Reminder: Technology Policy

No technology except for first four rows of left and right sides
Reminder: Due Feb 4

• Special accommodations letter:
  Scan documentation letter and email to FAI-L@cornell.edu

• Prelim and Final Exam conflicts (3420 – larger class rule)
Textbook

So far:
“Uninformed” Search: Sections 3.1-3.5

Today and next week:
“Informed” Search: Sections 3.6.1, 4.1
Adversarial Search: Sections 5.1, 5.2.1, 5.2.3
“Uninformed” Search

DFS
BFS
IDS

All you have is the structure of the space:
States, Initial State, Operators, Goal
“Uninformed” Search

To formulate a problem:

• States: S
• Operators: Ops
• Initial state
• Goal condition: goal(s)
“Informed” Search

To formulate a problem:

• States: \( S \)
• Operators: \( \text{Ops} \)
• Initial state
• Goal condition: \( \text{goal}(s) \)
• Heuristic Evaluation Function \( f(s): S \rightarrow \mathbb{R} \) (usually \( \geq 0 \))
  • \( f \) is an estimate of the merit of \( s \)
  • Typically \( f(s_1) < f(s_2) \) means \( s_1 \) is “better” than \( s_2 \)
"Informed" Search

To formulate a problem:

• States: S
• Operators: Ops
• Initial state
• Goal condition: goal(s)
• Heuristic Evaluation Function \( f(s) : S \rightarrow \mathbb{R} \) (usually \( \geq 0 \))
  • \( f \) is an estimate of the merit of \( s \)
  • Typically \( f(s_1) < f(s_2) \) means \( s_1 \) is “better” than \( s_2 \)

A heuristic technique (/hjuːˈrɪstɪk/; Ancient Greek: εὑρίσκω, "find" or "discover"), or a heuristic for short, is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect or rational, but which is nevertheless sufficient for reaching an immediate, short-term goal. [Wikipedia]
Search Algorithm Template

Initial call: \textbf{Search}(\text{initial state}, \text{ops}, \{\}, \{\})

\textbf{Search}(s, \text{ops}, \text{open}, \text{closed}) =
\begin{align*}
& \text{If goal}(s) \text{ Then return}(s); \\
& \text{Else If not}(s \in \text{closed}) \\
& \quad \text{Then} \\
& \quad \quad \text{successors } \leftarrow \{\}; \text{ add}(s, \text{closed}); \\
& \quad \quad \text{For each } o \in \text{ops} \text{ that applies to } s \\
& \quad \quad \quad \text{add apply}(o, s) \text{ to successors} \\
& \quad \quad \quad \text{open } \leftarrow \text{ add successors to open}; \\
& \quad \text{If not}(\text{empty}(\text{open})) \\
& \quad \quad s' \leftarrow \textbf{select}(\text{open}); \\
& \quad \quad \text{open } \leftarrow \text{ remove}(s', \text{open}); \\
& \quad \quad \textbf{search}(s', \text{ops}, \text{open}, \text{closed}) \\
\end{align*}
\begin{itemize}
\item Else return(FAIL)
\end{itemize}
Best First Search

Initial call: \textbf{BestFS}(\text{initialstate,ops,\{}{\}},\{}{\})

\textbf{BestFS}(s,\text{ops,open,closed}) =
  \text{If goal}(s) \text{ Then return}(s);
  \text{Else If not}(s \in \text{closed})
    \text{Then}
      \text{successors} \leftarrow \{}{\}; \text{add}(s, \text{closed});
      \text{For each } o \in \text{ops that applies to } s
        \text{add apply}(o,s) \text{ to successors}
      \text{open} \leftarrow \text{add successors to open};
  \text{If not}(\text{empty}(\text{open}))
    s' \leftarrow \text{best}(\text{open});
    \text{open} \leftarrow \text{remove}(s', \text{open});
    \textbf{BestFS}(s’,\text{ops,open,closed})
  \text{\textbackslash Else return(FAIL)}
Best First Search

Initial call: BestFS(initialstate,ops,{},{})

BestFS(s,ops,open,closed) =
    If goal(s) Then return(s);
Else If not(s ∈ closed)
    Then
        successors ← {}; add(s,closed);
        For each o ∈ ops that applies to s
            add apply(o,s) to successors
        open ← add successors to open;
    If not(empty(open))
        s' ← best(open);
        open ← remove(s',open);
        BestFS(s',ops,open,closed)
    Else return(FAIL)

Usually “best” means lowest value according to f:
best(open) = \arg\min_{s ∈ open} f(s)
Examples

Given uniform operator costs

\[ f(s) = \text{distance}(s, \text{initial state}) \]
Examples

Given uniform operator costs

\[ f(s) = \text{distance}(s, \text{initial state}) \]

Breadth-first search
Examples

Given uniform operator costs

\[ f(s) = \frac{1}{\text{distance}(s, \text{initial state})} \]
Examples

Given uniform operator costs

\[ f(s) = \frac{1}{\text{distance}(s, \text{initial state})} \]

Depth-first search
Examples

Given uniform operator costs

\[ f(s) = \frac{1}{\text{distance}(s, \text{initial state})} \]

Depth-first search

\[ \text{best(open)} = \text{argmin}_{s \in \text{open}} f(s) \]
Examples

Given uniform operator costs

\[ f(s) = \frac{1}{\text{distance}(s, \text{initial state})} \]

Depth-first search

\[ \text{best}(\text{open}) = \underset{s \in \text{open}}{\text{argmin}} f(s) \]
Examples

Given uniform operator costs

\[ f(s) = \text{distance}(s, \text{initialstate}) \]

Depth-first search

\[ \text{best(open)} = \arg\max_{s \in \text{open}} f(s) \]
Examples

Given uniform operator costs

\[ f(s) = \text{distance}(s, \text{initialstate}) \]

Depth-first search

\[ \text{best(open)} = \arg\max_{s \in \text{open}} f(s) \]

(Would not be what we typically do)
Examples

Given non-uniform operator costs
Examples

Given non-uniform operator costs

\[ f(s) = \text{(minimum) sum of costs from initial state to } s \]
Examples

Given non-uniform operator costs

\[ f(s) = \text{(minimum) sum of costs from initial state to } s \]

Uniform cost search
(Dijkstra’s algorithm)
Examples

Given non-uniform operator costs

\[ g(s) = \text{(minimum) sum of costs from initial state to } s \]

Uniform cost search
(Dijkstra’s algorithm)

We will typically use \( g(s) \) to designate this specific function
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]

[this is the observed cost to get to s]
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]

[this is the observed cost to get to s]

\[ h(s) = \text{estimate of cost from } s \text{ to nearest goal} \]
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]
\[ \text{[this is the observed cost to get to } s]\]

\[ h(s) = \text{estimate of cost from } s \text{ to nearest goal} \]
\[ \text{[this is a guess]} \]
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]
[This is the observed cost to get to s]

\[ h(s) = \text{estimate of cost from } s \text{ to nearest goal} \]
[This is a channel for giving information to the algorithm about the best direction to try]
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]
[This is the observed cost to get to s]

\[ h(s) = \text{estimate of cost from s to nearest goal} \]
[This is a channel for giving information to the algorithm about the best direction to try]

“Informed” Search
Informed Search

\[ g(s) = \text{sum of costs from initial state to } s \]
[This is the observed cost to get to \( s \)]

\[ h(s) = \text{estimate of cost from } s \text{ to nearest goal} \]
[This is a channel for giving information to the algorithm about the best direction to try]

“Informed” Search
Sometimes called “Heuristic Search”
Informed Search

\[ f(s) = g(s) + h(s) \]
Informed Search

\[ f(s) = g(s) + h(s) \]

[this is a guess about the cheapest solution to a goal if you go through state s]
Informed Search

\[ f(s) = g(s) + h(s) \]

[this is a guess about the cheapest solution to a goal if you go through state s]

Warning: sometimes “heuristic evaluation function” is used to refer just to \( h(s) \) and sometimes to all of \( f(s) \)
[they’re both “estimates”]
Best-first search with \( f(s) = g(s) + h(s) \) gets its own name

A* Search
Why \( h(s) = \) estimate of cost from \( s \) to nearest goal?
Why $h(s) = \text{estimate of cost from } s \text{ to nearest goal}$?

It is often easy to come up with
h(s) for Route Finding
h(s) for Route Finding

\[ h(s) = \text{sum of linear distances between } s \text{ and goal} \]
Heuristic Evaluation Function: 15 puzzle
Heuristic Evaluation Function:
15 puzzle

<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

→

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Heuristic Evaluation Function:
15 puzzle

4 1 2 3
5 6 7 11
8 9 10
12 13 14 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15
Heuristic Evaluation Function:
15 puzzle

<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

→

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Heuristic Evaluation Function: 15 puzzle

• $g(s) = \text{number of moves to get to } s$
• $h(s) = \text{number of out of place numbers in } s$

\[
\begin{array}{cccc}
4 & 1 & 2 & 3 \\
5 & 6 & 7 & 11 \\
8 & 9 & 10 & \\
12 & 13 & 14 & 15 \\
\end{array}
\quad
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & \\
\end{array}
\]

• $f(s) = \text{estimate of solution length from initial state to goal via } s$
Heuristic Evaluation Function: 15 puzzle

- $g(s) =$ number of moves to get to $s$
- $h(s) =$ number of out of place numbers in $s$
- $f(s) =$ estimate of solution length from initial state to goal via $s$

$h(s) = 12$

$$
\begin{array}{cccc}
4 & 1 & 2 & 3 \\
5 & 6 & 7 & 11 \\
8 & 9 & 10 & \\
12 & 13 & 14 & 15 \\
\end{array}
$$

$$
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & \\
\end{array}
$$
Heuristic Evaluation Function: 15 puzzle

- $g(s) =$ number of moves to get to $s$
- $h(s) =$ sum of “Manhattan” distances of each number from its correct position

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $f(s) =$ estimate of solution length from initial state to goal via $s$
Heuristic Evaluation Function: 15 puzzle

- **g(s)** = number of moves to get to s
- **h(s)** = sum of “Manhattan” distances of each number from its correct position

\[
\begin{array}{cccc}
4 & 1 & 2 & 3 \\
5 & 6 & 7 & 11 \\
8 & 9 & & 10 \\
12 & 13 & 14 & 15 \\
\end{array}
\quad
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & \\
\end{array}
\]

\[h(s) = 21\]

- **f(s)** = estimate of solution length from initial state to goal via s
Important Desirable Property of $h(s)$

- $h(s) =$ number of out of place numbers in $s$
- $h(s) =$ sum of “Manhattan” distances of each number from its correct position
- $h(s) =$ sum of linear distances between $s$ and goal
Important Desirable Property of $h(s)$

Denote the actual cost of an optimal path to a goal state $h^*(s)$
Important Desirable Property of \( h(s) \)

- \( h(s) = \) number of out of place numbers in \( s \)
- \( h(s) = \) sum of “Manhattan” distances of each number from its correct position
- \( h(s) = \) sum of linear distances between \( s \) and goal

\[
0 \leq h(s) \leq h^*(s)
\]

\( h(s) \) never overestimates the true cost to get to the closest goal
Admissible Heuristic Functions

- A function $h(s)$ is “admissible” if for every state $s$:

$$0 \leq h(s) \leq h^*(s)$$
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

$$0 \leq h(s) \leq h^*(s)$$

• If $h(s)$ is admissible then $A^*$ search is guaranteed to find an optimal solution
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

$$0 \leq h(s) \leq h^*(s)$$

Best-first search with $f(s) = g(s) + h(s)$

• If $h(s)$ is admissible then $A^*$ search is guaranteed to find an optimal solution
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

\[ 0 \leq h(s) \leq h^*(s) \]

Best-first search with $f(s) = g(s) + h(s)$

• If $h(s)$ is admissible then A* search is guaranteed to find an optimal solution

(With a couple of tweaks)
Search Algorithm Template

Initial call: \texttt{Search}(initialstate,ops,\emptyset,\emptyset)

\texttt{Search}(s,ops,open,closed) =

\hspace{1em} If \text{goal}(s) Then return(s);
\hspace{1em} Else If not(s \in \text{closed})
\hspace{1em} Then
\hspace{2em} successors \leftarrow \emptyset; \text{add}(s,\text{closed});
\hspace{2em} For each \text{o} \in \text{ops} that applies to \text{s}
\hspace{2em} \hspace{1em} add \text{apply}(o,s) to successors
\hspace{2em} \text{open} \leftarrow \text{add successors to open};
\hspace{1em} If not(empty(open))
\hspace{2em} s' \leftarrow \text{select}(open);
\hspace{2em} \text{open} \leftarrow \text{remove}(s',\text{open});
\hspace{2em} \text{search}(s',\text{ops,open,closed})
\hspace{1em} \text{Else return(FAIL)}
Search Algorithm Template

Initial call: Search(initialstate,ops,{},{})

Search(s,ops,open,closed) =
  If goal(s) Then return(s);
  Else If not(s ∈ closed)
    Then
      successors ← {}; add(s,closed);
      For each o ∈ ops that applies to s
        add apply(o,s) to successors
      open ← add successors to open;
      If not(empty(open))
        s’ ← select(open);
        open ← remove(s’,open);
        search(s’,ops,open,closed)
    Else return(FAIL)

Implicit:
  If a successor s is already in open
  don’t add it again
A* Search

Initial call: A*(initialstate,ops,{},{})

A*(s,ops,open,closed) =
    If goal(s) Then return(s);
    Else If not(s ∈ closed)
        Then
            successors ← {}; add(s,closed);
            For each o ∈ ops that applies to s
                add apply(o,s) to successors
            open ← add successors to open;
        If not(empty(open))
            s′ ← argmin_{s ∈ open} f(s);
            open ← remove(s′,open);
            A*(s’,ops,open,closed)
        Else return(FAIL)
    Else return(FAIL)
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^*(s, \text{ops}, \text{open}, \text{closed}) = \\
\quad \text{If goal}(s) \text{ Then return}(s); \\
\quad \text{Else If not}(s \in \text{closed}) \\
\quad \quad \text{Then successors} \leftarrow \{\}; \text{add}(s, \text{closed}); \\
\quad \quad \quad \text{For each } o \in \text{ops \ that \ applies \ to } s \text{ add apply}(o, s) \text{ to successors} \\
\quad \quad \text{open} \leftarrow \text{add successors to open}; \\
\]

\[
\quad \text{If not}(\text{empty}(\text{open})) \\
\quad \quad s' \leftarrow \text{argmin}_{s \in \text{open}}(f(s)); \\
\quad \quad \text{open} \leftarrow \text{remove}(s', \text{open}); \\
\quad \quad A^*(s', \text{ops}, \text{open}, \text{closed}) \\
\quad \text{Else return}(\text{FAIL})
\]
A* Search

Initial call: \( A^* (\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^* (s, \text{ops}, \text{open}, \text{closed}) = \begin{cases}
\text{If goal}(s) \text{ Then return}(s) ; \\
\text{Else If not}(s \in \text{closed}) \\
\quad \text{Then successors} \leftarrow \{\} ; \text{add}(s, \text{closed}) ; \\
\quad \text{For each } o \in \text{ops that applies to } s \text{ add apply}(o,s) \text{ to successors} \\
\quad \text{For each } s \in \text{successors} \\
\end{cases}
\]

If not(\text{empty}(\text{open}))

\[
s' \leftarrow \arg\min_{s \in \text{open}} f(s) ; \\
\text{open} \leftarrow \text{remove}(s', \text{open}) ; \\
A^* (s', \text{ops}, \text{open}, \text{closed}) \\
\quad \text{Else return}(\text{FAIL})
\]
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^*(s, \text{ops}, \text{open}, \text{closed}) =
\]
If goal(s) Then return(s);
Else If not(s \( \in \) closed)
Then successors \( \leftarrow \{\}; \text{add}(s, \text{closed}); \)
For each \( o \in \text{ops} \) that applies to \( s \) add apply(\( o, s \)) to successors
For each \( s \in \text{successors} \)
If \( s \in \text{open} \) and g(s) of current path is less, update g(s) in open

If not(empty(open))
\[
s' \leftarrow \text{argmin}_{s \in \text{open}} f(s);
\]
open \( \leftarrow \text{remove}(s', \text{open}); \)
\( A^*(s', \text{ops}, \text{open}, \text{closed}) \)
Else return(FAIL)
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \emptyset, \emptyset) \)

\[ A^*(s, \text{ops}, \text{open}, \text{closed}) = \]
\begin{align*}
& \text{If goal(s) Then return(s);} \\
& \text{Else If not(s } \in \text{ closed) Then} \\
& \hspace{1em} \text{successors } \leftarrow \{\}; \text{ add(s, closed);} \\
& \hspace{2em} \text{For each } o \in \text{ ops that applies to } s \text{ add apply(o,s) to successors} \\
& \hspace{2em} \text{For each } s \in \text{ successors} \\
& \hspace{3em} \text{If } s \in \text{ open and } g(s) \text{ of current path is less, update } g(s) \text{ in open} \\
& \text{If not(\text{empty(open)})} \\
& \hspace{1em} s' \leftarrow \text{argmin}_s f(s); \\
& \hspace{2em} \text{open } \leftarrow \text{remove}(s', \text{open}); \\
& \hspace{2em} A^*(s', \text{ops}, \text{open}, \text{closed}) \\
& \text{\textbackslash Else return(FAIL)}
\end{align*}

Handles multiple paths to a single state
A* Search

Initial call: \( A^* \text{(initialstate,ops,{},{} limb)} \)

\[
A^*(s, ops, open, closed) =
\begin{cases}
\text{If goal(s) Then return(s); } & \\
\text{Else If not(s} \in \text{closed) } & \\
\quad \text{Then successors } \leftarrow \{\}; \text{ add(s,closed); } & \\
\quad \text{For each } o \in \text{ops that applies to } s \text{ add apply}(o,s) \text{ to successors } & \\
\quad \text{For each } s \in \text{successors } & \\
\quad \quad \text{If } s \in \text{open and } g(s) \text{ of current path is less, update } g(s) \text{ in open } & \\
\quad \quad \text{Else If } s \in \text{closed and } g(s) \text{ of current path is less, add } s \text{ to open w/ new } g(s) & \\
\end{cases}
\]

\text{If not(empty(open))}
\[
s' \leftarrow \arg\min_{s \in \text{open}} (f(s)); \\
\text{open} \leftarrow \text{remove}(s',\text{open}); \\
A^*(s',\text{ops,open,closed}) & \\
\text{Else return(FAIL)}
\]
A* Search

Initial call: A*(initialstate,ops,{},{}).

A*(s,ops,open,closed) =
   If goal(s) Then return(s);  
   Else If not(s ∈ closed)
      Then successors ← {}; add(s,closed);
      For each o ∈ ops that applies to s add apply(o,s) to successors
      For each s ∈ successors
         If s ∈ open and g(s) of current path is less, update g(s) in open
         Else If s ∈ closed and g(s) of current path is less, add s to open w/ new g(s)
   If not(empty(open))
      s’ ← argmin_{s ∈ open} f(s);
      open ← remove(s’,open);
      A*(s’,ops,open,closed)
      Else return(FAIL)

All successors of s reflect a higher f(s) than they should
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^*(s, \text{ops}, \text{open}, \text{closed}) =
\]
\[
\text{If goal}(s) \text{ Then return}(s);
\]
\[
\text{Else If not}(s \in \text{closed})
\]
\[
\text{Then successors }\leftarrow \{\}; \text{ add}(s, \text{closed});
\]
\[
\text{For each } o \in \text{ops} \text{ that applies to } s \text{ add } \text{apply}(o, s) \text{ to successors}
\]
\[
\text{For each } s \in \text{successors}
\]
\[
\text{If } s \in \text{open} \text{ and } g(s) \text{ of current path is less, update } g(s) \text{ in open}
\]
\[
\text{Else If } s \in \text{closed} \text{ and } g(s) \text{ of current path is less, add } s \text{ to open w/ new } g(s)
\]

\[
\text{If not(}\text{empty(}\text{open})\text{)}
\]
\[
s' \leftarrow \text{argmin}_{s \in \text{open}} (f(s));
\]
\[
\text{open }\leftarrow \text{remove}(s', \text{open});
\]
\[
A^*(s', \text{ops}, \text{open}, \text{closed})
\]
\[
\text{Else return(FAIL)}
\]

Easier than doing bookkeeping and updating all successors
A* Search

Initial call: $A^*(\text{initialstate}, \text{ops}, \emptyset, \emptyset)$

$A^*(s, \text{ops}, \text{open}, \text{closed}) =$
If goal(s) Then return(s);
Else If not($s \in \text{closed}$)
Then successors $\leftarrow \emptyset$; add($s$, closed);
  For each $o \in \text{ops}$ that applies to $s$ add apply($o, s$) to successors
  For each $s \in \text{successors}$
    If $s \in \text{open}$ and $g(s)$ of current path is less, update $g(s)$ in open
    Else If $s \in \text{closed}$ and $g(s)$ of current path is less, add $s$ to open w/ new $g(s)$
    Else If $s \not\in \text{open}$ and $s \not\in \text{closed}$, add $s$ to open
If not(empty(open))
  $s' \leftarrow \arg\min_s f(s)$;
  $s \in \text{open}$
  open $\leftarrow \text{remove}(s', \text{open})$;
  $A^*(s', \text{ops}, \text{open}, \text{closed})$
Else return(FAIL)
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

$$0 \leq h(s) \leq h^*(s)$$

• If $h(s)$ is admissible then A* search is guaranteed to find an optimal solution
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

\[ 0 \leq h(s) \leq h^*(s) \]

• If $h(s)$ is admissible then A* search is guaranteed to find an optimal solution

WHY?
Admissible Heuristic Functions

• A function $h(s)$ is “admissible” if for every state $s$:

$$0 \leq h(s) \leq h^*(s)$$

• If $h(s)$ is admissible then A* search is guaranteed to find an optimal solution

   WHY?
   
   You would never take off a goal state if a better state is in open
Consistent Evaluation Functions

• A function $h(s)$ is “consistent” (or “monotonic”) if it satisfies the triangle inequality for all $s$:

$$h(s) \leq h(\text{apply}(a,s)) + \text{cost}(\text{apply}(a,s))$$

Intuitively: the estimate gets increasingly better over time
Consistency vs Admissibility

• Consistency is stronger than admissibility:
  
  If $h(s)$ is consistent then $h(s)$ is admissible

• Consistency lets you improve the A* algorithm
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^*(s, \text{ops}, \text{open}, \text{closed}) = \\
\text{If goal}(s) \text{ Then return}(s); \\
\text{Else If not}(s \in \text{closed}) \\
\text{Then successors } \leftarrow \{\}; \text{ add}(s, \text{closed}); \\
\text{For each } o \in \text{ops that applies to } s \text{ add apply}(o, s) \text{ to successors} \\
\text{For each } s \in \text{successors} \\
\text{If } s \in \text{open} \text{ and } g(s) \text{ of current path is less, update } g(s) \text{ in open} \\
\text{Else If } s \in \text{closed} \text{ and } g(s) \text{ of current path is less, add } s \text{ to open w/ new } g(s) \\
\text{Else If } s \notin \text{open} \text{ and } s \notin \text{closed}, \text{ add } s \text{ to open} \\
\text{If not(empty(open))} \\
\quad s' \leftarrow \text{argmin}(f(s)); \\
\quad s \in \text{open} \\
\quad \text{open } \leftarrow \text{remove}(s', \text{open}); \\
\quad A^*(s', \text{ops}, \text{open}, \text{closed}) \\
\text{Else return}(\text{FAIL})
\]
Consistency vs Admissibility

• Consistency is stronger than admissibility:
  If \( h(s) \) is consistent then \( h(s) \) is admissible

• Consistency lets you improve the A* algorithm

• A* is an optimal search method: No other algorithm using the same \( h(s) \) and the same tie-breaking rules will expand fewer nodes than A*
Consistency vs Admissibility

• Consistency is stronger than admissibility:
  If $h(s)$ is consistent then $h(s)$ is admissible

• Consistency lets you improve the A* algorithm

• A* is an optimal search method: No other algorithm using the same $h(s)$ and the same tie-breaking rules will expand fewer nodes than A*

• Consistency is often more difficult to show than admissibility
Consistency vs Admissibility

• Consistency is stronger than admissibility:
  If $h(s)$ is consistent then $h(s)$ is admissible

• Consistency lets you improve the A* algorithm

• A* is an optimal search method: No other algorithm using the same $h(s)$ and the same tie-breaking rules will expand fewer nodes than A*

• Consistency is often more difficult to show than admissibility
  (It takes some effort to come up with an admissible function that isn’t consistent)
Properties of A* Search

• If
  • search space is a finite graph and
  • all operator costs are positive

• Then
  • A* is guaranteed to terminate and
  • if there is a solution, A* will find a solution (not necessarily an optimal one)
Properties of A* Search

• If
  • search space is an *infinite* graph (but branching factor is finite) and
  • all operator costs are positive and are never less than some number \( \varepsilon \) (in other words, they cannot get arbitrarily close to 0)

• Then
  • if there is a solution, A* will terminate with a solution (not necessarily an optimal one)
    (no guarantee of termination if there is no solution)
Properties of A* Search

• If, in addition,
  • $h(s)$ is admissible
    (for all states $s$, $0 \leq h(s) \leq h^*(s)$)

• Then
  • If A* terminates with a solution it will be optimal
Properties of A* Search

• If, in addition,
  • $h(s)$ is consistent
    (for all states $s$, $h(s) \leq h(\text{apply}(a,s)) + \text{cost}(\text{apply}(a,s))$
  • Then
    • $h(s)$ is admissible,
    • the first path found to any state is guaranteed to have the lowest cost
      (do not need to check for this in the algorithm), and
    • no other algorithm using the same $h(s)$ and the same tie-breaking rules will expand fewer nodes than A*
Properties of A* Search

• If
  • the search space is a tree,
  • there is a single goal state, and
  • for all states s, $|h^*(s) - h(s)| = O(\log(h^*(s)))$
    (the error of $h(s)$ is never more than a logarithmic factor of $h^*(s)$)

• Then
  • A* runs in time polynomial in $b$ (branching factor)
A* Variants

Weighted A*:

• If
  • $h(s)$ is admissible and
  • $A^*$ is used with $h'(s) = c \times h(s)$ where $c > 1$

• Then
  • Any goal state that $A^*$ terminates with will have cost no more than $c$ times the cost of an optimal solution
A* Variants

IDA*:

- Use cost-bounded depth-first search with $h$(initial state) as the bound
- Any time a successor is greater than the bound don’t expand it
  - But store the lowest cost $C$ of any such state that you reach that exceeds the cost bound
- If you terminate without a goal state run cost-bounded depth-first search with depth bound $C$

($=$ Depth-first search emulation of A* search)
A* Variants

SMA*:

• A* search with a memory bound

• If you would generate a node but don’t have space to add it to Open, remove from open the node $s$ on Open with greatest $f(s)$ but keep track of its parent $s’$ and the cost of the removed node $f(s)$

• If you reach a node on Open whose cost is worse than this value, you re-expand $s’$