Lecture: Informed (Heuristic) Search
CS 583 (002) - Fall 2017

Amarda Shehu

Department of Computer Science
George Mason University, Fairfax, VA, USA

November 29, 2017
1 Outline of Today’s Class

2 Reflections/Insights on Uninformed Search

3 Informed Search
   - Uniform-cost Search
   - Best-first Search
   - A* Search
   - B* Search
   - D* Search
   - Informed Search Summary
**Insight:** All covered graph-search algorithms follow similar template:

- "Maintain" a set of explored vertices \( S \) and a set of unexplored vertices \( V - S \)
- "Grow" \( S \) by exploring edges with exactly one endpoint in \( S \) and the other in \( V - S \)
- "Keep" in fringe data structure the (border) vertices in \( V - S \) reachable from \( S \)
- When extracting from fringe, vertex removed from \( V - S \) and added to \( S \)

**Implication:** similar template → reusable code

**Data structure \( F \) for the fringe:** order in which vertices are extracted from \( V - S \) distinguishes search algorithms from one another

- **DFS:** Take edge from vertex discovered most recently \((F \text{ is a stack})\)
- **BFS:** Take edge from vertex discovered least recently \((F \text{ is a queue})\)

- What else could \( F \) be?
- Could we impose a different order?
- Can do in a priority queue
- Need priorities/costs associated with vertices
- What information in graph can we use that we have not used so far?
Find a **least-cost/shortest** path from initial vertex to goal vertex

- Make use of **costs/weights** in state-space graph

- **Informed** graph search algorithms:
  - Dijkstra’s Search [Edsger Dijkstra 1959]
  - Uniform-cost Search (a variant of Dijkstra’s)
  - Best-First Search [Judea Pearl 1984]
  - B* Search [Hans Berliner 1979]
  - D* Search [Stenz 1994]
  - More variants of the above

- **What we will not** cover in this class:
  - What to do if weights are negative
  - Dynamic Programming rather than greedy paradigm
  - Subject of CS583 (Algorithms) [Bellman-Ford’s, Floyd-Warshall’s]
The weight of a path \( p = (v_1, v_2, \ldots, v_k) \) is the sum of the weights of the corresponding edges: \( w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i) \).

The shortest path weight from a vertex \( u \) to a vertex \( v \) is:

\[
\delta(u, v) = \begin{cases} 
\min \{ w(p) : p = (u, \ldots, v) \} & \text{if } p \text{ exists} \\
\infty & \text{else}
\end{cases}
\]

A shortest path from \( u \) to \( v \) is any path \( p \) with weight \( \delta(u, v) \).

The tree of shortest paths is a spanning tree of \( G = (V, E) \), where the path from its root, the source vertex \( s \), to any vertex \( u \in V \) is the shortest path \( s \leadsto u \) in \( G \).

Tree grows from \( S \) to \( V - S \)
Start vertex first to be extracted from \( V - S \) and added to \( S \)
As \( S \) grows (\( V - S \) shrinks), tree grows
Tree grows in iterations, one vertex extracted from \( V - S \) at a time
When will I find \( s \leadsto g \)?
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
All you need to remember about informed search algorithms

- Associate an attachment cost $d[v]$ with each vertex $v$
- F becomes a priority queue: F keeps fringe/frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
All you need to remember about informed search algorithms

- Associate a(n attachment) cost \( d[v] \) with each vertex \( v \)
- \( F \) becomes a priority queue: \( F \) keeps fringe/frontier vertices, prioritized by \( d[v] \)
- Until \( F \) is empty, one vertex extracted from \( F \) at a time
  - Can terminate earlier? When? How does it relate to goal?
- \( v \) extracted from \( F \) @ some iteration is one with lowest cost among all those in \( F \)
All you need to remember about informed search algorithms

- Associate a(n attachment) cost \( d[v] \) with each vertex \( v \)
- \( F \) becomes a priority queue: \( F \) keeps fringe/frontier vertices, prioritized by \( d[v] \)
- Until \( F \) is empty, one vertex extracted from \( F \) at a time
  Can terminate earlier? When? How does it relate to goal?
- \( v \) extracted from \( F \) @ some iteration is one with lowest cost among all those in \( F \)
  ... so, vertices extracted from \( F \) in order of their costs
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  - Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  - ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
All you need to remember about informed search algorithms

- Associate an attachment cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  - Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  - ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
  - $v$ has been “removed” from $V - S$ and “added” to $S$
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  - Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  - ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
  - $v$ has been “removed” from $V – S$ and “added” to $S$
  - get to reach/see $v$’s neighbors and possibly update their costs
All you need to remember about informed search algorithms

- Associate an attachment cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  - Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  - ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
  - $v$ has been “removed” from $V - S$ and “added” to $S$
  - get to reach/see $v$’s neighbors and possibly update their costs

The rest are details, such as:

- What should $d[v]$ be? There are options...
  - backward cost (cost of $s \rightarrow v$)
  - forward cost (estimate of cost of $v \rightarrow g$)
  - back+forward cost (estimate of $s \rightarrow g$ through $v$)
- Which do I choose? This is how you end up with different search algorithms
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
  $v$ has been “removed” from $V - S$ and “added” to $S$
  get to reach/see $v$’s neighbors and possibly update their costs

The rest are details, such as:

- What should $d[v]$ be? There are options...
  - backward cost (cost of $s \rightarrow v$)
  - forward cost (estimate of cost of $v \rightarrow g$)
  - back+forward cost (estimate of $s \rightarrow g$ through $v$)
- Which do I choose? This is how to you end up with different search algorithms
All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex $v$
- $F$ becomes a priority queue: $F$ keeps fringe/frontier vertices, prioritized by $d[v]$
- Until $F$ is empty, one vertex extracted from $F$ at a time
  - Can terminate earlier? When? How does it relate to goal?
- $v$ extracted from $F$ @ some iteration is one with lowest cost among all those in $F$
  - ... so, vertices extracted from $F$ in order of their costs
- When $v$ extracted from $F$:
  - $v$ has been “removed” from $V - S$ and “added” to $S$
  - get to reach/see $v$’s neighbors and possibly update their costs

The rest are details, such as:

- What should $d[v]$ be? There are options...
  - backward cost (cost of $s \leadsto v$)
  - forward cost (estimate of cost of $v \leadsto g$)
  - back+forward cost (estimate of $s \leadsto g$ through $v$)
- Which do I choose? This is how you end up with different search algorithms
Dijkstra’s Search Algorithm

Dijkstra extracts vertices from fringe (adds to \( S \)) in order of their backward costs

**Claim:** When a vertex \( v \) is extracted from fringe \( F \) (thus “added” to \( S \)), the shortest path from \( s \) to \( v \) has been found.

**Proof:** by induction on \(|S|\) (Base case \(|S| = 1\) is trivial).

Assume invariant holds for \(|S| = k \geq 1\).

- Let \( v \) be vertex about to be extracted from fringe (added to \( S \)), so has lowest backward cost
- Last time \( d[v] \) updated when parent \( u \) extracted from fringe
- When \( d[v] \) is lowest in the fringe, should we extract \( v \) or wait?
- Could \( d[v] \) get lower later through some other vertex \( y \) in fringe?

\[
\begin{align*}
    w(P) &\geq w(P') + w(x, y) & \text{nonnegative weights} \\
    &\geq d[x] + w(x, y) & \text{inductive hypothesis} \\
    &\geq d[y] & \text{definition of } d[y] \\
    &\geq d[v] & \text{Dijkstra chose } v \text{ over } y
\end{align*}
\]
Dijkstra's Algorithm in Pseudocode

- **Fringe**: $F$ is a priority queue/min-heap
- **arrays**: $d$ stores attachment (backward) costs, $\pi[v]$ stores parents
- **$S$** not really needed, only for clarity below

```
Dijkstra(G, s, w)
1: $F \leftarrow s$, $S \leftarrow \emptyset$
2: $d[v] \leftarrow \infty$ for all $v \in V$
3: $d[s] \leftarrow 0$
4: while $F \neq \emptyset$ do
5: $u \leftarrow \text{Extract-Min}(F)$
6: $S \leftarrow S \cup \{u\}$
7: for each $v \in \text{Adj}(u)$ do
8: $F \leftarrow v$
9: Relax($u$, $v$, $w$)
```

```
Relax($u$, $v$, $w$)
1: if $d[v] > d[u] + w(u, v)$ then
2: $d[v] \leftarrow d[u] + w(u, v)$
3: $\pi[v] \leftarrow u$
```

- The process of relaxing tests whether one can improve the shortest-path estimate $d[v]$ by going through the vertex $u$ in the shortest path from $s$ to $v$
- If $d[u] + w(u, v) < d[v]$, then $u$ replaces the predecessor of $v$
- Where would you put an earlier termination to stop when $s \rightsquigarrow g$ found?
Dijkstra's Algorithm in Pseudocode

- **Fringe**: F is a priority queue/min-heap
- **Arrays**: $d$ stores attachment (backward) costs, $\pi[v]$ stores parents
- $S$ not really needed, only for clarity below

$$\text{Dijkstra}(G, s, w)$$

1. $F \leftarrow s, S \leftarrow \emptyset$
2. $d[v] \leftarrow \infty$ for all $v \in V$
3. $d[s] \leftarrow 0$
4. **while** $F \neq \emptyset$ **do**
5. $u \leftarrow \text{Extract-Min}(F)$
6. $S \leftarrow S \cup \{u\}$
7. **for each** $v \in \text{Adj}(u)$ **do**
8. $F \leftarrow v$
9. Relax($u, v, w$)

**Relax**($u, v, w$)

1. **if** $d[v] > d[u] + w(u, v)$ **then**
2. $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

in another implementation, F is initialized with all V, and line 8 is removed.

- The process of relaxing tests whether one can improve the shortest-path estimate $d[v]$ by going through the vertex $u$ in the shortest path from $s$ to $v$
- If $d[u] + w(u, v) < d[v]$, then $u$ replaces the predecessor of $v$
- Where would you put an earlier termination to stop when $s \leadsto g$ found?
Dijsktra’s Algorithm in Action

Figure: Graph $G = (V, E)$

Figure: Shortest paths from $B$

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Initial $d$</th>
<th>Initial $\pi$</th>
<th>Pass1 $d$</th>
<th>Pass1 $\pi$</th>
<th>Pass2 $d$</th>
<th>Pass2 $\pi$</th>
<th>Pass3 $d$</th>
<th>Pass3 $\pi$</th>
<th>Pass4 $d$</th>
<th>Pass4 $\pi$</th>
<th>Pass5 $d$</th>
<th>Pass5 $\pi$</th>
<th>Pass6 $d$</th>
<th>Pass6 $\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\infty$</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\infty$</td>
<td>5</td>
<td>B</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>11</td>
<td>$D$</td>
<td>9</td>
<td>$E$</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm in Action

Figure: Graph $G = (V, E)$

Figure: Shortest paths from $B$

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Initial</th>
<th>Pass1</th>
<th>Pass2</th>
<th>Pass3</th>
<th>Pass4</th>
<th>Pass5</th>
<th>Pass6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\infty$</td>
<td>3</td>
<td>$B$</td>
<td>3</td>
<td>$B$</td>
<td>3</td>
<td>$B$</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>−</td>
<td>0</td>
<td>−</td>
<td>0</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>$\infty$</td>
<td>5</td>
<td>$B$</td>
<td>4</td>
<td>$A$</td>
<td>4</td>
<td>$A$</td>
</tr>
<tr>
<td>D</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>6</td>
<td>$C$</td>
<td>6</td>
<td>$C$</td>
</tr>
<tr>
<td>E</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>8</td>
<td>$C$</td>
<td>8</td>
<td>$C$</td>
</tr>
<tr>
<td>F</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>11</td>
<td>$D$</td>
<td>9</td>
</tr>
</tbody>
</table>

If not earlier goal termination criterion, Dijkstra’s search tree is spanning tree of shortest paths from $s$ to any vertex in the graph.
### Take-home Exercise

#### Informed Search

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Initial</th>
<th>Pass1</th>
<th>Pass2</th>
<th>Pass3</th>
<th>Pass4</th>
<th>Pass5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis of Dijkstra’s Algorithm

- Updating the heap takes at most $O(lg(|V|))$ time
- The number of updates equals the total number of edges
- So, the total running time is $O(|E| \cdot lg(|V|))$
- Running time can be improved depending on the actual implementation of the priority queue

$$\text{Time} = \theta(V) \cdot T(\text{Extract - Min}) + \theta(E) \cdot T(\text{Decrease - Key})$$

<table>
<thead>
<tr>
<th>$F$</th>
<th>$T(\text{Extr.-Min})$</th>
<th>$T(\text{Decr.-Key})$</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Binary heap</td>
<td>$O(1)$</td>
<td>$O(lg</td>
<td>V</td>
</tr>
<tr>
<td>Fib. heap</td>
<td>$O(lg</td>
<td>V</td>
<td>)$</td>
</tr>
</tbody>
</table>

How does this compare with BFS?
How does BFS get away from a $lg(|V|)$ factor?
Some Quotes

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture.

In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete??
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete?? Yes, if step cost $\geq \epsilon$
Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on
Complete?? Yes, if step cost $\geq \epsilon$

Time??
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on
Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+[C^*/\epsilon]})$
where $C^*$ is the cost of the optimal solution
Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space??
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+[C^*/\epsilon]})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+[C^*/\epsilon]})$
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete? Yes, if step cost $\geq \epsilon$

Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$

Optimal?
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$
   where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$

Optimal?? Yes, nodes expanded in increasing order of $g$
Uniform-cost Search

Lazier Dijkstra’s
Terminates when goal removed from $V - S$
Equivalent to BFS if step costs all equal

Let’s use $g$ for backward cost from now on

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lceil C^*/\epsilon \rceil})$

Optimal?? Yes, nodes expanded in increasing order of $g$
**Main Idea:** use an evaluation function $f$ for each vertex $v$
- may not use weights at all
→ Extract from fringe vertex $v$ with lowest $f[v]$

**Special Cases:**

- **Greedy best-first search:** $f[v] = h[v]$ (forward cost)
- **A* search:** $f[v] = g[v] + h[v]$ (backward + forward cost)

**Greedy-best first search:**
- Extracts from fringe (so, expands first) vertex that appears to be closest to goal
- cannot see weights has not seen, so uses heuristic to “estimate” cost of $v \leadsto g$
- Evaluation function, **forward cost** $h(v)$ (heuristic)
  = estimate of cost from $v$ to the closest goal
- E.g., $h_{SLD}(v) = $ straight-line distance from $v$ to Bucharest
<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobroeti</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadi</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Rmnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Greedy Best-first Search in Action

Arad
366
Greedy Best-first Search in Action
Greedy Best-first Search in Action

![Diagram of a search tree with cities and distances]

- Arad (366)
- Fagaras (176)
- Oradea (380)
- Rimnicu Vilcea (193)
- Sibiu
- Timisoara (329)
- Zerind (374)
Greedy Best-first Search in Action

Informed Search

Amarda Shehu (583)
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time $O(b^m)$, but a good heuristic can give dramatic improvement

Space $O(b^m)$—keeps all nodes in memory

Optimal No... plotting a trip on a map...
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement
Complete in finite space with repeated-state checking

**Time??** $O(b^m)$, but a good heuristic can give dramatic improvement

**Space??**
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time $\mathcal{O}(b^m)$, but a good heuristic can give dramatic improvement

Space $\mathcal{O}(b^m)$—keeps all nodes in memory
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal??
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time?? \( O(b^m) \), but a good heuristic can give dramatic improvement

Space?? \( O(b^m) \)—keeps all nodes in memory

Optimal?? No
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No ... plotting a trip on a map ...
Summary of Greedy Best-first Search

Complete in finite space with repeated-state checking

Time: $O(b^m)$, but a good heuristic can give dramatic improvement

Space: $O(b^m)$—keeps all nodes in memory

Optimal: No ... plotting a trip on a map ...
**A* Search**

**Idea:** avoid expanding paths that are already expensive

Evaluation function \( f(v) = g(v) + h(v) \):
Combines Dijkstra’s/uniform cost with greedy best-first search

- \( g(v) \) = (actual) cost to reach \( v \) from \( s \)
- \( h(v) \) = estimated lowest cost from \( v \) to goal
- \( f(v) \) = estimated lowest cost from \( s \) through \( v \) to goal

Same implementation as before, but prioritize vertices in min-heap by \( f[v] \)

A* is both complete and optimal provided \( h \) satisfies certain conditions:
  - for searching in a tree: admissible/optimistic
  - for searching in a graph: consistent (which implies admissibility)
Admissible Heuristic

What do we want from $f[v]$?

not to overestimate cost of path from source to goal that goes through $v$

Since $g[v]$ is actual cost from $s$ to $v$, this “do not overestimate” criterion is for the forward cost heuristic, $h[v]$

A* search uses an admissible/optimistic heuristic

i.e., $h(v) \leq h^*(v)$ where $h^*(v)$ is the true cost from $v$

(Also require $h(v) \geq 0$, so $h(G) = 0$ for any goal $G$)

Example of an admissible heuristic: $h_{SLD}(v)$ never overestimates the actual road distance
What do we want from $f[v]$?
   not to overestimate cost of path from source to goal that goes through $v$

Since $g[v]$ is actual cost from $s$ to $v$, this “do not overestimate” criterion is for the forward cost heuristic, $h[v]$

A* search uses an admissible/optimistic heuristic
i.e., $h(v) \leq h^*(v)$ where $h^*(v)$ is the true cost from $v$
(Also require $h(v) \geq 0$, so $h(G) = 0$ for any goal $G$)

Example of an admissible heuristic: $h_{SLD}(v)$ never overestimates the actual road distance

Let’s see A* with this heuristic in action
A* Search in Action

- Arad
- $366 = 0 + 366$
A* Search in Action

Informed Search

Sibiu: 393 = 140 + 253
Timisoara: 447 = 118 + 329
Zerind: 449 = 75 + 374
A* Search in Action

[Diagram showing a search tree with nodes labeled Arad, Sibiu, Timisoara, and Zerind. Edge weights are shown for each branch.]
A* Search in Action

![A* Search Tree Diagram]

- **Arad**
  - Sibiu
    - Arad
    - Fagaras
    - Oradea
  - Rimnicu Vilcea
    - Craiova
    - Pitesti
    - Sibiu
- **Timisoara**
  - 447 = 118 + 329
- **Zerind**
  - 449 = 75 + 374

Costs:
- Arad to Sibiu: 646 = 280 + 366
- Fagaras to Sibiu: 415 = 239 + 176
- Oradea to Sibiu: 671 = 291 + 380
- Rimnicu Vilcea to Sibiu: 526 = 366 + 160
- Rimnicu Vilcea to Pitesti: 417 = 317 + 100
- Rimnicu Vilcea to Sibiu: 553 = 300 + 253
A* Search in Action

Diagram showing the A* search process with distances between cities:

- Arad to Sibiu: 646 = 280 + 366
- Fagaras to Sibiu: 591 = 338 + 253
- Oradea to Sibiu: 450 = 450 + 0
- Timisoara: 447 = 118 + 329
- Zerind: 449 = 75 + 374
- Arad to Bucharest: 418 = 418 + 0
- Craiova: 615 = 455 + 160
- Pitesti: 607 = 414 + 193
- Sibiu to Pitesti: 553 = 300 + 253

Informed Search
Optimality of A*

- Tree-search version of A* is optimal if $h$ is admissible
does not overestimate lowest cost from a vertex to the goal

- Graph-search version additionally requires that $h$ be consistent
  estimated cost of reaching goal from a vertex $n$ is not greater than
cost to go from $n$ to its successors and then the cost from them to the goal

  Consistency is stronger, and it implies admissibility

Need to show:

- Lemma 1: If $h$ is consistent, then values of $f$ along any path are nondecreasing

- Lemma 2: If $h$ is admissible, whenever A* selects a vertex $v$ for expansion (extracts from fringe), optimal path to $v$ has been found (where else we have proved this?)
Tree-search version of A* is optimal if \( h \) is admissible
does not overestimate lowest cost from a vertex to the goal

Graph-search version additionally requires that \( h \) be consistent
estimated cost of reaching goal from a vertex \( n \) is not greater than cost to
go from \( n \) to its successors and then the cost from them to the goal

Consistency is stronger, and it implies admissibility

Need to show:

- Lemma 1: If \( h \) is consistent, then values of \( f \) along any path are nondecreasing

- Lemma 2: If \( h \) is admissible, whenever A* selects a vertex \( v \) for expansion (extracts
  from fringe), optimal path to \( v \) has been found (where else we have proved this?)
Proof of Lemma 1: Consistency $\rightarrow$ Nondecreasing $f$ along a Path

A heuristic is **consistent** if:

$$h(n) \leq c(n, a, n') + h(n')$$

If $h$ is consistent, we have

$$f(n') = g(n') + h(n')$$
$$= g(n) + c(n, a, n') + h(n')$$
$$\geq g(n) + h(n)$$
$$= f(n)$$

I.e., $f(n)$ is nondecreasing along any path.
Proof of Lemma 2: Consistency \( \rightarrow \) Admissibility

\[ h(n): \text{does not overestimate cost of lowest-cost path from } n \text{ to } g \]

\[ h(n) \leq \delta(n, g) \]
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

$$h(n) \leq \delta(n, g)$$

... on the other hand

$$h(n) \leq c(n, a, n') + h(n')$$

Why?

... and

$$h(n') \leq \delta(n', g)$$

Why?

... so

$$h(n) \leq c(n, a, n') + \delta(n', g)$$

for all successors $n'$ of $n$.

... what does the above mean?

... what else do you need so that you put the two and two together?

... how does $c(n, a, n') + \delta(n', g)$ relate to $\delta(n, g)$ when you consider $\forall n'$ of $n$?

Practically done - mull it over at home...
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

$h(n) \leq \delta(n, g)$

... on the other hand

\[ h(n) \leq c(n, a, n') + h(n') \]

Why?
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$
$h(n) \leq \delta(n, g)$

... on the other hand
$h(n) \leq c(n, a, n') + h(n')$  \hspace{1cm} \text{Why?}

... and
$h(n') \leq \delta(n', g)$
Proof of Lemma 2: Consistency → Admissibility

\( h(n) \): does not overestimate cost of lowest-cost path from \( n \) to \( g \)
\[
h(n) \leq \delta(n, g)
\]

... on the other hand
\[
h(n) \leq c(n, a, n') + h(n')
\]

... and
\[
h(n') \leq \delta(n', g)
\]
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

\[ h(n) \leq \delta(n, g) \]

... on the other hand

\[ h(n) \leq c(n, a, n') + h(n') \]

Why?

... and

\[ h(n') \leq \delta(n', g) \]

Why?

... so

\[ h(n) \leq c(n, a, n') + \delta(n', g) \]

for all successors $n'$ of $n$
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

\[ h(n) \leq \delta(n, g) \]

... on the other hand

\[ h(n) \leq c(n, a, n') + h(n') \]

Why?

... and

\[ h(n') \leq \delta(n', g) \]

Why?

... so

\[ h(n) \leq c(n, a, n') + \delta(n', g) \]

for all successors $n'$ of $n$

... what does the above mean?
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

$$h(n) \leq \delta(n, g)$$

... on the other hand

$$h(n) \leq c(n, a, n') + h(n')$$  \hspace{2cm} \text{Why?}

... and

$$h(n') \leq \delta(n', g)$$  \hspace{2cm} \text{Why?}

... so

$$h(n) \leq c(n, a, n') + \delta(n', g)$$  \hspace{2cm} \text{for all successors } n' \text{ of } n

... what does the above mean?

... what else do you need so that you put the two and two together?
Proof of Lemma 2: Consistency \(\rightarrow\) Admissibility

\[ h(n) \text{: does not overestimate cost of lowest-cost path from } n \text{ to } g \]
\[ h(n) \leq \delta(n, g) \]

... on the other hand
\[ h(n) \leq c(n, a, n') + h(n') \]

... and
\[ h(n') \leq \delta(n', g) \]

... so
\[ h(n) \leq c(n, a, n') + \delta(n', g) \]

... what does the above mean?
... what else do you need so that you put the two and two together?
... how does \( c(n, a, n') + \delta(n', g) \) relate to \( \delta(n, g) \) when you consider \( \forall n' \) of \( n \)?
Proof of Lemma 2: Consistency → Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

\[ h(n) \leq \delta(n, g) \]

... on the other hand

\[ h(n) \leq c(n, a, n') + h(n') \quad \text{Why?} \]

... and

\[ h(n') \leq \delta(n', g) \quad \text{Why?} \]

... so

\[ h(n) \leq c(n, a, n') + \delta(n', g) \quad \text{for all successors $n'$ of $n$} \]

... what does the above mean?

... what else do you need so that you put the two and two together?

... how does $c(n, a, n') + \delta(n', g)$ relate to $\delta(n, g)$ when you consider $\forall n'$ of $n$?

Practically done - mull it over at home...
Proof of Lemma 2: Consistency $\rightarrow$ Admissibility

$h(n)$: does not overestimate cost of lowest-cost path from $n$ to $g$

\[ h(n) \leq \delta(n, g) \]

... on the other hand

\[ h(n) \leq c(n, a, n') + h(n') \]

Why?

... and

\[ h(n') \leq \delta(n', g) \]

Why?

... so

\[ h(n) \leq c(n, a, n') + \delta(n', g) \]

for all successors $n'$ of $n$

... what does the above mean?

... what else do you need so that you put the two and two together?

... how does $c(n, a, n') + \delta(n', g)$ relate to $\delta(n, g)$ when you consider $\forall n'$ of $n$?

Practically done - mull it over at home...
Corollary from consistency: A* expands nodes in order of increasing $f$ value.

Gradually adds “$f$-contours” of nodes (cf. breadth-first adds layers).
Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$

So, why does this guarantee optimality?
First time we see goal will be the time it has lowest $f = g$ (h is 0)
Other occurrences have no lower $f$ ($f$ non-decreasing)
Why do I need Consistency on Graphs?

- Consistency needed when searching over a graph
- Admissibility only when searching over a tree
- Why?

- What can graphs have that trees do not have?
  - Redundant connectivity
  - ... and Cycles!!!
Why do I need Consistency on Graphs?

- Consistency needed when searching over a graph
- Admissibility only when searching over a tree
- Why?

- What can graphs have that trees do not have?
  Redundant connectivity
  ... and Cycles!!!

- Does consistency allow negative-weight edges?
Why do I need Consistency on Graphs?

- Consistency needed when searching over a graph
- Admissibility only when searching over a tree
- Why?

- What can graphs have that trees do not have? 
  Redundant connectivity
  ... and Cycles!!!

- Does consistency allow negative-weight edges?

  - Big deal with edges of negative weight!
    Lower f values along a path
    Cannot guarantee optimality
    Negative-weight cycles make f arbitrarily small
Why do I need Consistency on Graphs?

- Consistency needed when searching over a graph
- Admissibility only when searching over a tree
- Why?

- What can graphs have that trees do not have?
  - Redundant connectivity
  - ... and Cycles!!!

- Does consistency allow negative-weight edges?

- Big deal with edges of negative weight!
  - Lower f values along a path
  - Cannot guarantee optimality
  - Negative-weight cycles make f arbitrarily small

- What do we do when we have negative-weight edges and cycles?
  - Cannot use best-first/greedy paradigm anymore, need Dynamic Programming
Why do I need Consistency on Graphs?

- Consistency needed when searching over a graph
- Admissibility only when searching over a tree
- Why?

- What can graphs have that trees do not have?
  Redundant connectivity
  ... and Cycles!!!

- Does consistency allow negative-weight edges?

- Big deal with edges of negative weight!
  Lower f values along a path
  Cannot guarantee optimality
  Negative-weight cycles make f arbitrarily small

- What do we do when we have negative-weight edges and cycles?
  Cannot use best-first/greedy paradigm anymore, need Dynamic Programming
Complete??

Time

Exponential in \[\text{path length} \times \delta(s,g) - h(s)\]

Space

Keeps all generated nodes in memory (worse drawback than time)

Optimal

Yes—cannot expand \(f_i + 1\) until \(f_i\) is finished

Optimally efficient for any given consistent heuristic: A* expands all nodes with \(f(v) < \delta(s,g)\)

A* expands some nodes with \(f(v) = \delta(s,g)\)

A* expands no nodes with \(f(v) > \delta(s,g)\)
Yes, unless there are infinitely many nodes with $f \leq f(G)$
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with \( f \leq f(G) \)

Time??

Keep all generated nodes in memory (worse drawback than time)
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $[\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]$
Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $[\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]$ 

Space??

Summary of A* Search
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $[\text{path length} \times \frac{\delta(s, g) - h(s)}{\delta(s, g)}]$.

Space?? Keeps all generated nodes in memory (worse drawback than time).
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $\left[ \text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)} \right]$ 

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand $f_i + 1$ until $f_i$ is finished

Optimally efficient for any given consistent heuristic: A* expands all nodes with $f(v) < \delta(s, g)$

A* expands some nodes with $f(v) = \delta(s, g)$

A* expands no nodes with $f(v) > \delta(s, g)$
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with \( f \leq f(G) \)

Time?? Exponential in \([\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]\)

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand \( f_{i+1} \) until \( f_i \) is finished
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with \( f \leq f(G) \)

Time?? Exponential in \([\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]\)

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand \( f_{i+1} \) until \( f_i \) is finished

Optimally efficient for any given consistent heuristic:
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with \( f \leq f(G) \)

Time?? Exponential in \([\text{path length } \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]\)\)

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand \( f_{i+1} \) until \( f_i \) is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with \( f(v) < \delta(s,g) \)
Summary of A* Search

---

**Complete?** Yes, unless there are infinitely many nodes with $f \leq f(G)$

**Time?** Exponential in $[\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]$ 

**Space?** Keeps all generated nodes in memory (worse drawback than time)

**Optimal?** Yes—cannot expand $f_{i+1}$ until $f_i$ is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s, g)$
A* expands some nodes with $f(v) = \delta(s, g)$
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $[\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]$}

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand $f_{i+1}$ until $f_i$ is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s, g)$
A* expands some nodes with $f(v) = \delta(s, g)$
A* expands no nodes with $f(v) > \delta(s, g)$
Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in $[\text{path length} \times \frac{\delta(s,g) - h(s)}{\delta(s,g)}]$ 

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand $f_{i+1}$ until $f_i$ is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$
A* expands some nodes with $f(v) = \delta(s,g)$
A* expands no nodes with $f(v) > \delta(s,g)$
Admissible Heuristics

E.g., for the 8-puzzle:
\[ h_1(v) = \text{number of misplaced tiles} \]
\[ h_2(v) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \text{blank} \\
\end{array}
\]

Start State \quad Goal State

\[ h_1(S) = ?? \]
Admissible Heuristics

E.g., for the 8-puzzle:

- $h_1(v) = \text{number of misplaced tiles}$
- $h_2(v) = \text{total Manhattan distance}$

(i.e., no. of squares from desired location of each tile)

Start State

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1
\end{array}
\]

Goal State

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

\[
h_1(S) = 6
\]
Admissible Heuristics

E.g., for the 8-puzzle:

\[ h_1(v) = \text{number of misplaced tiles} \]

\[ h_2(v) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\]

Start State

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Goal State

\[ h_1(S) = ?? \]

\[ h_2(S) = ?? \]

Amarda Shehu (583)
Admissible Heuristics

E.g., for the 8-puzzle:

\( h_1(v) = \text{number of misplaced tiles} \)

\( h_2(v) = \text{total Manhattan distance} \)

(i.e., no. of squares from desired location of each tile)

\[ \begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array} \quad \begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array} \]

Start State  
Goal State

\[
\begin{align*}
\overline{h_1(S)} &= 6 \\
\overline{h_2(S)} &= 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14
\end{align*}
\]

start with tile 1, 2, and so on, not counting the blank tile
Admissible Heuristics

E.g., for the 8-puzzle:

\[ h_1(v) = \text{number of misplaced tiles} \]
\[ h_2(v) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Start State  Goal State

\[
h_1(S) = ?? \quad 6
\]
\[
h_2(S) = ?? \quad 4+0+3+3+1+0+2+1 = 14
\]

start with tile 1, 2, and so on, not counting the blank tile
If $h_2(v) \geq h_1(v)$ for all $v$ (both admissible)
then $h_2$ dominates $h_1$ and is better for search

Typical search costs:
\[ \begin{align*}
    d = 14 & \quad \text{IDS} = 3,473,941 \text{ nodes} \\
                & \quad A^*(h_1) = 539 \text{ nodes} \\
                & \quad A^*(h_2) = 113 \text{ nodes} \\
    d = 24 & \quad \text{IDS} \approx 54,000,000,000 \text{ nodes} \\
                & \quad A^*(h_1) = 39,135 \text{ nodes} \\
                & \quad A^*(h_2) = 1,641 \text{ nodes}
\end{align*} \]

Given any admissible heuristics $h_a, h_b,$

\[ h(v) = \max(h_a(v), h_b(v)) \]

is also admissible and dominates $h_a, h_b$
Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(v)$ gives the shortest solution.

If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(v)$ gives the shortest solution.

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.
Relaxed Problems

Well-known example: travelling salesperson problem (TSP)
Find the shortest tour visiting all cities exactly once

Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour
Proposed by Berliner in 1979 as a Best-first search algorithm.

Instead of single point-valued estimates, B* uses intervals for nodes of the tree.

Leaves can be searched until one of the top level nodes has an interval which is clearly “best.”

**Intervals backup**: a parent’s upper bound is set to the maximum of the upper bounds of the children. A parent’s lower bound is set to the maximum of the lower bound of the children. Note that different children might supply these bounds.

Applied to two-player deterministic zero-sum games. Palay applied to chess. B* implemented in Scrabble program.

Optimality depends on interval evaluations.
Main Idea behind D* Search

- Very popular in robot path/motion planning.
- Follows similar template as tree search algorithms
- Initiated at goal rather than start node
- In this way, each expanded node knows its exact cost to the goal, not an estimate
- Uses current and minimum cost
- Terminates when start node is to be expanded
- Variants have been proposed
  - focused D*: uses heuristic for expansion
  - D* Lite: nothing to do with D*, combines ideas of A* and Dynamic SSF-FP
Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest $h$
– incomplete and not always optimal

A* search expands lowest $g + h$
– complete and optimal
– also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems
Additionally consider scenarios where greedy substructure does not lead to optimality.

For instance, how can one modify Dijkstra and the other algorithms to deal with negative weights?

How does one efficiently find all pairwise shortest/least-cost paths?

**Dynamic Programming** is the right alternative in these scenarios.