Lecture: Informed (Heuristic) Search [and Admissible Heuristics]

CS 483 (001) - Spring 2017

Amarda Shehu

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

Spring 2017

1. Outline of Today's Class

2. Reflections/Insights on Uninformed Search

3. Informed Search
   - Best-first Search
   - A* Search
   - Informed Search Summary

Insights: Reflection on Uninformed Search

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 $V - S$
- What do we actually store in the fringe?

Implication: similar template $\rightarrow$ reusable code

Data structure F for the fringe: order vertices are extracted from $V - S$ distinguishes search algorithms from one another

- DFS: Take edge from vertex discovered most recently (F is a stack)
- BFS: Take edge from vertex discovered least recently (F is a queue)

- What does order affect? Completeness or optimality?
- 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 state-space graph can we use that we have not used so far?

(Discrete) Informed Search Algorithms

Find a least-cost/shortest path from initial vertex to goal vertex

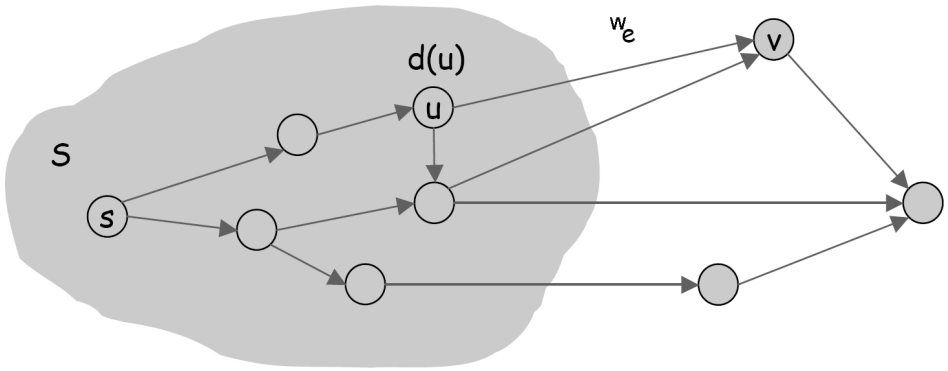
- Make use of costs/weights in state-space graph
- **Informed** graph search algorithms:
  - Dijkstra's Search [Edsger Dijkstra 1959]
  - Uniform-cost Search (a variant of Dijkstra's)
  - Best-First Search [Judea Pearl 1984]
  - A* Search [Petter Hart, Nils Nilsson, Bertram Raphael 1968]
  - B* Search [Hans Berliner 1979]
  - D* Search [Stenz 1994]
  - More variants of the above
- What we will <u>not</u> 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]

Finding Shortest Paths in Weighted Graphs

- 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 $s \rightsquigarrow u$ 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 $s \rightsquigarrow g$?

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v

Essence of All Informed 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 frontier vertices, prioritized by $d[v]$

Essence of All Informed 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 frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time

Essence of All Informed 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 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?

Essence of All Informed 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 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

Essence of All Informed 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 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

Essence of All Informed 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 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:

Essence of All Informed 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 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

Essence of All Informed 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 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

Essence of All Informed 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 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:

Essence of All Informed 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 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 $s \rightsquigarrow v$)
  - forward cost (estimate of cost of $v \rightsquigarrow g$)
  - back+for ward cost (estimate of $s \rightsquigarrow g$ through v)
- Which do I choose? This is how to you end up with different search algorithms

Essence of All Informed 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 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 $s \rightsquigarrow v$)
  - forward cost (estimate of cost of $v \rightsquigarrow g$)
  - back+for ward cost (estimate of $s \rightsquigarrow g$ through v)
- Which do I choose? This is how to you end up with different search algorithms

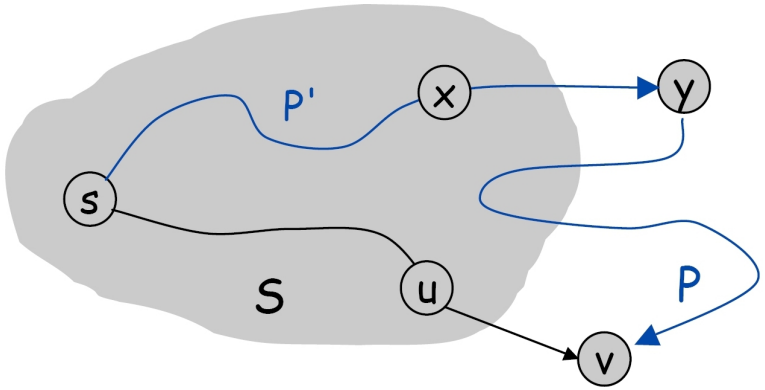
Dijkstra extracts vertices from fringe (adds to S) in order of their backward costs

Claim: When a vertex v is extracted from fringe F (thus "added" to S), the shortest path from s to v has been found. $\leftarrow$ invariant

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?

$$
\begin{aligned}
w(P) &\geq w(P') + w(x,y) && \text{nonnegative weights} \\
&\geq d[x] + w(x,y) && \text{inductive hypothesis} \\
&\geq d[y] && \text{definition of } d[y] \\
&\geq d[v] && \text{Dijkstra chose } v \text{ over } y
\end{aligned}
$$



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)

```
F ← s, S ← ∅
d[v] ← ∞ for all v ∈ V
d[s] ← 0
while F ≠ ∅ do
    u ← Extract-Min(F)
    S ← S ∪ {u}
    for each v ∈ Adj(u) do
        F ← v
        Relax(u, v, w)
```

Relax(u, v, w)

```
if d[v] > d[u] + w(u, v) then
    d[v] ← d[u] + w(u, v)
    π[v] ← 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

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: $\quad u \leftarrow$ Extract-Min(F)
6: $\quad S \leftarrow S \cup \{u\}$
7: $\quad$ **for** each $v \in \mathrm{Adj(u)}$ **do**
8: $\quad\quad F \leftarrow v$
9: $\quad\quad$ Relax(u, v, w)

Relax(u, v, w)

1: **if** $d[v] > d[u] + w(u, v)$ **then**
2: $\quad d[v] \leftarrow d[u] + w(u, v)$
3: $\quad \pi[\mathrm{v}] \leftarrow \mathrm{u}$

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 $s \rightsquigarrow g$ found?

Dijsktra's Algorithm in Action

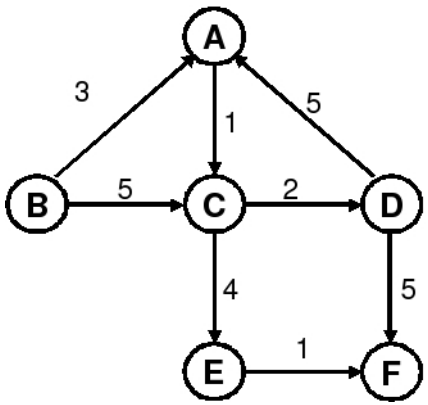


Figure: Graph $G = (V, E)$

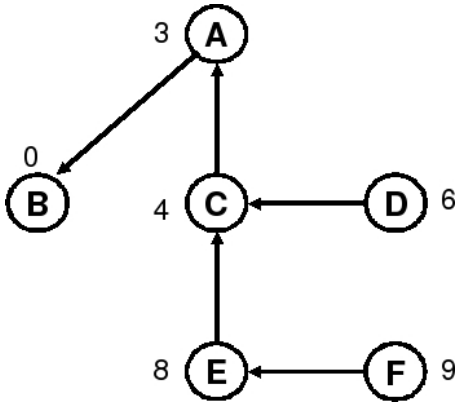


Figure: Shortest paths from B

	Initial		Pass1		Pass2		Pass3		Pass4		Pass5		Pass6	
Vertex	d	π	d	π	d	π	d	π	d	π	d	π	d	π
A	∞		3	B	3	B	3	B	3	B	3	B	3	B
B	0	–	0	–	0	–	0	–	0	–	0	–	0	–
C	∞		5	B	4	A	4	A	4	A	4	A	4	A
D	∞		∞		∞		6	C	6	C	6	C	6	C
E	∞		∞		∞		8	C	8	C	8	C	8	C
F	∞		∞		∞		∞		11	D	9	E	9	E

Dijsktra's Algorithm in Action

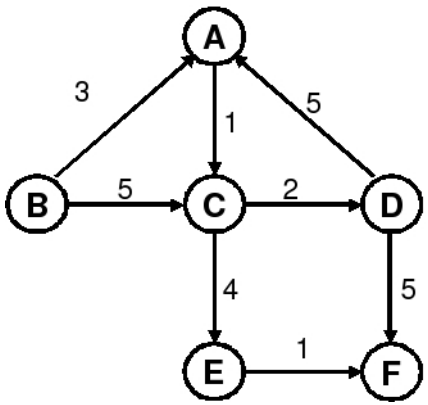


Figure: Graph $G = (V, E)$

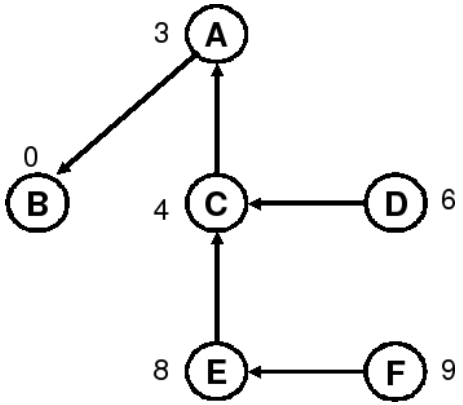
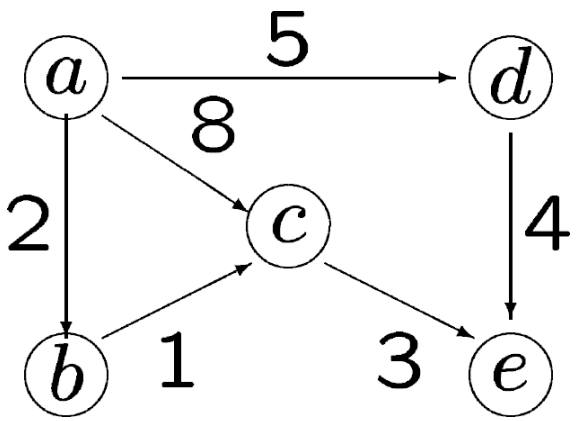


Figure: Shortest paths from B

	Initial		Pass1		Pass2		Pass3		Pass4		Pass5		Pass6	
Vertex	d	π	d	π	d	π	d	π	d	π	d	π	d	π
A	∞		3	B	3	B	3	B	3	B	3	B	3	B
B	0	–	0	–	0	–	0	–	0	–	0	–	0	–
C	∞		5	B	4	A	4	A	4	A	4	A	4	A
D	∞		∞		∞		6	C	6	C	6	C	6	C
E	∞		∞		∞		8	C	8	C	8	C	8	C
F	∞		∞		∞		∞		11	D	9	E	9	E

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



	Initial		Pass1		Pass2		Pass3		Pass4		Pass5	
Vertex	d	π	d	π	d	π	d	π	d	π	d	π
a	0	–										
b	∞											
c	∞											
d	∞											
e	∞											

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

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

F	T(Extr.-Min)	T(Decr.-Key)	Total
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(1)$	$O(lg\|V\|)$	$O(\|E\| \cdot lg\|V\|)$
Fib. heap	$O(lg\|V\|)$	$O(1)$	$O(\|E\| + \|V\| \cdot lg\|V\|)$

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

Short Interlude

Edsger Dijkstra: 1930-2002

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.

Best-first Search

Main Idea: use an evaluation function f for each vertex v
– may not use weights at all
$\rightarrow$ 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 $v \rightsquigarrow g$
- Evaluation function, **forward cost** $h(v)$ (**h**euristic)
  $=$ estimate of cost from v to the closest goal
- E.g., $h_{\mathrm{SLD}}(v) =$ straight-line distance from v to Bucharest

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??

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

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??

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

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: $h_{\mathrm{SLD}}(v)$ never overestimates the actual road distance

Admissible Heuristic

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

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

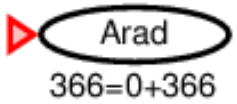
Since $g[v]$ is actual cost from s to v, this "do not overestimate" criterion is for the forward cost heuristic, $h[v]$

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: $h_{\mathrm{SLD}}(v)$ never overestimates the actual road distance

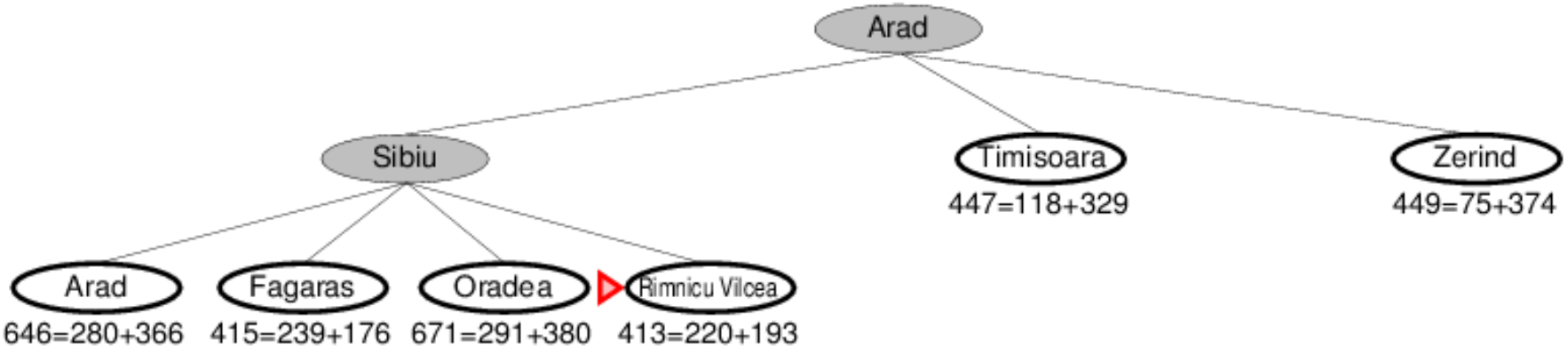
Let's see A* with this heuristic in action

A* Search in Action

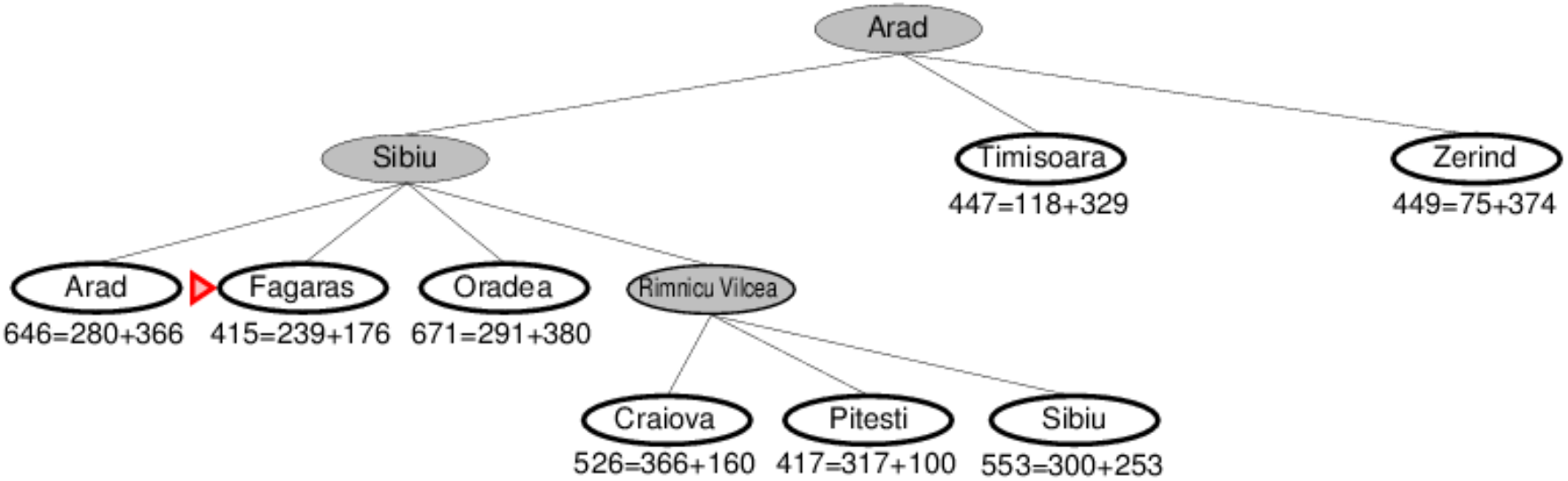


A* Search in Action

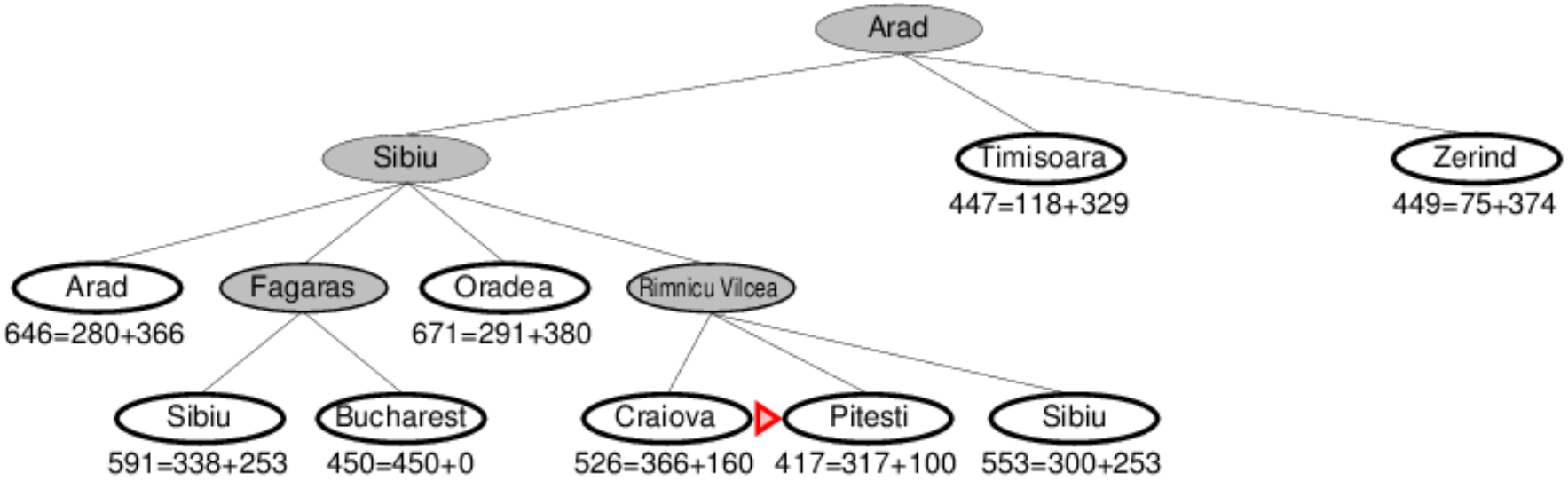
A* Search in Action



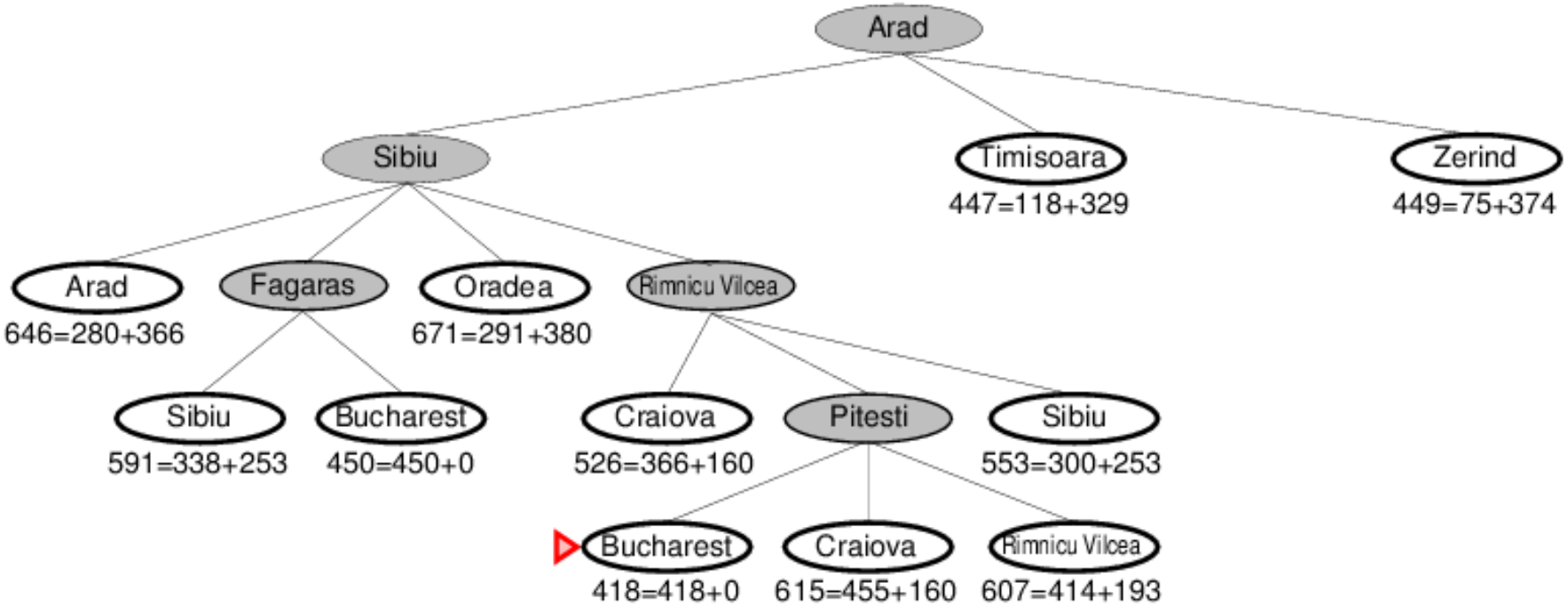
A* Search in Action



A* Search in Action



A* Search in Action



Analysis of A* - Advanced Material

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?)

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

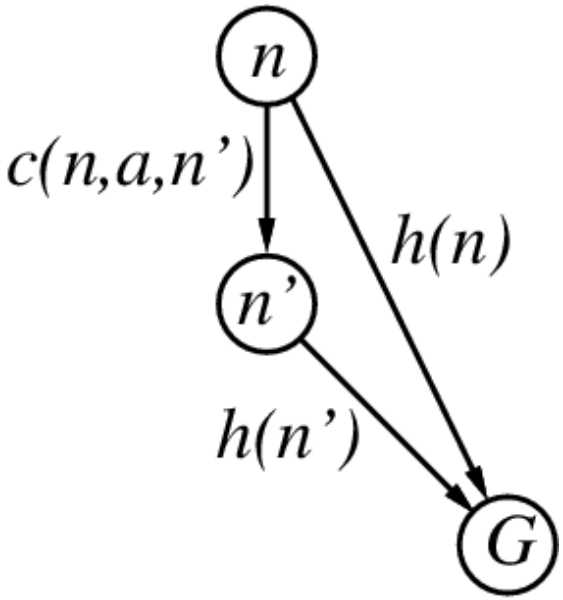
  Consistency is stronger, and it implies admissibility

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?)

Proof of Lemma1: Consistency → 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

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.

Proof of Lemma2: Consistency $\rightarrow$ Admissibility

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

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

Proof of Lemma2: 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

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

Proof of Lemma2: 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 Lemma2: 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

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

Proof of Lemma2: 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?

Proof of Lemma2: 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 Lemma2: 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 Lemma2: 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?

Proof of Lemma2: 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?

Proof of Lemma2: 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 Lemma2: 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

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

... and

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

... so

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

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

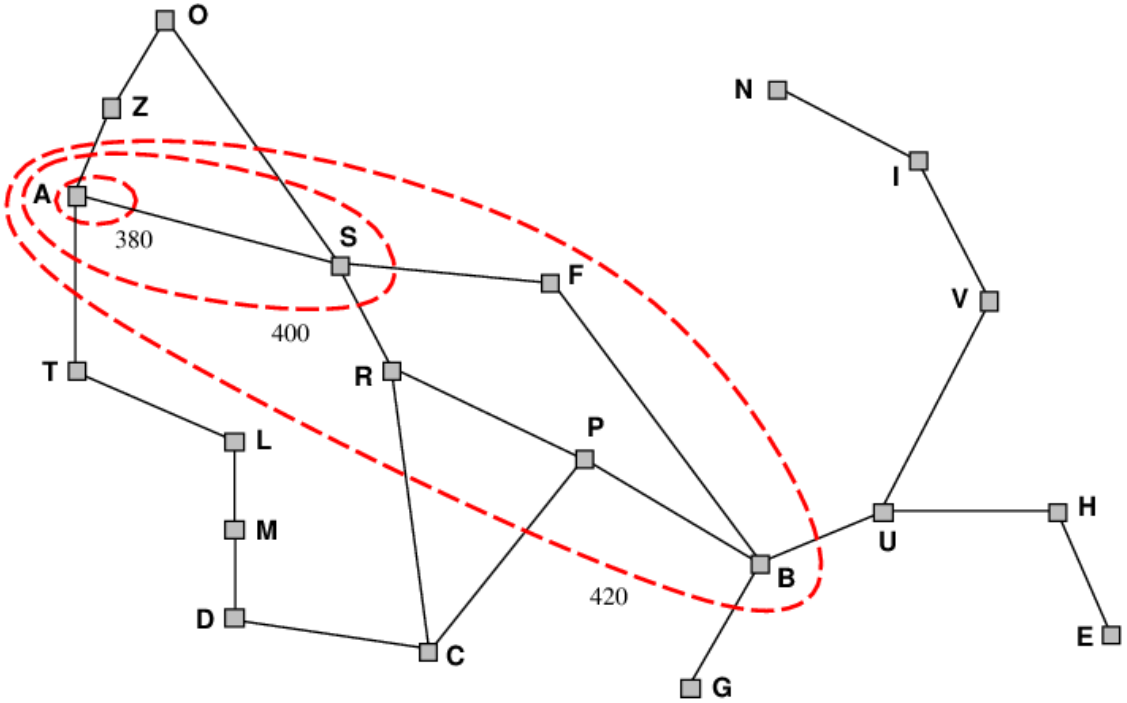
Practically done - mull it over at home...

Optimality of A*

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

Summary of A* Search

Complete??

Summary of A* Search

Complete?? 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??

Summary of A* Search

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

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

Summary of A* Search

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

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

Space??

Summary of A* Search

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

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

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

Optimal??

Summary of A* Search

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

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

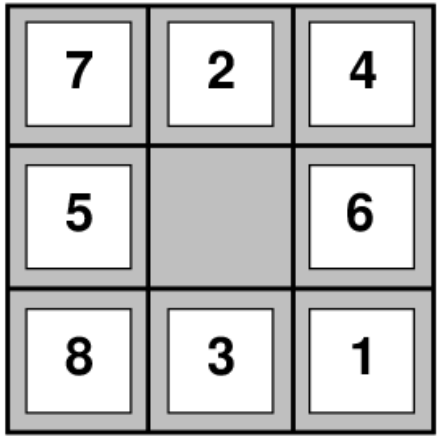
Space?? Keeps all generated nodes in memory (worse drawback than time)

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

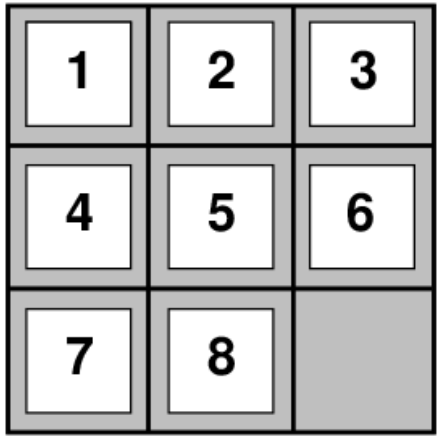
Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$
A* expands some nodes with $f(v) = \delta(s,g)$
A* expands no nodes with $f(v) > \delta(s,g)$

E.g., for the 8-puzzle:
$h_1(v)$ = number of misplaced tiles
$h_2(v)$ = total Manhattan distance

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



Start State

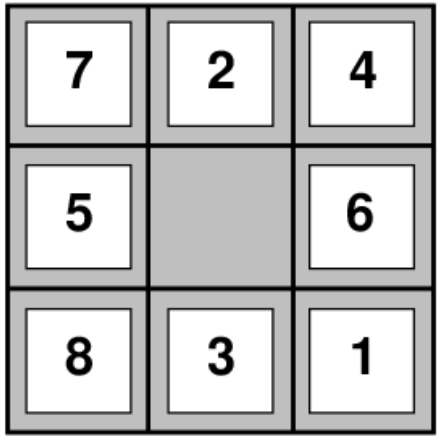


Goal State

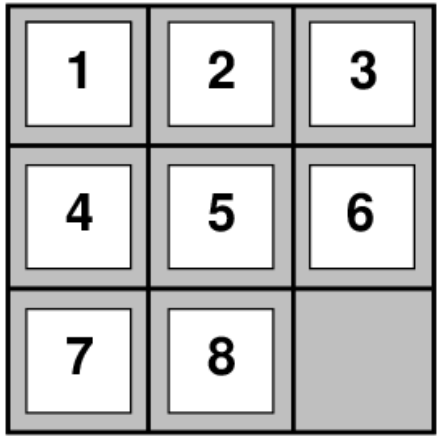
$h_1(S) = ??$

E.g., for the 8-puzzle:
$h_1(v)$ = number of misplaced tiles
$h_2(v)$ = total Manhattan distance

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



Start State

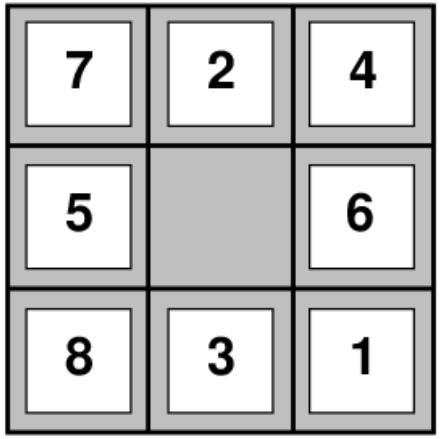


Goal State

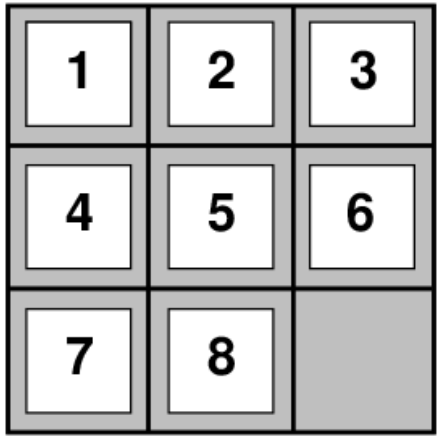
$h_1(S)$ =?? 6

E.g., for the 8-puzzle:
$h_1(v)$ = number of misplaced tiles
$h_2(v)$ = total Manhattan distance

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



Start State



Goal State

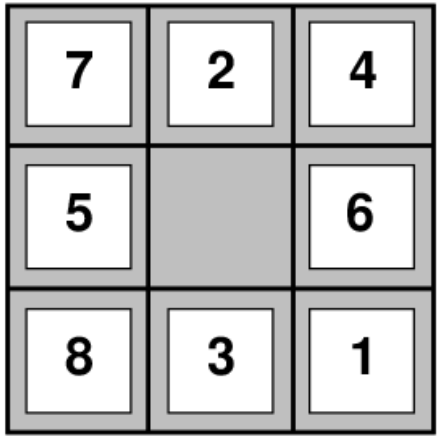
$h_1(S) =$?? 6

$h_2(S) =$??

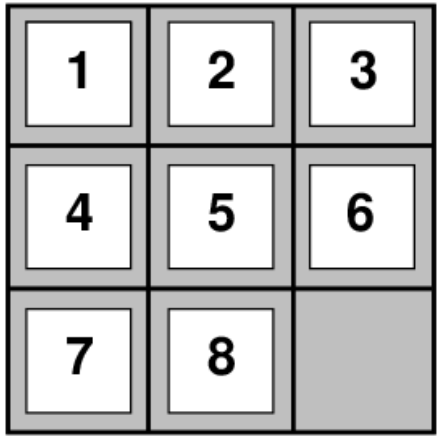
Admissible Heuristics

E.g., for the 8-puzzle:
$h_1(v)$ = number of misplaced tiles
$h_2(v)$ = total Manhattan distance

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



Start State



Goal State

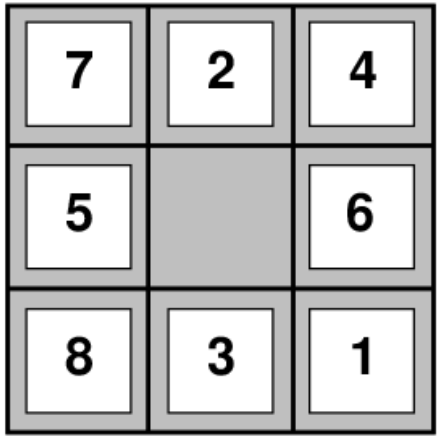
$h_1(S) =$?? 6

$h_2(S) =$?? 4+0+3+3+1+0+2+1 = 14

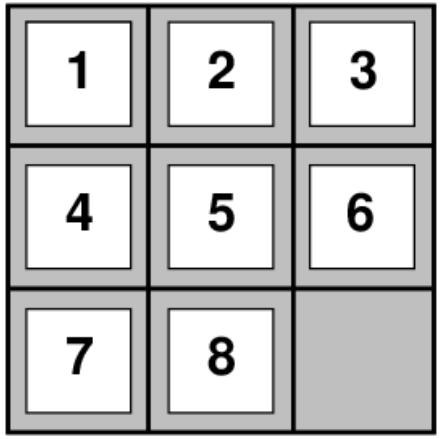
start with tile 1, 2, and so on, not counting the blank tile

E.g., for the 8-puzzle:
$h_1(v)$ = number of misplaced tiles
$h_2(v)$ = total Manhattan distance

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



Start State



Goal State

$h_1(S) =$?? 6

$h_2(S) =$?? 4+0+3+3+1+0+2+1 = 14

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

Summary

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

Greedy not Always Optimal

CS583 additionally considers 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

More graph exploration and search algorithms considered in CS583