \]
\[ a_4 = \text{navigate}(r1,\text{loc}1,\text{loc}2) \]
\[ a_5 = \text{put}(r1,\text{loc}2,o7) \]

S. M. Nau – Lecture slides for *Automated Planning and Acting*
Using Planning in Acting

Run-Lookahead($\Sigma, g$)

while ($s \leftarrow$ abstraction of observed state $\xi$) $\not\equiv g$ do

$\pi \leftarrow$ Lookahead($\Sigma, s, g$)

if $\pi = \text{failure}$ then return failure

$a \leftarrow \text{pop-first-action}(\pi)$; perform$(a)$

- Lookahead is the planner
- Receding horizon:
  - Call Lookahead, obtain $\pi$, perform 1st action, call Lookahead again …
  - Like game-tree search (chess, checkers, etc.)
- Useful when unpredictable things are likely to happen
  - Replans immediately
- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if $\xi$ changes during the wait?
Using Planning in Acting

Run-Lazy-Lookahead($\Sigma, g$)

$s \leftarrow$ abstraction of observed state $\xi$

while $s \not\models g$ do

$\pi \leftarrow$ Lookahead($\Sigma, s, g$)

if $\pi =$ failure then return failure

while $\pi \neq \langle \rangle$ and $s \not\models g$ and Simulate($\Sigma, s, g, \pi$) $\neq$ failure do

$a \leftarrow$ pop-first-action($\pi$); perform($a$)

$s \leftarrow$ abstraction of observed state $\xi$

- Call Lookahead, execute the plan as far as possible, don’t call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - Could just compute $\gamma(s, \pi)$, or could do something more detailed
    - lower-level refinement, physics-based simulation
- Potential problems
  - may might miss opportunities to replace $\pi$ with a better plan
  - without Simulate, may not detect problems until it’s too late
Using Planning in Acting

\[
\text{Run-Concurrent-Lookahead}(\Sigma, g)
\]

\[
\pi \leftarrow \emptyset; \quad s \leftarrow \text{abstraction of observed state } \xi
\]

thread 1: // threads 1 and 2 run concurrently

loop

\[
\pi \leftarrow \text{Lookahead}(\Sigma, s, g)
\]

thread 2:

loop

if \( s \models g \) then return success

else if \( \pi = \text{failure} \) then return failure

else if \( \pi \neq \emptyset \) and \( \text{Simulate}(\Sigma, s, g, \pi) \neq \text{failure} \) then

\[
a \leftarrow \text{pop-first-action}(\pi); \quad \text{perform}(a)
\]

\[
s \leftarrow \text{abstraction of observed state } \xi
\]

- May detect opportunities earlier than Run-Lazy-Lookahead
  - But may miss some that Run-Lazy-Lookahead would find
- Without Simulate, may fail to detect problems until it’s too late
  - Not as bad at this as Run-Lazy-Lookahead
  - Possible work-around: restart Lookahead each time \( s \) changes
How to do Lookahead

- **Subgoaling**
  - Instead of planning for \( g \), plan for a subgoal \( g' \)
  - Once \( g' \) is achieved, plan for next subgoal

- **Receding horizon**
  - Return a plan that goes just part-way to \( g' \)
  - \( E.g., \) cut off search at
    - every plan whose cost exceeds some value \( c_{\text{max}} \)
    - or whose length exceeds some value \( l_{\text{max}} \)
    - or when no time is left

Planning stage
Acting stage
Receding-Horizon Search

Deterministic-Search(Σ, s₀, g)

Frontier ← \{⟨⟩, s₀\}
Expanded ← ∅

while Frontier ≠ ∅ do
    select a node ν = (π, s) ∈ Frontier (i)
    remove ν from Frontier
    add ν to Expanded
    if s satisfies g then return π (ii)
    \( \text{Children} \leftarrow \{ (\pi.a, γ(s,a)) \mid s \text{ satisfies } \text{pre}(a) \} \)
    \( \text{prune 0 or more nodes from } \text{Children, Frontier, Expanded} \) (iii)
    Frontier ← Frontier ∪ Children

return failure

- After line (i), put something like these:

  - cost-based cutoff:
    
    \[
    \text{if } \text{cost}(\pi) + h(s) > c_{\text{max}} \text{ then return } \pi
    \]

  - length-based cutoff:
    
    \[
    \text{if } |\pi| > l_{\text{max}} \text{ then return } \pi
    \]

  - time-based cutoff:
    
    \[
    \text{if } \text{time-left}() = 0 \text{ then return } \pi
    \]
Partial or Non-Optimal Plans

- **Sampling**
  - Planner is a modified version of greedy algorithm
    - Make randomized choice in line 4
    - Run several times, get several solutions
    - Return best one
  - Actor calls the planner repeatedly as it acts
    - An analogous technique is used in the game of go

\[
\text{Greedy}(\Sigma, s, g, Visited)
\]
1. if $s$ satisfies $g$ then return $\pi$
2. $Act \leftarrow \{a \in A \mid s \text{ satisfies pre}(a) \text{ and } \gamma(s, a) \not\in Visited\}$
3. if $Act = \emptyset$ then return failure
4. $a \leftarrow \text{arg min}_{a \in Act} h(\gamma(s, a))$
5. $\pi \leftarrow \text{Greedy}(\Sigma, \gamma(s, a), g, Visited \cup \{s\})$
6. if $\pi \neq \text{failure}$ then return $a.\pi$
7. return failure
Example

- **Killzone 2**
  - “First-person shooter” game
  - Special-purpose AI planner
    - Plans enemy actions at the squad level
      - Subproblems; solution plans are maybe 4–6 actions long
    - Different planning algorithm than what we’ve discussed so far
      - Hierarchical refinement as in Chapter 3
    - Quickly generates a plan for a subgoal
    - Replans several times per second as the world changes

- Why it worked:
  - Don’t *want* to get the best possible plan
  - Need actions that appear believable and consistent to human users
  - Need them very quickly
Summary

- 2.6 Incorporating Planning into an actor
  - Things that can go wrong while acting
  - Algorithms
    - Run-Lookahead,
    - Run-Lazy-Lookahead,
    - Run-Concurrent-Lookahead
  - Lookahead
    - subgoaling
    - receding-horizon search
    - sampling
2.4 Backward Search

- Forward search starts at the initial state
  - Choose applicable action
  - Compute state transition $s' = \gamma(s,a)$

- Backward search starts at the goal
  - Chooses *relevant* action
    - A possible “last action” before the goal
  - Computes *inverse* state transition $g' = \gamma^{-1}(g,a)$
    - $g'$ = properties a state $s'$ should satisfy in order for $\gamma(s',a)$ to satisfy $g$

- Sometimes has a lower branching factor
  - Forward: 7 applicable actions
    - five load actions, two move actions
  - Backward: $g = \{\text{loc}(r1)=d3\}$
    - two relevant actions: $\text{move}(r1,d1,d3), \text{move}(r2,d1,d3)$
Relevance

- Idea: when can \( a \) be useful as the last action of a plan \( \pi \) for achieving \( g \)?
  - \( a \) can make at least one atom in \( g \) true that wasn’t true already
  - \( a \) doesn’t make any part of \( g \) false

- \( a \) is \textit{relevant} for \( g = \{x_1=c_1, \ldots, x_k=c_k\} \) if
  - at least one atom in \( g \) is also in \( \text{eff}(a) \)
    - i.e., \( g \) contains \( x=c \) and \( \text{eff}(a) \) contains \( x\leftarrow c \)
  - for every atom \( x=c \) in \( g \)
    - \( a \) doesn’t make \( x=c \) false
      - i.e., \( \text{eff}(a) \) doesn’t contain \( x\leftarrow c' \) for some \( c' \)
    - if \( \text{pre}(a) \) requires \( x=c \) to be false, then \( \text{eff}(a) \) makes it true
      - i.e., if \( \text{pre}(a) \) contains \( x\neq c \) or \( x=c' \), then \( \text{eff}(a) \) contains \( x\leftarrow c \)
Relevance

\[ s = \{ \text{loc}(c1) = d1, \text{loc}(c2) = d1, \text{loc}(c3) = d1, \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(r2) = d2, \text{cargo}(r2) = \text{nil} \} \]

adjacent = \{(d1,d2), (d1,d3), (d2,d1), (d2,d3), (d3,d1), (d3,d2)\}

\[ g = \{ \text{loc}(c1) = r1, \text{loc}(r1) = d3 \} \]

\begin{align*}
\text{move}(r,l,m) & \quad \text{pre: } \text{loc}(r) = l, \text{adjacent}(l,m) \quad \text{eff: } \text{loc}(r) \leftarrow m \\
\text{take}(r,l,c) & \quad \text{pre: } \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l \quad \text{eff: } \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r \\
\text{put}(r,l,c) & \quad \text{pre: } \text{loc}(r) = l, \text{loc}(c) = r \quad \text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l
\end{align*}

Range(r) = \textit{Robots} = \{r1,r2\}

Range(l) = \text{Range}(m) = \textit{Locs} = \{d1,d2,d3\}

Range(c) = \textit{Containers} = \{c1,c2,c3\}

\textbf{Poll:} for each action below, is it relevant for \( g \)?

- take(r1,c1,d1)
- take(r1,c1,d2)
- put(r2,c1,d3)
- move(r1,d1,d3)
- move(r1,d3,d1)
- move(r1,d2,d3)
Inverse State Transitions

- If $a$ is relevant for $g$, then $\gamma^{-1}(g,a) = \text{pre}(a) \cup (g - \text{eff}(a))$

- If $a$ isn’t relevant for $g$, then $\gamma^{-1}(g,a)$ is undefined

Example:

- $g = \{\text{loc}(c1)=r1\}$
- What is $\gamma^{-1}(g, \text{load}(r1,c1,d3))$?
- What is $\gamma^{-1}(g, \text{load}(r2,c1,d1))$?
Backward Search

Backward-search($\Sigma, s_0, g_0$)

$\pi \leftarrow \emptyset$; $g \leftarrow g_0$

loop

if $s_0$ satisfies $g$ then return $\pi$

$A' \leftarrow \{a \in A \mid a$ is relevant for $g\}$

if $A' = \emptyset$ then return failure

nondeterministically choose $a \in A'$

$g \leftarrow \gamma^{-1}(g, a)$

$\pi \leftarrow a.\pi$

(i) (ii) (iii)

Cycle checking:

- After line (i), put $Solved \leftarrow \{g\}$
- After line (iii), put either this:
  
  if $g \in Solved$ then return failure

  $Solved \leftarrow Solved \cup \{g\}$

  or this:

  if $\exists g' \in Solved$ s.t. $g \subseteq g'$ then return failure

  $Solved \leftarrow Solved \cup \{g\}$

- With cycle checking, sound and complete

  If $(\Sigma, s_0, g_0)$ is solvable, then at least one of the execution traces will find a solution
Motivation for Backward-search was to reduce the branching factor
- As written, doesn’t accomplish that
- Solve this by *lifting*:
  - When possible, leave variables uninstantiated

\[ g = \{ \text{loc}(r1)=d3 \} \]
Lifted Backward Search

- Like Backward-search but much smaller branching factor
- Must keep track of what values were substituted for which parameters
  - I won’t discuss the details
  - PSP (later) does something similar

\[
\text{Backward-search}(\Sigma, s_0, g_0)
\]
\[
\pi \leftarrow \langle \rangle; \quad g \leftarrow g_0
\]
\[
\text{loop}
\]
\[
\text{if } s_0 \text{ satisfies } g \text{ then return } \pi
\]
\[
A' \leftarrow \{ a \in A \mid a \text{ is relevant for } g \}
\]
\[
\text{if } A' = \emptyset \text{ then return failure}
\]
\[
\text{nondeterministically choose } a \in A'
\]
\[
g \leftarrow \gamma^{-1}(g, a)
\]
\[
\pi \leftarrow a.\pi
\]

```
Lifted-backward-search(\mathcal{A}, s_0, g)
\pi \leftarrow \text{the empty plan}
\text{loop}
\quad \text{if } s_0 \text{ satisfies } g \text{ then return } \pi
\quad A \leftarrow \{ (o, \theta) \mid o \text{ is a standardization of an atom of } g \text{ and an atom of } \text{effects}^+(o), \text{ and } \gamma^{-1}(\theta(g), \theta(o)) \text{ is defined} \}
\quad \text{if } A = \emptyset \text{ then return failure}
\quad \text{nondeterministically choose a pair } (o, \theta) \in A
\quad \pi \leftarrow \text{the concatenation of } \theta(o) \text{ and } \theta(\pi)
\quad g \leftarrow \gamma^{-1}(\theta(g), \theta(o))
```
Summary

- 2.4 Backward State-Space Search
  - Relevance, $\gamma^{-1}$
  - Backward search, cycle checking
  - Lifted backward search (briefly)
2.5 Plan-Space Search

- Formulate planning as a constraint satisfaction problem
  - Use constraint-satisfaction techniques to produce solutions that are more flexible than ordinary plans
    - E.g., plans in which the actions are partially ordered
    - Postpone ordering decisions until the plan is being executed
      - the actor may have a better idea about which ordering is best
- First step toward temporal planning (Chapter 4)

- Basic idea:
  - Backward search from the goal
  - Each node of the search space is a partial plan that contains flaws
    - Remove the flaws by making refinements
  - If successful, we’ll get a partially ordered solution
Definitions

- **Partially ordered plan**
  - partially ordered set of nodes
  - each node contains an action

- **Partially ordered solution**
  - partially ordered plan \( \pi \) such that every total ordering of \( \pi \) is a solution

- **Partial plan**
  - partially ordered set of nodes that contain partially instantiated actions
  - *inequality constraints*, e.g. \( z \neq x \) or \( w \neq p1 \)
  - *causal links* (dashed arcs)
    - use action \( a \) to establish precondition \( p \) of action \( b \)

\[
\begin{align*}
\text{move}(d,a,p1) & \quad \text{move}(c,b,p4) \\
\text{move}(a,p3,d) & \quad \text{move}(b,p4,c)
\end{align*}
\]

\[
\begin{align*}
\text{foo}(y) & \rightarrow \text{pre: loc}(y)=p1 \\
\text{bar}(y) & \rightarrow \text{eff: loc}(y)=p1 \\
\text{baz}(z) & \quad z \neq x
\end{align*}
\]
Flaws: 1. Open Goals

- A precondition $p$ of an action $b$ is an open goal if there is no causal link for $p$
- Resolve the flaw by creating a causal link
  - Find an action $a$ (either already in $\pi$, or can add it to $\pi$) that can establish $p$
    - can precede $b$
    - can have $p$ as an effect
  - Do substitutions on variables to make $a$ assert $p$
    - e.g., replace $x$ with $y$
  - Add an ordering constraint $a < b$
  - Create a causal link from $a$ to $p$

 replacements

pre: $\text{loc}(y) = p_1$

$\text{bar}(y)$

$\text{foo}(x)$

eff: $\text{loc}(x) = p_1$

$\text{pre: loc}(y) = p_1$

$\text{bar}(y)$

$\text{foo}(y)$

$\text{eff: loc}(y) = p_1$

replace $x$ with $y$
Flaws: 2. Threats

- Suppose we have a causal link from action $a$ to precondition $p$ of action $b$
- Action $c$ threatens the link if $c$ may affect $p$ and may come between $a$ and $b$
  - $c$ is a threat even if it makes $p$ true rather than false
    - Causal link means $a$, not $c$, is supposed to establish $p$ for $b$
    - The plan in which $c$ establishes $p$ will be generated on another path in the search space
- Three possible ways to resolve the flaw:
  - Make $c \prec a$
  - Make $b \prec c$
  - Add inequality constraints to prevent $c$ from affecting $p$
PSP Algorithm

\[ \text{PSP}(\Sigma, \pi) \]

\[
\begin{align*}
\text{loop} & \\
\text{if } \text{Flaws}(\pi) = \emptyset & \text{ then return } \pi \\
\text{arbitrarily select } f & \in \text{Flaws}(\pi) \\
R & \leftarrow \{\text{all feasible resolvers for } f\} \\
\text{if } R = \emptyset & \text{ then return failure} \\
\text{nondeterministically choose } & \rho \in R \\
\pi & \leftarrow \rho(\pi) \\
\text{return } \pi
\end{align*}
\]

- Initial plan is always \{Start, Finish\} with Start \prec Finish
  - Start has no preconditions; effects are the atoms in \(s_0\)
  - Finish has no effects; preconditions are the atoms in \(g\)
- PSP is sound and complete
  - It returns a partially ordered plan \(\pi\) such that any total ordering of \(\pi\) will achieve \(g\)
  - In some environments, could execute actions in parallel

\[ \text{Start} \quad \text{eff: } s_0 \]

\[ \text{Finish} \quad \text{pre: } g \]
Example

- Finish has two open goals: pos(a)=d, pos(b)=c

\[
\text{PSP}(\Sigma, \pi)
\]

\[
\text{loop}
\]

\[
\text{if } Flaws(\pi) = \emptyset \text{ then return } \pi
\]

arbitrarily select \( f \in Flaws(\pi) \)

\[
R \leftarrow \{\text{all feasible resolvers for } f\}
\]

if \( R = \emptyset \) then return failure

nondeterministically choose \( \rho \in R \)

\[
\pi \leftarrow \rho(\pi)
\]

return \( \pi \)

\[
\text{move}(c, y, z)
\]

pre: pos(c)=y, clear(c)=T, clear(z)=T

eff: pos(c)\leftarrow z, clear(y)\leftarrow T, clear(z)\leftarrow F

Range(c) = Containers; Range(y) = Range(z) = Container \cup pallets

\[
\begin{align*}
\text{clear(p1)}=\text{T} & \quad \text{clear(p2)}=\text{T} & \quad \text{clear(p3)}=\text{F} & \quad \text{clear(p4)}=\text{F} \\
\text{clear(a)}=\text{F} & \quad \text{clear(b)}=\text{F} & \quad \text{clear(c)}=\text{T} & \quad \text{clear(d)}=\text{T} \\
\text{pos(a)}=\text{p3} & \quad \text{pos(b)}=\text{p4} & \quad \text{pos(c)}=\text{b} & \quad \text{pos(d)}=\text{a}
\end{align*}
\]
Example

- For each open goal, add a new action
  - Every new action $a$ must have $\text{Start} < a < \text{Finish}$

$\text{PSP}(\Sigma, \pi)$

loop
  if $\text{Flaws}(\pi) = \emptyset$ then return $\pi$
  arbitrarily select $f \in \text{Flaws}(\pi)$
  if $R = \emptyset$ then return failure
  nondeterministically choose $\rho \in R$
  $\pi \leftarrow \rho(\pi)$
return $\pi$

$\text{move}(c, y, z)$
pre: $\text{pos}(c) = y$, $\text{clear}(c) = \text{T}$, $\text{clear}(z) = \text{T}$
\[ \text{eff: pos}(c) \leftarrow z, \text{clear}(y) \leftarrow \text{T}, \text{clear}(z) \leftarrow \text{F} \]
$\text{Range}(c) = \text{Containers}$; $\text{Range}(y) = \text{Range}(z) = \text{Container} \cup \text{pallets}$

![Diagram of a robotic arm moving objects between positions and indicating states of clear, pos, and move actions.](image)

- clear(p1)=T, clear(p2)=T, clear(p3)=F, clear(p4)=F
- pos(a)=p3, pos(b)=p4, pos(c)=b, pos(d)=a

- initial state:
  - clear(d)=T, clear(a)=T, pos(a)=y1
  - move(a, y1, d)
- next state:
  - pos(a)=d
  - pos(b)=c
  - move(b, y2, c)

Final state:
- pos(b)=y2, clear(b)=T, clear(c)=T
- move(b, y2, c)
- pos(a)=p3, pos(b)=p4, pos(c)=b, pos(d)=a

- example:
  - For each open goal, add a new action
    - Every new action $a$ must have $\text{Start} < a < \text{Finish}$

![Diagram of a robotic arm moving objects between positions and indicating states of clear, pos, and move actions.](image)
Example

- Resolve four open goals using the Start action
  - substitute $y_1 = p3$, $y_2 = p4$

$PSP(\Sigma, \pi)$

loop
  if $Flaws(\pi) = \emptyset$ then return $\pi$
  arbitrarily select $f \in Flaws(\pi)$
  $R \leftarrow \text{all feasible resolvers for } f$
  if $R = \emptyset$ then return failure
  nondeterministically choose $\rho \in R$
  $\pi \leftarrow \rho(\pi)$
return $\pi$

move($c$, $y$, $z$)
  pre: $\text{pos}(c) = y$, $\text{clear}(c) = T$, $\text{clear}(z) = T$
  eff: $\text{pos}(c) \leftarrow z$, $\text{clear}(y) \leftarrow T$, $\text{clear}(z) \leftarrow F$

Range($c$) = Containers; Range($y$) = Range($z$) = Container $\cup$ pallets

move($a$, $p3$, $d$)
  $\text{pos}(a) = d$

move($b$, $p4$, $c$)
  $\text{pos}(b) = c$

clear($p1$) = $T$ clear($p2$) = $T$ clear($p3$) = $F$ clear($p4$) = $F$

clear($a$) = $F$ clear($b$) = $T$ clear($c$) = $T$ clear($d$) = $T$

$\text{pos}(a) = p3$ $\text{pos}(b) = p4$ $\text{pos}(c) = b$ $\text{pos}(d) = a$
New action to resolve open goal

1\textsuperscript{st} threat has one resolver: $z_3 \neq d$

2\textsuperscript{nd} threat has two resolvers:

- $\text{move}(b, p4, c) < \text{move}(x_3, a, z_3)$
- $z_3 \neq c$

(move \(c, y, z\))

pre: $\text{pos}(c) = y, \text{clear}(c) = T, \text{clear}(z) = T$

eff: $\text{pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F$

Range(c) = \text{Containers}; \ Range(y) = \text{Range}(z) = \text{Container} \cup \text{pallets}
move (c, y, z)

pre: pos (c) = y, clear (c) = T, clear (z) = T
eff: pos (c) ← z, clear (y) ← T, clear (z) ← F

Range (c) = Containers; Range (y) = Range (z) = Container ∪ pallets
Example

- 1st threat has two resolvers:
  - An ordering constraint,
    and $z_4 \neq d$

- 2nd threat has three resolvers:
  - Two ordering constraints, and $z_4 \neq a$

- 3rd threat has one: $z_4 \neq c$

- $clear(z_3) = T$  $clear(x_3) = T$  $pos(x_3) = a$
- $move(x_3, a, z_3)$

- $clear(d) = T$  $clear(a) = T$  $pos(a) = p3$
- $move(a, p3, d)$

- $pos(a) = d$

- $pos(b) = c$
- $pos(x_4) = b$  $clear(x_4) = T$  $clear(z_4) = T$
- $move(x_4, b, z_4)$

- $pos(b) = p4$
- $clear(b) = T$  $clear(c) = T$
- $move(b, p4, c)$

- $move(c, y, z)$
  - Pre: $pos(c) = y$, $clear(c) = T$, $clear(z) = T$
  - Eff: $pos(c) \leftarrow z$, $clear(y) \leftarrow T$, $clear(z) \leftarrow F$

- Range($c$) = Containers; Range($y$) = Range($z$) = Container $\cup$ pallets
Example

- Resolve the three threats using inequality constraints

\[ z_3 \neq c \quad z_4 \neq a \quad z_3 \neq d \quad z_4 \neq c \quad z_4 \neq d \]

\[
\text{move}(x_3, a, z_3) \\
\text{move}(a, p3, d) \\
\text{pos}(a) = d \\
\text{pos}(b) = c
\]

\[
\text{pos}(x_4) = b \\
\text{clear}(x_4) = T \\
\text{clear}(z_4) = T
\]

\[
\text{move}(x_4, b, z_4) \\
\text{move}(b, p4, c) \\
\text{clear}(c) = T
\]

\[
\text{move}(c, y, z) \\
\text{pre: pos}(c) = y, \text{clear}(c) = T, \text{clear}(z) = T \\
\text{eff: pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F
\]

Range(c) = Containers; Range(y) = Range(z) = Container \cup pallets
Example

- Resolve five open goals using the Start action
  - substitute
    - $x_3 = d$, $x_4 = c$, $z_3 = p1$

move($c, y, z$)
pre: $\text{pos}(c) = y$, clear($c$) = $T$, clear($z$) = $T$

$\text{eff: } \text{pos}(c) \leftarrow z$, clear($y$) $\leftarrow T$, clear($z$) $\leftarrow F$

Range($c$) = Containers; Range($y$) = Range($z$) = Container $\cup$ pallets
Example

- Threatened causal link
- Resolvers:
  - move(d,a,p1) < move(c,b,z_4)
  - z_4 \neq p1

move(c, y, z)

pre: pos(c)=y, clear(c)=T, clear(z)=T

eff: pos(c)\leftarrow z, clear(y)\leftarrow T, clear(z)\leftarrow F

Range(c) = Containers; Range(y) = Range(z) = Container \cup pallets
• Threat resolved

move(c, y, z)

pre: pos(c)=y, clear(c)=T, clear(z)=T
eff: pos(c)←z, clear(y)←T, clear(z)←F

Range(c) = Containers; Range(y) = Range(z) = Container ∪ pallets
Example

- Resolve open goal using the Start action
  - substitute $z_4 = p2$
- No more flaws, so we’re done!

move($c$, $y$, $z$)
  pre: $\text{pos}(c) = y$, $\text{clear}(c) = T$, $\text{clear}(z) = T$
  eff: $\text{pos}(c) \leftarrow z$, $\text{clear}(y) \leftarrow T$, $\text{clear}(z) \leftarrow F$

Range($c$) = Containers; Range($y$) = Range($z$) = Container $\cup$ pallets
Example

- PSP returns this solution:

  \[
  \text{move}(d,a,p1) \\
  \text{move}(a,p3,d) \\
  \text{move}(c,b,p2) \\
  \text{move}(b,p4,c)
  \]
Example 2

- Go back to the last threat, choose the other resolver:
  - move(d,a,p1) < move(c,b,z₄)
  - z₄≠p1
Example 2

- Threat resolved

- $\text{clear}(p1) = T$, $\text{clear}(d) = T$, $\text{pos}(d) = a$

- $\text{move}(d, a, p1)$

- $\text{clear}(d) = T$, $\text{clear}(a) = T$, $\text{pos}(a) = p3$

- $\text{move}(a, p3, d)$

- $\text{pos}(a) = d$

- $\text{move}(c, b, z_4)$

- $\text{pos}(c) = b$, $\text{clear}(c) = T$, $\text{clear}(z_4) = T$

- $\text{pos}(b) = p4$, $\text{clear}(b) = T$, $\text{clear}(c) = T$

- $\text{move}(b, p4, c)$

- $\text{clear}(c) = T$

- $\text{move}(d, a, p1)$

- $\text{clear}(d) = T$, $\text{clear}(p1) = T$, $\text{pos}(d) = a$

- $\text{pos}(c) = b$, $\text{clear}(c) = T$, $\text{clear}(z_4) = T$

- $\text{move}(b, p4, c)$

- $\text{clear}(c) = T$, $\text{clear}(p1) = T$, $\text{pos}(d) = a$

- $\text{pos}(a) = d$
Example 2

- Resolve open goal
  - substitute $z_4 = p_2$
- No more flaws, so we’re done

\[
\begin{align*}
\text{clear}(p_1) &= T \\
\text{clear}(d) &= T \\
\text{pos}(d) &= a \\
\text{move}(d,a,p_1) \\
\text{move}(a,p_3,d) \\
\text{pos}(a) &= d \\
\text{pos}(b) &= c \\
\text{move}(c,b,p_2) \\
\text{pos}(c) &= b \\
\text{clear}(c) &= T \\
\text{clear}(p_2) &= T \\
\text{clear}(b) &= T \\
\text{pos}(b) &= c \\
\text{move}(b,p_4,c) \\
\text{pos}(b) &= c \\
\text{clear}(c) &= T \\
\text{clear}(p_2) &= T \\
\text{clear}(p_1) &= T \\
\text{pos}(c) &= b \\
\text{clear}(c) &= T \\
\text{clear}(p_2) &= T \\
\text{pos}(b) &= c \\
\end{align*}
\]

\[z_4 \neq a, \quad z_4 \neq c, \quad z_4 \neq d, \quad \text{satisfied}\]
Example 2

- Like previous solution, but has another ordering constraint
Node-Selection Heuristics

- Analogy to constraint-satisfaction problems
  - Resolving a flaw in PSP ≈ assigning a value to a variable in a CSP
- What flaw to work on next?
  - *Fewest Alternatives First (FAF)*:
    - choose a flaw having the fewest resolvers
      ≈ Minimum Remaining Values (MRV) heuristic for CSPs
- To resolve the flaw, which resolver to try first?
  - *Least Constraining Resolver (LCR)*:
    - choose a resolver that rules out the fewest resolvers for the other flaws
      ≈ Least Constraining Value (LCV) heuristic for CSPs
Example

- Fewest Alternatives First:
  - 1\textsuperscript{st} threat has two resolvers: an ordering constraint, and \(z_4 \neq d\)
  - 2\textsuperscript{nd} threat has three resolvers: 2 ordering constraints, and \(z_4 \neq a\)
  - 3\textsuperscript{rd} threat has one resolver: \(z_4 \neq c\)
- So resolve the 3\textsuperscript{rd} threat first

\[
\begin{align*}
\text{clear}(z_3) &= T & \text{clear}(x_3) &= T & \text{pos}(x_3) &= a \\
\text{move}(x_3, a, z_3) & & \text{move}(a, p3, d) & & \text{pos}(a) &= d & \text{pos}(b) &= c & \text{pos}(x_4) &= b & \text{clear}(x_4) &= T & \text{clear}(z_4) &= T \\
\text{clear}(d) &= T & \text{clear}(a) &= T & & \text{clear}(b) &= T & \text{clear}(c) &= T \\
\text{clear}(z_3) &= T & \text{clear}(z_3) &= T & \text{clear}(z_3) &= T & \text{clear}(z_3) &= T & \text{clear}(z_3) &= T
\end{align*}
\]
Node-Selection Heuristics

- Analogy to constraint-satisfaction problems
  - Resolving a flaw in PSP \(\approx\) assigning a value to a variable in a CSP

- What flaw to work on next?
  - *Fewest Alternatives First (FAF):*
    - choose a flaw having the fewest resolvers
      \(\approx\) Minimum Remaining Values (MRV) heuristic for CSPs

- To resolve the flaw, which resolver to try first?
  - *Least Constraining Resolver (LCR):*
    - choose a resolver that rules out the fewest resolvers for the other flaws
      \(\approx\) Least Constraining Value (LCV) heuristic for CSPs

- In PSP, introducing a new action introduces new flaws to resolve
  - The plan can get arbitrarily large; want it to be as small as possible
    - Not like CSPs, where the search tree always has a fixed depth
  - Avoid introducing new actions unless necessary

- To choose between actions \(a\) and \(b\), estimate distance from \(s_0\) to \(\text{Pre}(a)\) and \(\text{Pre}(b)\)
  - Can use the heuristic functions we discussed earlier
Discussion

* Problem: how to prune infinitely long paths in the search space?
  - Loop detection is based on recognizing states or goals we’ve seen before
    \[ \ldots \xrightarrow{} s \xrightarrow{} s' \xrightarrow{} s \]
  - In a partially ordered plan, we don’t know the states
  
* Can we prune if \( \pi \) contains the same *action* more than once?
  - \( \langle a_1, a_2, \ldots, a_1, \ldots \rangle \)
  - No. Sometimes we might need the same action several times in different states of the world
  - E.g., the Towers of Hanoi problem
    - Do this action many times:
      - stack disk1 onto disk2
A Weak Pruning Technique

- Can prune all partial plans of $n$ or more actions, where $n = |S|
  - Not very helpful

- I’m not sure whether there’s a good pruning technique for plan-space planning
2.5 Plan-Space Search
- Partially ordered plans and solutions
- partial plans, causal links
- flaws: open goals, threats, resolvers
- PSP algorithm, long example, node-selection heuristics

Two additional sets of lecture slides, if you’re interested
- Section 2.7.7, HTN Planning
- Section 2.7.8, Planning with Control Rules