Searching the State Space

Part 2: Pruning, informed search, other strategies

Lowest-cost-first search (LCFS)

- Sometimes there are costs associated with arcs. The **cost of a path** is the sum of the costs of its arcs.

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.

- The frontier is a priority queue ordered by path cost.

- LCFS finds an optimal solution: a least-cost path to a goal node.

- Another commonly-used name for this algorithm is uniform-cost search (which is somewhat misleading)

Priority queue refresher

1. A container in which each element has a priority (cost).

2. An element (path) with higher priority (lower cost) is always selected/removed/dequeued before an element with lower priority (higher cost).

3. We require the priority queue to be stable: if two or more elements have the same priority, elements that were enqueued earlier are dequeued earlier.

4. In Python you can use `heapq`. You need to store objects in a way that the above properties hold.

Priority queue example

On an empty frontier, after executing:

```
+ a, 5
+ b, 10
+ c, 5
+ d, 10
```

a sequence of selection/removals yields:

```
- a, 5
- c, 5
- b, 10
- d, 10
```

Example: tracing frontier in LCFS

Given the following graph

nodes={a, b, c, d, g},

edge_lists=[(a,b,4), (a,c,2), (a,d,1),
(b,g,4), (c,g,2), (d,g,4)],

starting_nodes = [a],

goal_nodes = {g}

trace the frontier in lowest-cost-first search (LCFS).

Answer:

```
+ a, 0
- a, 0
+ ab, 4
+ ac, 2
+ ad, 1
- ad, 1
+ adg, 5
- ac, 2
+ acg, 4
- ab, 4
+ abg, 8
- acg, 4
```

The problem with cycles and multiple paths

There are two issues that affect all search strategies in the framework of generic graph search:

1. Cycle: leads to an infinite search tree.

2. Expanding multiple paths to the same node: leads to cycles (the first issue) and also wasted computation dues to multiple branches going to the same node.

- The latter subsumes the former.

- Idea: let's "prune" unnecessary branches of the search tree.

Pruning

Principle: Do not expand paths to nodes to which we have already found a path.

- The frontier keeps track of **expanded** (aka "*closed*") nodes.

- When trying to add a new path to the frontier, it is added only if another path to the same end-node has not been already expanded, otherwise the path is discarded (pruned).

- When asking for the next path to be returned by the frontier, a path is selected and removed but it is returned only if the end-node has not been expanded before, otherwise the path is discarded (pruned) and not returned. The selection and removal is repeated until a path is returned (or the frontier becomes empty). If a path is returned, its end-node will be remembered as an expanded node.

In frontier traces, every time a path is pruned (when trying to add or when asking for the next path), we add an exclamation mark '**!**' at the end of the line.

Example: LCFS with pruning

Trace LCFS with pruning on the following graph:

```
nodes = {S, A, B, G},
edge_list=[(S,A,3), (S,B,1), (B,A,1), (A,B,1), (A,G,5)],
starting_nodes = [S],
goal_nodes = {G}.
```

Answer:

```
                # expanded={}
+ S,0
- S,0           # expanded={S}
+ SA,3
+ SB,1
- SB,1          # expanded={S,B}
+ SBA,2
- SBA,2         # expanded={S,B,A}
+ SBAB,3!         # not added!
+ SBAG,7
- SA,3!           # not returned!
- SBAG,7        # expanded={S,B,A,G}
```

How does LCFS behave?

LCFS explores increasing cost contours.

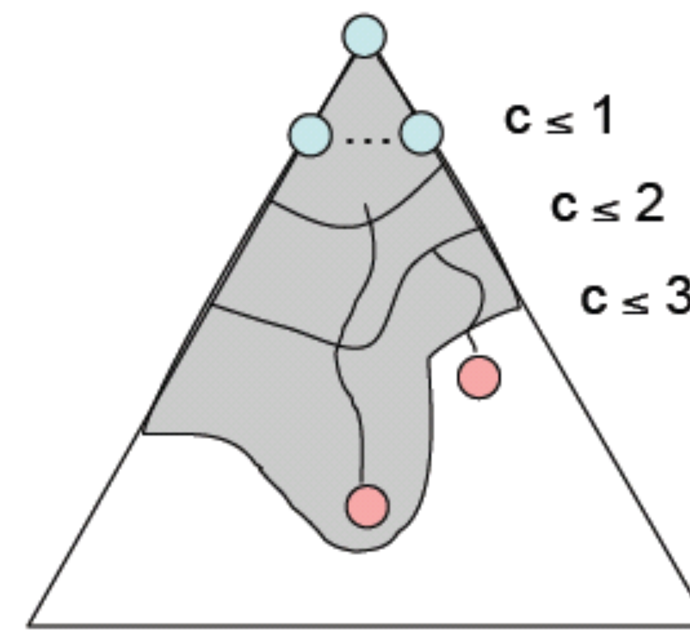
The good:

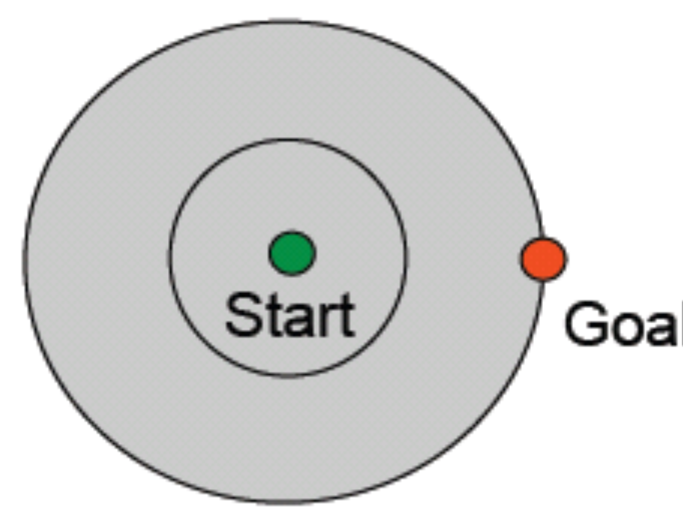
- Finds an optimal solution.

The bad:

- Explores options in every direction

- No information about goal location





Search heuristic

Idea: don't ignore the goal when selecting paths. Often there is extra knowledge that can be used to guide the search: **heuristics**.

h(n) is an estimate of the cost of the shortest path from node *n* to a goal node in an instance of a search problem.

h(n) needs to be efficient to compute.

h can be extended to paths: $\mathbf{h(\langle n_0,\ldots,n_k\rangle) = h(n_k)}$.

h(n) is an **underestimate** if there is no path from *n* to a goal node with cost less than *h(n)*. In other words *h(n)* is **less than or equal** to the actual cost of getting from *n* to a goal node.

An **admissible** heuristic is a nonnegative heuristic function that is an underestimate of the actual cost of a path to a goal node.

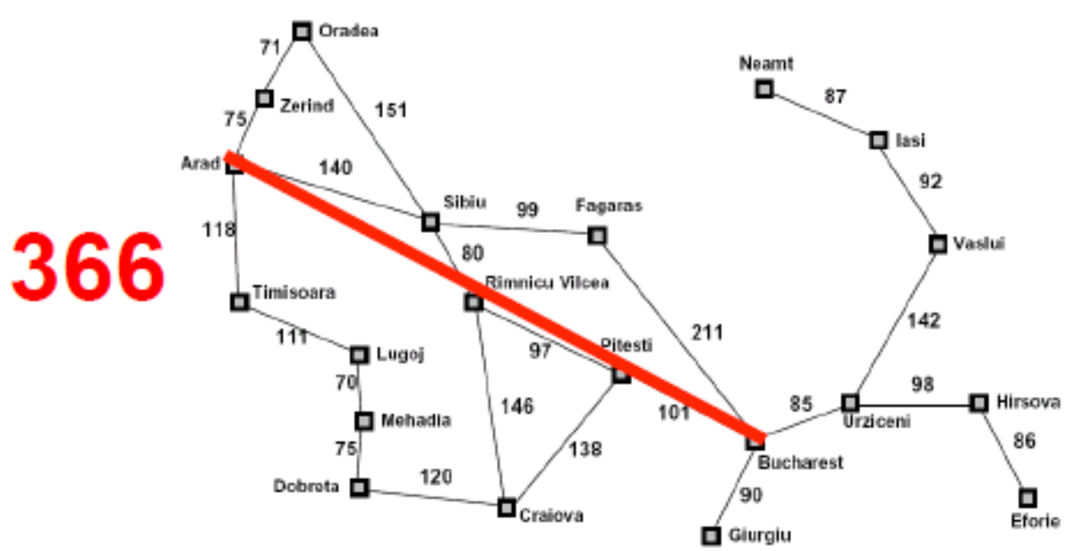
Example heuristic functions

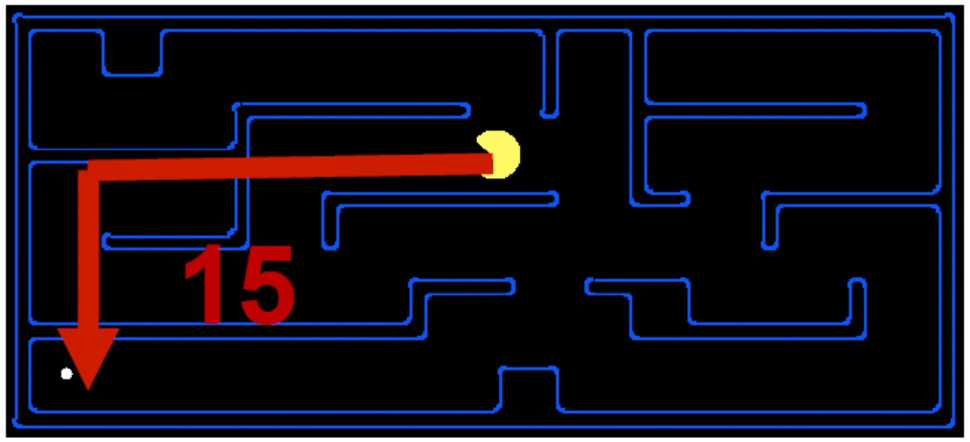
- If the nodes are points on a Euclidean plane and the cost is the distance, *h(n)* can be the straight-line distance from *n* to the closest goal.

- If the nodes are locations and cost is time, *h(n)* can be the distance to a goal divided by the maximum speed.

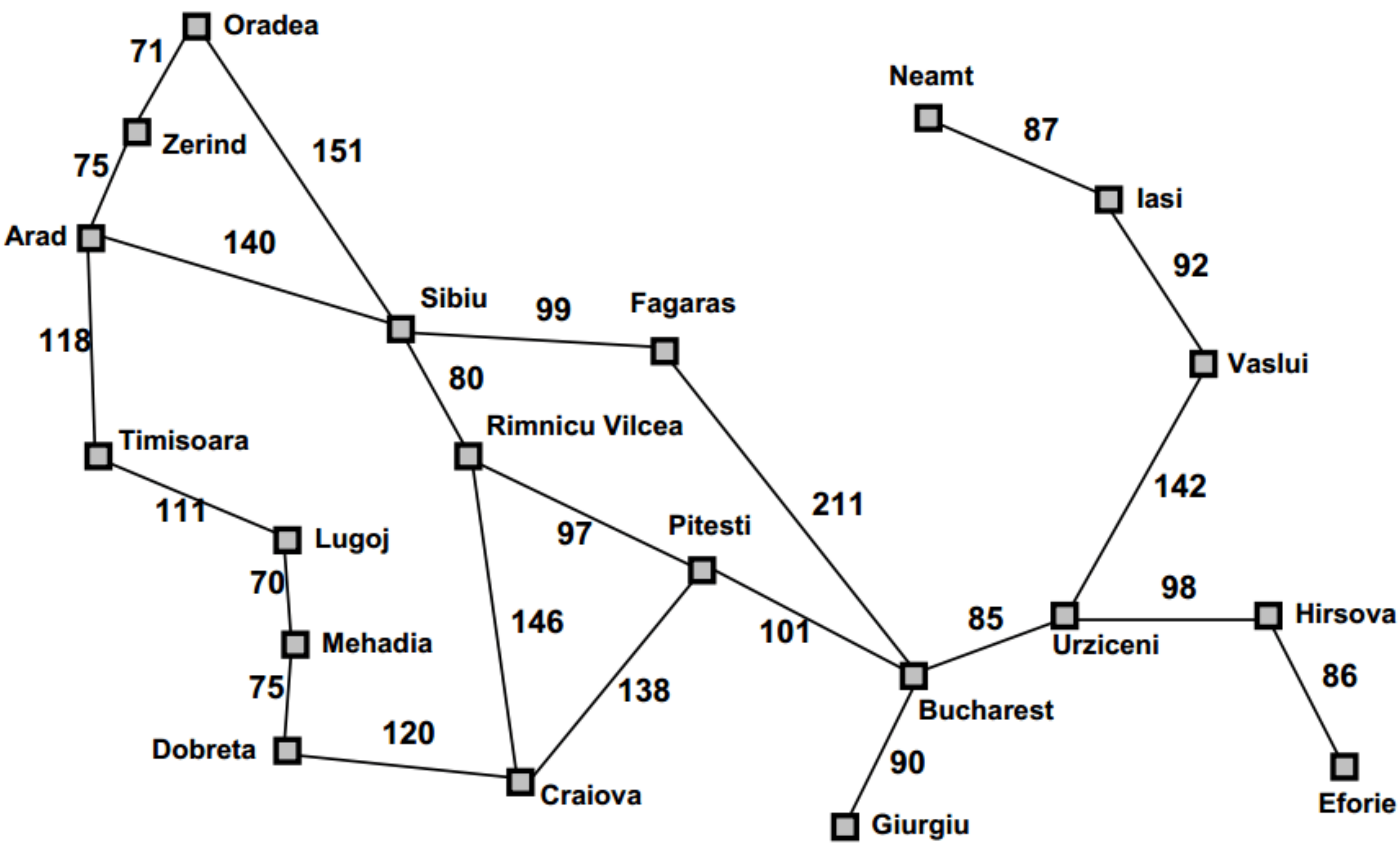
- If the nodes are locations in a maze where the agent can move in four directions, *h(n)* can be Manhattan distance.

- A heuristic function can be found by solving a simpler (less constrained) version of the problem.





Example: Euclidean distance



Straight–line distance
to Bucharest

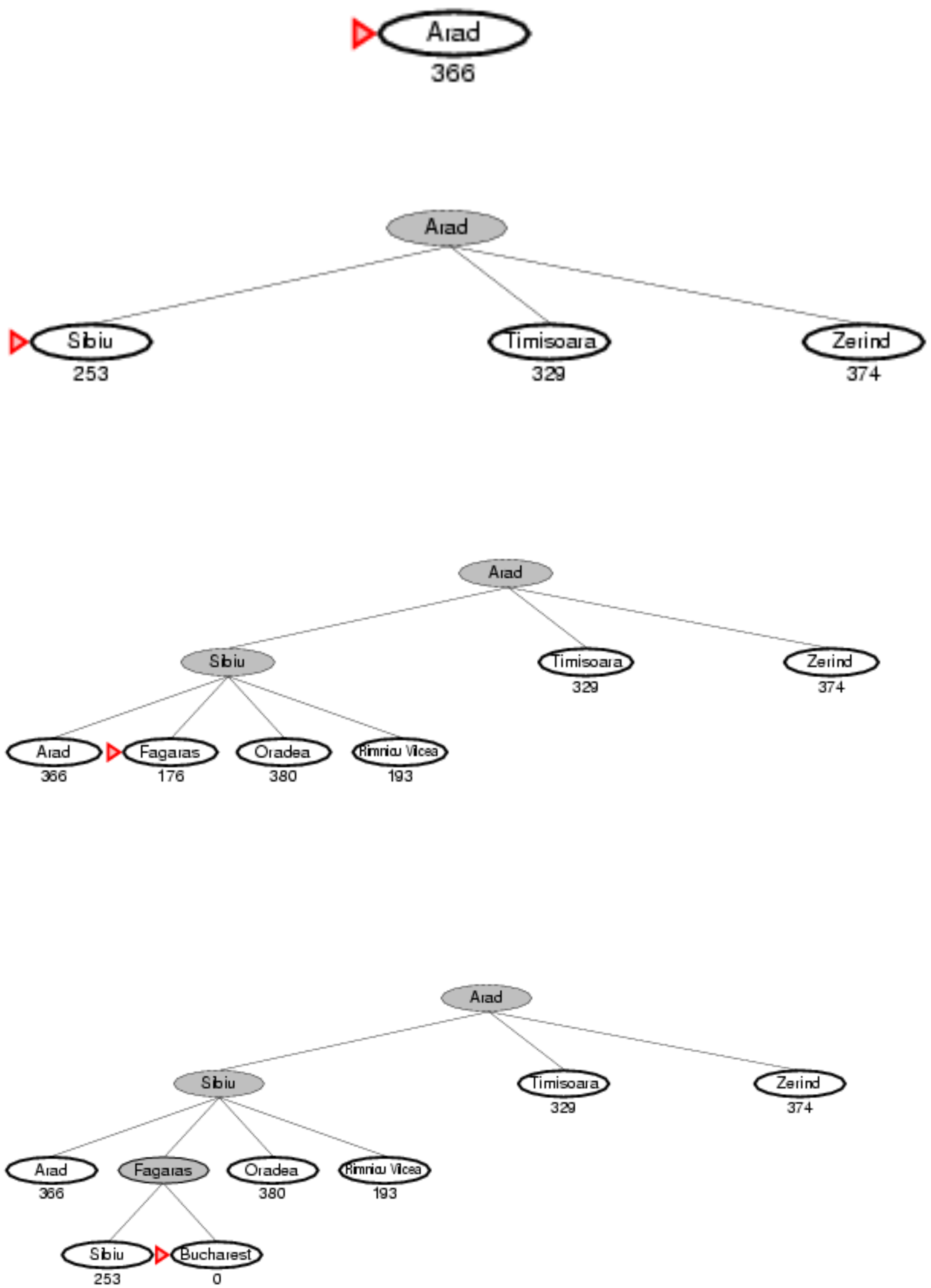
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-first Search

- Idea: select the path whose end is closest to a goal according to the heuristic function.
- Best-first search is a greedy strategy that selects a path on the frontier with minimal *h*-value.
- It treats the frontier as a priority queue ordered by *h*.
- By exploring more "promising" paths first, in many instances, it can find a solution faster than LCFS.
- Main drawback: does <u>not</u> guarantee finding an optimal solution.

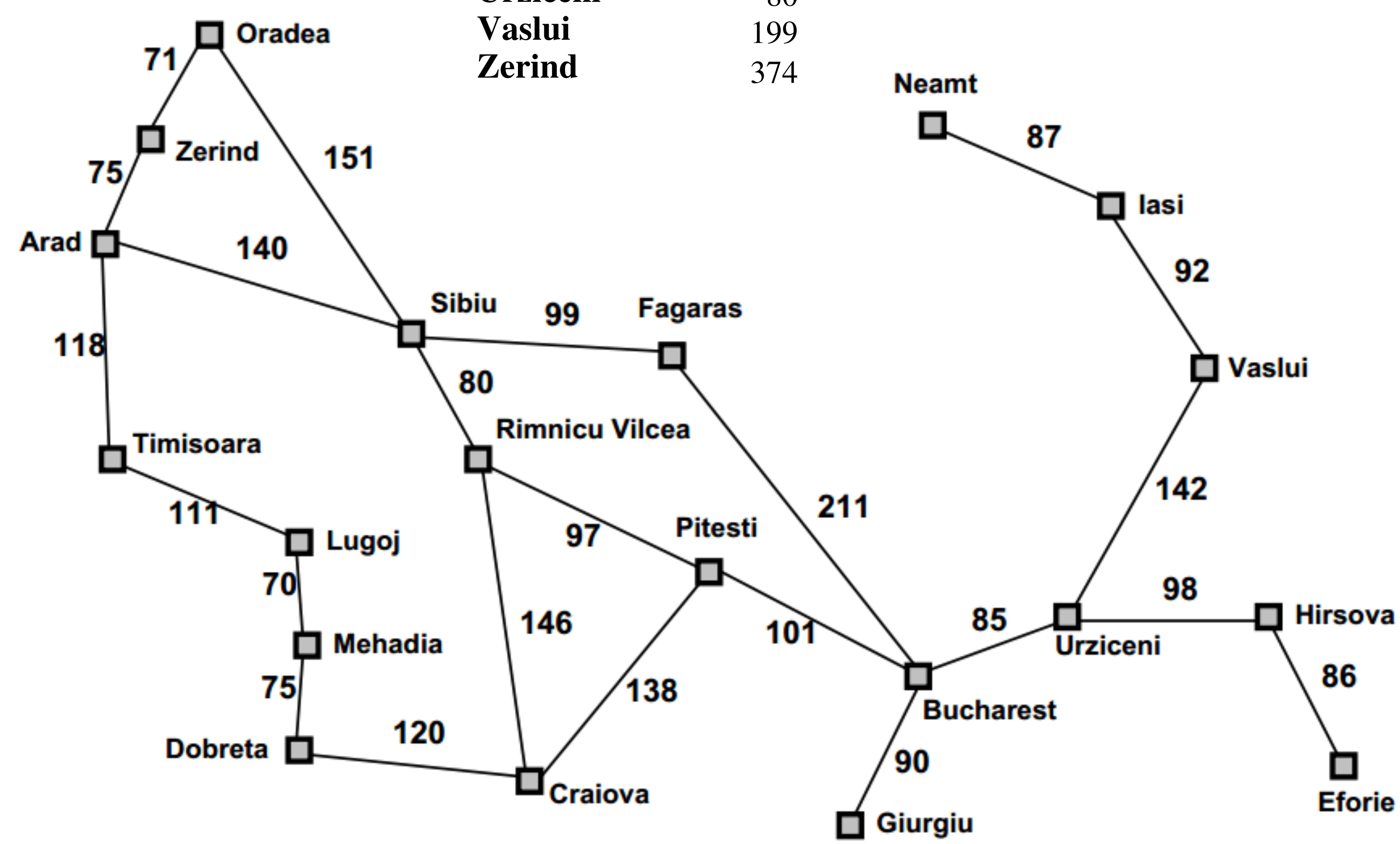
Best-first search: example

stages of search tree and frontier



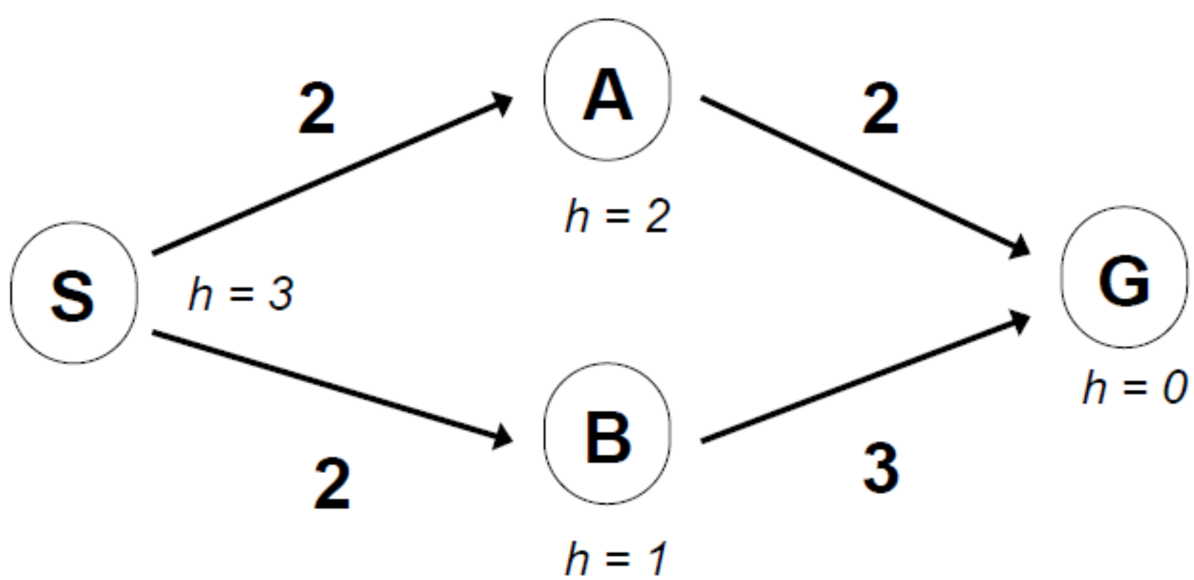
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Example: tracing best-first search

- Trace the frontier when using the best-first (greedy) search strategy for the following graph.
- The starting node is S and the goal node is G.
- Heuristic are given next to each node.
- SA comes before SB.



Answer:

```
+ S,3
- S,3
+ SA,2
+ SB,1
- SB,1
+ SBG,0
- SBG,0
```

A* search strategy

Idea:

- Don't be as wasteful as LCFS
- Don't be as greedy as best-first search.
- Avoid expanding paths that are already expensive.

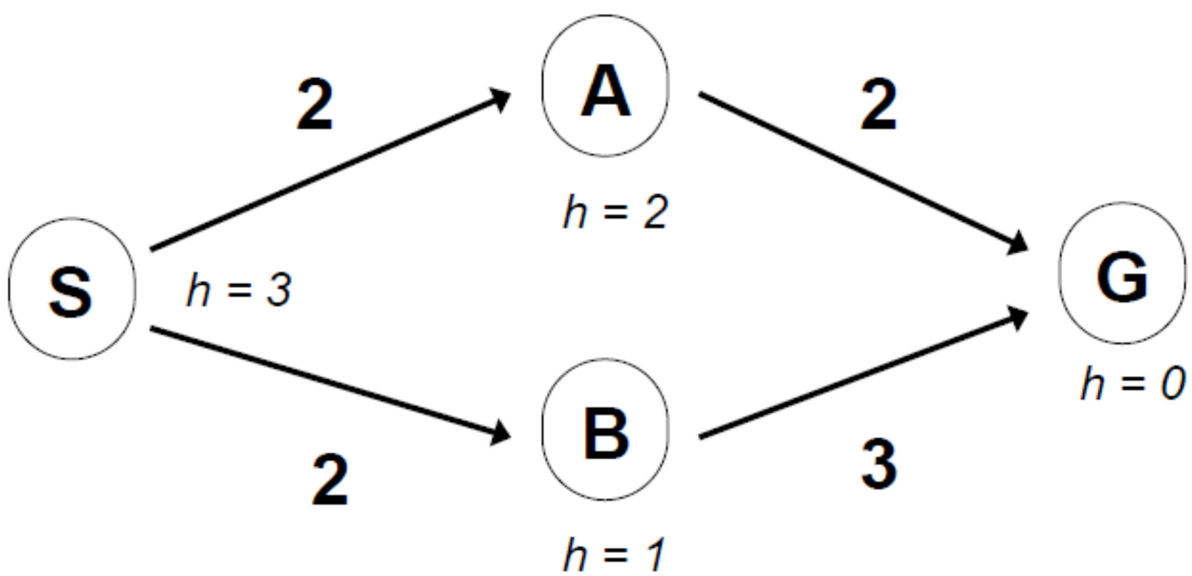
Evaluation function ***f(p) = cost(p) + h(n)***

- *p* is a path, *n* is the last node on *p*
- *cost(p)* = cost of path *p* (This is the real cost from the starting node to node *n*)
- *h(n)* = an estimate of cost from *n* goal (This is the estimated cost from n to the closest goal node)
- *f(p)* = estimated total cost of path through *p* to goal

The frontier is a priority queue ordered by *f(p)*.

Example: tracing A* search

- Trace the frontier when using the A* search strategy for the following graph.
- The starting node is S and the goal node is G.
- Heuristic are given next to each node.
- SA comes before SB.



Note: These small examples only show the inner working of A*. They do not demonstrate its advantage over LCFS.

heuristic function

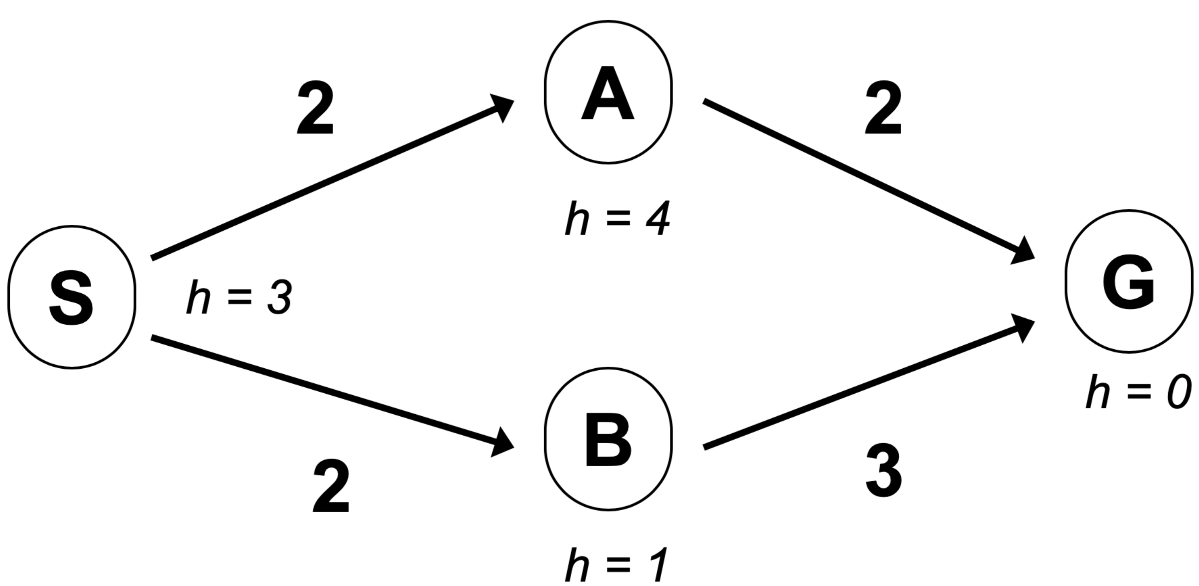
h(S) = 3
h(A) = 2
h(B) = 1
h(G) = 0

Answer:

```
+ S,3     # 0 + 3 = 3
- S,3
+ SA,4    # 2 + 2 = 4
+ SB,3    # 2 + 1 = 3
- SB,3
+ SBG,5   # 5 + 0 = 5
- SA,4
+ SAG,4   # 4 + 0 = 4
- SAG,4
```

Example: tracing A* search

- Same example as the one before just assume *h(A) = 4* instead.



Answer:

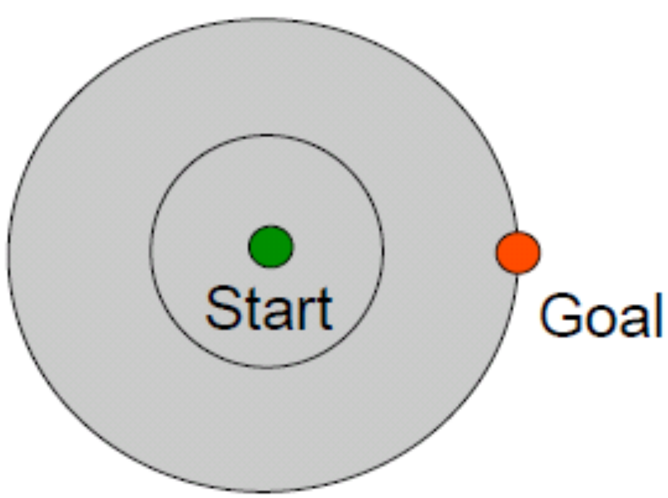
```
+ S,3     # 0 + 3 = 3
- S,3
+ SA,6    # 2 + 4 = 6
+ SB,3    # 2 + 1 = 3
- SB,3
+ SBG,5   # 5 + 0 = 5
- SBG,5
```

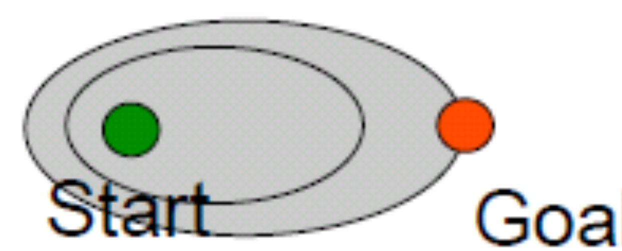
Non-optimal solution! Why?

Properties of A*

- A* always finds an optimal solution (a solution with the lowest costs) as long as:
  - there is a solution;
  - there is no pruning; and
  - the heuristic function is admissible.

- Does it halt on every graph?
- How about time and space complexity?
- LCFS vs A* (in average):





A*: proof of optimality

When using A* (without pruning) the first path *p* from a starting node to a goal node that is selected and removed from the frontier has the lowest cost.

Sketch of proof:

- Suppose to the contrary that there is another path from one of the starting nodes to a goal node with a lower cost.

- There must be a path *p*' on the frontier such that one of its continuations leads to the goal with a lower overall cost than *p*.

- Since *p* was removed before *p*':

$$f(p) \leq f(p') \implies cost(p) + h(p) \leq cost(p') + h(p') \implies cost(p) \leq cost(p') + h(p')$$

- Let *c* be a continuation of *p*' that goes to a goal node; that is, we have a path *p*'*c* from a start node to a goal node. Since *h* is admissible, we have:

$$cost(p'c) = cost(p') + cost(c) \geq cost(p') + h(p')$$

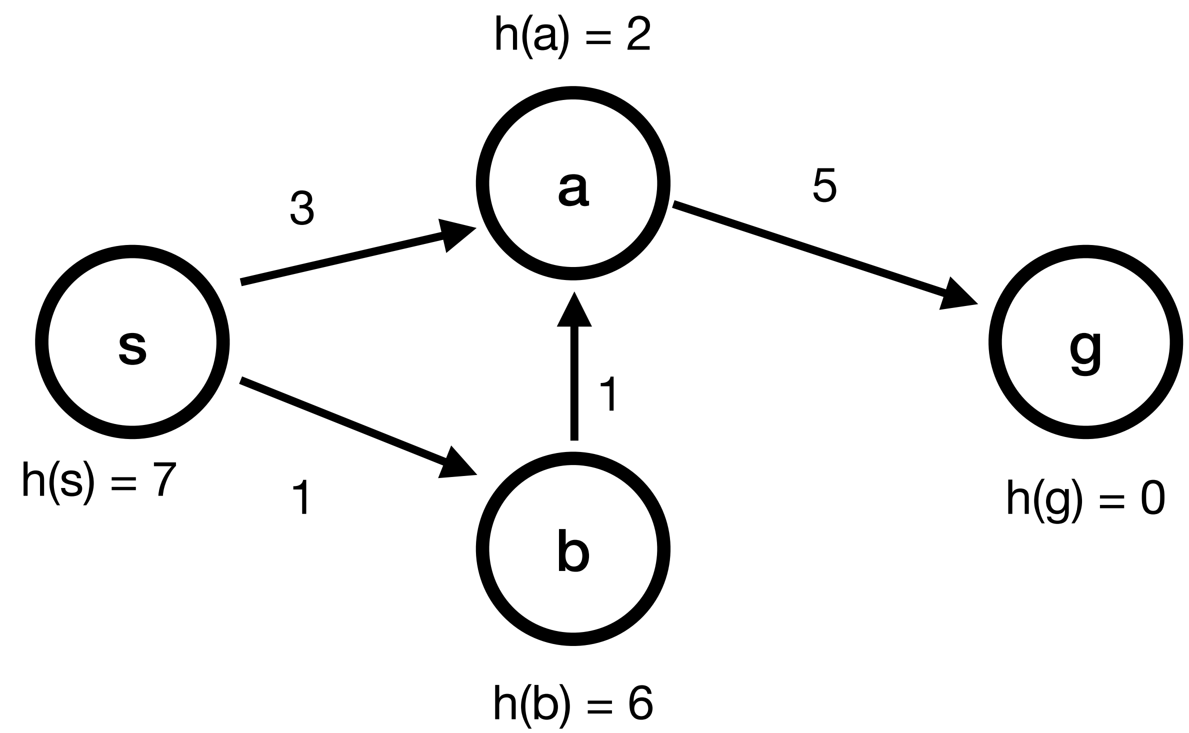
- Thus:

$$cost(p) \leq cost(p') + h(p') \leq cost(p') + cost(c) = cost(p'c)$$

Effect of pruning on A*

Trace the frontier in A* search for the following graph, with and without pruning.

```
nodes={s, a, b, g},
estimates = {s:7, a:2, b:6, g:0},
edge_list=[(s,a,3), (s,b,1),
           (b,a,1), (a,g,5)],
starting_nodes = [s],
goal_nodes = {g}.
```



Answer <u>without</u> pruning

```
+ S, 7
- S, 7
+ SA, 5
+ SB, 7
- SA, 5
+ SAG, 8
- SB, 7
+ SBA, 4
- SBA, 4
+ SBAG, 7
- SBAG, 7
```

Answer **with** pruning

```
# expanded={}
+ S, 7
- S, 7         # expanded={S}
+ SA, 5
+ SB, 7
- SA, 5        # expanded={S,A}
+ SAG, 8
- SB, 7        # expanded={S,A,B}
+ SBA, 4!
- SAG, 8
```

Non-optimal solution!

What went wrong?

- An expensive path, *sa*, was expanded before a cheaper one *sba* could be discovered because *f(sa)* < *f(sb)*.

- Is the heuristic function *h* admissible?

- So why?
  - *h(a)* is too low compared to *h(b)*, this makes *sa* look better. [or equivalently *h(b)* is relatively too high making *sb* look worse.]
  - we see that once, *sb* is expanded to *sba*, the f-value drops.

- So what can we do?
  - We need a stronger condition than admissibility to stop this from happening.

Monotonicity

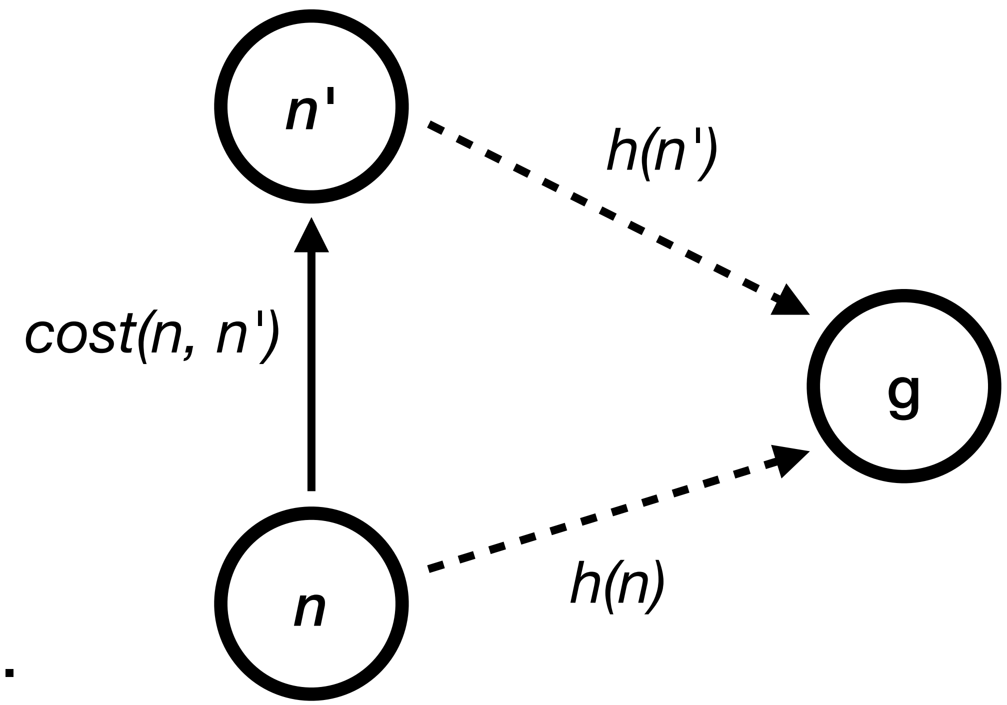
A heuristic function is **monotone** (or **consistent**) if for every two nodes *n*, and *n'* which is reachable from *n:*

$h(n) \leq cost(n,n') + h(n')$

With monotone restriction, we have:

$$
\begin{aligned}
f(n') &= cost(s,n') + h(n') \\
&= cost(s,n) + cost(n,n') + h(n') \\
&\geq cost(s,n) + h(n) \\
&= f(n)
\end{aligned}
$$

that is, *f(n)* is non-decreasing along any path.



Another interpretation for monotone restriction: real cost must always exceed reduction in heuristic!

Monotonicity is stronger condition than admissibility.

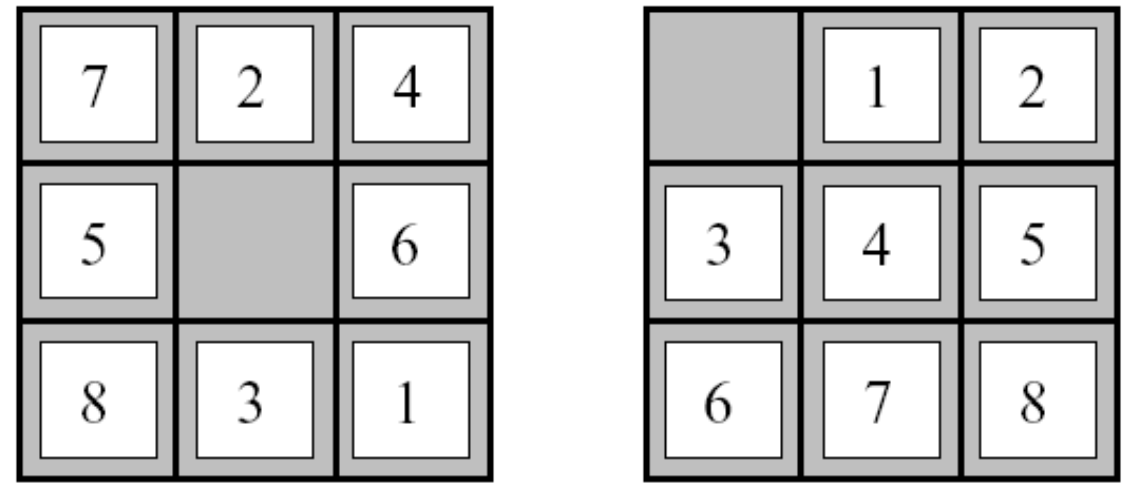
If *h* meets the monotone requirement, A* using multiple-path pruning yields optimal solutions.

Finding good heuristics

- Most of the work is in coming up with admissible heuristics.
- A common approach is to solve a less constrained (simpler) version of the problem.
- Good news: usually admissible heuristics are also consistent.
- Even inadmissible heuristics are often quite effective if we are OK with sacrificing optimality to some degree (or when we have no choice).
- In fact a known hack is to use $a * h(n)$ where h is admissible and $a > 1$ (i.e. make an inadmissible heuristic) in order to save more time (but lose optimality).
- [In COSC367 programming exercises we do not use inadmissible heuristics.]

Example heuristic in 8-puzzle

- Number of tiles misplaced?
- Why is it admissible?
- h(start) = 8

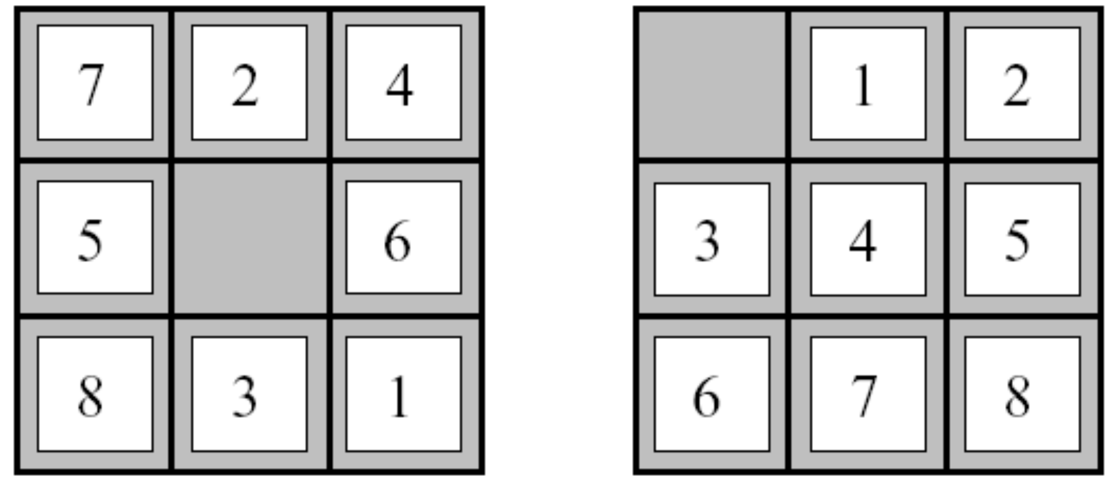


Start State Goal State

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
LCFS	112	6,300	3.6×10^6
A* - TILES	13	39	227

Example heuristic in 8-puzzle (cont'd)

- What if we had an easier 8-puzzle where any tile could slide any one direction at any time?
- Total *Manhattan* distance
- Why admissible?
- h(start) = 3 + 1 + 2 + … = 18



Start State Goal State

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
A* - TILES	13	39	227
A* - MAN-HATTAN	12	25	73

Best heuristic?

How about using the actual cost as a heuristic?

- Would it be a valid heuristic?
- Would we save on nodes expanded?
- What’s wrong with it?

Choosing a heuristic: a trade-off between quality of estimate and work per node!

Dominance relation

- Dominance: $h_a \geq h_c$ if

$$\forall n : h_a(n) \geq h_c(n)$$

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible

$$h(n) = max(h_a(n), h_b(n))$$

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic