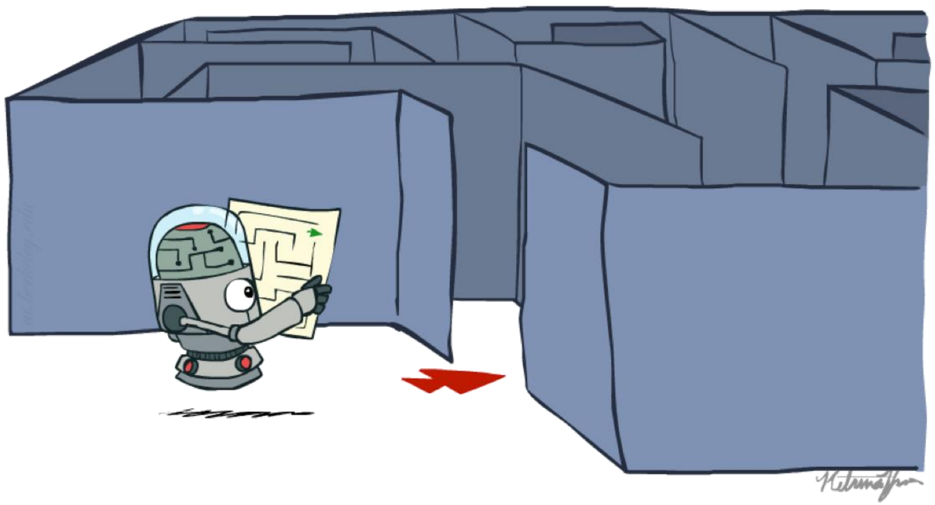
Part 1:
Solving Problems with Search

[Aknowledgment: Some Slides adapted from Dan Klein and Pieter Abbeel]

http://ai.berkeley.edu.]

Tesla on **autopilot** crashes into *parked* Fire truck at 65mph

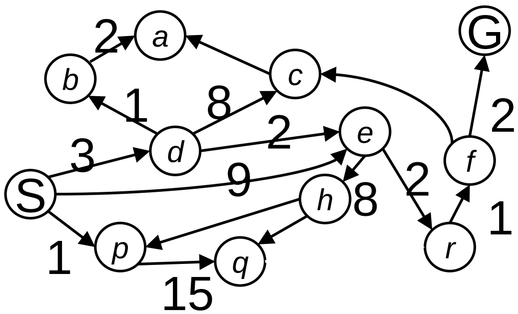
Search, continued



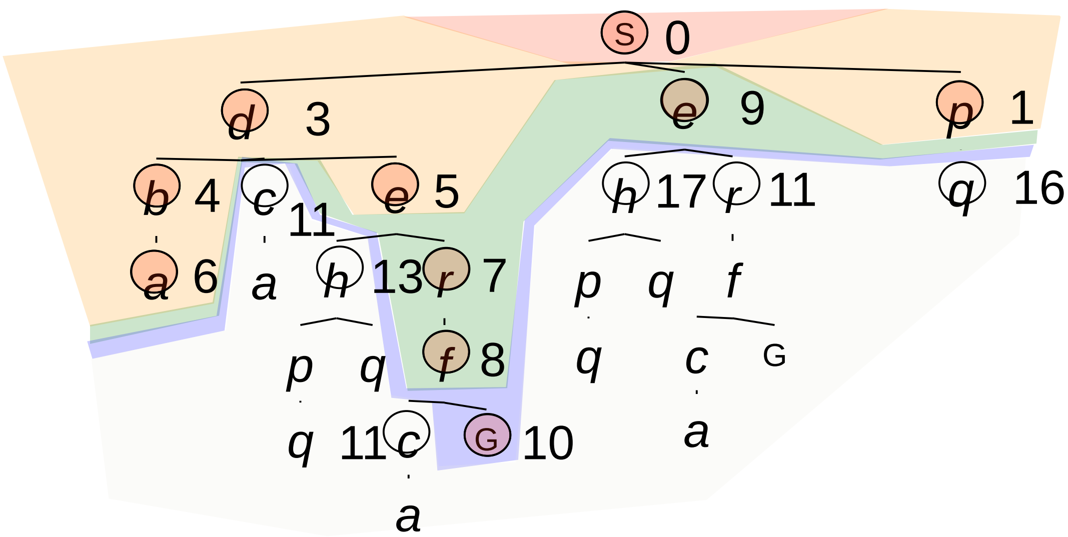
Uniform Cost Search (Review)

Strategy: ***expand a cheapest node first:***

Fringe is ***a priority queue*** *(priority: cumulative cost)*



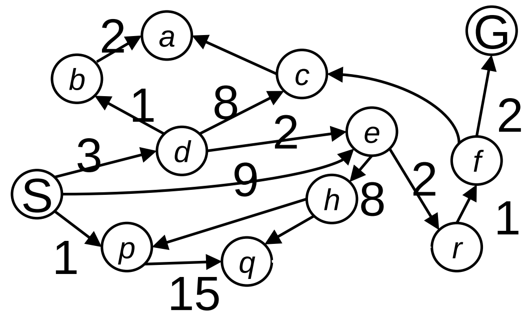
Cost contours



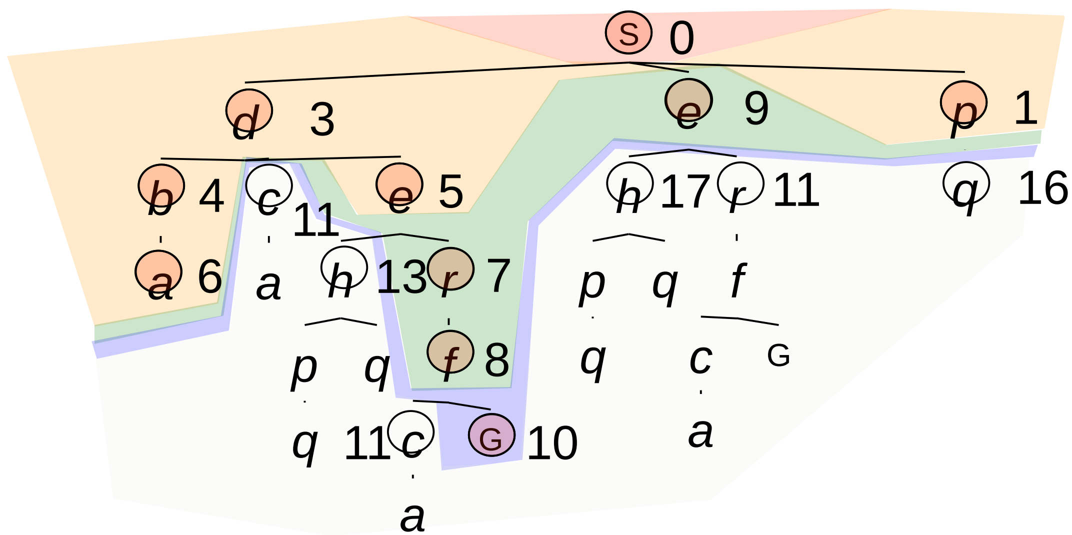
UCS Search (Reminder)

Strategy: expand a cheapest node first:

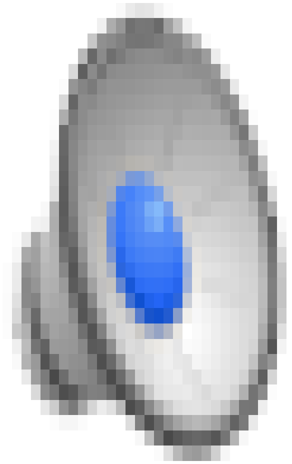
*Fringe is **a priority queue** (priority: cumulative cost)*



Cost contours



UCS over Maze with Deep/Shallow Water



UCS … in 1 line ☺

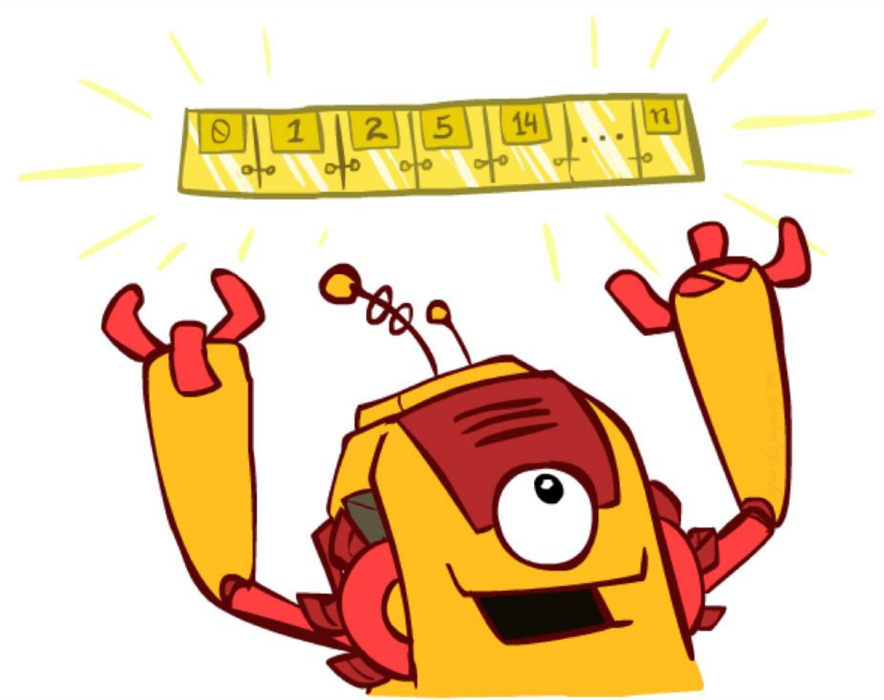
```
def BFS(problem):
    """Search the shallowest nodes in the search tree first."""
    return tree_search(problem,
                 util.PriorityQueueWithFunction(Node.getCost))
```

HUH?

http://www.mathcs.emory.edu/~eugene/cs425/p1/docs/util.html

One Queue to rule them all...

- All these search algorithms are the same except for fringe strategies
  - Conceptually, **all fringes are priority queues (i.e. collections of nodes with attached priorities)**
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stack and queues
  - Python Hint: can make one general graph search implementation that takes a variable **Fringe** object as a parameter
  - Use utils.pm for Stack, Queue, PriorityQueue classes.



Priority Queue Refresher

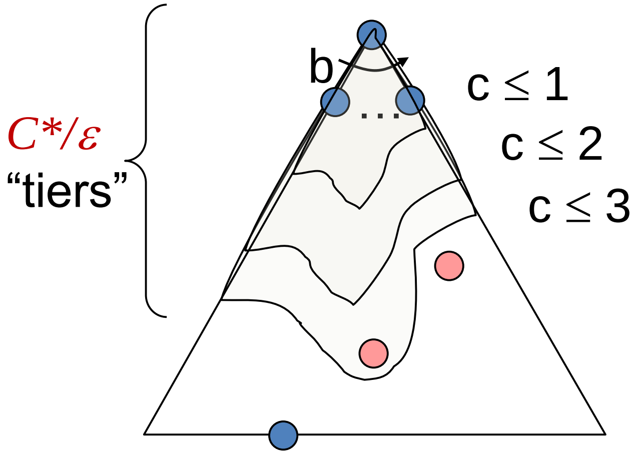
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

pq.push(key, value)	inserts (key, value) into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

- You can decrease a key's priority by pushing it again
- Unlike a regular queue, insertions aren't constant time, usually O(log n)
- We'll need priority queues for cost-sensitive search methods

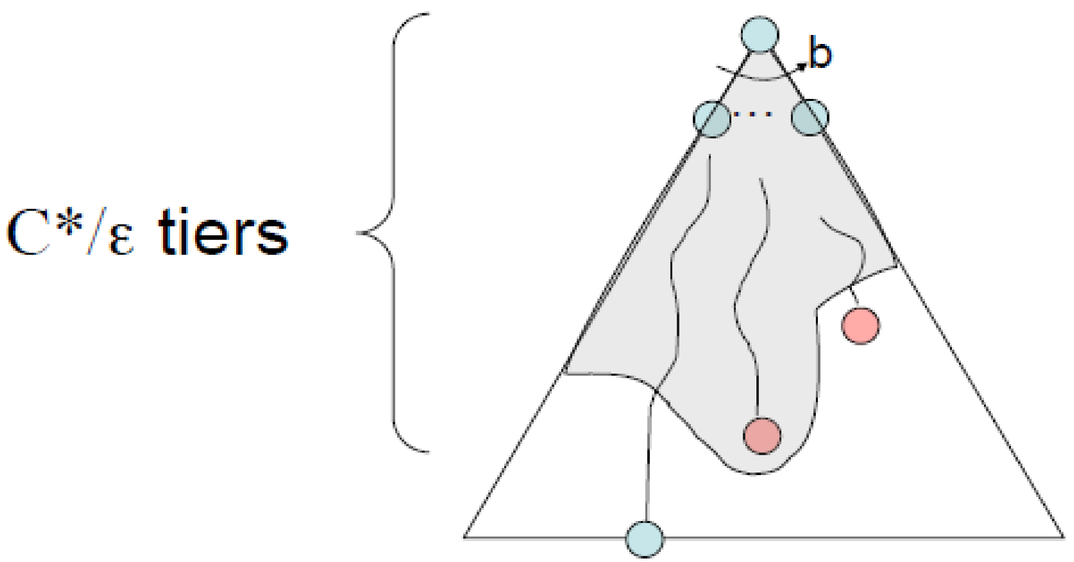
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly size of the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof soon via A* algorithm)



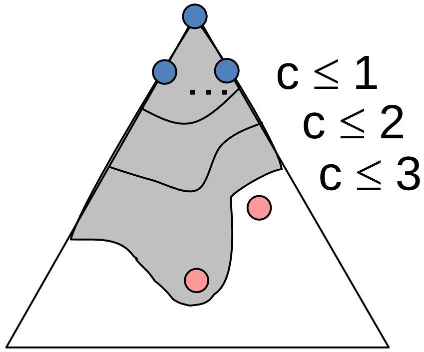
Performance Comparison

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	Y*	$O(b^d)$	$O(b^d)$
UCS		Y*	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$



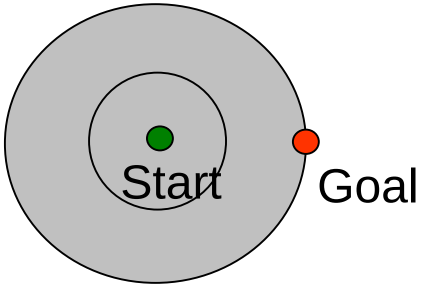
Uniform Cost Issues

- Remember: UCS explores increasing cost contours



- The good: UCS is complete and optimal!

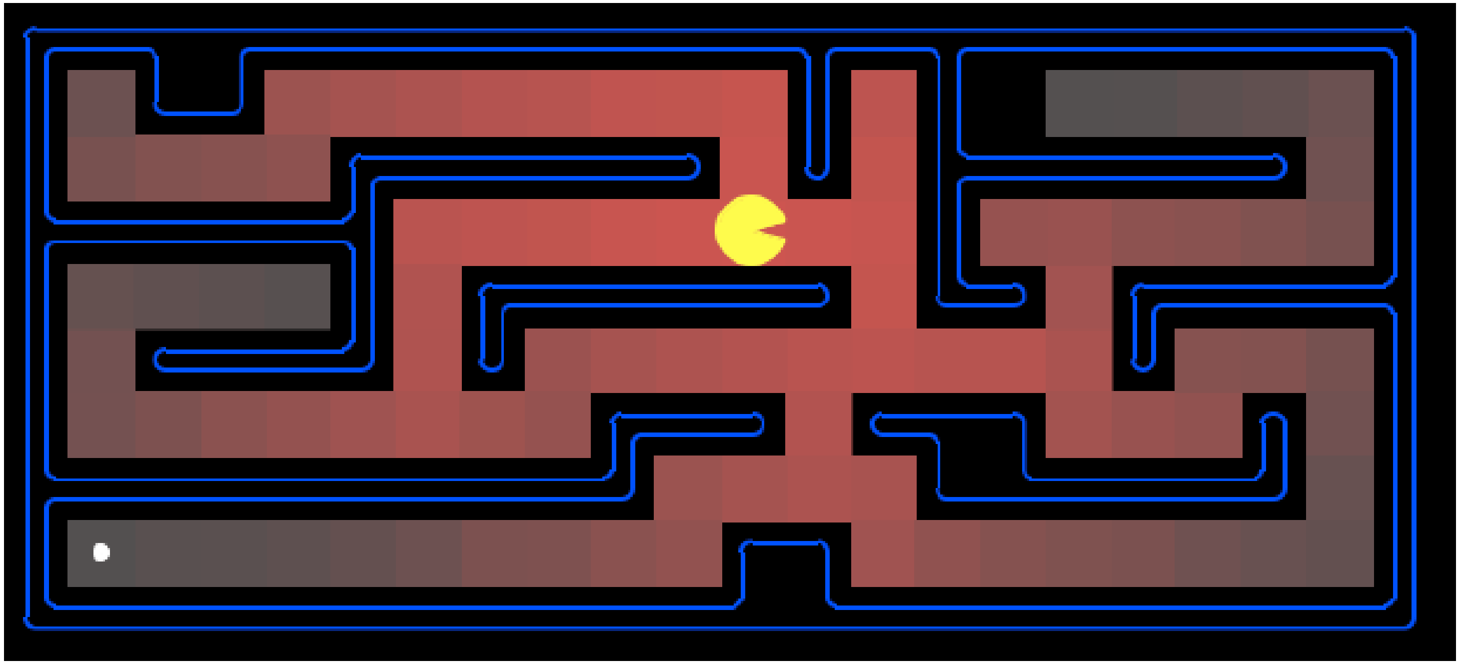
- The bad:
  - Explores options in every "direction"
  - No information about goal location



- We'll fix that next!

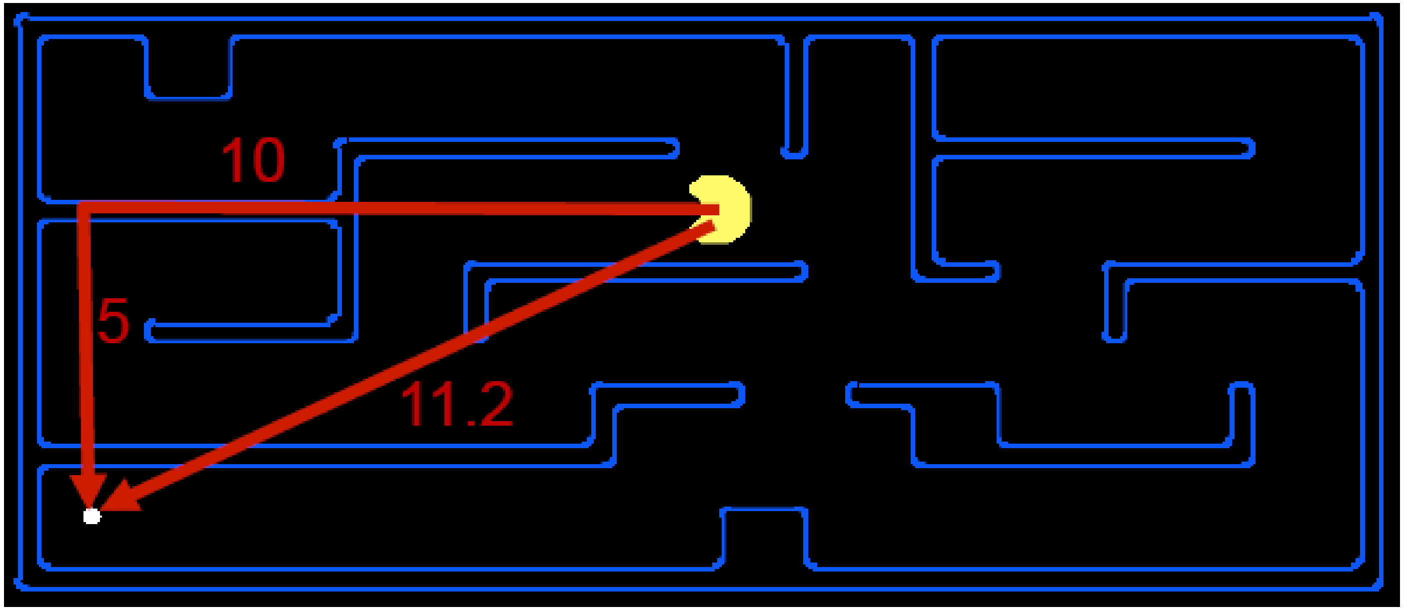
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one



Search Heuristics

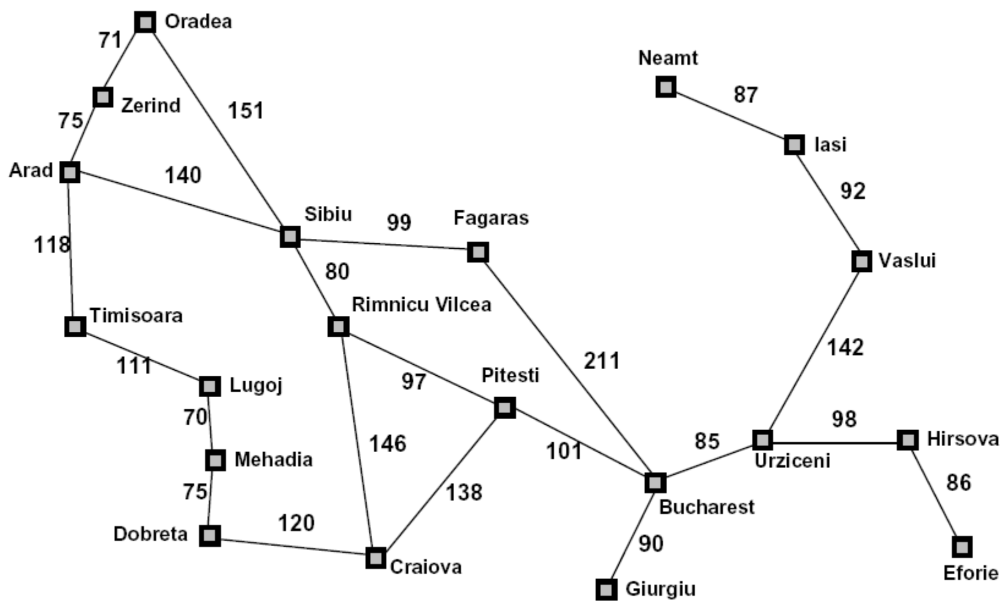
- Any estimate of how close a state is to a goal
- Designed for a particular search problem



- Examples: Manhattan distance, Euclidean distance

https://qiao.github.io/PathFinding.js/visual/

Example: Heuristic Function

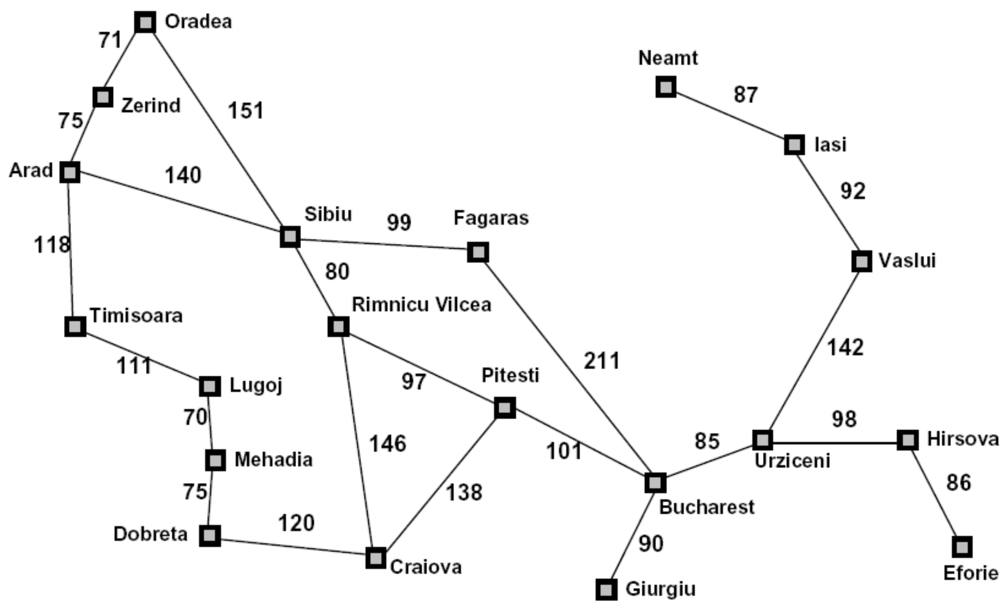


Straight−line distance to Bucharest

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

h(x)

Example: Heuristic Function

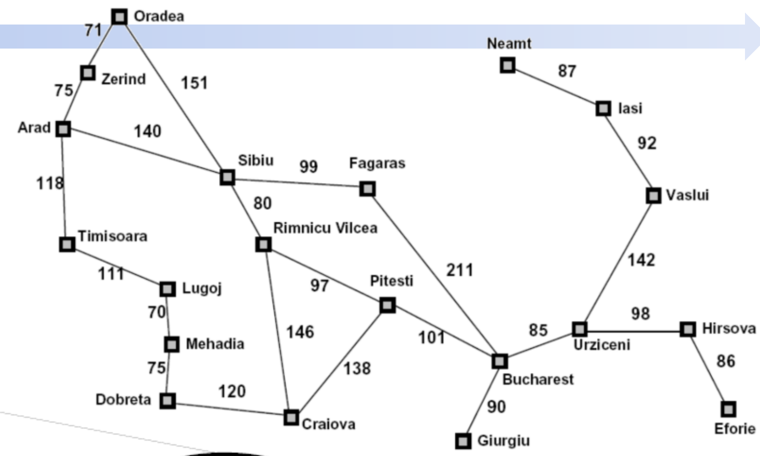


Straight−line distance to Bucharest

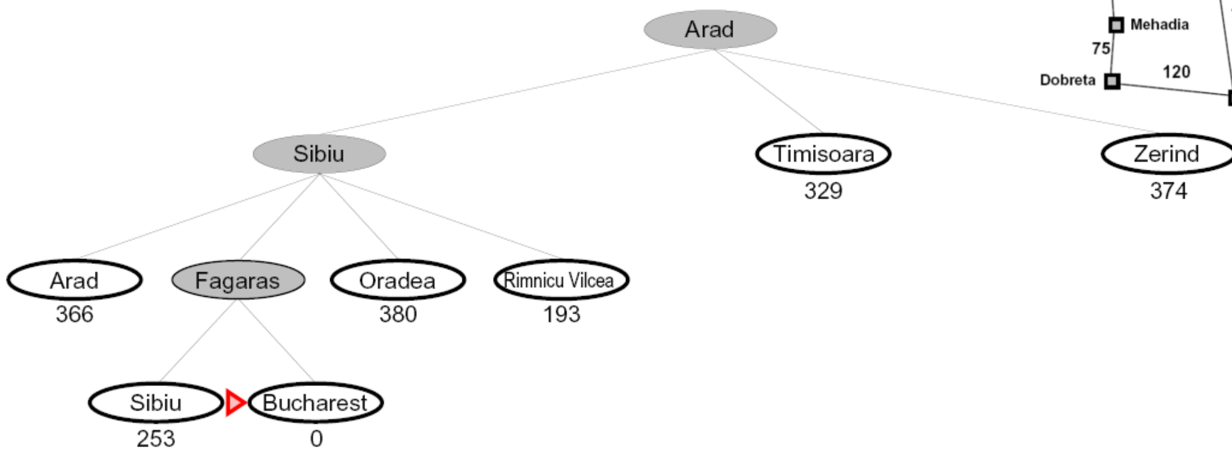
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

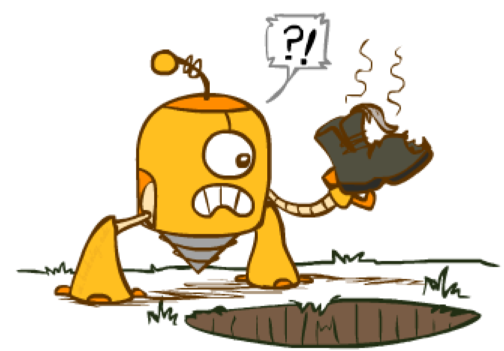
Greedy Search



- Expand the node that seems closest…

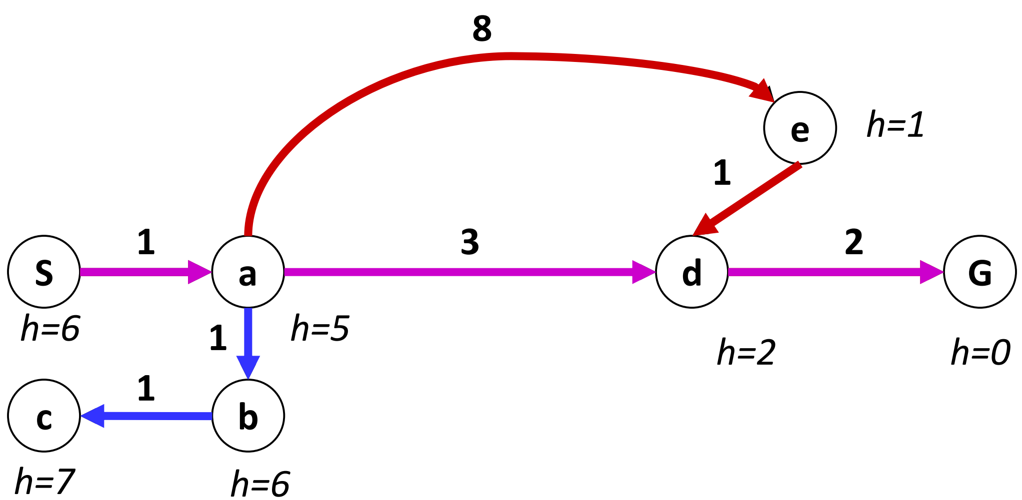


- What can go wrong?



Exercise: Greedy Search

- Greedy orders PQ by goal proximity, or *heuristic cost* h(n)



- What is the greedy solution?

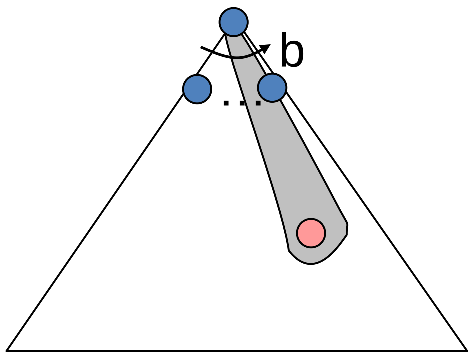
Greedy … in 3 lines ☺

```
def BFS(problem):
    """Search the shallowest nodes in the search tree first."""
    return greedy_search(problem,
                    util.PriorityQueueWithFunction(heuristic))
```

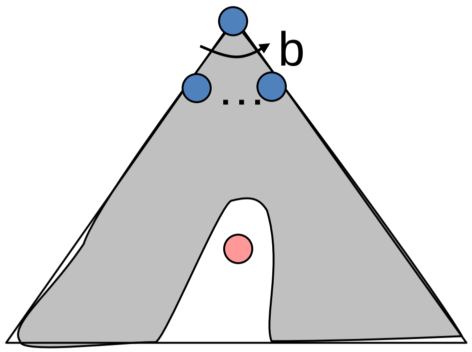
```
    def heuristic(Node n):
         return 42
```

Greedy Search (analysis sketch)

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state

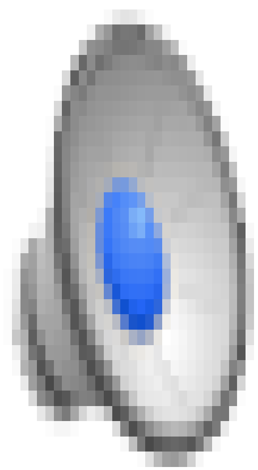


- A common case:
  - Best-first takes you straight to the (wrong) goal

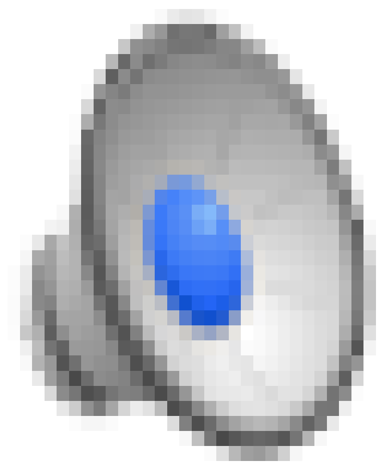


- Worst-case: like a badly-guided DFS

Video of Demo Contours Greedy (Empty)



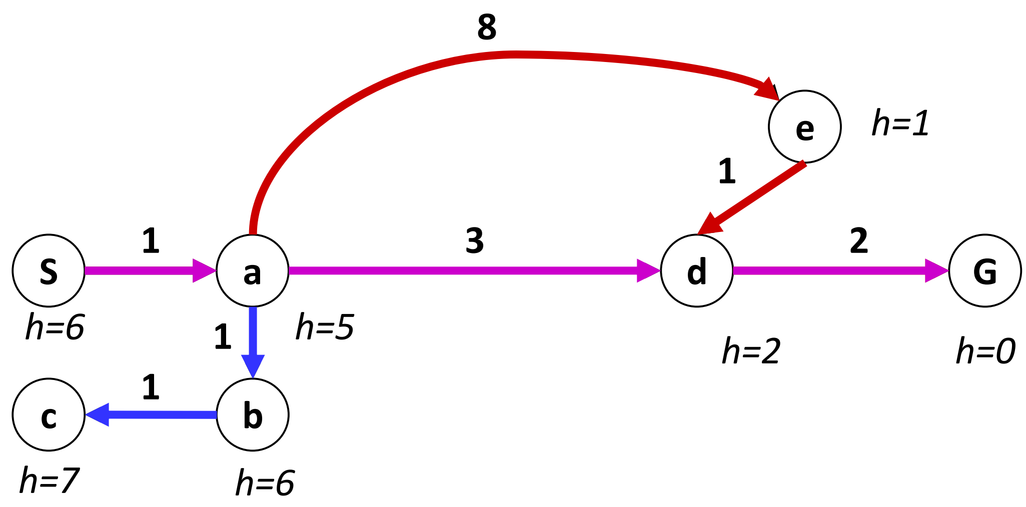
Video of Demo Contours Greedy (Pacman Small Maze)



A* Search

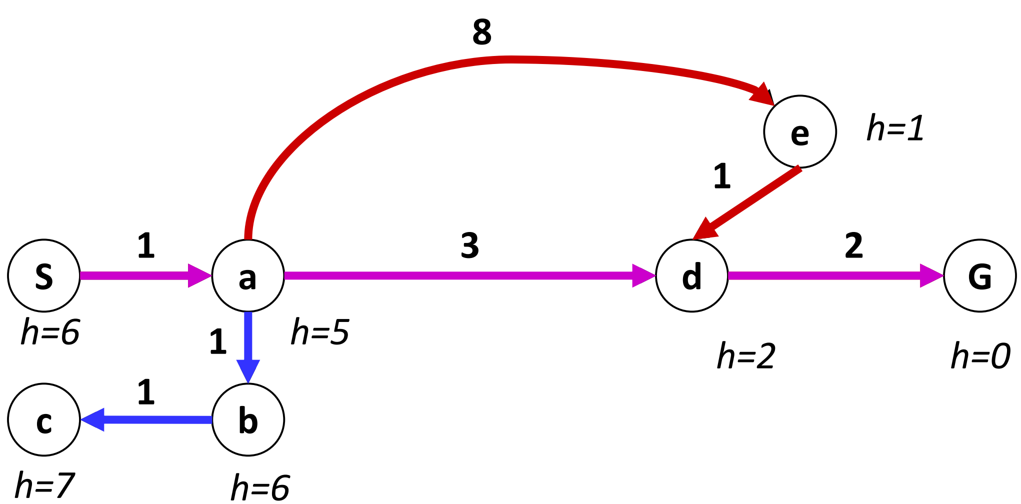
Can we Combine UCS and Greedy?

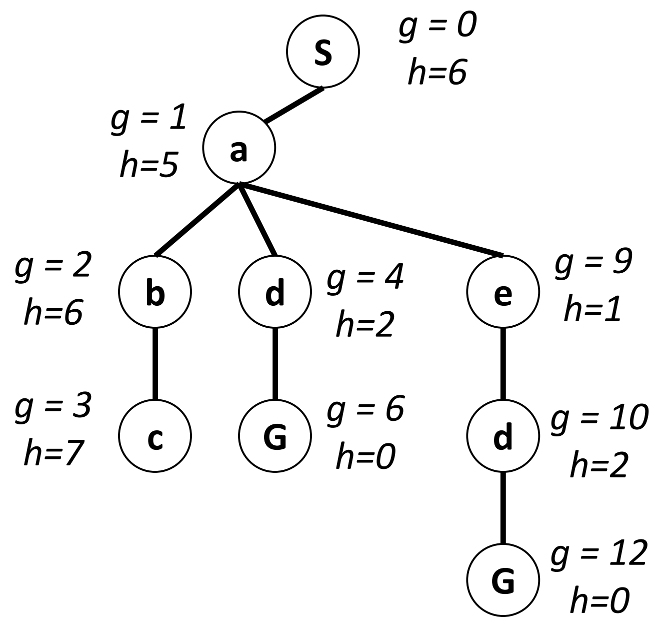
- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)



UCS + Greedy = A*

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)



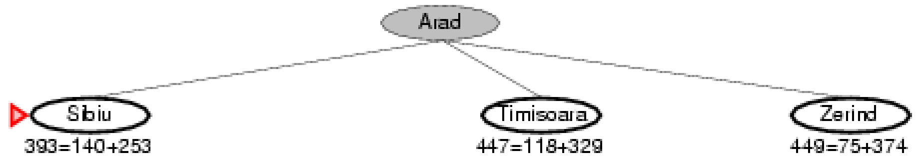


A* Search orders by the sum: f(n) = g(n) + h(n)

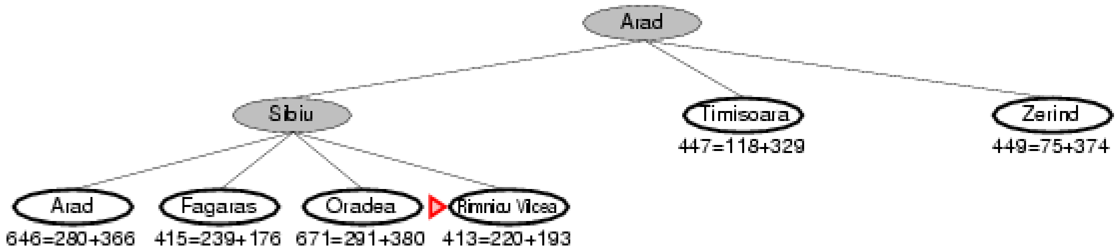
Example: Teg Grenager

A* search example

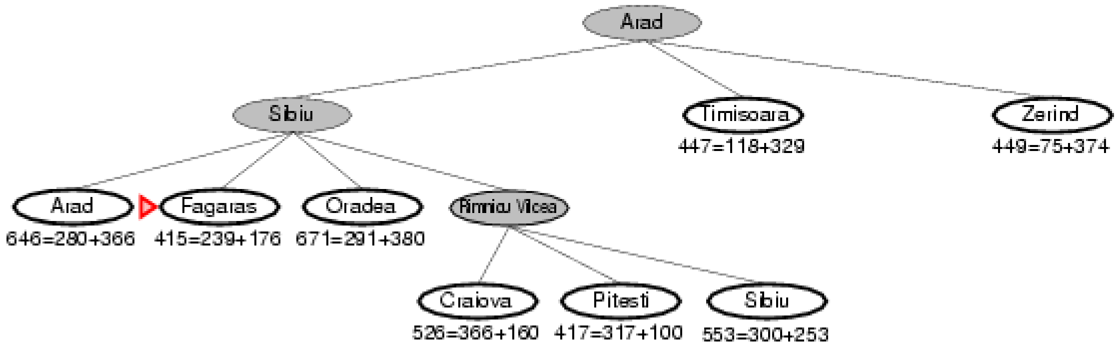
A* search example



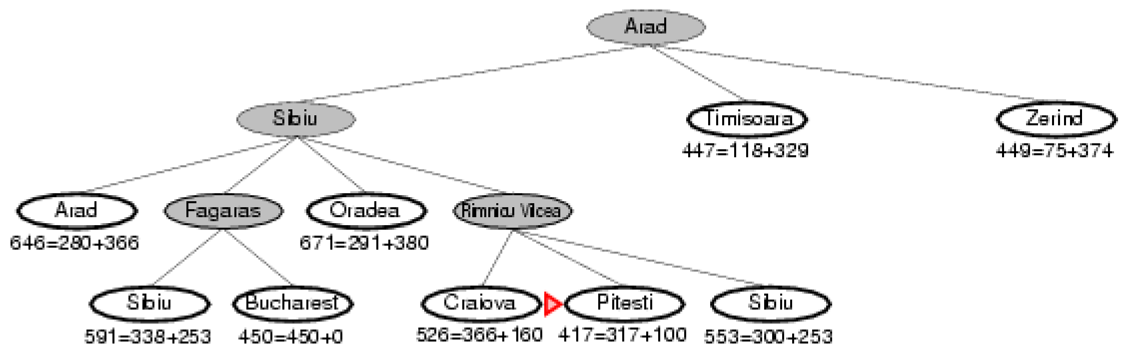
A* search example



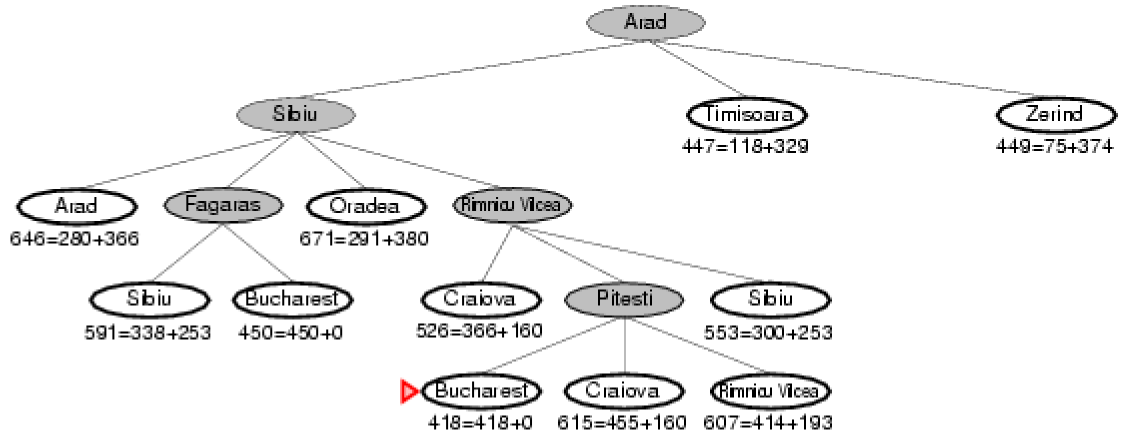
A* search example



A* search example



A* search example



A*… in 3 lines ☺

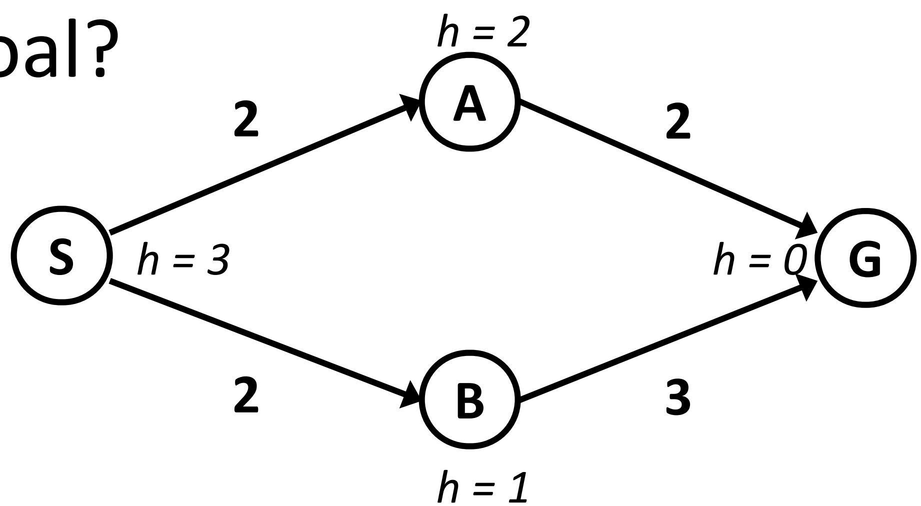
```
def BFS(problem):
    """Search the shallowest nodes in the search tree first."""
    return astar_search(problem,
                util.PriorityQueueWithFunction(fcost))
```

```
def heuristic(Node n):
     return ….
```

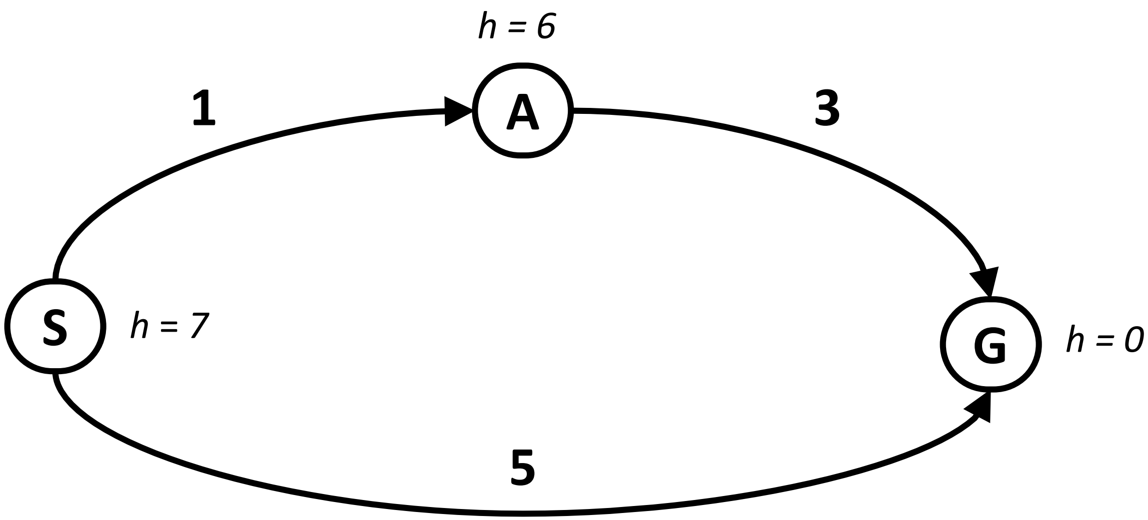
```
def fcost(Node n):
     return heuristic(Node n) + n.getCost()
```

When should A* terminate?

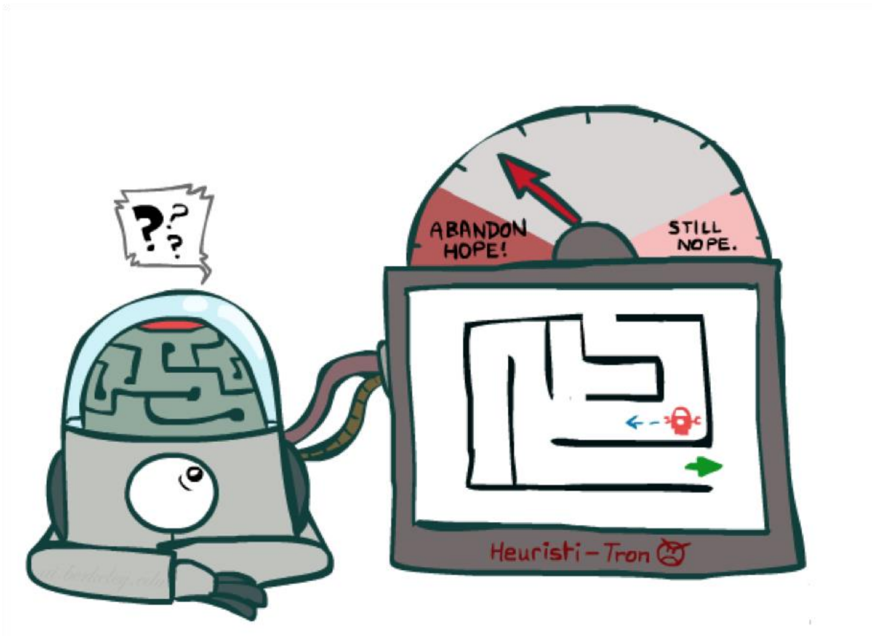
- Should we stop when we enqueue a goal?



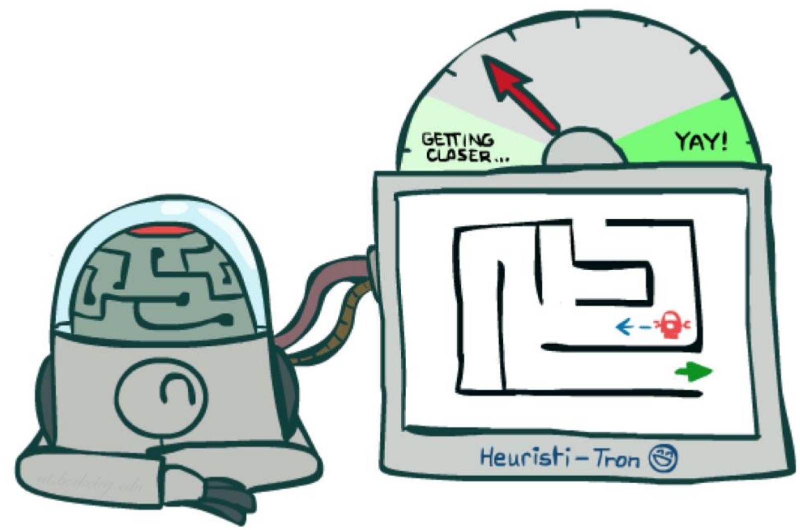
Exercise: A* solution for this graph?



Idea: Admissibility



Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe



Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

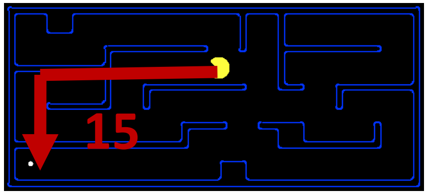
Admissible Heuristics

- A heuristic *h* is *admissible* (optimistic) if:

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

  where $h^*(n)$ is the true cost to a nearest goal

- Examples:



- Coming up with admissible heuristics is most of what's involved in using A* in practice.

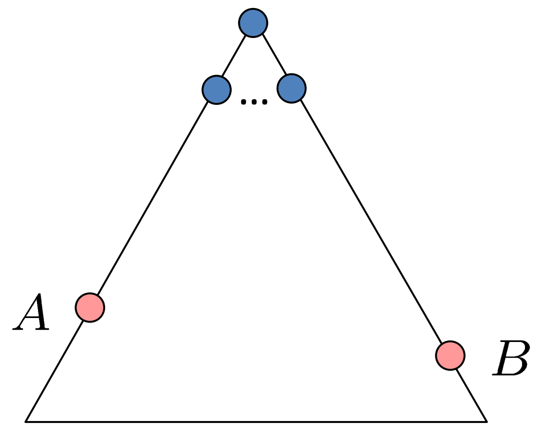
Optimality of A* Tree Search

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

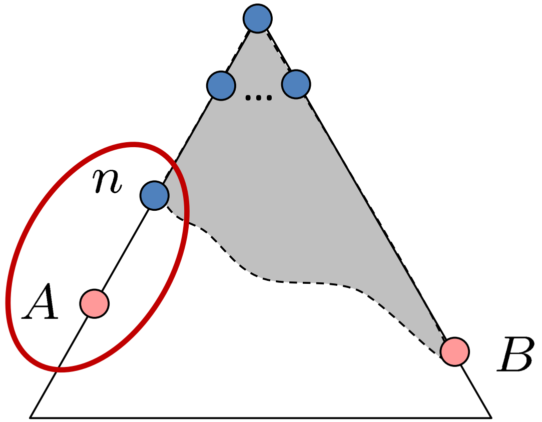
- A will exit the fringe before B



Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
    1. f(n) is less or equal to f(A)

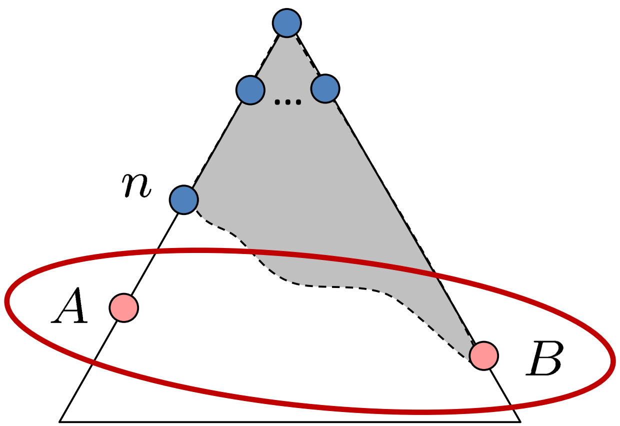


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)



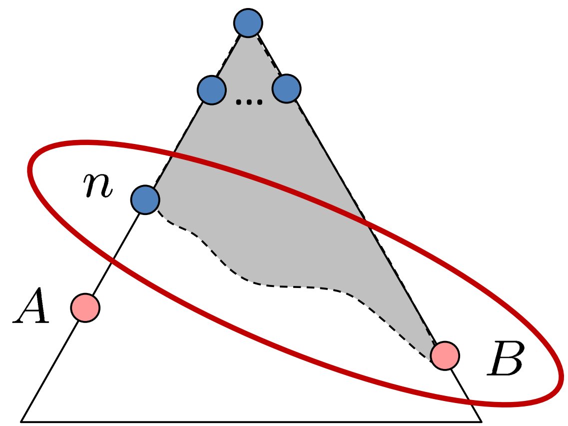
$g(A) < g(B)$ B is suboptimal

$f(A) < f(B)$ h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

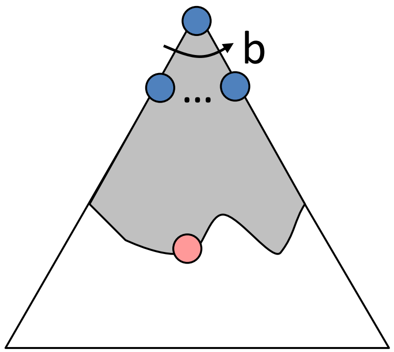
- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
    1. f(n) is less or equal to f(A)
    2. f(A) is less than f(B)
    3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



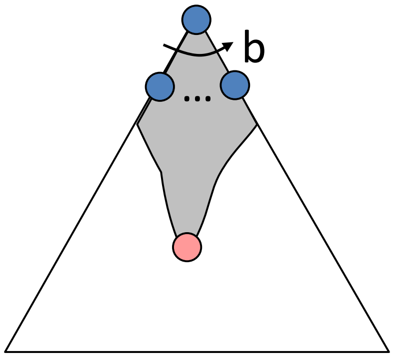
$$f(n) \leq f(A) < f(B)$$

Properties of A*

Uniform-Cost

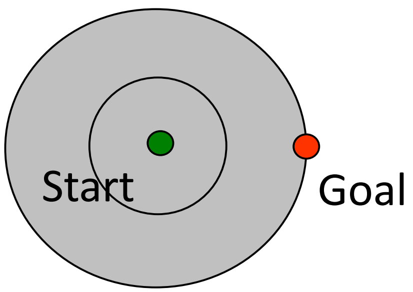


A*

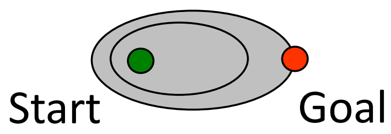


UCS vs A* Contours

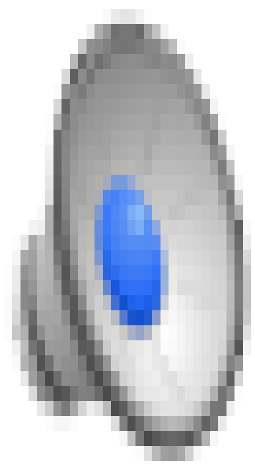
- Uniform-cost expands equally in all "directions"



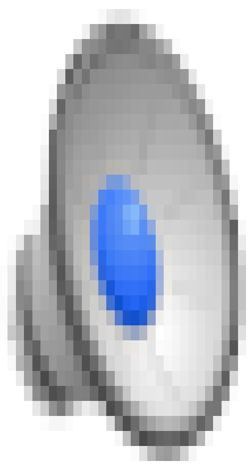
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



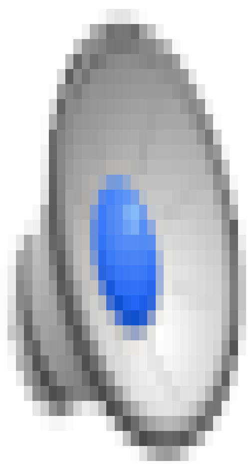
Video of Demo Contours (Empty) -- UCS



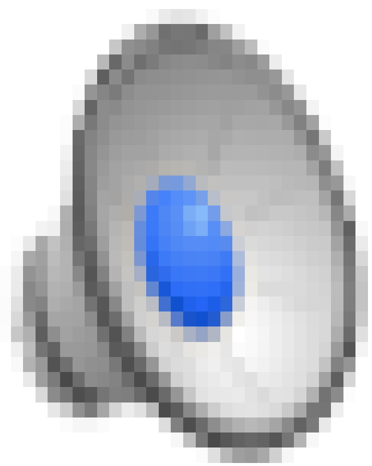
Video of Demo Contours (Empty) -- Greedy



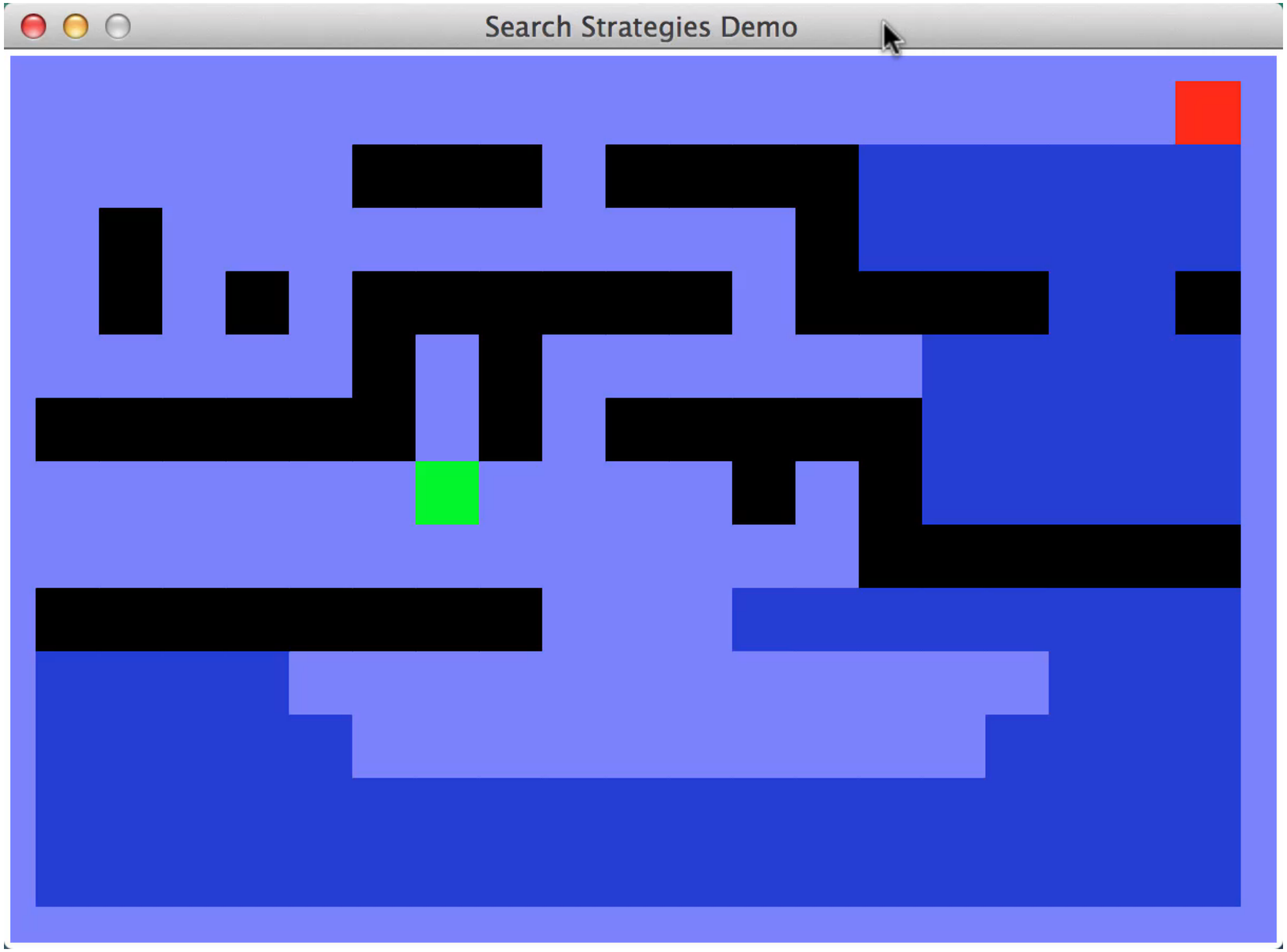
Video of Demo Contours (Empty) – A*



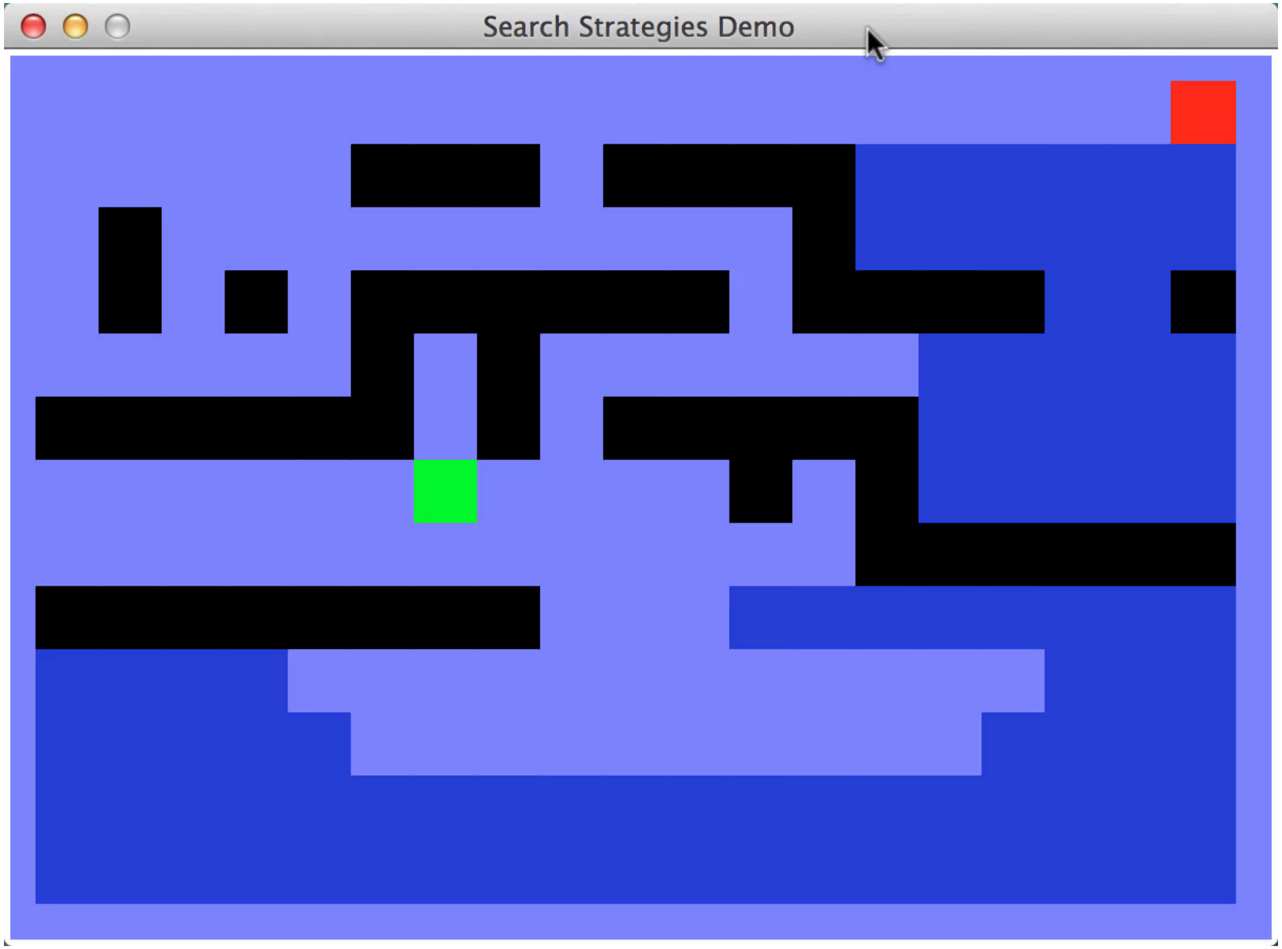
Video of Demo Contours (Pacman Small Maze) – A*



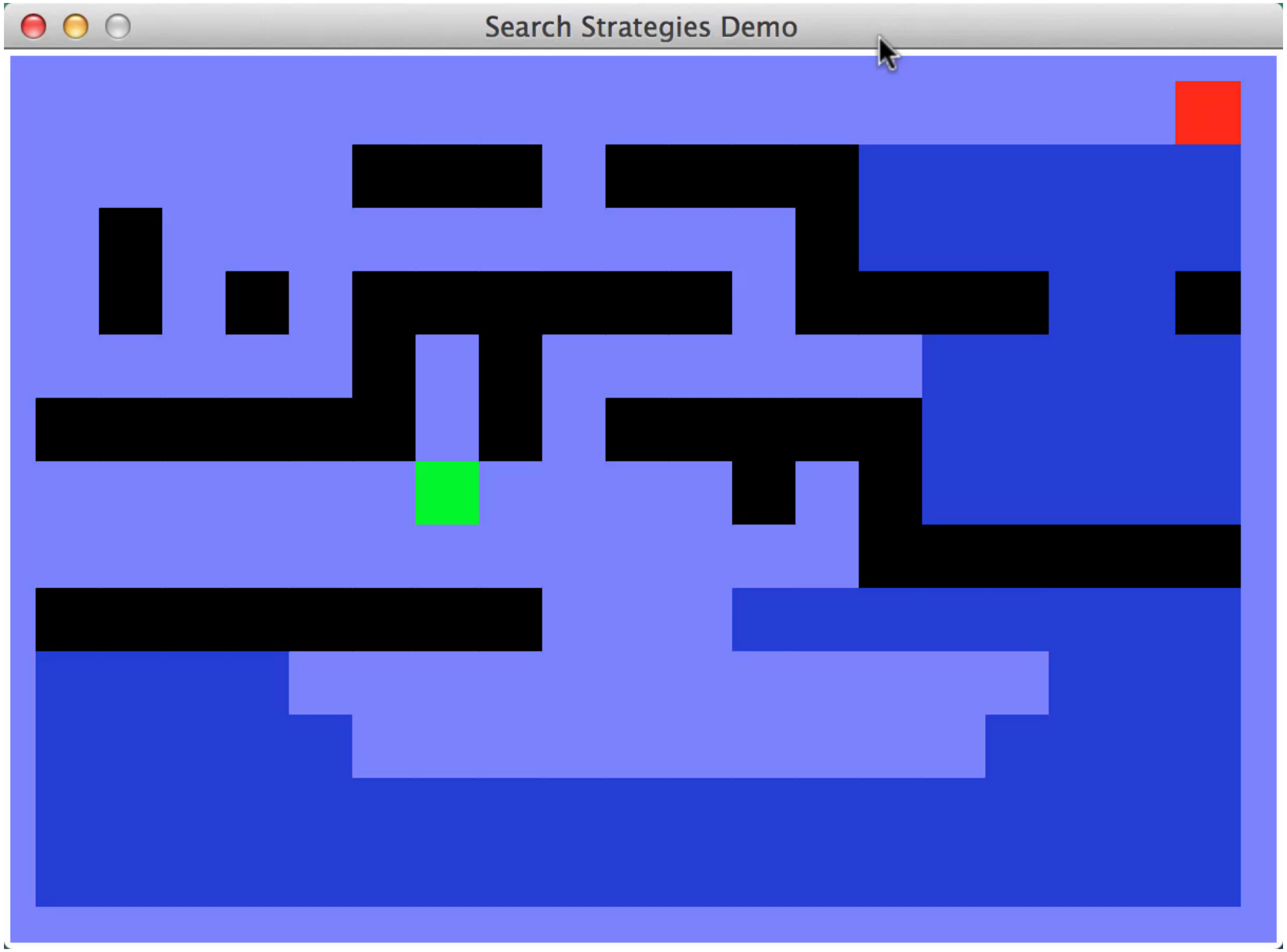
Which Algorithm (1)?



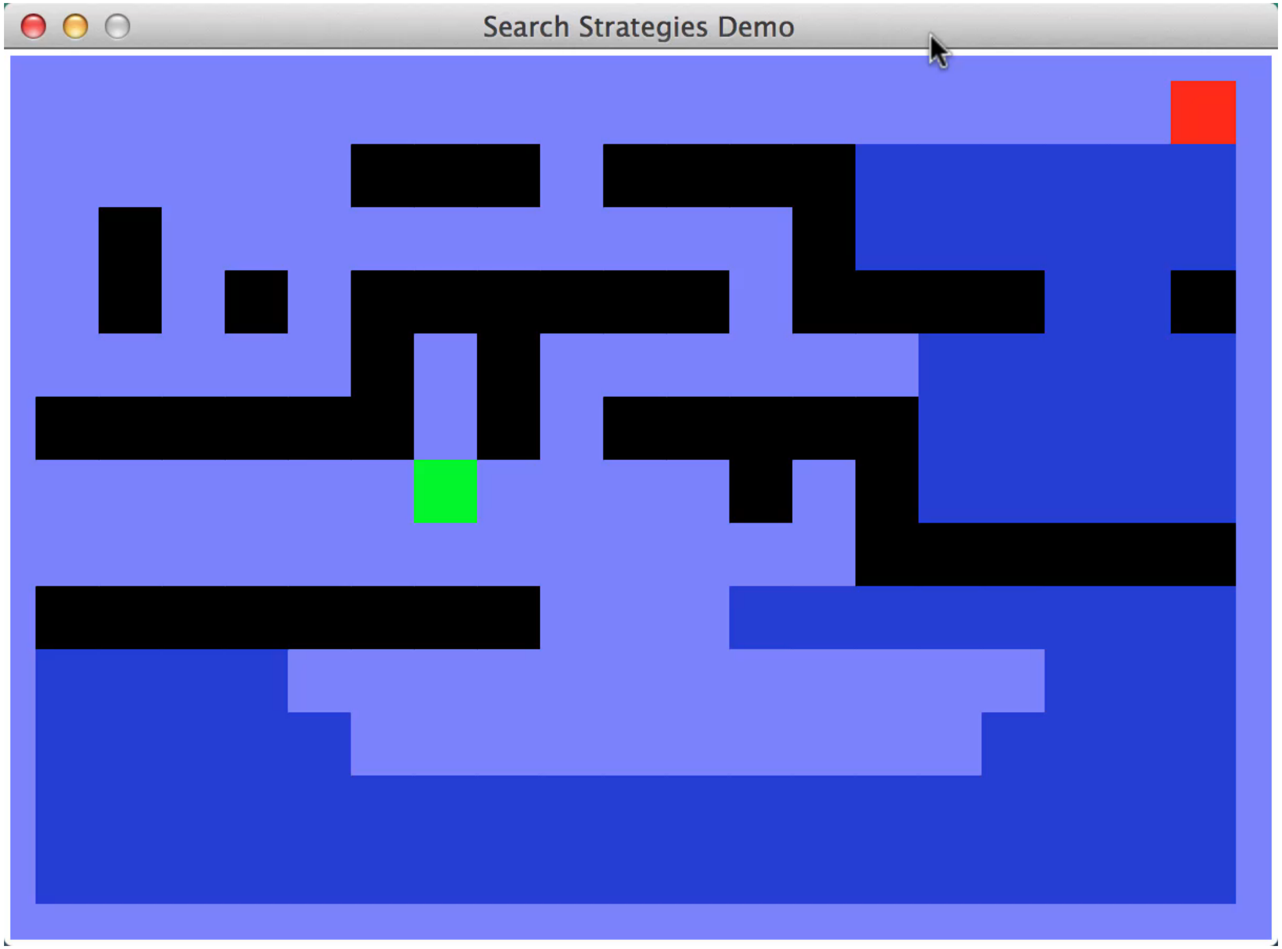
Which Algorithm (2)



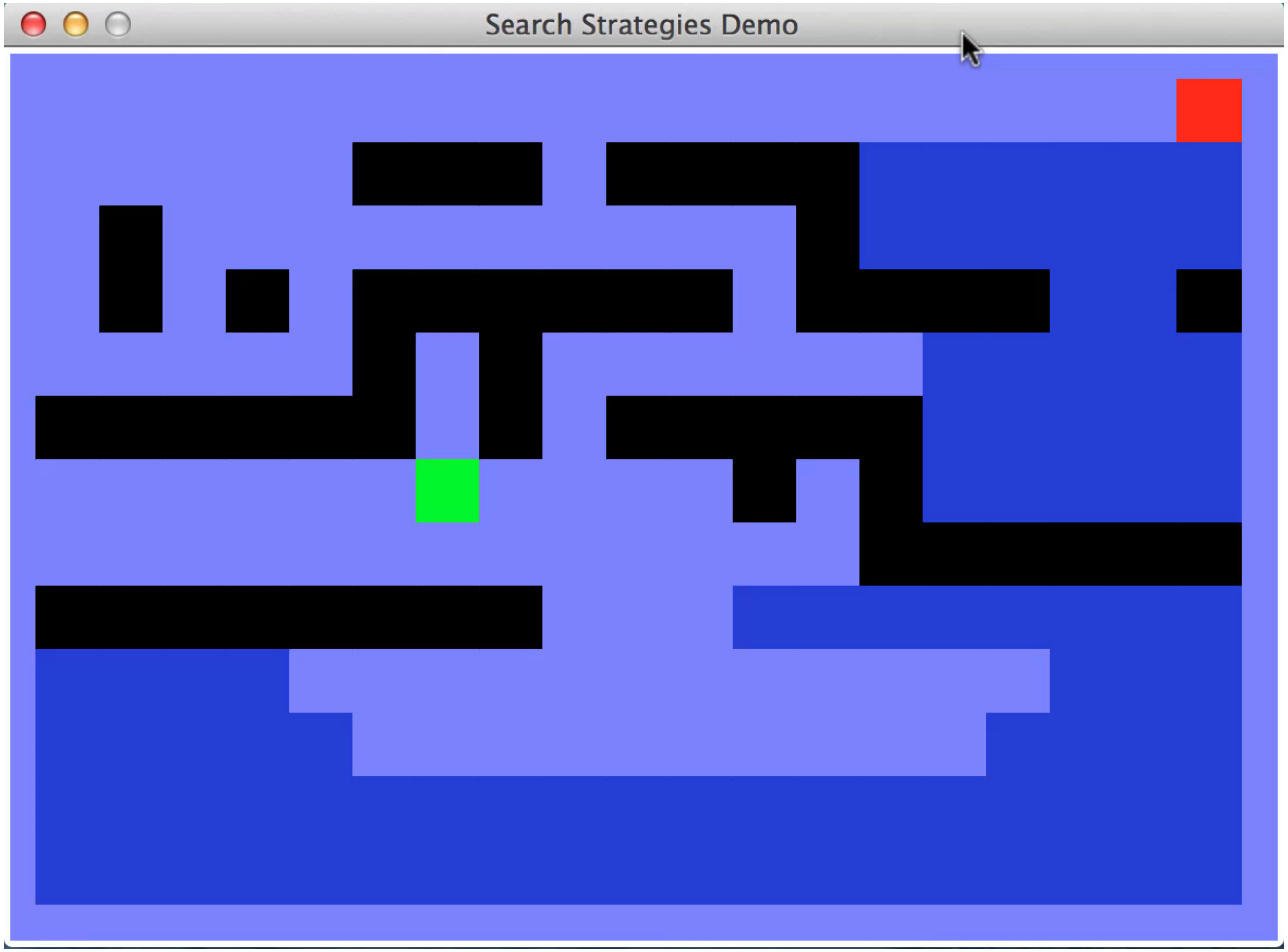
Which Algorithm (3)



Which Algorithm (4)?



Which Algorithm (5)



Comparison: Summary

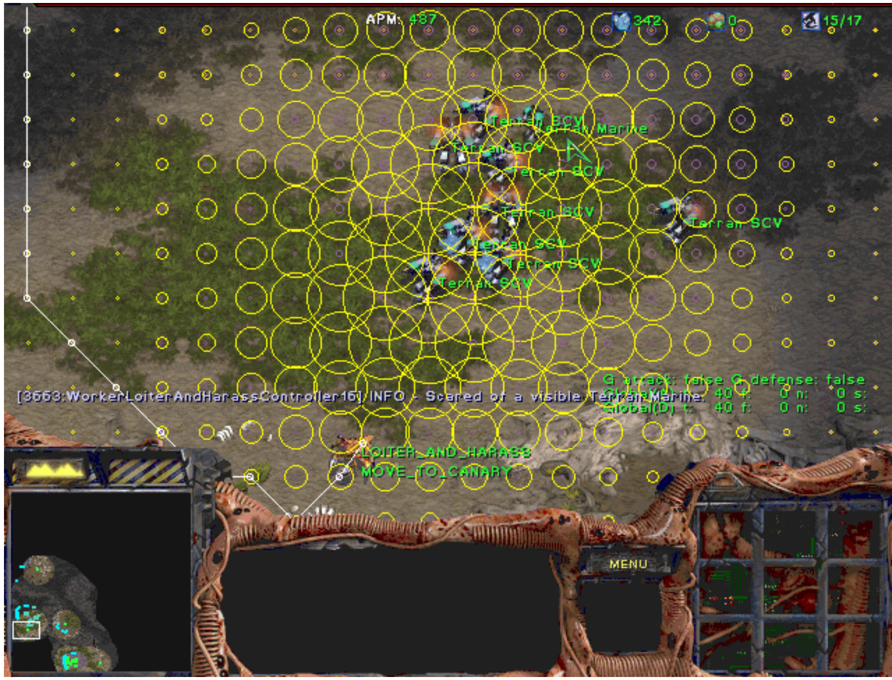
Greedy

Uniform Cost

A*

A* Applications

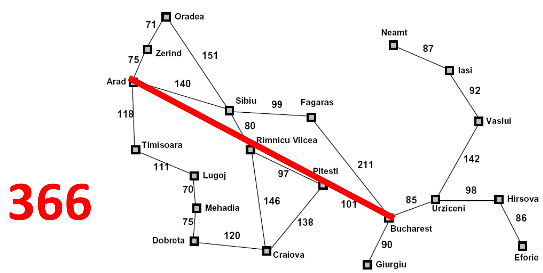
- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

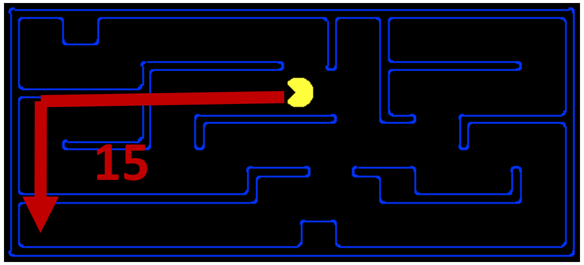


Creating Admissible Heuristics

- **Most of the work in solving hard search problems optimally is in coming up with admissible heuristics**

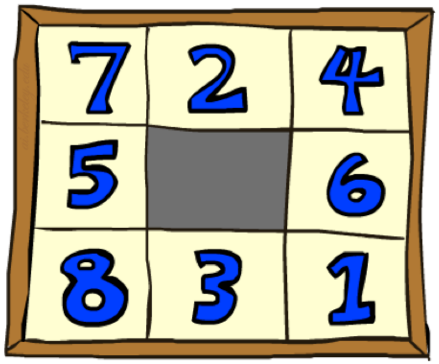
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available



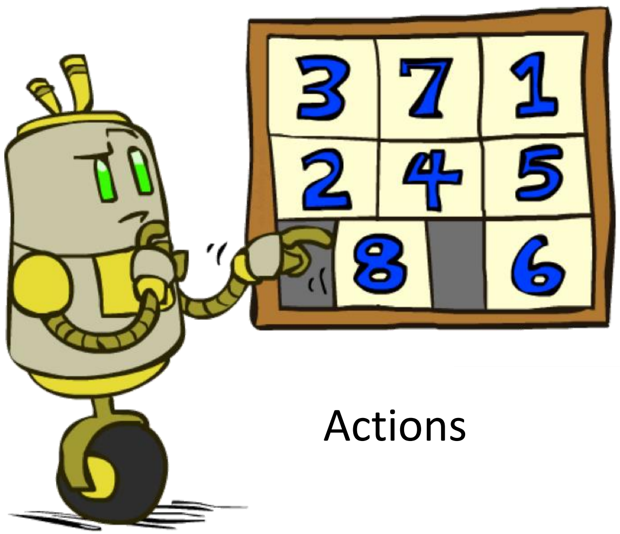


- Inadmissible heuristics can be useful too!

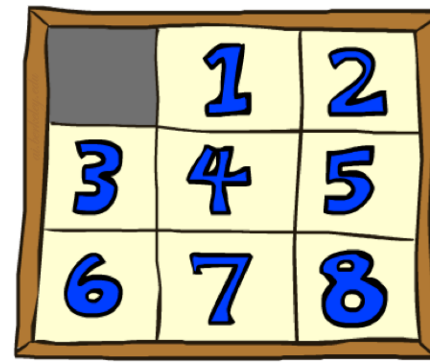
Example: 8 Puzzle



Start State



Actions

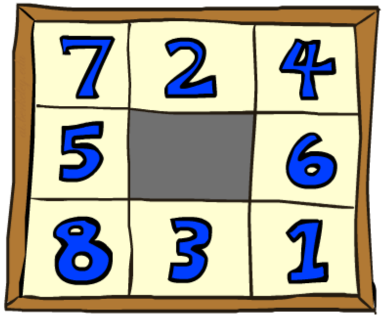


Goal State

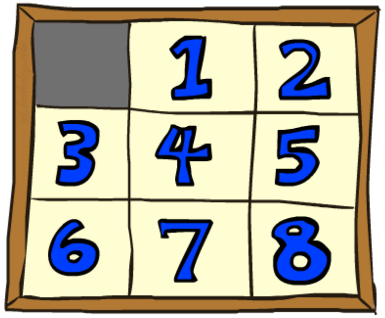
- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

8 Puzzle I

- Heuristic:
- Is it admissible?
- h(start) =
- h(goal) =



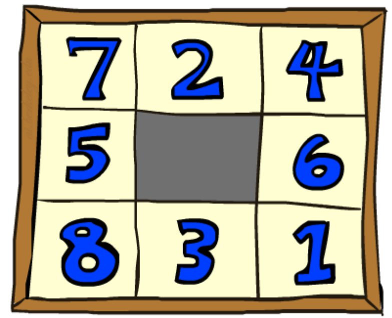
Start State



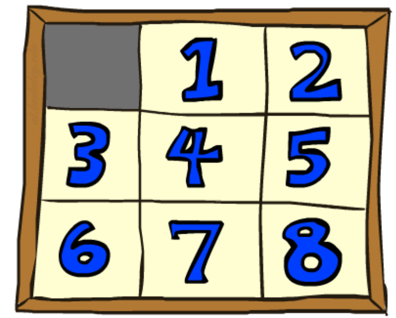
Goal State

8 Puzzle: **Tiles** heuristic

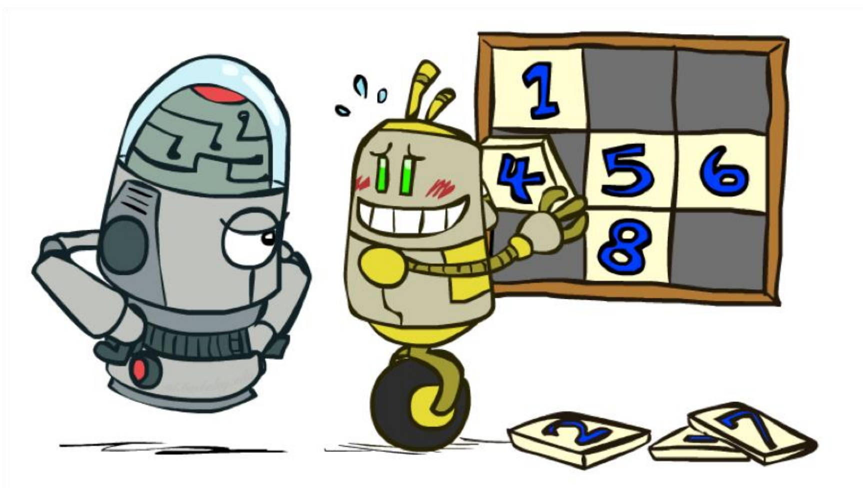
- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic



Start State



Goal State



	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

Statistics from Andrew Moore

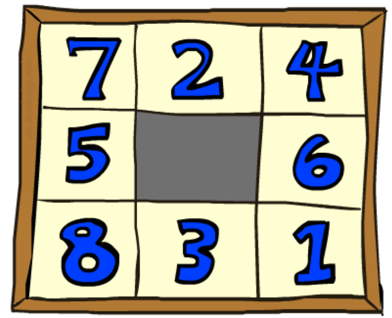
8 Puzzle II: **Manhattan** heuristic

- **<u>Relaxation</u>**: easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?

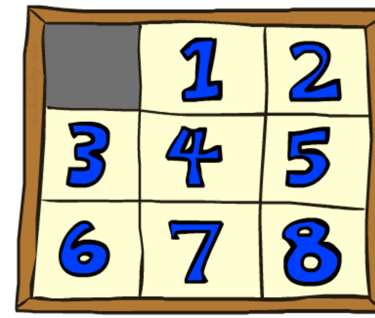
- Total *Manhattan* distance from correct location

- Is it admissible?

- h(start) = 3 + 1 + 2 + … = 18



Start State

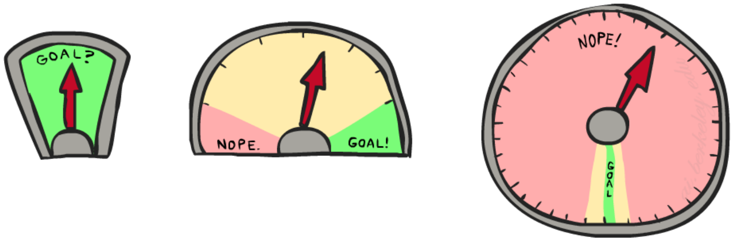


Goal State

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

8 Puzzle III: **Oracle** heuristic

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Recap: **Problem Relaxation**

- A problem with fewer restrictions on the actions is called a relaxed problem

- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

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

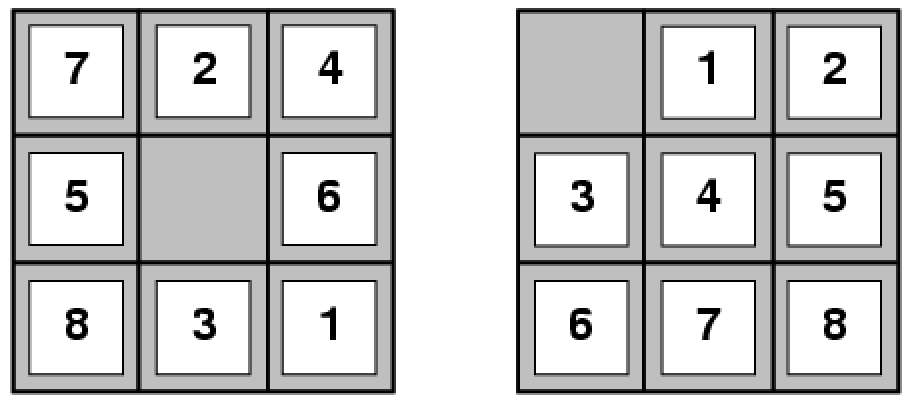
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Designing heuristics (cont'd)

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

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



Start State Goal State

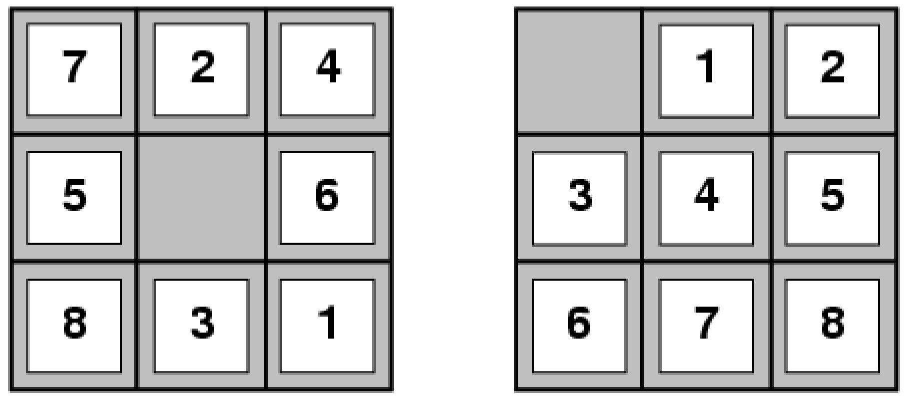
- $h_1(S)$ = ?
- $h_2(S)$ = ?

Heuristics: cont’d

E.g., for the 8-puzzle:

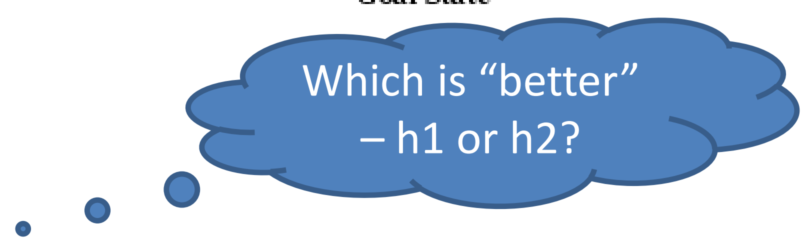
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

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



Start State Goal State

- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18



Idea: Heuristic **dominance**

- If $h_2(n) \geq h_1(n)$ for all n (both admissible, i.e., < true cost)
  then h_2 dominates h_1

  h_2 is better for search

- Typical search costs (average number of nodes expanded):

- *d=12* IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- *d=24* IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

Example: Heuristics for Chess

- To select next move, must evaluate expected benefit of successor position:
  - **Value of the pieces** (count value of your pieces – value of opponents pieces)
  - **Space**: threatened/controlled space by you – space controlled by opponent
  - **Pawn** structure
  - …
- Examples:
  - https://www.quora.com/What-are-some-heuristics-for-quickly-evaluating-chess-positions
  - https://chessprogramming.wikispaces.com/Killer+Heuristic

Example: Heuristics for Motion Planning

- Robot motion: many moving (body) parts
- What's the most efficient way to accomplish goal?

https://www.youtube.com/watch?v=dSwDZmvtGZY

Designing Heuristics

- A good heuristic is:
  - ✓ Admissible (optimistic)
  - ➢ Consistent (non-decreasing)
  - ✓ “Accurate”

Trivial Heuristics, Dominance

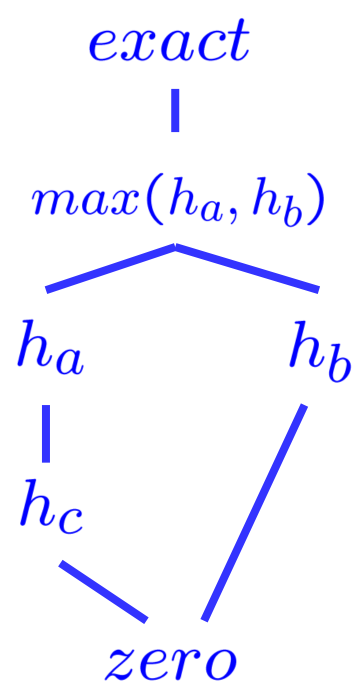
- Dominance: $h_a \geq h_c$ if

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

- Max of admissible heuristics is admissible:

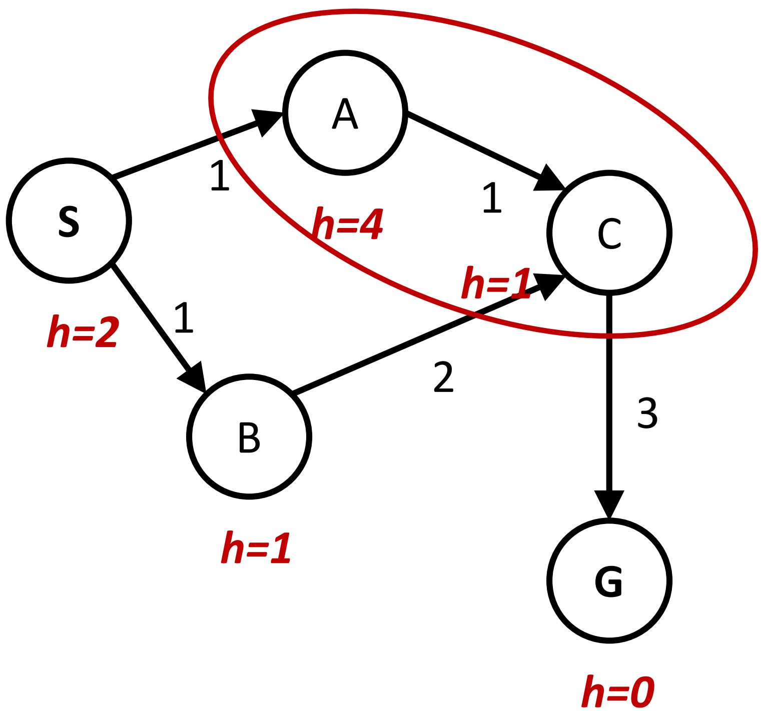
$$h(n) = max(h_a(n), h_b(n))$$

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

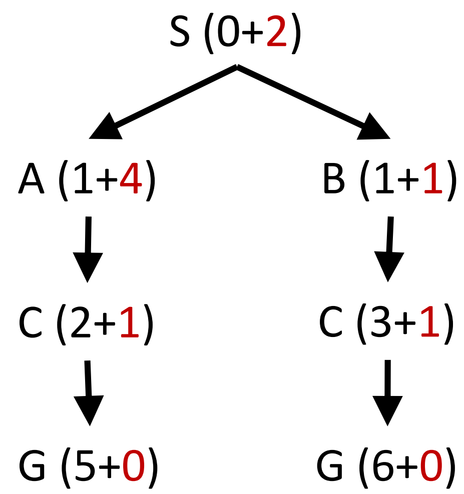


A* Graph Search Gone Wrong?

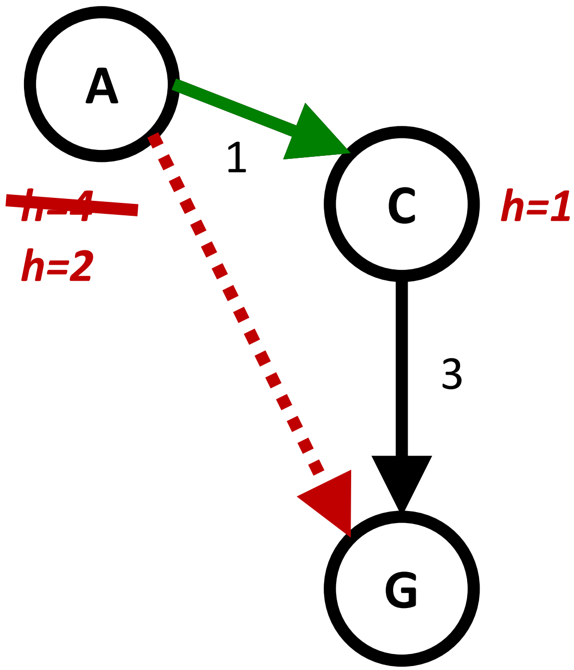
State space graph



Search tree



Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

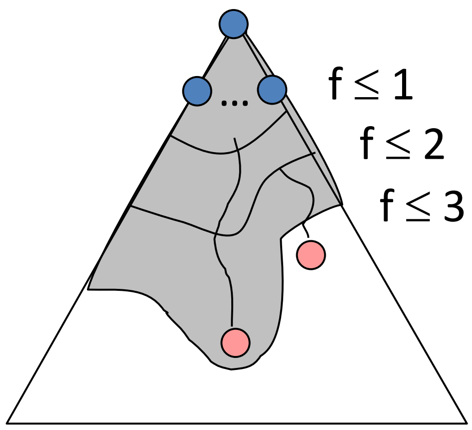
    h(A) – h(C) ≤ cost(A to C)

- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Optimality of A* Graph Search

- Sketch of proof: consider what A* does with a **consistent** heuristic:
  - **Fact 1:** In tree search, A* expands nodes in increasing total f value (f-contours)
  - **Fact 2:** For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal



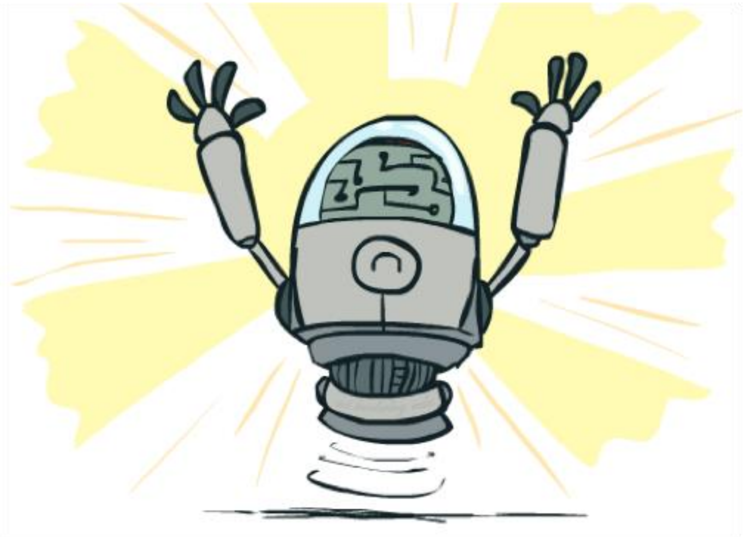
Optimality (2): Tree vs. Graph Search

- **Tree search:**
  - A* is optimal if heuristic is **admissible**
  - UCS is a special case (h = 0)

- **Graph search:**
  - A* optimal if heuristic is **consistent**
  - UCS optimal (h = 0 is consistent)

- Consistency implies admissibility

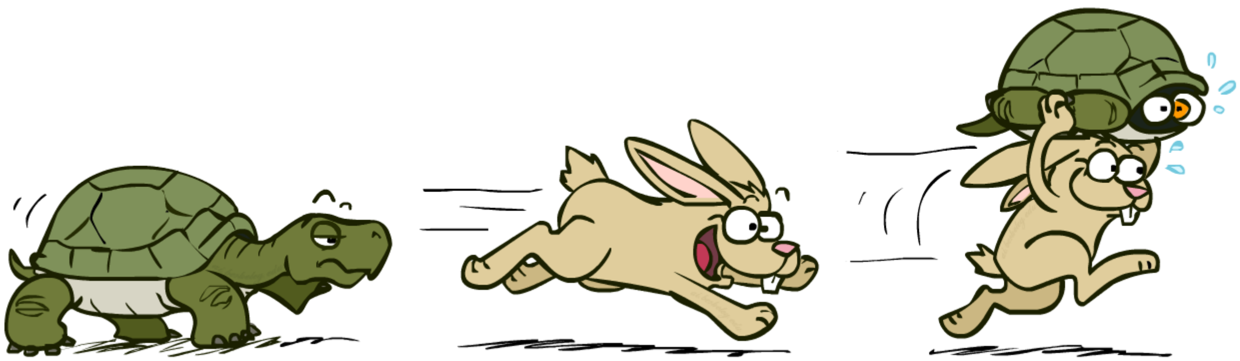
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A*: Summary

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



Project 1: Due **Friday, Feb 9**

- Read FAQ on Canvas before posting questions:
- **Questions 1-4**: if you develop a <u>correct</u> solution for DFS, the rest will be easy modifications
- **Run autograder** after <u>*every* question</u>. Until you perfectly pass all the test cases, assume your code has bugs.
- Example (incomplete!) implementations: https://github.com/aimacode/aima-python/blob/master/search.py

Tips for Project 1 (cont'd)

- Problems 5-8 <u>depend on code in 1-4</u>. Get that right (and tested) first, before moving on!
- P5/Corners problem: must visit all corners in *single* path
  - Implications for search tree, state info to update
- Heuristics for p6-8: start simple. For extra credit, think back to graph traversal algorithms from cs323.

Project 1 final hints

- Use Discussions, read FAQ before posting questions
- Suggestion: use Node class or similar. Easier to do Questions 5-8.
- Questions 5-8: more fun/creative. Leave enough time, start early.
- Most importantly: Don't Panic! Eat the elephant one question at a time.