Chapter 2
Deliberation with Deterministic Models

Section 2.1: Forward Search
Section 2.3: Heuristic Functions
Section 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

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

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
Nearly all planning procedures are search procedures.

- Search tree: the data structure the procedure uses to keep track of which paths it has explored.

Search-Tree Terminology

- **Node:** a pair \( \nu = (\pi, s) \), where \( s = \gamma(s_0, \pi) \)
- In practice, \( \nu \) may contain other things
  - pointer to parent, cost(\( \pi \)), …
  - \( \pi \) not always stored explicitly, can be computed from the parent pointers
- **Children** of \( \nu \) =
  \[ \{ (\pi.a, \gamma(s,a)) \mid a \text{ is applicable in } s \} \]
- **Successors** of \( \nu \):
  - children, children of children, etc.
- **Ancestors** of \( \nu \) =
  \{ nodes that have \( \nu \) as a successor \}
- **Initial or starting node:** \( \nu_0 = (\emptyset, s_0) \)
  - root of the search tree
- **Path** in the search space:
  - sequence \( \langle \nu_0, \nu_1, \ldots, \nu_n \rangle \)
  - such that each \( \nu_i \) is a child of \( \nu_{i-1} \)

- **Height** of search space = length of longest acyclic path from \( \nu_0 \)
- **Depth** of \( \nu \) = length(\( \pi \)) = length of path from \( \nu_0 \) to \( \nu \)
- **Branching factor** of \( \nu \) = number of children
- **Branching factor** of search tree = max branching factor of the nodes
- **Expand** \( \nu \): generate all children
Forward Search

Forward-search \((\Sigma, s_0, g)\)

\[
\begin{align*}
  s & \leftarrow s_0; \quad \pi \leftarrow \langle \rangle \\
  \text{loop} & \\
  \quad \text{if } s \text{ satisfies } g \text{ then return } \pi \\
  \quad A' & \leftarrow \{ a \in A \mid a \text{ is applicable in } s \} \\
  \quad \text{if } A' = \emptyset \text{ then return failure} \\
  \quad \text{nondeterministically choose } a \in A' \\
  \quad s & \leftarrow \gamma(s,a); \quad \pi \leftarrow \pi.a
\end{align*}
\]

- Nondeterministic algorithm
  - Sound: if an execution trace returns a plan \(\pi\), it’s a solution
  - Complete: if the planning problem is solvable, at least one of the possible execution traces will return a solution
- Represents a class of deterministic search algorithms
  - Depends on how you implement the nondeterministic choice
    - Which nodes, in which order
  - Won’t necessarily be complete
Deterministic Version

Deterministic-Search($\Sigma, s_0, g$)

- $\text{Frontier} \leftarrow \{(\langle \rangle, s_0)\}$
- $\text{Expanded} \leftarrow \emptyset$

while $\text{Frontier} \neq \emptyset$ do

- select a node $v = (\pi, s) \in \text{Frontier}$ (i)
- remove $v$ from $\text{Frontier}$
- add $v$ to $\text{Expanded}$
- if $s$ satisfies $g$ then return $\pi$ (ii)

- $\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$
- prune 0 or more nodes from $\text{Children, Frontier, Expanded}$ (iii)
- $\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$

return failure

- Special cases:
  - depth-first, breadth-first, A*, many others

- Classify by
  - how they select nodes (step i)
  - how they prune nodes (step iii)

- Pruning often includes cycle-checking:
  - Remove from $\text{Children}$ every node $(\pi,s)$ that has an ancestor $(\pi',s')$ such that $s' = s$
  - In classical planning problems, $S$ is finite, so cycle-checking will guarantee termination
Breadth-First Search (BFS)

Deterministic-Search($\Sigma, s_0, g$)

$$\text{Frontier} \leftarrow \{(\langle \rangle, s_0)\}$$

$$\text{Expanded} \leftarrow \emptyset$$

while $\text{Frontier} \neq \emptyset$ do

select a node $\nu = (\pi, s) \in \text{Frontier}$ (i)

remove $\nu$ from $\text{Frontier}$

add $\nu$ to $\text{Expanded}$

if $s$ satisfies $g$ then return $\pi$ (ii)

$$\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$$

prune 0 or more nodes from $\text{Children, Frontier, Expanded}$ (iii)

$$\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$$

return failure

(i): select $(\pi, s) \in \text{Frontier}$ with smallest length($\pi$)

- tie-breaking rule: select oldest

(iii): remove every $(\pi, s) \in \text{Children} \cup \text{Frontier}$ such that $s$ is in $\text{Expanded}$

- Thus expand states at most once

- Properties
  - Terminates
  - Returns solution if one exists
    - shortest, but not least-cost
  - Worst-case complexity:
    - memory $O(|S|)$
    - running time $O(b|S|)$
Depth-First Search (DFS)

Deterministic-Search($\Sigma, s_0, g$)

1. $Frontier \leftarrow \{(\langle \rangle, s_0)\}$
2. $Expanded \leftarrow \emptyset$

while $Frontier \neq \emptyset$ do

1. select a node $v = (\pi, s) \in Frontier$ (i)
2. remove $v$ from $Frontier$
3. add $v$ to $Expanded$
4. if $s$ satisfies $g$ then return $\pi$ (ii)

$Children \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } pre(a)\}$

prune 0 or more nodes from

$Children, Frontier, Expanded$ (iii)

$Frontier \leftarrow Frontier \cup Children$

return failure

(i): Select $(\pi, s) \in Children$ that has largest length($\pi$)

- Possible tie-breaking rules: left-to-right, smallest $h(s)$

(iii): do cycle-checking, then prune all

- nodes that recursive depth-first search would discard

- Repeatedly remove from $Expanded$

- any node that has no children in $Children \cup Frontier \cup Expanded$

Properties

- Terminates
- Returns solution if there is one
  - No guarantees on quality
- Worst-case running time $O(b^l)$
- Worst-case memory $O(bl)$
  - $b = \text{ max branching factor}$
  - $l = \text{ max depth of any node}$
Uniform-Cost Search

Deterministic-Search($\Sigma, s_0, g$)

1. **Frontier** ← $\{(\langle\rangle, s_0)\}$
2. **Expanded** ← $\emptyset$

while Frontier ≠ $\emptyset$ do

   1. select a node $v = (\pi, s) \in \text{Frontier}$ (i)
   2. remove $v$ from Frontier
   3. add $v$ to Expanded

if $s$ satisfies $g$ then return $\pi$ (ii)

$\text{Children} ← \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies pre}(a)\}$

prune 0 or more nodes from \text{Children}, Frontier, Expanded (iii)

Frontier ← Frontier $\cup$ Children

return failure

(i): Select $(\pi,s) \in \text{Children}$ that has smallest cost($\pi$)

(iii): Prune every $(\pi,s) \in \text{Children} \cup \text{Frontier}$ such that Expanded already contains a node $(\pi',s)$

- cost($\pi'$) ≤ cost($\pi$), so we only keep the least-cost path to $s$

- Properties

- Terminates
- Finds optimal solution if one exists
- Worst-case time $O(b|S|)$
- Worst-case memory $O(|S|)$

- $b = \text{max branching factor}$
- $|S| = \text{number of states in $S$}$
Uniform-Cost Search

Deterministic-Search($\Sigma, s_0, g$)

Frontier $\leftarrow \{(\langle \rangle, s_0)\}$

Expanded $\leftarrow \emptyset$

while Frontier $\neq \emptyset$ do

select a node $\nu = (\pi, s) \in$ Frontier (i)

remove $\nu$ from Frontier

add $\nu$ to Expanded

if $s$ satisfies $g$ then return $\pi$ (ii)

Children $\leftarrow$

\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies pre(a)}\}

prune 0 or more nodes from

Children, Frontier, Expanded (iii)

Frontier $\leftarrow$ Frontier $\cup$ Children

return failure

(i): Select $(\pi, s) \in \text{Children}$ that has smallest cost($\pi$)

(iii): Prune every $(\pi, s) \in \text{Children} \cup \text{Frontier}$ such that Expanded already contains a node $(\pi', s)$

- cost($\pi'$) $\leq$ cost($\pi$), so we only keep the least-cost path to $s$

• Properties
  - Terminates
  - Finds optimal solution if one exists
  - Worst-case time $O(b|S|)$
  - Worst-case memory $O(|S|)$

- $b = \text{max branching factor}$
- $|S| = \text{number of states in } S$

Poll 2d: If node $\nu$ is expanded before node $\nu'$, then what can we say about cost($\nu$) and cost($\nu'$)?
Heuristic Function

- Motivation: get to a solution quickly by selecting nodes close to the goal
- Let $h^*(s) = \min\{\text{cost}(\pi) \mid \gamma(s,\pi) \text{ satisfies } g\}$
  - Note that $h^*(s) \geq 0$ for all $s$

heuristic function $h(s)$: returns an estimate of $h^*(s)$
- We’ll assume $h(s) \geq 0$ for all $s$

Terminology
- $h$ is admissible if for every $s$, $h(s) \leq h^*(s)$
- $h$ is $\varepsilon$-admissible if for every $s$, $h(s) \leq h^*(s) + \varepsilon$
- $h$ is monotone if for every node $s$ and action $a$ that’s applicable in $s$, $h(s) \leq \text{cost}(s, a) + h(\gamma(s, a))$
  - if $h$ is monotone and $h(s) = 0$ at goal nodes, then $h$ is admissible
- $h$ dominates $h'$ if $h'(s) \leq h(s) \leq h^*(s)$ for every $s$

- $f(\nu) = \text{cost}(\pi) + h(s)$, where $\nu = (\pi,s)$
  - If $h$ is admissible then $f(\nu) \leq \min\{\text{cost}(\pi,\pi') \mid \gamma(s,\pi,\pi') \text{ satisfies } g\}$
  - If $h$ is $\varepsilon$-admissible then $f(\nu) \leq \varepsilon + \min\{\text{cost}(\pi,\pi') \mid \gamma(s,\pi,\pi') \text{ satisfies } g\}$
Deterministic-Search($\Sigma$, $s_0$, $g$)

$\text{Frontier} \leftarrow \{(\langle \rangle, s_0)\}$

$\text{Expanded} \leftarrow \emptyset$

while $\text{Frontier} \neq \emptyset$ do

select a node $\nu = (\pi, s) \in \text{Frontier}$ (i)

remove $\nu$ from $\text{Frontier}$

add $\nu$ to $\text{Expanded}$

if $s$ satisfies $g$ then return $\pi$ (ii)

$\text{Children} \leftarrow$

$\{ (\pi.a, \gamma(s,a)) \mid s \text{ satisfies pre}(a) \}$

prune 0 or more nodes from $\text{Children}, \text{Frontier}, \text{Expanded}$ (iii)

$\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$

return failure

- **Node selection**
  Select a node $\nu = (\pi,s)$ in $\text{Frontier}$ that has smallest value of $f(\nu) = g(\pi) + h(s)$
  (see next slides)

  - Tie-breaking rule: choose oldest

- **Pruning:**
  for every node $\nu = (\pi,s)$ in $\text{Children}$, if $\text{Children} \cup \text{Frontier} \cup \text{Expanded}$ contains more than one node for the same state $s$, then it has multiple paths to $s$

  - Keep only the one with the lowest $f$-value

  - Tie-breaking rule: keep oldest

- If the node is $\nu$, will expand $s$ again

Next, an example
• A domain-specific heuristic function
  - $h(s) =$ straight-line distance from city $s$ to Bucharest
  - admissible and monotone

• Later I’ll give you some that are domain-independent
  - can be used with any classical planning problem

straight-line dist. from $s$ to Bucharest
- Arad 366
- Bucharest 0
- Craiova 160
- Dobreta 242
- Fagaras 176
- Iasi 226
- Lugoj 244
- Mehadia 241
- Neamt 234
- Oradea 380
- Pitesti 100
- Rimnicu Vilcea 193
- Sibiu 253
- Timisoara 329
- Urziceni 80
- Vaslui 199
- Zerind 374

from Russell & Norvig, Artificial Intelligence: A Modern Approach
straight-line dist. from s to Bucharest

Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

Nau – Lecture slides for Automated Planning and Acting
straight-line dist. from $s$ to Bucharest:

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobreta: 242
- Fagaras: 176
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 100
- Rimnicu Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Vaslui: 199
- Urziceni: 80
- Zerind: 374
straight-line dist. from $s$ to Bucharest:
- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrota: 242
- Fagaras: 176
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 100
- Rimnicu Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374

Nau – Lecture slides for Automated Planning and Acting
Properties of A*

- Terminates and returns solution there is one
  - If $h$ is admissible then this solution will be optimal
  - If $h$ is $\varepsilon$-admissible then the solution will be $\varepsilon$-optimal
  - If $h$ is monotone then
    - $f(v) \leq f(v')$ for every child $v'$ of a node $v$
    - Nodes will be expanded in non-decreasing order of $f$ values
    - A* will never prune any nodes from Expanded
    - A* will expand no state more than once
- If $h$ dominates $h'$ then (assuming same tie-breaking rule)
  - A* will never expand more nodes with $h$ than with $h'$
  - In most cases A* will expand fewer nodes with $h$ than with $h'$
- A* needs to store every node it visits
  - Running time and memory both $O(b|S|)$ in worst case
  - With good heuristic function, usually much smaller
Greedy Best-First Search (GBFS)

Deterministic-Search(\(\Sigma, s_0, g\))

1. \(\text{Frontier} \leftarrow \{(\langle\rangle, s_0)\}\)
2. \(\text{Expanded} \leftarrow \emptyset\)

while \(\text{Frontier} \neq \emptyset\) do

1. select a node \(\nu = (\pi, s) \in \text{Frontier}\) (i)
2. remove \(\nu\) from \(\text{Frontier}\)
3. add \(\nu\) to \(\text{Expanded}\)
4. if \(s\) satisfies \(g\) then return \(\pi\) (ii)
5. \(\text{Children} \leftarrow\)
   \(\{(\pi.a, \gamma(s,a)) | s\) satisfies \(\text{pre}(a)\}\)
6. prune 0 or more nodes from \(\text{Children}, \text{Frontier}, \text{Expanded}\) (iii)
7. \(\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}\)

return failure

- Often want a low-cost or least-cost solution
  - Select nodes that are likely to be on the least-cost path

- Node selection
  - Select a node \((\pi,s) \in \text{Frontier}\)
    that has smallest \(h(s)\)

- Pruning: same as in A*
  - If \(\text{Children} \cup \text{Frontier} \cup \text{Expanded}\)
    contains more than one node for the same state \(s\), then it has multiple paths to \(s\)
  - Keep only the one with the lowest \(f\)-value
  - If more than one such node, keep the oldest

- Properties
  - Terminates
  - Returns a solution if one exists
  - Often near-optimal
  - will usually find it quickly
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrogeia 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 221
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrogea 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamț 234
Oradea 380
Pitesti 100
Rimnicu Vâlcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

- expanded 4 nodes instead of 6
- solution cost 450 instead of 418
Depth-First Branch and Bound (DFBB)

Deterministic-Search($\Sigma$, $s_0$, $g$)

$\text{Frontier} \leftarrow \{(\emptyset, s_0)\}$

$\text{Expanded} \leftarrow \emptyset$

$c^* \leftarrow \infty$; $\pi^* \leftarrow \text{failure}$

while $\text{Frontier} \neq \emptyset$ do

select a node $\nu = (\pi, s) \in \text{Frontier}$ (i)

remove $\nu$ from $\text{Frontier}$ and add it to $\text{Expanded}$

if $s$ satisfies $g$ then return $\pi$ (ii)

if $s$ satisfies $g$ and $\text{cost}(\pi) < c^*$ then

$c^* \leftarrow \text{cost}(\pi)$; $\pi^* \leftarrow \pi$

else if $f(\nu) < c^*$ then

$\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

prune 0 or more nodes from $\text{Children, Frontier, Expanded}$ (iii)

$\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$

return $\text{failure}$ $\pi^*$

- Node selection same as in DFS:
  - Select $\nu = (\pi, s) \in \text{Children}$ that has largest length($\pi$)
  - Tie-breaking: smallest $h(s)$

- Pruning
  - Like DFS, do cycle-checking and prune what recursive depth-first search would discard

- Additional pruning during node expansion:
  - If $f(\nu) \geq c^*$ then discard $\nu$ instead

- Properties
  - Termination, completeness, optimality same as A*
  - Usually less memory than A*, but more time
  - Worst-case like DFS: $O(b\ell)$ memory, $O(b^\ell)$ running time
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest:
- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrogea: 242
- Fagaras: 176
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamț: 234
- Oradea: 380
- Pitesti: 100
- Rimnicu Vilea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374

\[
\begin{align*}
447 &= 118 + 329 \\
449 &= 75 + 374 \\
646 &= 280 + 366 \\
413 &= 220 + 193 \\
671 &= 291 + 380
\end{align*}
\]
straight-line dist. from s to Bucharest
Arad
Bucharest
Craiova
Dobreta
Fagaras
Iasi
Lugoj
Mehadia
Neamt
Oradea
Pitesti
Rimnicu Vilcea
Sibiu
Timisoara
Urziceni
Vaslui
Zerind
Arad
Bucharest
Craiova
Dobreta
Fagaras
Iasi
Lugoj
Mehadia
Neamt
Oradea
Pitesti
Rimnicu Vilcea
Sibiu
Timisoara
Urziceni
Vaslui
Zerind

Arad

Fagaras

Oradea

Rimnicu Vilcea

Craiova

Pitesti

Sibiu

Arad

Fagaras

Oradea

Rimnicu Vilcea

Craiova

Pitesti

Sibiu
straight-line dist.
from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrogea 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

Nau – Lecture slides for Automated Planning and Acting
\(c^*=418\) 
\(\pi^*=\langle a_{AS}, a_{SR}, a_{RP}, a_{PB}\rangle\)
straight-line dist. from $s$ to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrogea 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamţ 234
Oradea 380
Pitesti 100
Rimnicu Vîlcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
straight-line dist. from $s$ to Bucharest

Arad  366
Bucharest  0
Craiova  160
Dobrogea  242
Fagaras  176
Iasi  226
Lugoj  244
Mehadia  241
Neamt  234
Oradea  380
Pitesti  100
Rimnicu Vilcea  193
Sibiu  253
Timisoara  329
Urziceni  80
Vaslui  199
Zerind  374

$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
\[ c^* = 418 \]
\[ \pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle \]
straight-line dist. from $s$ to Bucharest:
- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrota: 242
- Fagaras: 176
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 100
- Rimnicu Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374

$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
Iterative Deepening (IDS)

IDS(Σ, s₀, g)
for k = 1 to ∞ do
  do a depth-first search, backtracking at every node of depth k
  if the search found a solution then return it
  if the search generated no nodes of depth k then return failure

- Example:
  Expand a
  Expand a, b, c
  Expand a, b, c, d, e, f, g
  Expand a, b, c, d, e, f, g, h, i, j, k, l, m, n, o
  Solution path ⟨a, c, g, o⟩
  Total number of node expansions:
    1 + 3 + 7 + 15 = 26

- If goal is at depth d and branching factor is 2:
  \[ \sum_{i=1}^{d} (2^i - 1) = 2^{d+1} - d - 2 = O(2^d) \]
Iterative Deepening (IDS)

IDS(Σ, s₀, g)
for k = 1 to ∞ do
    do a depth-first search, backtracking at every node of depth k
    if the search found a solution then return it
    if the search generated no nodes of depth k then return failure

- Termination, completeness, optimality
  - same as breadth-first search
- Worst-case memory requirement $O(bd)$
  - vs. $O(b^d)$ for breadth-first search
- If the number of nodes at depth $d$ grows exponentially with $d$:
  - worst-case running time $O(b^d)$, vs. $O(b^l)$ for DFS

- $b = \text{max branching factor}$
- $d = \text{min solution depth if there is one, otherwise max depth of any node}$
**IDA**

\[ \text{IDA}^*(\Sigma, s_0, g) \]

\[ c \leftarrow 0 \]

loop

\[ \text{do a depth-first search, backtracking whenever } f(\nu) > c \]

if the search found a solution then return it

if the search didn’t generate an \( f(\nu) > c \) then return failure

\[ c \leftarrow \text{the smallest } f(\nu) > c \text{ where backtracking occurred} \]

- Termination, completeness, and optimality same as A*
- If \( h \) is admissible, worst-case memory requirement \( O(bd) \) rather than \( O(b^d) \)
- If the number of nodes grows exponentially with \( c \), worst-case running time \( O(b^d) \) (like DFS)
  - Can be much worse if the number of nodes grows subexponentially
    - e.g., real-valued costs
Discussion

- If $h$ is admissible, both A* and DFBB will return optimal solutions
  - Usually DFBB takes more time, A* takes more memory
  - A* better than DFBB in highly connected graphs (many paths to states)
    - DFBB can have exponentially worse running time than A*
  - DFBB best in problems where $S$ is a tree of uniform height, all solutions at the bottom (e.g., constraint satisfaction)
    - DFBB and A* have similar running time, A* takes exponentially more memory than DFBB
- DFS returns the first solution it finds
  - less backtracking than DFBB, but solution can be very far from optimal
- GBFS returns the first solution it finds
  - with a good heuristic function, usually near-optimal without much backtracking
  - used by most classical planners nowadays
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

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

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
2.3 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

![Graph of city connections with distances](image-url)
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 \( x=w \)

\[
\begin{align*}
\gamma^+(s,a) & = \gamma^+(s_0, \text{move}(r1,d3,d1)) \\
& = \{ \text{loc}(r1) = d3, \text{loc}(r1) = d1, \\
& \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \}
\end{align*}
\]

\[
\begin{align*}
s_0 & = \{ \text{loc}(r1) = d3, \\
& \text{cargo}(r1) = \text{nil}, \\
& \text{loc}(c1) = d1 \}
\end{align*}
\]
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 \( \{s\} \)

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

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

- 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 \( \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n \) 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 \( \gamma^+(\hat{s}_1,a_2) = \hat{s}_2 \)
  - \( \ldots \)
  - 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)
pre: \text{loc}(r1) = d3
eff: \text{loc}(r1) \leftarrow d1

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

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

- $h^+(s) = \text{minimum cost of all relaxed solutions}$
  - 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*
Fast-Forward Heuristic

- If $a_1, a_2, \ldots, a_n$ are r-applicable, can apply them in any order and get same result:
  - $\gamma^+(\hat{s}, \{a_1, a_2, \ldots, a_n\}) = \hat{s} \cup \text{eff}(a_1) \cup \text{eff}(a_2) \cup \ldots \cup \text{eff}(a_n)$

Given r-state $\hat{s}_0$ and goal $g$:
For $i = 1$ by 1 until we get an $\hat{s}_k$ that r-satisfies $g$
  - $A_i = \{\text{all actions r-applicable in } \hat{s}_{i-1}\}; \hat{s}_i = \gamma^+(s_{i-1}, A_i)$

$\langle A_1, A_2, \ldots, A_k \rangle$ is a relaxed solution

Extract minimal relaxed solution
  - $\hat{a}_k = \text{minimal subset of } A_k \text{ that r-achieves } g$
  - $\hat{a}_{k-1} = \text{minimal subset of } A_{k-1} \text{ that r-achieves } \text{pre}(\hat{a}_k)$
    - $\ldots$
  - $\hat{a}_1 = \text{minimal subset of } A_1 \text{ that r-achieves } \text{pre}(\hat{a}_2)$

- $h^\text{FF}(s) = \sum \text{costs of } \hat{a}_1, \ldots, \hat{a}_k = \text{an upper bound on } h^+$
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 two slides: $h^{FF}(s_1)$ and $h^{FF}(s_2)$

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

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

Atoms in \( \hat{s}_0 = s_1 \): Actions in \( A_1 \): Atoms in \( \hat{s}_1 \):
\( \text{loc}(r1) = d1 \) \( \text{move}(r1,d1,d2) \) \( \text{loc}(r1) = d2 \)
\( \text{loc}(c1) = d1 \) \( \text{move}(r1,d1,d3) \) \( \text{loc}(r1) = d3 \)
\( \text{cargo}(r1) = \text{nil} \) \( \text{load}(r1,c1,d1) \) \( \text{loc}(c1) = r1 \)
\( \text{cargo}(r1) = \text{c1} \) \( \text{cargo}(r1) = \text{c1} \)

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

\( \hat{a}_1 = \{ \text{move}(r1,d1,d3), \text{load}(r1,c1,d1) \} \)
\( h^{FF}(s_1) = 2 \)

\( s_1 = \{ \text{loc}(r1)=d1, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1 \} \)

\( g = \{ \text{loc}(r1)=d3, \text{loc}(c1)=r1 \} \)
**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 \( A_1 \):**
- \( \text{move}(r1,d2,d3) \)
- \( \text{move}(r1,d2,d1) \)

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

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

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

**\( \hat{a}_2 = \{ \text{move}(r1,d1,d3), \text{load}(r1,c1,d1) \} \)**

**\( \hat{a}_1 = \{ \text{move}(r1,d2,d1) \} \)**

**\( h^{FF}(s_2) = 3 \)**

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

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

Given r-state $\hat{s}_0$ and goal $g$:
For $i = 1$ by $1$ until we get an $\hat{s}_k$ that r-satisfies $g$
  - $A_i = \{\text{all actions r-applicable in } \hat{s}_{i-1}\}; \hat{s}_i = \gamma^+(s_{i-1}, A_i)$

$\langle A_1, A_2, \ldots, A_k \rangle$ is a relaxed solution

Extract minimal relaxed solution
  - $\hat{a}_k = \text{minimal subset of } A_k \text{ that r-achieves } g$
  - $\hat{a}_{k-1} = \text{minimal subset of } A_{k-1} \text{ that r-achieves } pre(\hat{a}_k)$
  - $\hat{a}_1 = \text{minimal subset of } A_1 \text{ that r-achieves } pre(\hat{a}_2)$

- Running time is polynomial in $|A| + \sum_{x \in X} |\text{Range}(x)|$
- $h^{FF}(s) = \sum \text{costs of } \hat{a}_1, \ldots, \hat{a}_k$
- Ambiguous value: depends on which minimal subsets we choose
  - $h^+(s) = \text{smallest cost of any relaxed solution } \leq h^{FF}(s)$
  - $h^{FF}$ not admissible
2.3.3 Landmark Heuristics

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

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

- $\text{loc}(r1)=d1$
- $\text{loc}(r1)=d3 \lor \text{loc}(r1)=d2$
- $\text{loc}(r1)=d3$
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 $m_2$
  - ...

\[ s_0 \xrightarrow{P_1} m_1 \xrightarrow{P_2} m_2 \xrightarrow{P_3} m_3 \xrightarrow{P_4} g \]
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 $\phi$ is a landmark, can get new landmarks from the preconditions of the actions that achieve $\phi$

- **Example:**
  - goal $g$
  - $\{a_1, a_2\} =$ 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 $g_1$: $\text{move}(r1,d3,d1)$ and $\text{move}(r1,d2,d1)$
- Preconditions $\text{loc}(r1)=d3$ and $\text{loc}(r1)=d2$
- New landmark:
  $\phi' = \text{loc}(r1)=d3 \lor \text{loc}(r1)=d2$

\[
\begin{align*}
\text{move}(r, d, e) & \quad \text{pre: } \text{loc}(r)=d \\
& \quad \text{eff: } \text{loc}(r) \leftarrow e
\end{align*}
\]

\[
\begin{align*}
\text{load}(r, c, l) & \quad \text{pre: } \text{cargo}(r)=\text{nil}, \text{loc}(c)=l, \text{loc}(r)=l \\
& \quad \text{eff: } \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r
\end{align*}
\]

\[
\begin{align*}
\text{unload}(r, c, l) & \quad \text{pre: } \text{loc}(c)=r, \text{loc}(r)=l \\
& \quad \text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l
\end{align*}
\]

\[
\begin{align*}
{s_0} & = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}
\end{align*}
\]
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\}; \; \text{Landmarks} \leftarrow \emptyset \\
\text{while } \text{queue} \neq \emptyset & \\
\text{remove a } g_i \text{ from } \text{queue}; \; \text{add it to } \text{Landmarks} & \\
\text{R} & \leftarrow \{\text{actions whose effects include } g_i\} \\
\text{if } s_0 \text{ satisfies pre}(a) \text{ for some } a \in R & \text{ then return } \text{Landmarks} \\
\text{generate RPG from } s_0 \text{ using } A \setminus R, \text{ stopping when } \hat{s}_k = \hat{s}_{k-1} & \\
\text{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} \\
\text{loop (over all combinations of preconditions below)} & \\
\text{for each action } a_j \text{ in } N & \\
\text{p}_j & \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0 \\
\varphi & \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n \\
\text{add } \varphi \text{ to } \text{queue} & \\
\text{return } \text{Landmarks} &
\end{align*}
RPG-based Landmark Computation

\begin{align*}
\text{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 & \\
& \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 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} \\
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} \\
\text{loop (over all combinations of preconditions below)} & \\
& \quad \text{for each action } a_j \text{ in } N \\
& \quad \quad p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0 \\
& \quad \quad \varphi \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n \\
& \quad \quad \text{add } \varphi \text{ to } \text{queue} \\
\text{return } \text{Landmarks}
\end{align*}
RPG-based Landmark Computation

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

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

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

loop (over all combinations of preconditions below)

for each action $a_j$ in $N$

$p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0$

$\varphi \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n$

add $\varphi$ to queue

return Landmarks

--

$N$ means “necessary”

- to achieve $g$, need at least one of these

---

Nau – Lecture slides for Automated Planning and Acting
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\}; \text{Landmarks} \leftarrow \emptyset \\
\text{while } \text{queue} \neq \emptyset &\quad \text{remove a } g_i \text{ from } \text{queue}; \text{ add it to } \text{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 } \text{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} \\
\text{loop (over all combinations of preconditions below)} &\quad \text{for each action } a_j \text{ in } N \\
\quad &\quad p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0 \\
\varphi &\leftarrow p_1 \lor p_2 \lor \ldots \lor p_n \\
\text{add } \varphi &\text{ to } \text{queue} \\
\text{return } \text{Landmarks}
\end{align*}
Example

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

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

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

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

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

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

loop (over all combinations of preconditions below)

for each action \(a_j\) in \(N\)

\[
p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0
\]

\[
\phi \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n
\]

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

return \(\text{Landmarks}\)

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

\[
g = \{\text{loc}(r1)=d3, \text{loc}(c1)=r1\}
\]
RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

queue ← \{gi ∈ g | s₀ doesn’t satisfy gi\}; Landmarks ← ∅

while queue ≠ ∅

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

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

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

generate RPG from $s_0$ using $A \setminus R$, stopping when $s_k = s_{k-1}$

$N ← \{\text{all actions in } R \text{ that are } r\text{-applicable in } s_k\}$

if $N = ∅$ then return failure

loop (over all combinations of preconditions below)

for each action $a_j$ in $N$

$p_j ← \text{a precondition of } a_j \text{ not satisfied in } s_0$

$\varphi ← p_1 ∨ p_2 ∨ \ldots ∨ p_n$

add $\varphi$ to queue

return Landmarks

---

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

$s_0 = \{\text{loc}(r_1)=d_3,$
\text{cargo}(r_1)=\text{nil},$
\text{loc}(c_1)=d_1\}$

---

$\text{load}(r, c, l)$

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

eff: cargo($r$)←c, loc($c$)←r

$\text{move}(r, d, e)$

pre: loc($r$)=d

eff: loc($r$)←e

$\text{unload}(r, c, l)$

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

eff: cargo($r$)←nil, loc($c$)←l

---

queue = ∅

Landmarks = \{loc($c_1$)=r₁\}

$R = \{\text{load}(r_1,c_1,d_1),$
\text{load}(r_1,c_1,d_2),$
\text{load}(r_1,c_1,d_3)\}$
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 $\text{pre}(a)$ 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

loop (over all combinations of preconditions below)

    for each action $a_j$ in $N$

        $p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0$

        $\phi \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n$

        add $\phi$ to $queue$

return $Landmarks$

---

Example

$load(r, c, l)$

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

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

$move(r, d, e)$

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

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

$unload(r, c, l)$

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

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

$queue = \emptyset$  

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

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

$N = \{\text{load(r1,c1,d1)}\}$
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 \(\hat{s}_k = \hat{s}_{k-1}\)

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

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

loop (over all combinations of preconditions below)

for each action \(a_j\) in \(N\)

\(p_j \leftarrow \text{a precondition of } a_j \text{ not satisfied in } s_0\)

\(\varphi \leftarrow p_1 \lor p_2 \lor \ldots \lor p_n\)

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

return \(\text{Landmarks}\)

\[\begin{align*}
\text{load}(r, c, l) \\
\text{pre: } &\text{cargo}(r) = \text{nil}, \ \text{loc}(c) = l, \\
&\text{loc}(r) = l \\
\text{eff: } &\text{cargo}(r) \leftarrow c, \ \text{loc}(c) \leftarrow r
\end{align*}\]

\[\begin{align*}
\text{load}(r1,c1,d1) \\
\text{pre: } &\text{cargo}(r1) = \text{nil}, \ \text{loc}(c1) = d1, \\
&\text{loc}(r1) = d1
\end{align*}\]

\[\begin{align*}
\text{load}(r, c, l) \\
\text{pre: } &\text{cargo}(r) = \text{nil}, \ \text{loc}(c) = d1, \\
&\text{loc}(r) = d1
\end{align*}\]

\[\begin{align*}
\text{load}(r1,c1,d1) \\
\text{pre: } &\text{cargo}(r1) = \text{nil}, \ \text{loc}(c1) = d1
\end{align*}\]
Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h(s) =$ number of landmarks to be accomplished from $s$
- Is this heuristic admissible?
Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - \( h(s) = \) number of landmarks to be accomplished from \( s \)
- Is this heuristic admissible?
  - 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
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

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

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

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

- Plans don’t always work
  - *The best-laid plans of mice and men often go awry*
    – Robert Burns
    *(translated from Scots dialect)*

- What to do about it?
Service Robot

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

**Function Definitions:**
- **go(r, l, m)**
  - pre: adjacent(l,m), loc(r)=l
  - eff: loc(r) ← m
- **navigate(r; l, m)**
  - pre: ¬adjacent(l, m), loc(r)=l
  - eff: loc(r) ← m
- **take(r; l, o)**
  - pre: loc(r)=l, loc(o)=l, cargo(r)=nil
  - eff: loc(o) ← r, cargo(r) ← o

**Service Robot Tasks:**
- Respond to user requests
- Bring o7 to loc2
- Go to hallway
- Navigate to loc1
- Fetch o7
- Navigate to loc2
- Deliver o7
- Move to door
- Open door
- Get out
- Close door
- Identify type of door
- Move close to knob
- Grasp knob
- Turn knob
- Maintain
- Pull
- Move back
- Ungrasp
- Monitor
- Monitor
Service Robot

\( s_0 = \{\text{loc(r1)=loc3, loc(o7)=loc1, cargo(r1)=nil}\} \)
\( g = \{\text{loc(o7)=loc2}\} \)
\( \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \)
  \( a_1 = \text{go(r1,loc3,hall)} \)
  \( a_2 = \text{navigate(r1,hall,loc1)} \)
  \( a_3 = \text{take(r1,loc1,o7)} \)
  \( a_4 = \text{navigate(r1,loc1,loc2)} \)
  \( a_5 = \text{put(r1,loc2,o7)} \)

- **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 \( \text{loc(o7)} \)
Using Planning in Acting

Run-Lookahead($\Sigma, g$)

while ($s \leftarrow$ abstraction of observed state $\xi) \neq g$ do

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

if $\pi = $ failure then return failure

$a \leftarrow$ 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($\Sigma, g$)

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

while $s \not\models g$ do

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

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

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

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

- 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

Run-Concurrent-Lookahead(Σ, g)

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

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 Simulate(Σ, s, g, π) ≠ failure then
      \[ a \leftarrow \text{pop-first-action}(\pi); \ \text{perform}(a) \]
      \[ s \leftarrow \text{abstraction of observed state } \xi \]

- May detect opportunities earlier than Run-Lazy-Lookahead
  - But may miss some that Run-Lookahead would find
- Without Simulate, may fail to detect problems before 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
Receding-Horizon Search

Deterministic-Search(Σ, s₀, g)

\[ \text{Frontier} \leftarrow \{ (\langle \rangle, s₀) \} \]

\[ \text{Expanded} \leftarrow \emptyset \]

while Frontier ≠ ∅ do

select a node \( ν = (\pi, s) \in \text{Frontier} \) (i)

remove \( ν \) from \( \text{Frontier} \)

add \( ν \) to \( \text{Expanded} \)

if \( s \) satisfies \( g \) then return \( π \) (ii)

\[ \text{Children} \leftarrow \{(\pi.a, γ(s,a)) | s \text{ satisfies } \text{pre}(a)\} \]

prune 0 or more nodes from \( \text{Children, Frontier, Expanded} \) (iii)

\[ \text{Frontier} \leftarrow \text{Frontier} \cup \text{Children} \]

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

**Greedy**(\(\Sigma, s, g, Visited\))

1. if \(s\) satisfies \(g\) then return \(\pi\)
2. \(Act \leftarrow \{a \in A \mid s\) satisfies \(pre(a)\) and \(\gamma(s, a) \notin Visited\}\)
3. if \(Act = \emptyset\) then return failure
4. \(a \leftarrow \arg \min_{a \in Act} h(\gamma(s, a))\)
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 (Continued)

- **2.2 Forward State-Space Search**
  - Forward-search, Deterministic-Search
  - Breadth-first, depth-first, uniform-cost, $A^*$, GBFS, DFBB, IDS, IDA*

- **2.3 Heuristic Functions**
  - Straight-line distance
  - Delete relaxation, $h^+$, $h^{FF}$
  - Landmark heuristics, RPG-Landmarks

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