Iterative Deepening Search

IDS(s,ops)
    i=0
    loop
        result = DBDFS(s,ops,{}, {}, i)
        if result = fail then i=i+1
        else return(result)
Case 3b: Search Graph Using Graph Search

(Assuming finite state space, maximum distance from initial state m)

<table>
<thead>
<tr>
<th>Method</th>
<th>DFS</th>
<th>BFS</th>
<th>IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^m)$</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(bm)$</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>
# Case 3b: Search Graph Using Graph Search

(Assuming finite state space, maximum distance from initial state m)

<table>
<thead>
<tr>
<th>Method</th>
<th>DFS</th>
<th>BFS</th>
<th>IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^m)$</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(bm)$</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>

Why?

---

DFS: Depth-First Search
BFS: Breadth-First Search
IDS: Iterative Deepening Search
Time Complexity of IDS

- Time for depth-bounded depth-first search to level $i$: $O(b^i)$
Time Complexity of IDS

- Time for depth-bounded depth-first search to level $i$: $O(b^i)$
- Time for IDS is the sum of this time for $i = 0, ..., d$
Time Complexity of IDS

• Time for depth-bounded depth-first search to level \(i\): \(O(b^i)\)

• Time for IDS is the sum of this time for \(i = 0, \ldots, d\)

\[
\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}
\]
Time Complexity of IDS

• Time for depth-bounded depth-first search to level \( i \): \( O(b^i) \)
• Time for IDS is the sum of this time for \( i = 0, \ldots, d \)

\[
\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}
\]

This is \( O(b^d) \)
Time Complexity of IDS

• Time for depth-bounded depth-first search to level i: $O(b^i)$
• Time for IDS is the sum of this time for $i = 0, ..., d$

$$
\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}
$$

This is $O(b^d)$
(You should be able to do this)
“Uninformed” Search

DFS
BFS
DBDFS
IDS

All you have is the structure of the space:
States, Initial State, Operators, Goal
“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 $s_2$
Best-First Search

\[
\text{BestFS}(s, \text{ops}, \text{open}, \text{closed}) = \\
\begin{aligned}
&\text{If goal}(s) \text{ Then return}(s); \\
&\text{Else If not}(s \in \text{closed}) \\
&\quad \text{Then} \\
&\quad \quad \text{successors} \leftarrow \{\}; \text{ add}(s, \text{closed}); \\
&\quad \quad \text{For each } o \in \text{ops} \text{ that applies to } s \\
&\quad \quad \quad \text{add apply}(o, s) \text{ to successors} \\
&\quad \quad \text{open} \leftarrow \text{ add successors to open;} \\
&\text{If not}(\text{empty(open)}) \\
&\quad s' \leftarrow \min_f(\text{open}); \\
&\quad \text{open} \leftarrow \text{ remove}(s', \text{open}); \\
&\quad \text{BestFS}(s', \text{ops}, \text{open}, \text{closed}) \\
&\text{Else return}(\text{FAIL})
\end{aligned}
\]

Initial call: \text{BestFS}(\text{initialstate, ops, \{\}, \{\}})
Best-First Search Generalizes Other Search Methods

• Given uniform operator costs
  • If \( f(s) = \text{distance}(s, \text{initial state}) \), get Best-First Search
  • If \( f(s) \) inverts this (for example, \( -\text{distance}(s, \text{initial state}) \) or \( 1/\text{distance}(s, \text{initial state}) \)), get Depth-First Search

• Non-uniform operator costs
  • If \( f(s) = \text{sum of costs from initial state to } s \)
Best-First Search Generalizes Other Search Methods

• Given uniform operator costs
  • If $f(s) = \text{distance}(s, \text{initial state})$, get Best-First Search
  • If $f(s)$ inverts this (for example, $-\text{distance}(s, \text{initial state})$ or $1/\text{distance}(s, \text{initial state})$), get Depth-First Search

• Non-uniform operator costs
  • If $f(s) = \text{sum of costs from initial state to } s$
    Uniform Cost Search, aka
    Dijkstra’s Algorithm
Best-First Search Generalizes Other Search Methods

• Given uniform operator costs
  • If \( f(s) = \text{distance}(s, \text{initial state}) \), get Best-First Search
  • If \( f(s) \) inverts this (for example, \(- \text{distance}(s, \text{initial state})\) or \(1/\text{distance}(s, \text{initial state})\)), get Depth-First Search

• Non-uniform operator costs
  • If \( f(s) = \text{sum of costs from initial state to } s \)
    
    Uniform Cost Search, aka
    
    Dijkstra’s Algorithm
  • Often use \( g(s) \) instead of \( f(s) \) for this special case
Best-First Search Generalizes Other Search Methods

• Given uniform operator costs
  • If \( f(s) = \text{distance}(s, \text{initial state}) \), get Best-First Search
  • If \( f(s) \) inverts this (for example, \(- \text{distance}(s, \text{initial state})\) or \(1/\text{distance}(s, \text{initial state})\)), get Depth-First Search

• Non-uniform operator costs
  • If \( f(s) = \text{sum of costs from initial state to } s \)
    Uniform Cost Search, aka
    Dijkstra’s Algorithm
  • Often use \( g(s) \) instead of \( f(s) \) for this special case

I said \( f(s) \) estimates the quality of \( s \) – but there’s no estimation here ????
A* Search

• Special case of best-first search with a specific form of evaluation function:

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

where

• \( g(s) \) is the cost of getting to \( s \) (same as before)
• \( h(s) \) is an estimate of the cost getting to a goal state
A* Search

- Special case of best-first search with a specific form of evaluation function:

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

where

- \( g(s) \) is the cost of getting to \( s \) (same as before)
- \( h(s) \) is an estimate of the cost getting to a goal state

\( f(s) \) is an estimate of the cost of getting from the initial state to a goal state going through \( s \)
A* Search

• Special case of best-first search with a specific form of evaluation function:

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

where

• \( g(s) \) is the cost of getting to \( s \) (same as before)
• \( h(s) \) is an estimate of the cost getting to a goal state

\( f(s) \) is an estimate of the cost of getting from the initial state to a goal state going through \( s \)

(Warning: sometimes “heuristic evaluation function” is used to refer just to \( h(s) \) and sometimes to all of \( f(s) \) – they’re both “estimates”
}
Why \( f(s) = g(s) + h(s) \)?

Answer 1:
For many problems it isn’t too difficult to come up with an \( h(s) \)
Heuristic Evaluation Function: Route Finding

[Image of a graph showing various cities connected by lines and numbers indicating distances. The cities are labeled, and two nodes are highlighted: one as the Initial State and another as the Goal State.]
Heuristic Evaluation Function:
Route Finding

- \( g(s) = \) sum of costs from Arad to a city \( s \)
- \( h(s) = \) distance between \( s \) and Bucharest on a map
- \( f(s) = \) estimate of distance from Arad to Bucharest through \( s \)

\[ h(s) = \text{distance to Bucharest} \]
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
Heuristic Evaluation Function: 15 puzzle

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

→

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

- \( g(s) = \) number of moves to get to \( s \)
- \( h(s) = \) number of out of place numbers in \( s \)

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

- \( f(s) = \) estimate of solution length from initial state to goal via \( s \)

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

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

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

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

$h(s) = 12$

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

• $g(s) =$ number of moves to get to $s$

• $h(s) =$ sum of distances of each number from its correct position

<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>

• $f(s) =$ 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 distances of each number from its correct position

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

- \( f(s) \) = 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 distances of each number from its correct position

$$h(s) = 21$$

- $f(s)$ = estimate of solution length from initial state to goal via $s$

“Manhattan Distance”
Common Property of These Two $h(s)$ Examples

Denote the actual cost of an optimal path to a goal state $h^*(s)$
Common Property of These Two $h(s)$ Examples

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

$$0 \leq h(s) \leq h^*(s)$$
Common Property of These Two $h(s)$ Examples

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

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

Map distance never over-estimates driving distance
Manhattan distance never over-estimates solution distance
Admissible Heuristic Functions

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

$$0 \leq h(s) \leq h^*(s)$$
Why $f(s) = g(s) + h(s)$?

Answer 1:
For many problems it isn’t too difficult to come up with an $h(s)$
Why $f(s) = g(s) + h(s)$?

Answer 2:

Knowing something about $h(s)$ can give you guarantees on the outcome of search
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

(need slight modification to Best-First Search)
Search Algorithm Template

\[\text{Search}(s, \text{ops}, \text{open}, \text{closed}) =\]

\[
\begin{align*}
\text{If goal}(s) \text{ Then return}(s); \\
\text{Else If not}(s \in \text{closed}) \\
\quad \text{Then} \\
\quad \text{successors} \leftarrow \{\}; \text{add}(s, \text{closed}); \\
\quad \text{For each } o \in \text{ops} \text{ that applies to } s \\
\quad \quad \text{add apply}(o, s) \text{ to successors} \\
\quad \text{open} \leftarrow \text{add successors to open}; \\
\text{If not}(\text{empty}(\text{open})) \\
\quad s' \leftarrow \text{select}(\text{open}); \\
\quad \text{open} \leftarrow \text{remove}(s', \text{open}); \\
\quad \text{search}(s', \text{ops}, \text{open}, \text{closed}) \\
\quad \text{Else return}(\text{FAIL})
\end{align*}
\]
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 that are not already in open to open;
            If not(empty(open))
                s’ ← select(open); 
                open ← remove(s’,open);
                search(s’,ops,open,closed)
            Else return(FAIL)
A* Search

\[
\text{Search}(s, \text{ops, open, closed}) = \\
\quad \text{If goal}(s) \text{ Then return}(s); \\
\quad \text{Else If not}(s \in \text{closed}) \\
\quad \quad \text{Then} \\
\quad \quad \quad \text{successors} \leftarrow \{\}; \text{ add}(s, \text{closed}); \\
\quad \quad \quad \text{For each } o \in \text{ops} \text{ that applies to } s \\
\quad \quad \quad \quad \text{add apply}(o,s) \text{ to successors} \\
\quad \quad \quad \text{open} \leftarrow \text{add successors that are not already in open to open}; \\
\quad \quad \text{If not}(\text{empty}(\text{open})) \\
\quad \quad \quad s' \leftarrow \text{select}(\text{open}); \\
\quad \quad \quad \text{open} \leftarrow \text{remove}(s', \text{open}); \\
\quad \quad \quad \text{search}(s', \text{ops, open, closed}) \\
\quad \quad \quad \text{'Else return}(\text{FAIL})
\]

If a successor is in open and the new g(s) is smaller update g(s) in open
A* Search

\[
\text{Search}(s, \text{ops}, \text{open}, \text{closed}) = \\
\quad \text{If } \text{goal}(s) \text{ Then return }(s) ; \quad \text{Else If not}(s \in \text{closed}) \quad \text{Then} \quad \\
\quad \quad \text{successors} \leftarrow \{\} ; \text{add}(s, \text{closed}) ; \quad \text{For each } o \in \text{ops} \text{ that applies to } s \quad \\
\quad \quad \quad \quad \text{add } \text{apply}(o, s) \text{ to successors} \quad \text{add successors that are not already in open to open;} \quad \text{If not}(\text{empty}(\text{open})) \quad \\
\quad \quad \quad \quad \quad \text{If a successor is in open and the new } g(s) \text{ is smaller update } g(s) \text{ in open} \quad \\
\quad \quad \quad \quad \quad \text{If a successor is in closed and the new } g(s) \text{ is smaller, put } s \text{ back on open} \quad \\
\quad \quad \quad \quad \quad \text{If not}(\text{empty}(\text{open})) \quad \\
\quad \quad \quad \quad \quad \quad \text{s'} \leftarrow \text{select}(\text{open}) ; \quad \text{If a successor is in open and the new } g(s) \text{ is smaller update } g(s) \text{ in open} \quad \\
\quad \quad \quad \quad \quad \quad \text{If a successor is in closed and the new } g(s) \text{ is smaller, put } s \text{ back on open} \quad \\
\quad \quad \quad \quad \quad \quad \text{open} \leftarrow \text{remove}(s', \text{open}) ; \quad \text{search}(s', \text{ops}, \text{open}, \text{closed}) \quad \\
\quad \text{Else return } (\text{FAIL}) \quad \\
\]
A* Search

- If a successor is in open and the new $g(s)$ is smaller:
  Handles multiple paths to the same state
- If a successor is in closed and the new $g(s)$ is smaller:
  Necessary to guarantee optimality
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
A* Search

• If a successor is in open and the new $g(s)$ is smaller:
  Handles multiple paths to the same state

• If a successor is in closed and the new $g(s)$ is smaller:
  Necessary to guarantee optimality
A* Search

• If a successor is in open and the new $g(s)$ is smaller:
  Handles multiple paths to the same state
• If a successor is in closed and the new $g(s)$ is smaller:
  Necessary to guarantee optimality
  $s$ might have been explored further, need to now follow that through
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

\[
\text{Search}(s, \text{ops}, \text{open}, \text{closed}) =
\text{If goal}(s) \text{ Then return}(s);
\text{Else If not}(s \in \text{closed}) \text{ Then}
\quad \text{successors} \leftarrow \emptyset; \text{ add}(s, \text{closed});
\quad \text{For each } o \in \text{ops} \text{ that applies to } s
\quad \text{ add apply}(o, s) \text{ to successors}
\quad \text{open} \leftarrow \text{ add successors that are not already in open to open};
\text{If not}(\text{empty}(\text{open}))
\quad s' \leftarrow \text{select}(\text{open});
\quad \text{open} \leftarrow \text{ remove}(s', \text{open});
\quad \text{search}(s', \text{ops}, \text{open}, \text{closed})
\quad \text{Else return}(\text{FAIL})
\]