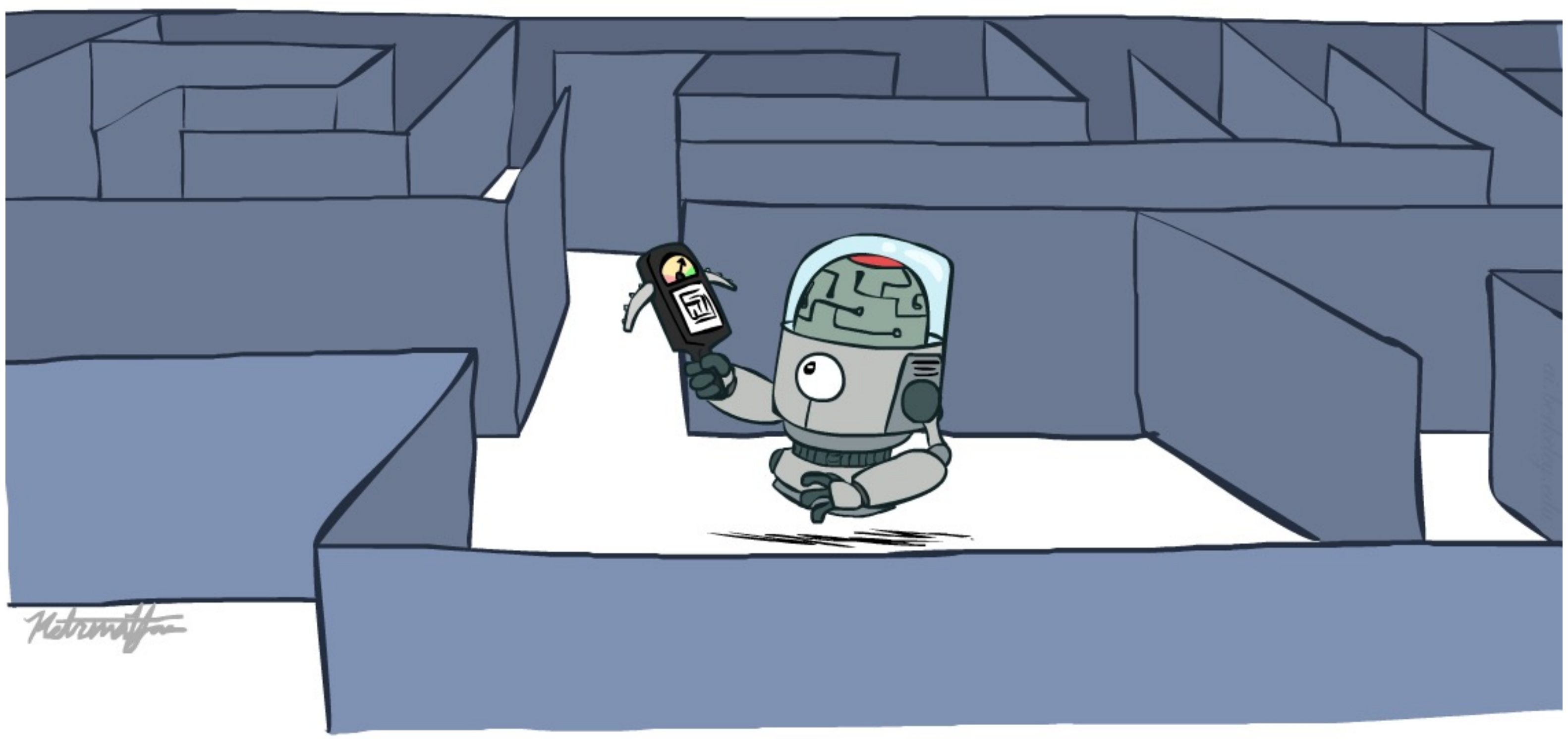
Ve492: Introduction to Artificial Intelligence

Informed Search and Graph Search



Paul Weng

UM-SJTU Joint Institute

Slides adapted from http://ai.berkeley.edu, AIMA, UM, CMU

Announcements

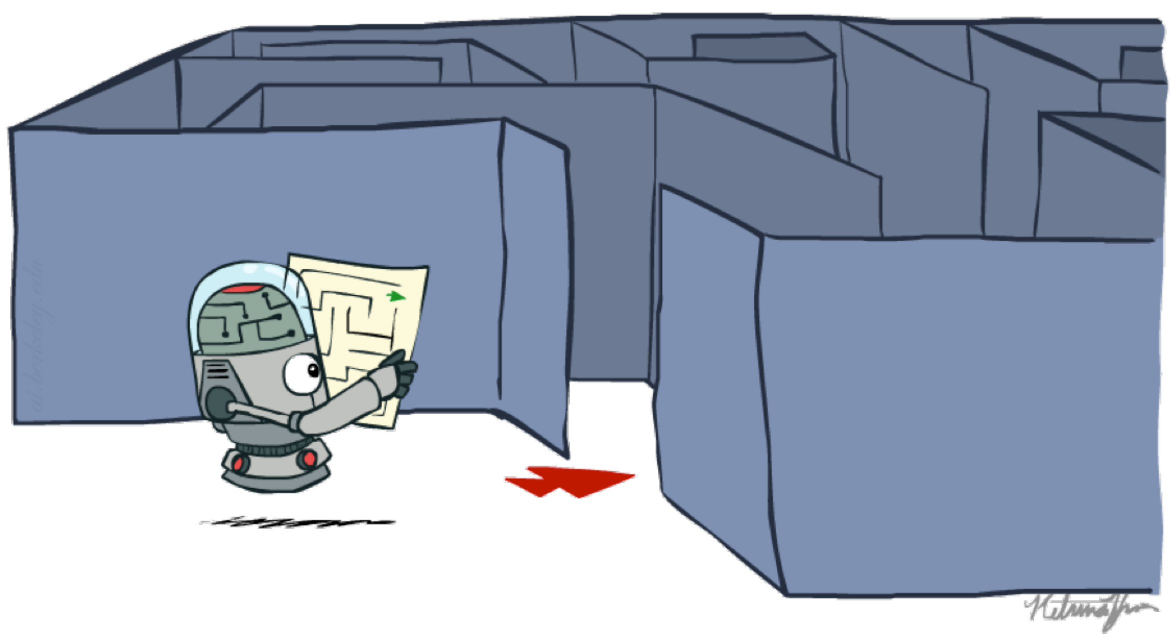
- Homework 1: Search
  - Part I AND Part II due Fri. May 21 at 11:59pm.
  - Part I through Online Judge.
  - Part II through Canvas -- submit pdf
- Project 1: Search
  - Released today. Due Mon May 31 at 11:59pm
  - Start early and ask questions. It's longer than most!
- Grading policy for HW and Project
  - 20% deduction per day of late submission
  - Drop 2 lowest grades for electronic HW part and 2 lowest grades for written HW part
  - 5 slip days for projects; maximum 2 slip days for a given project

Outline

- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search
- Graph Search

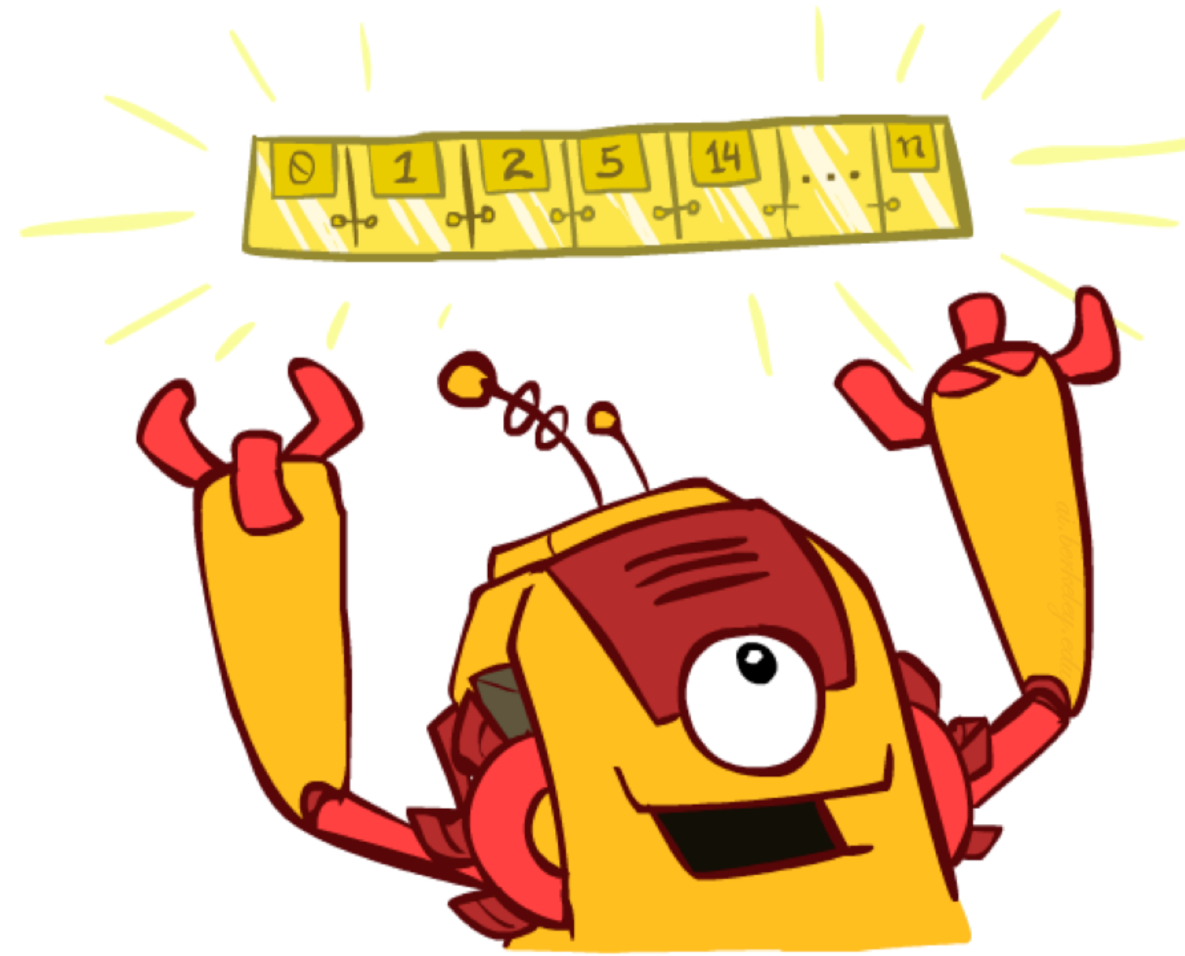
Recap: Search

- Search problem:
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)
- Search algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Optimal: finds least-cost plans



The One Queue

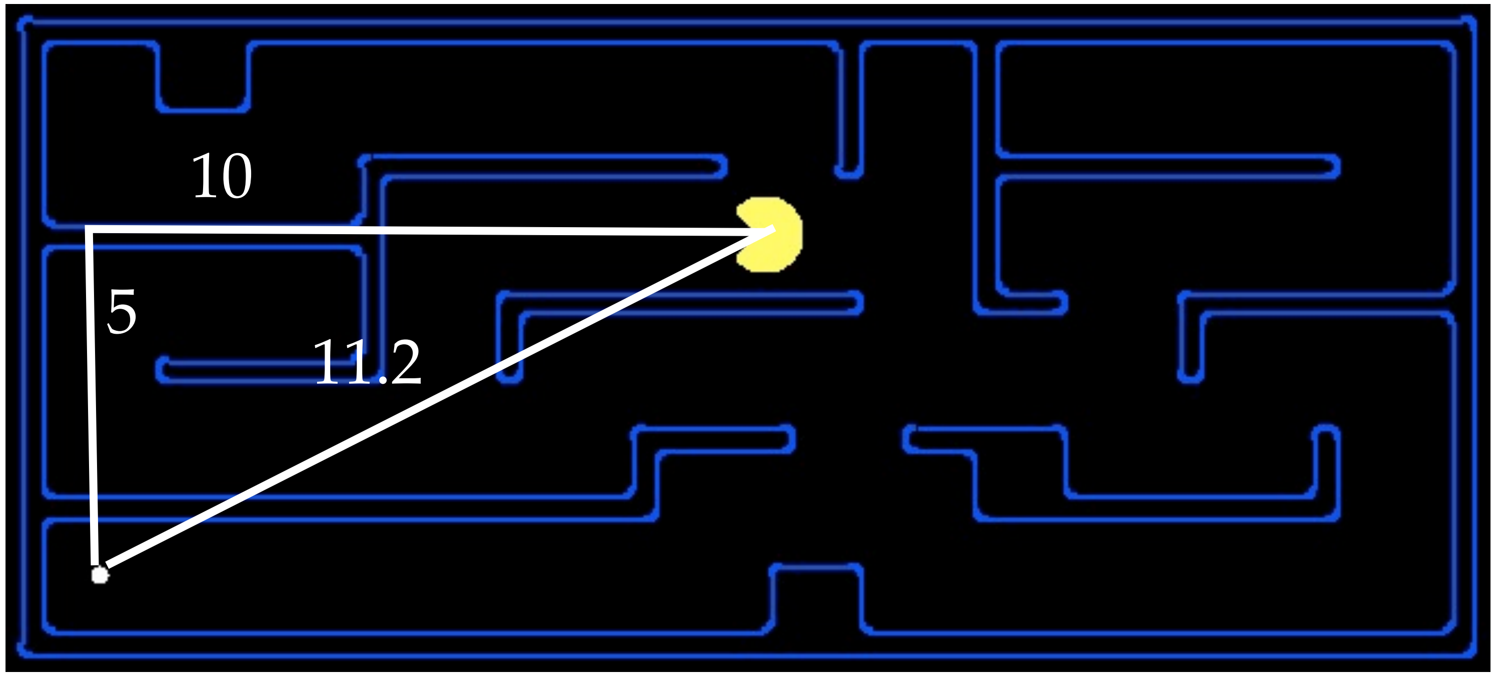
- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object

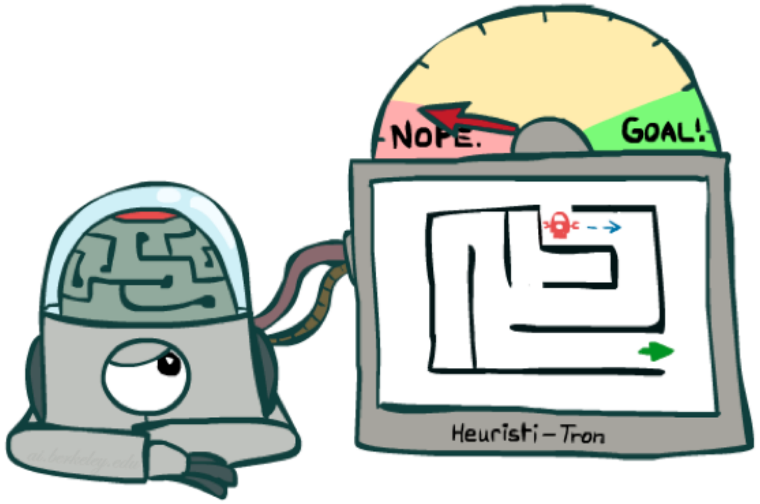


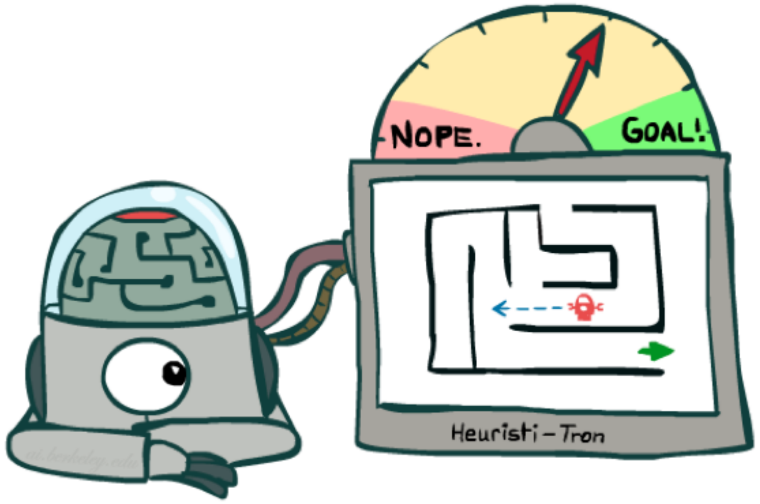
Informed Search

Search Heuristics

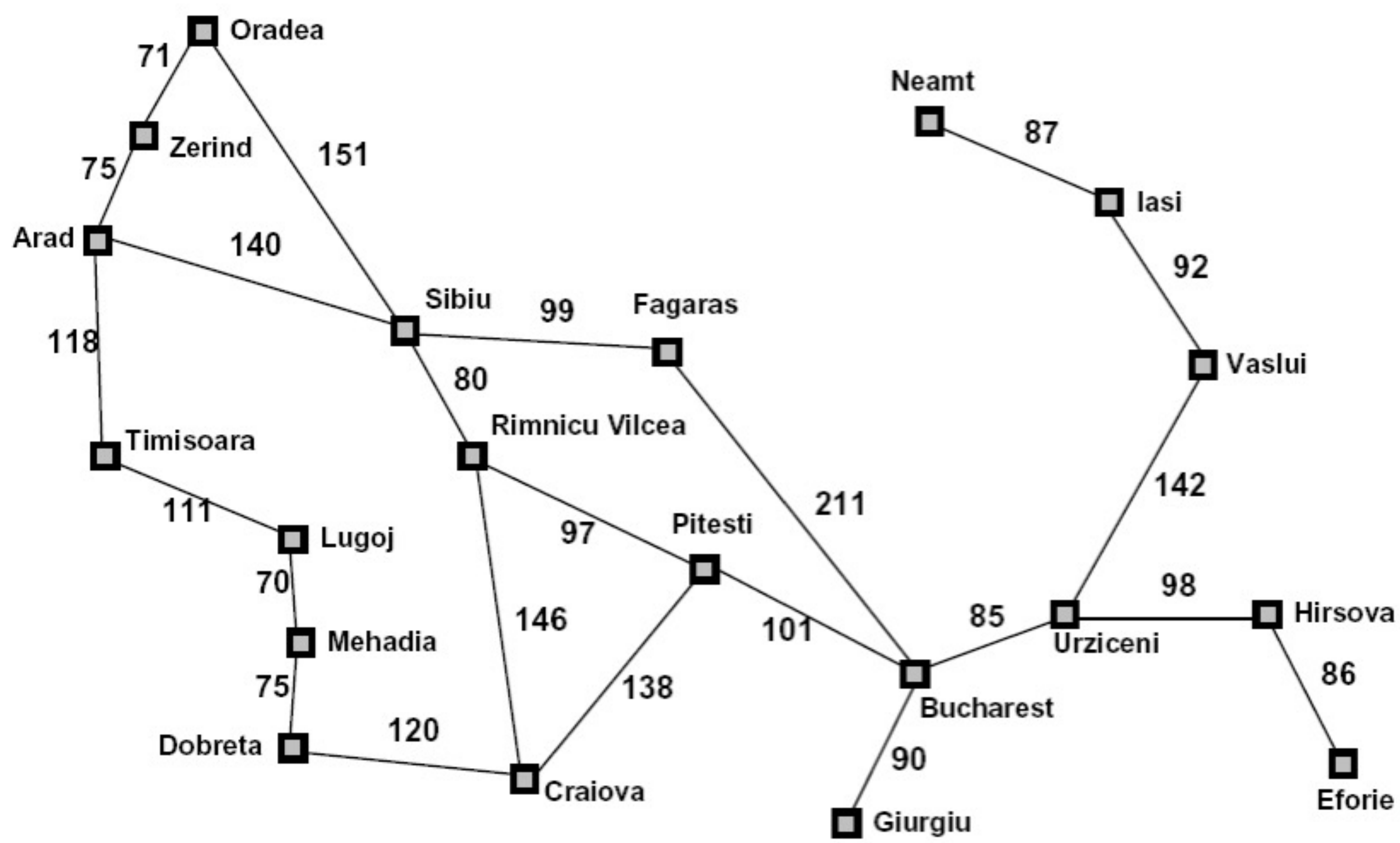
- A heuristic (function) is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for navigation







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

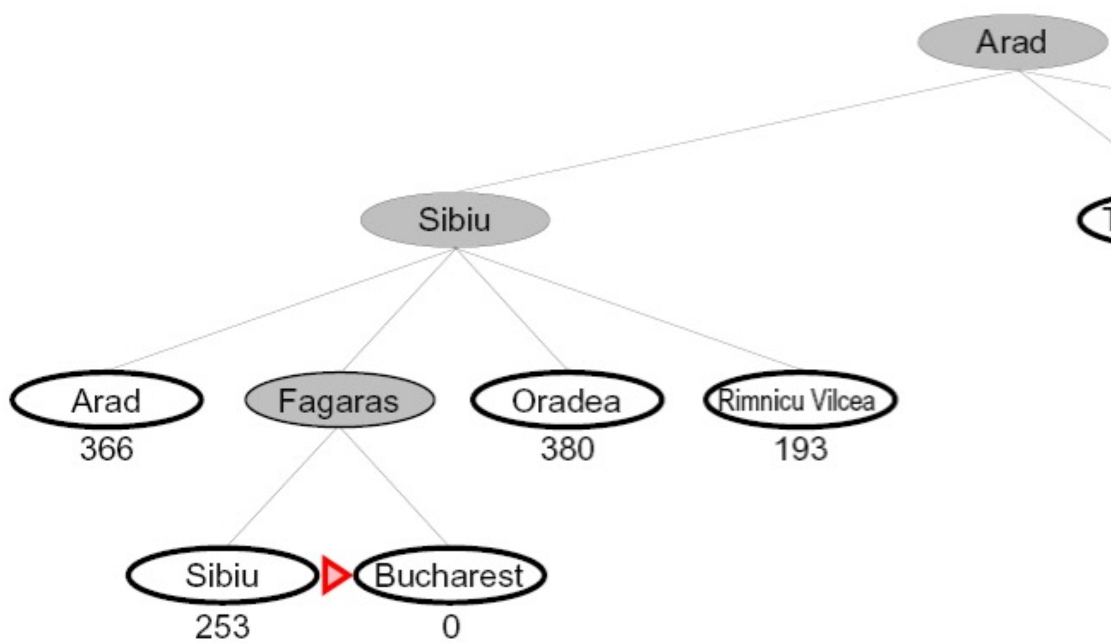
h(x)

Informed Search

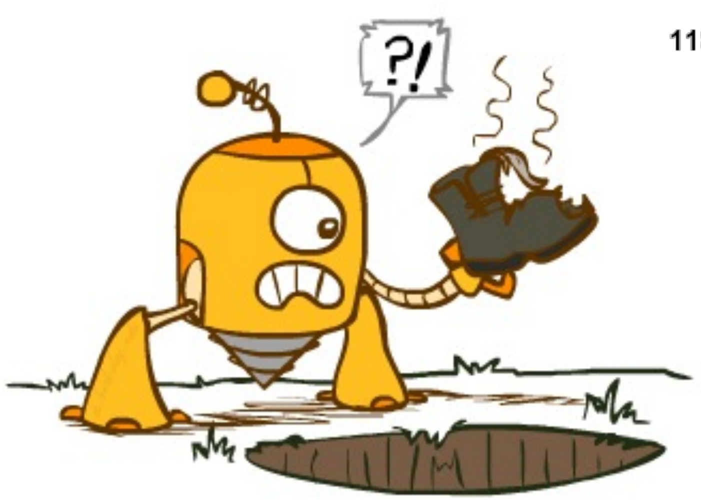
Greedy Search

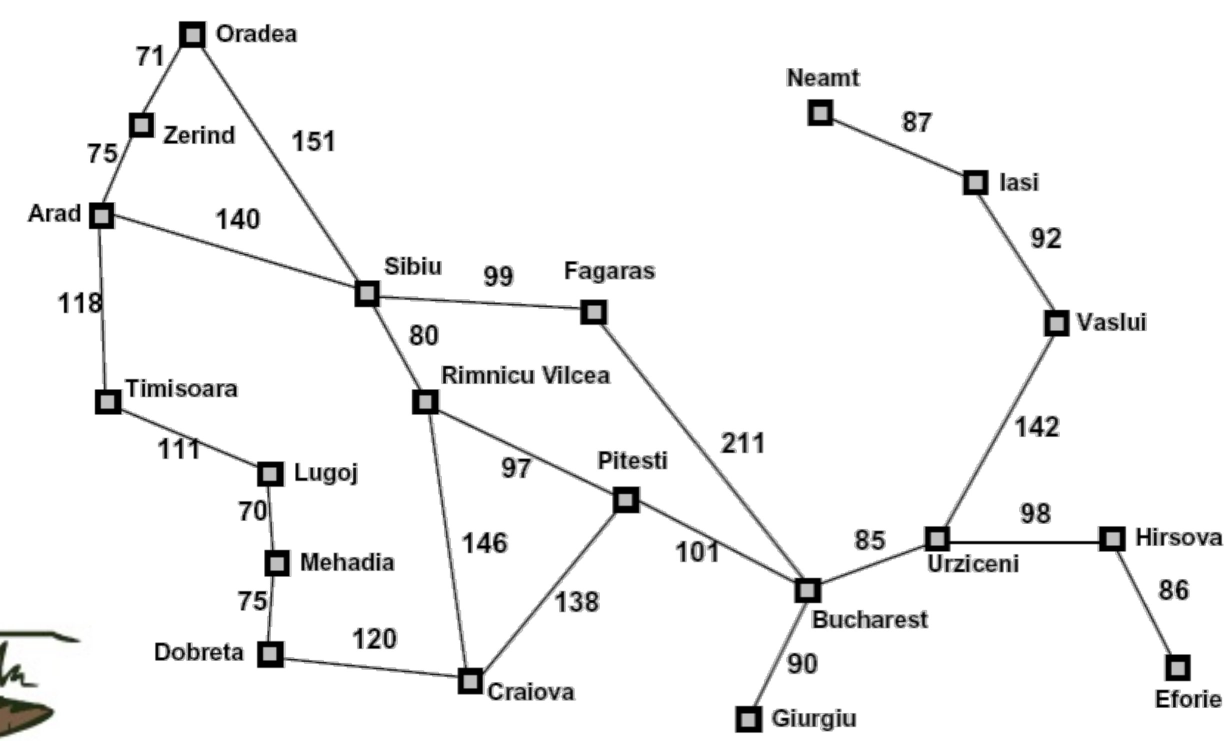
Greedy Search

- Expand the node that seems closest…



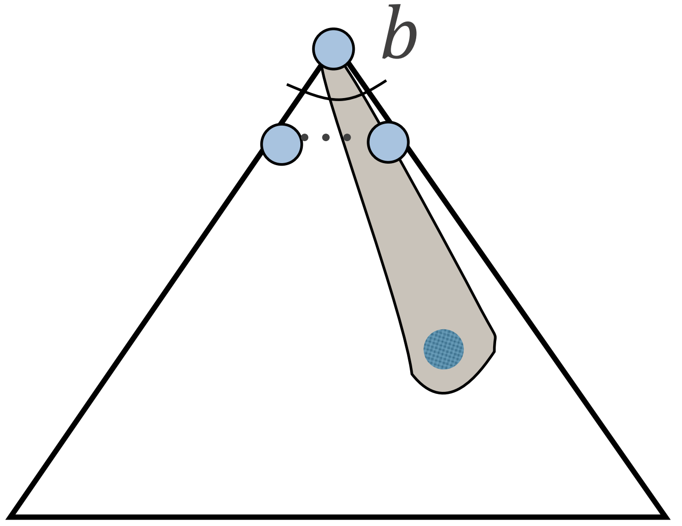
- What can go wrong?

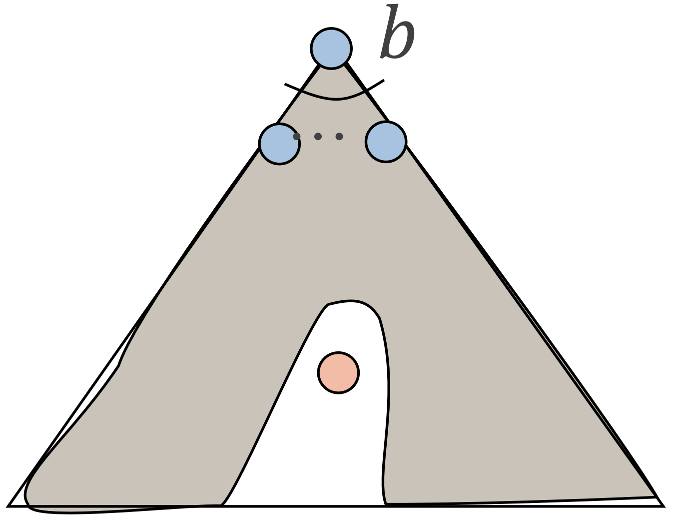




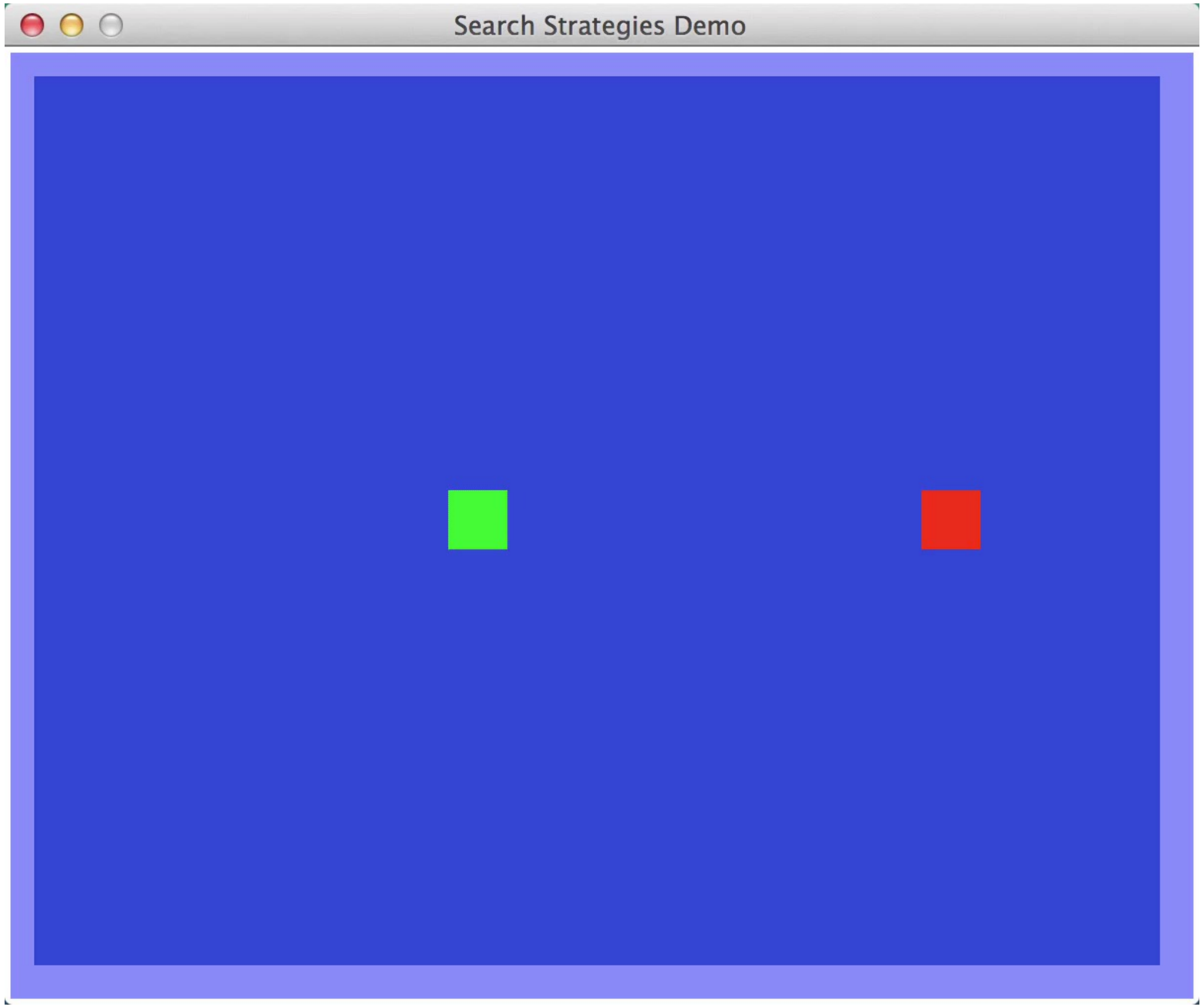
Greedy Search

- 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

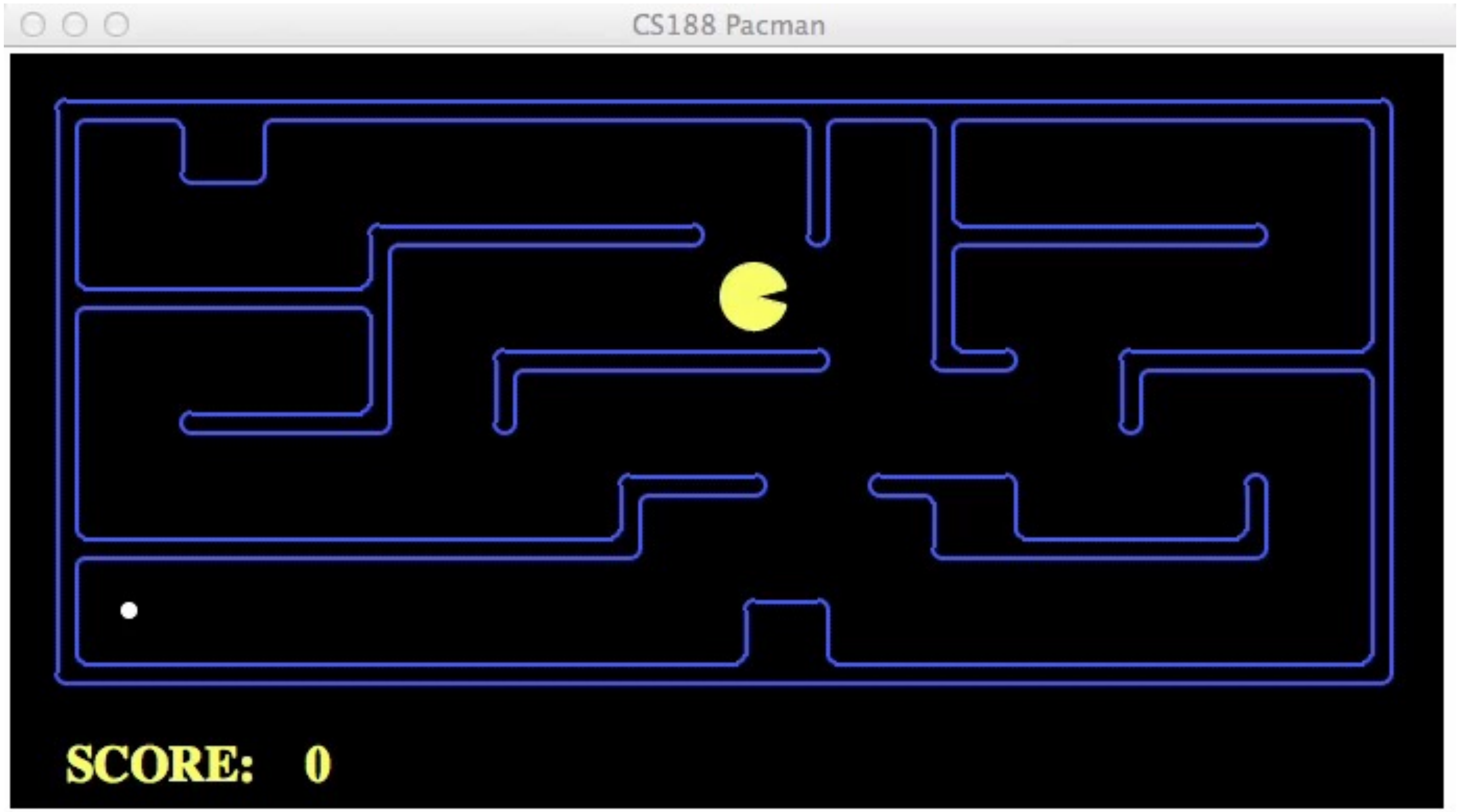




Video of Demo Contours Greedy (Empty)



Video of Demo Contours Greedy (Pacman Small Maze)



Informed Search

A* Search

A* Search

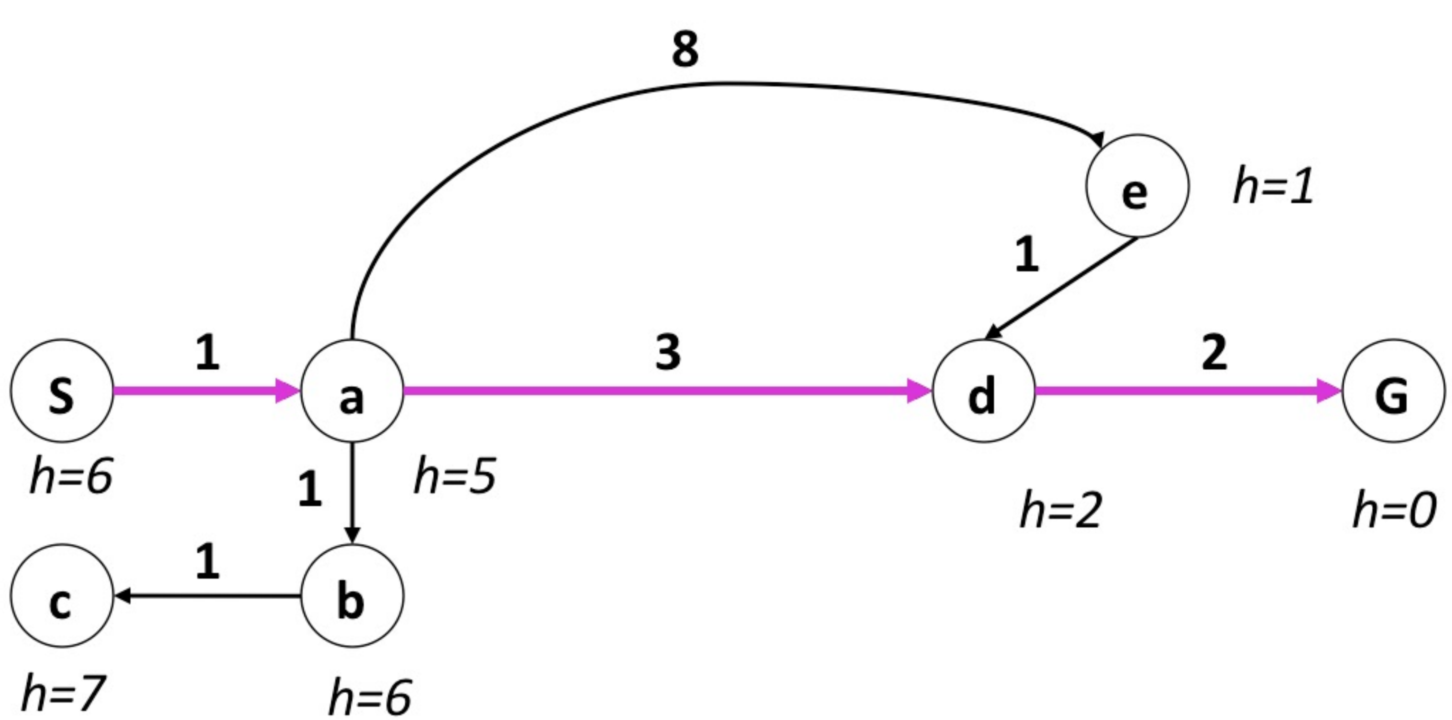
UCS

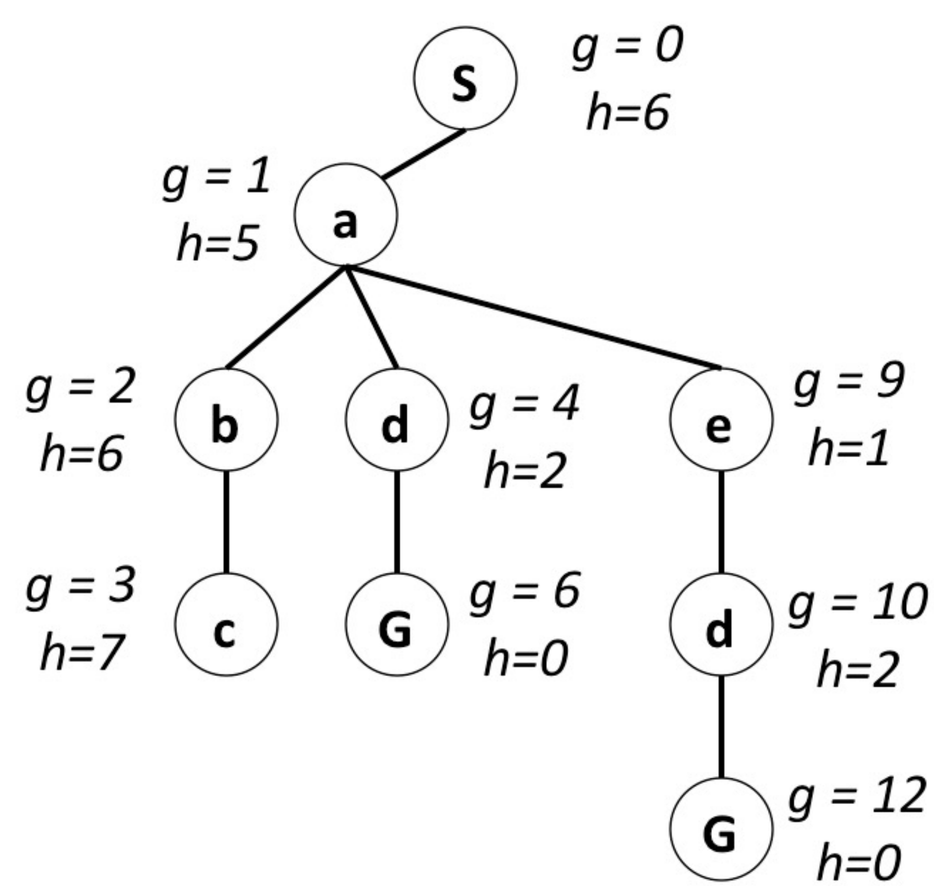
Greedy

A*

Combining UCS and Greedy

- 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)

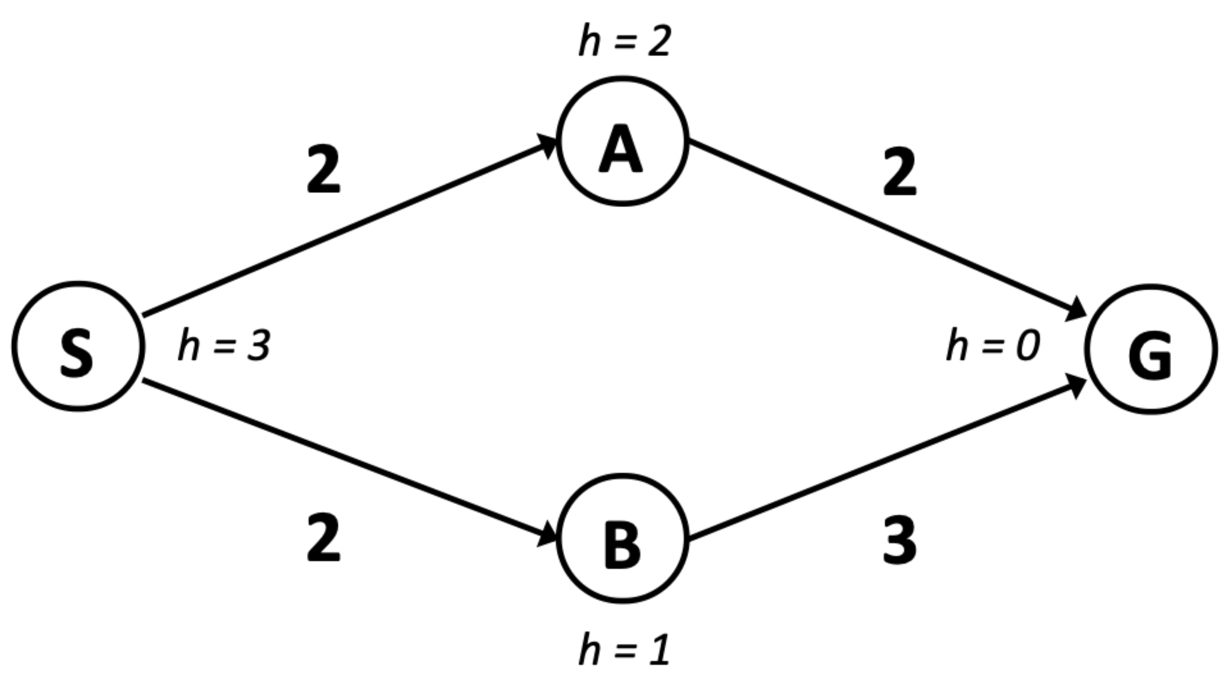




Example: Teg Grenager

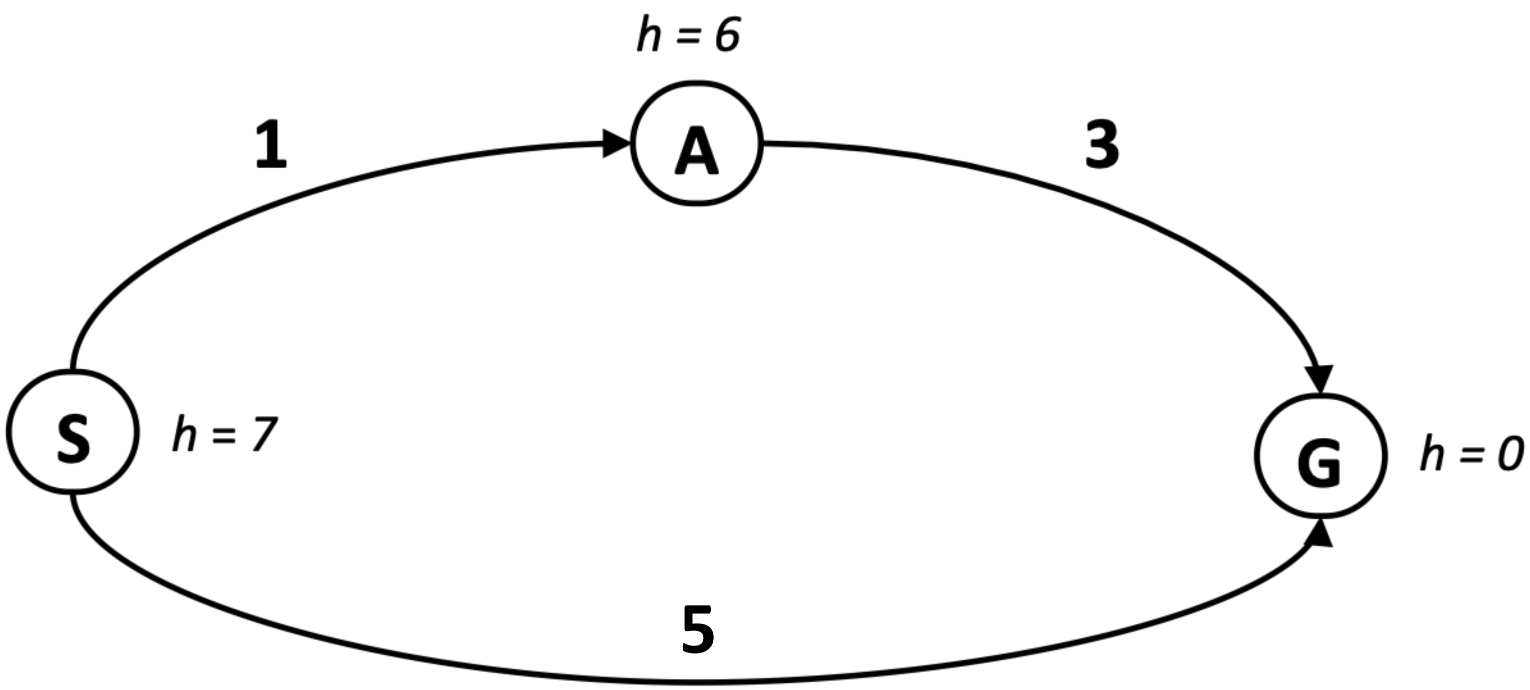
When should A* terminate?

- Should we stop when we enqueue a goal?



- No: only stop when we dequeue a goal

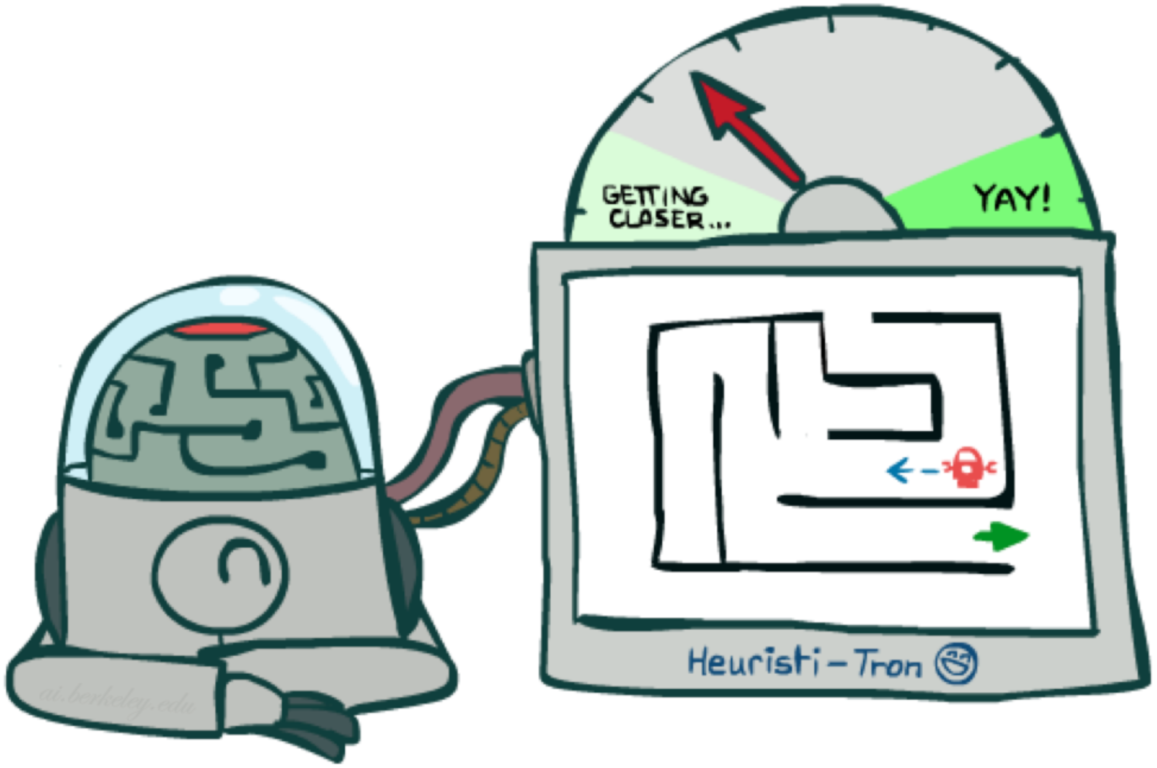
Is A* Optimal?



- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

*Informed Search: A**

Admissible Heuristics



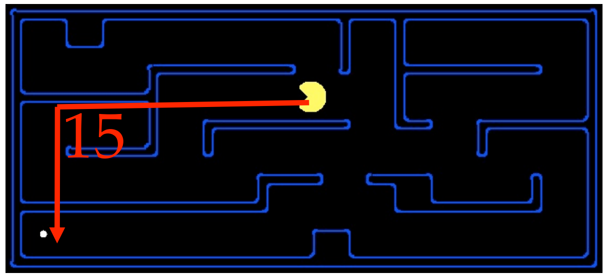
Admissible Heuristics

- 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:

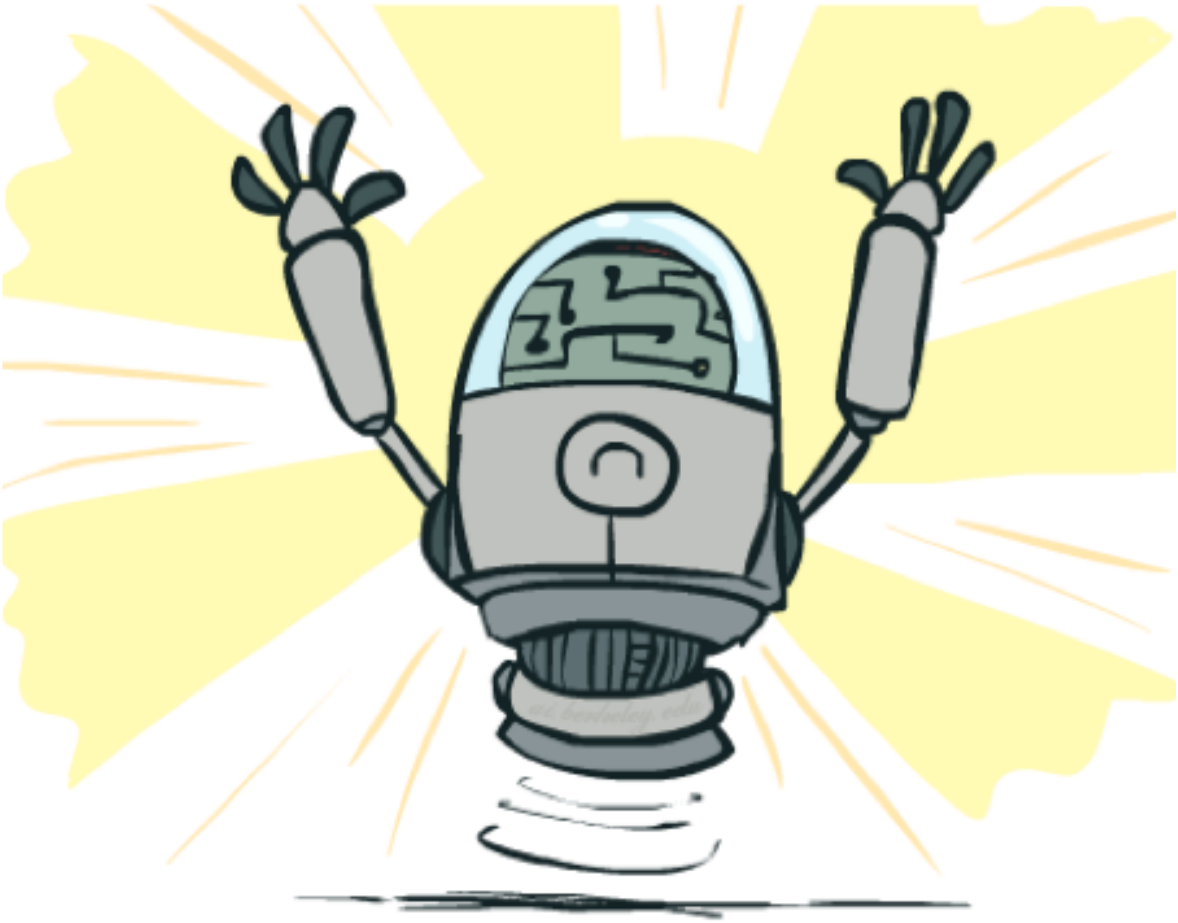


4

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

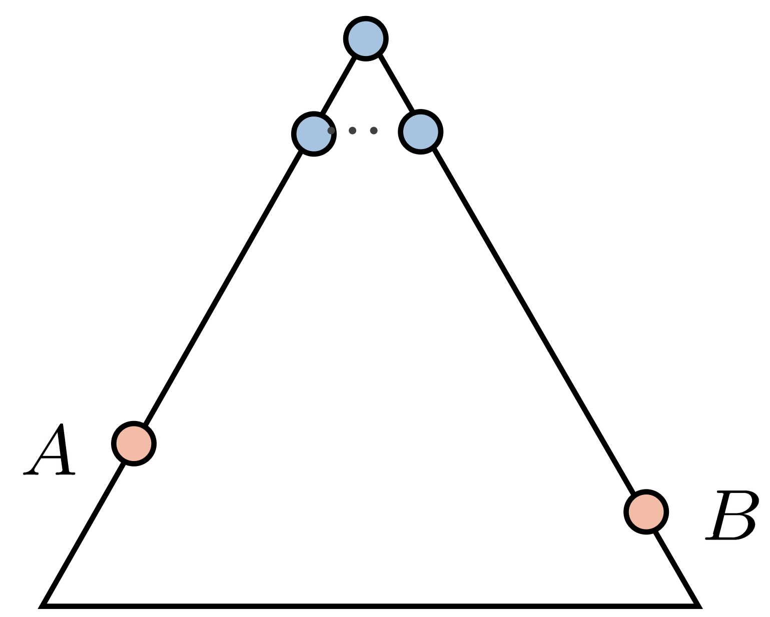
*Informed Search: A**

Optimality of A*
Tree Search



Optimality of A* Tree Search

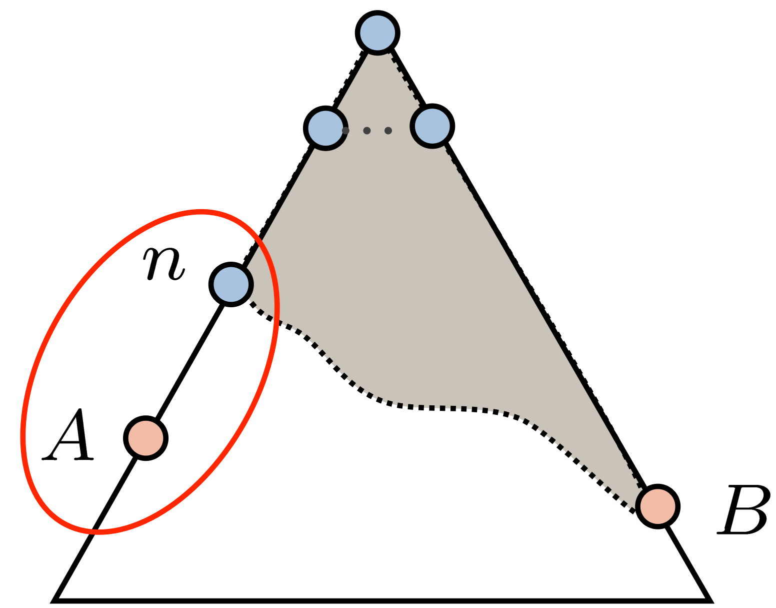
- Assume:
  - A is an optimal goal node
  - B is a suboptimal goal node
  - h is admissible
- Claim:
  - A will exit the fringe before B



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
  - f(n) is less or equal to f(A)

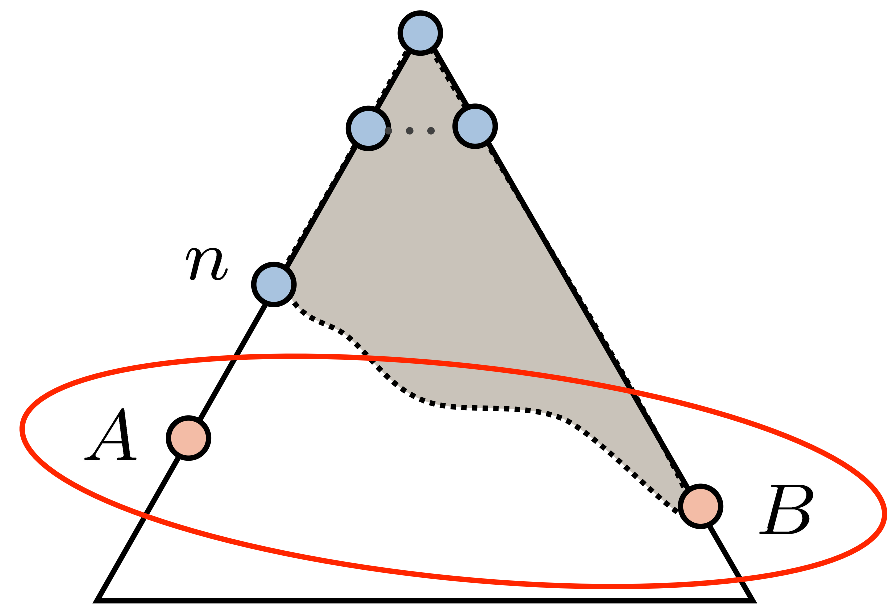


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

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
  - f(n) is less or equal to f(A)
  - f(A) is less than f(B)



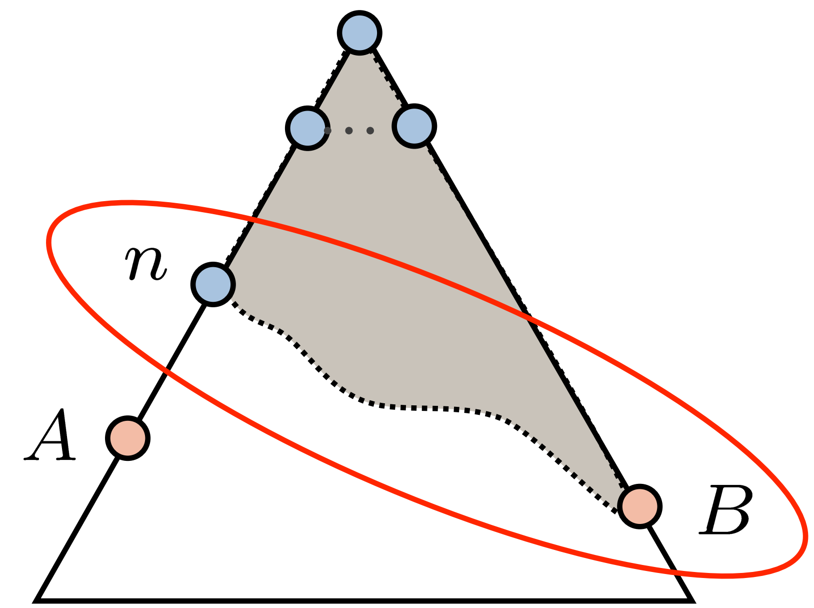
$g(A) < g(B)$ B is suboptimal

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

Optimality of A* Tree Search: Blocking

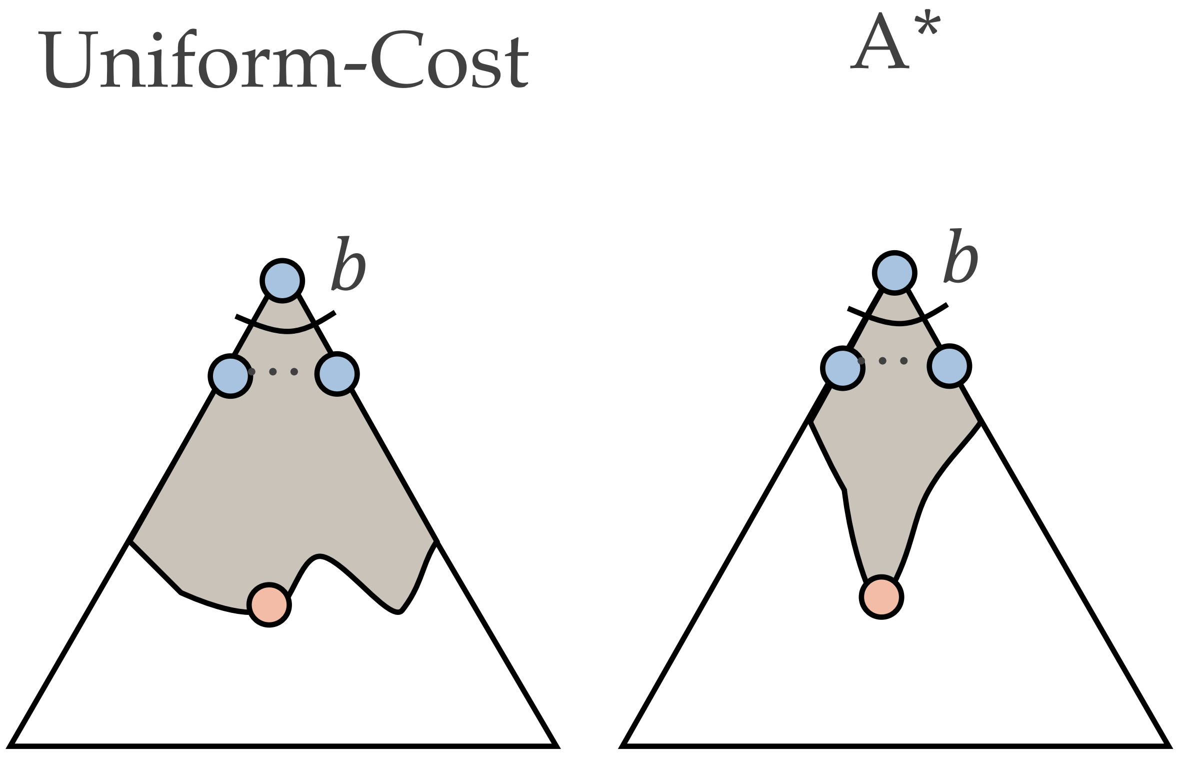
Proof:

- 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
  - f(n) is less or equal to f(A)
    - f(A) is less than f(B)
    - n expands before B
  - All ancestors of A expand before B
  - A expands before B
  - A* search is optimal



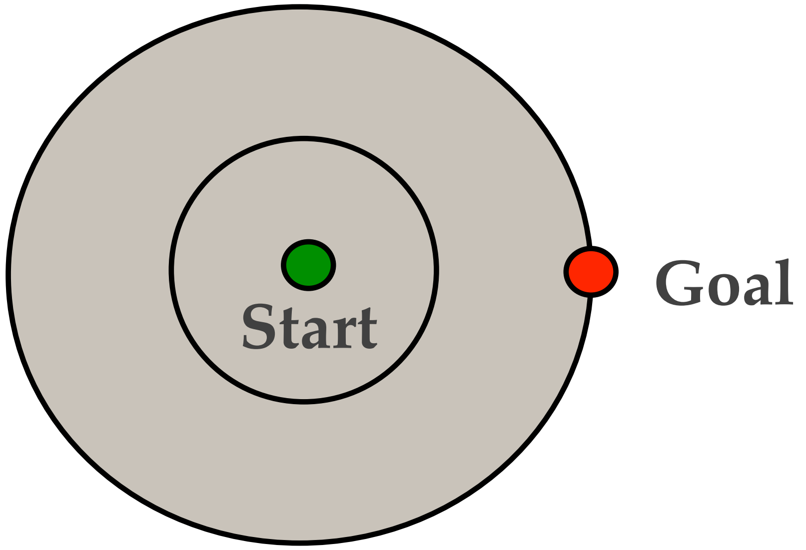
$$f(n) \leq f(A) < f(B)$$

Properties of A*

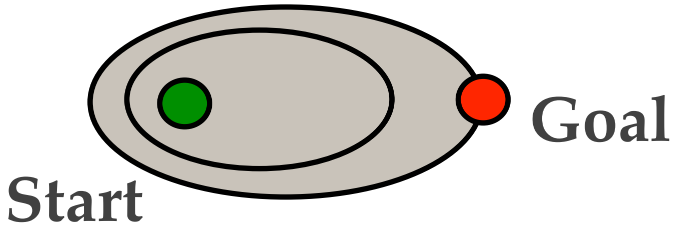


UCS vs A* Contours

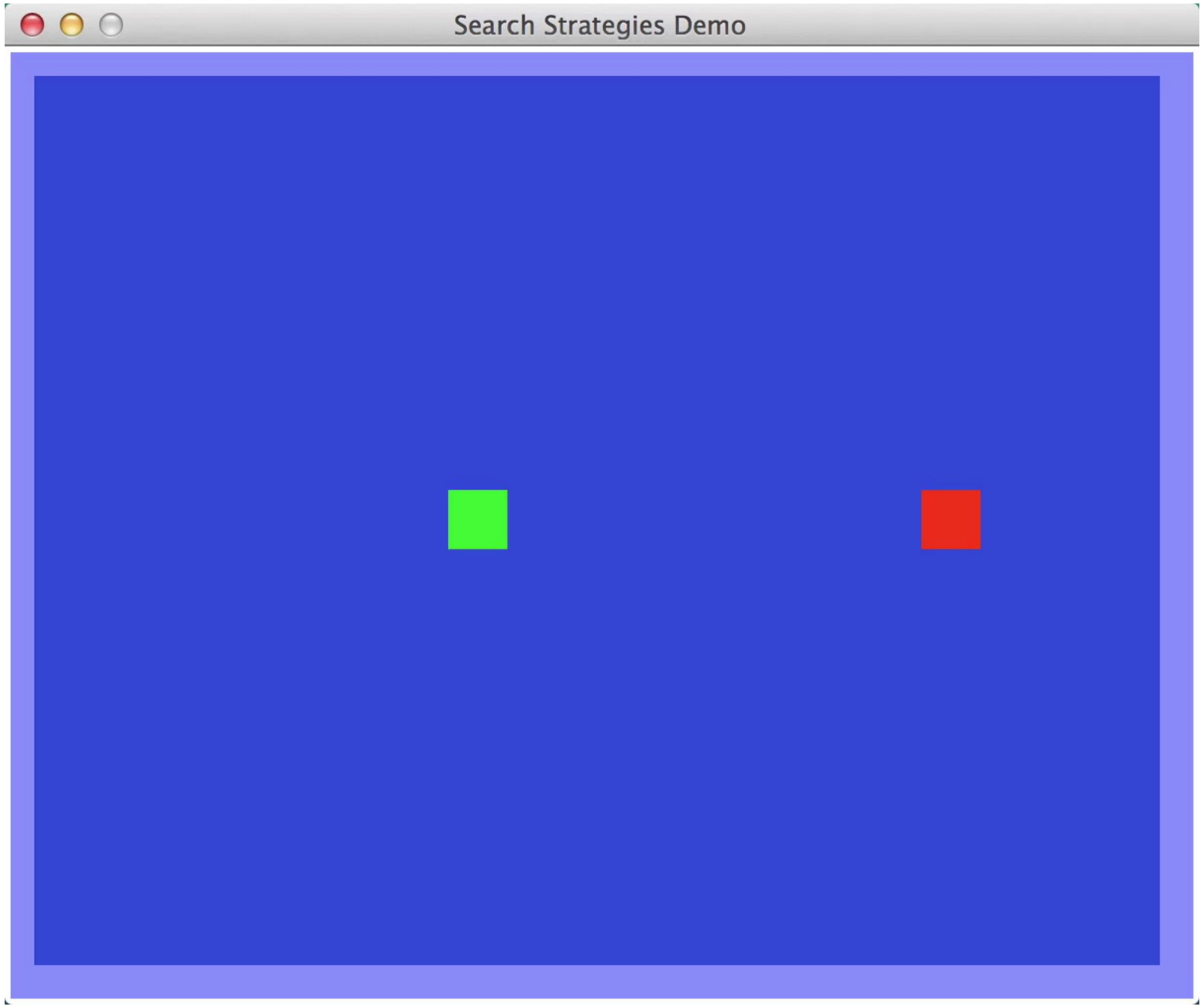
- Uniform-cost expands equally in all "directions"



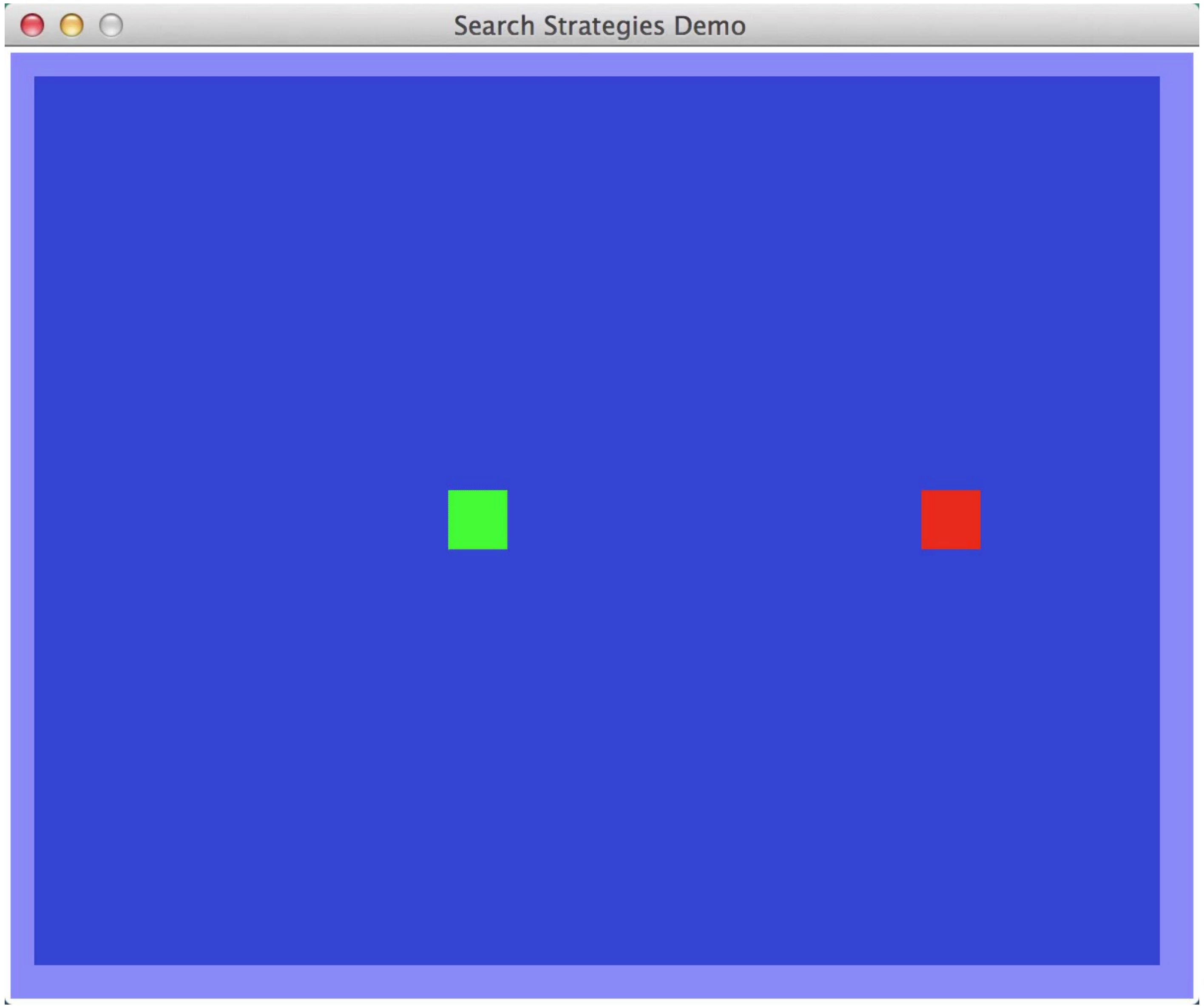
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



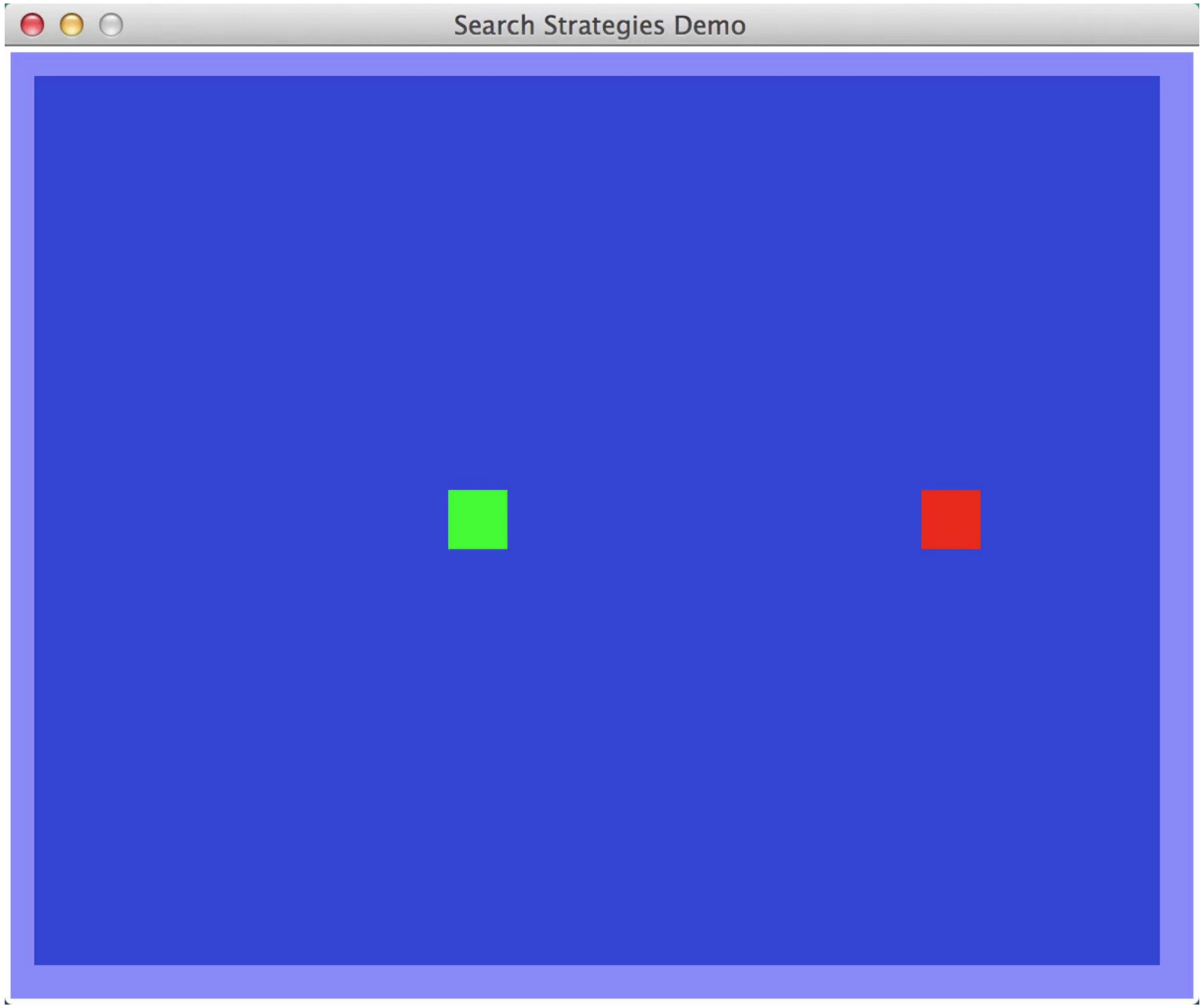
Video of Demo Contours (Empty) -- UCS



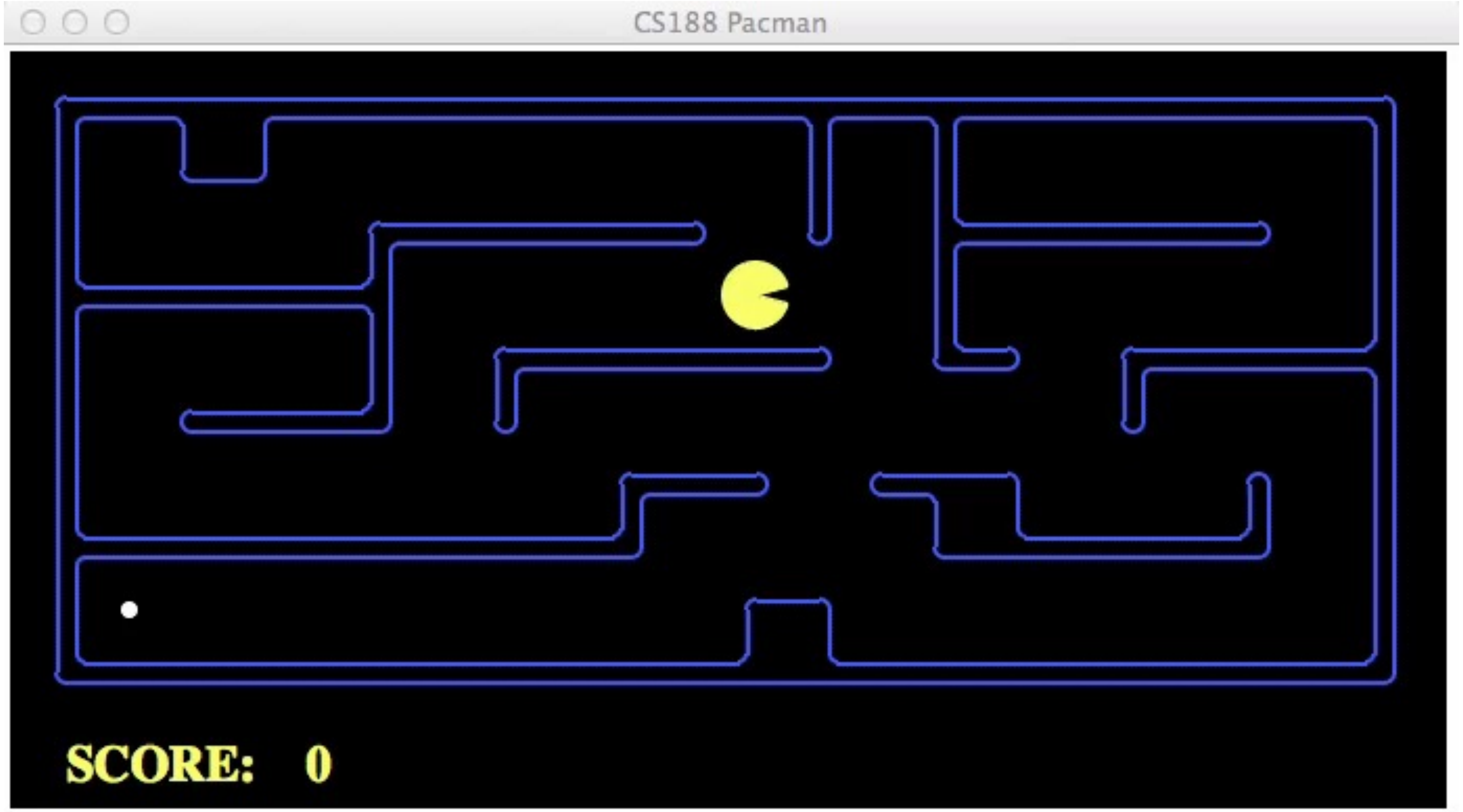
Video of Demo Contours (Empty) -- Greedy



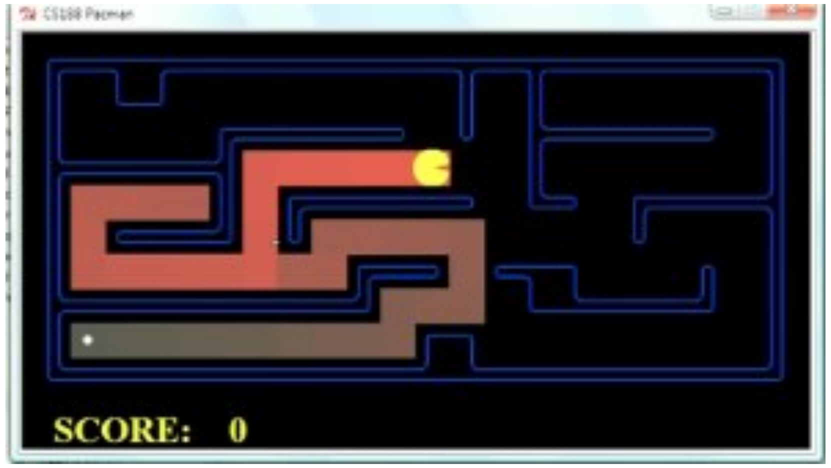
Video of Demo Contours (Empty) – A*



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



Comparison



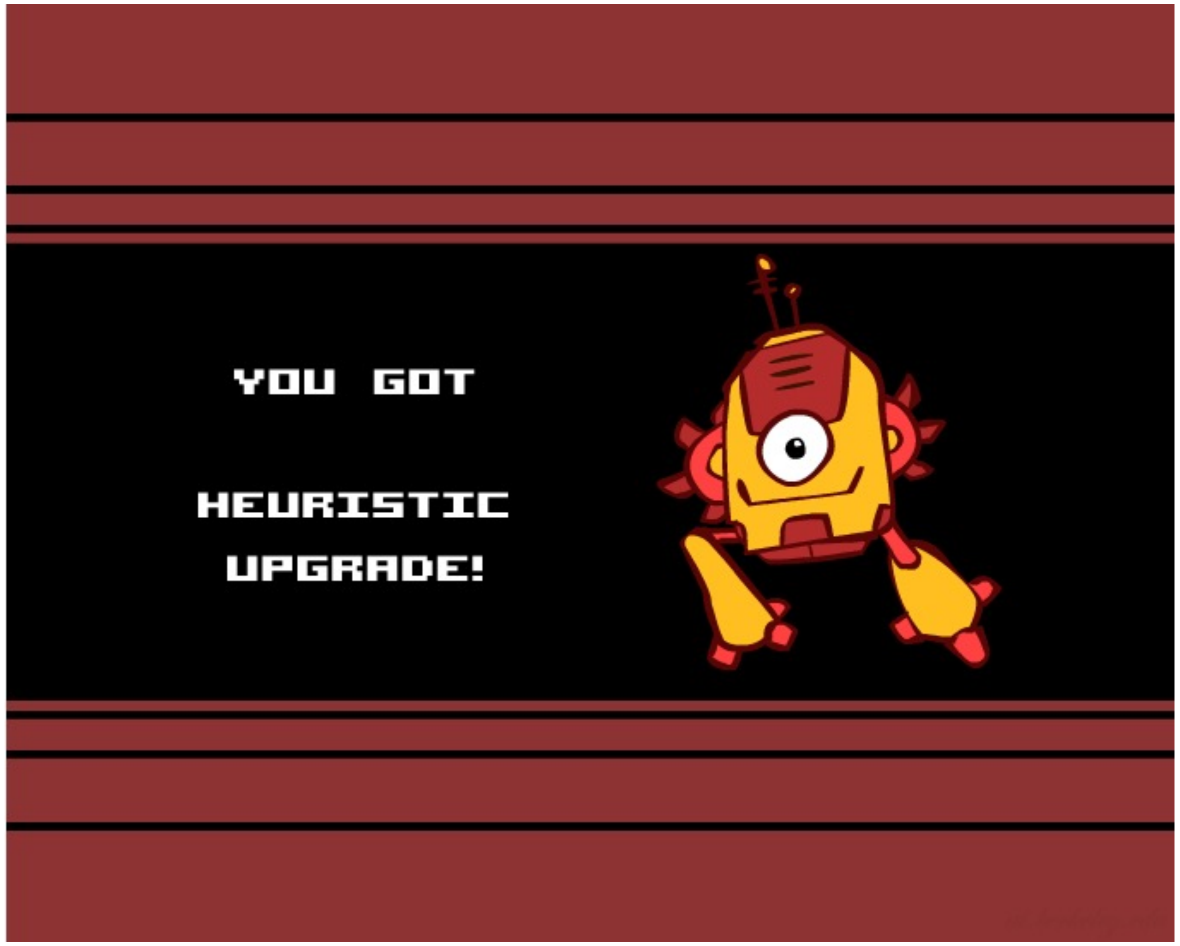
Greedy

Uniform Cost

A*

*Informed Search: A**

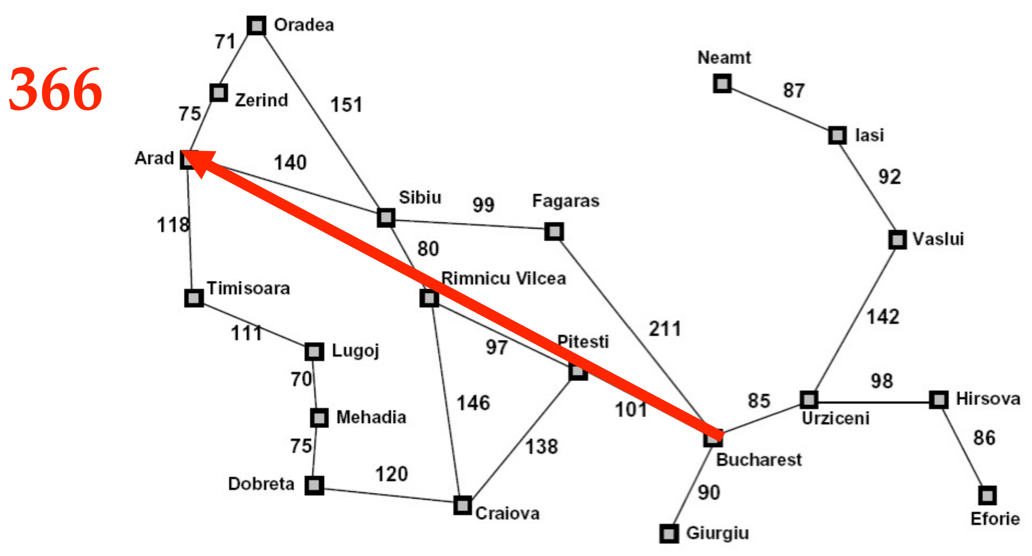
Creating Heuristics

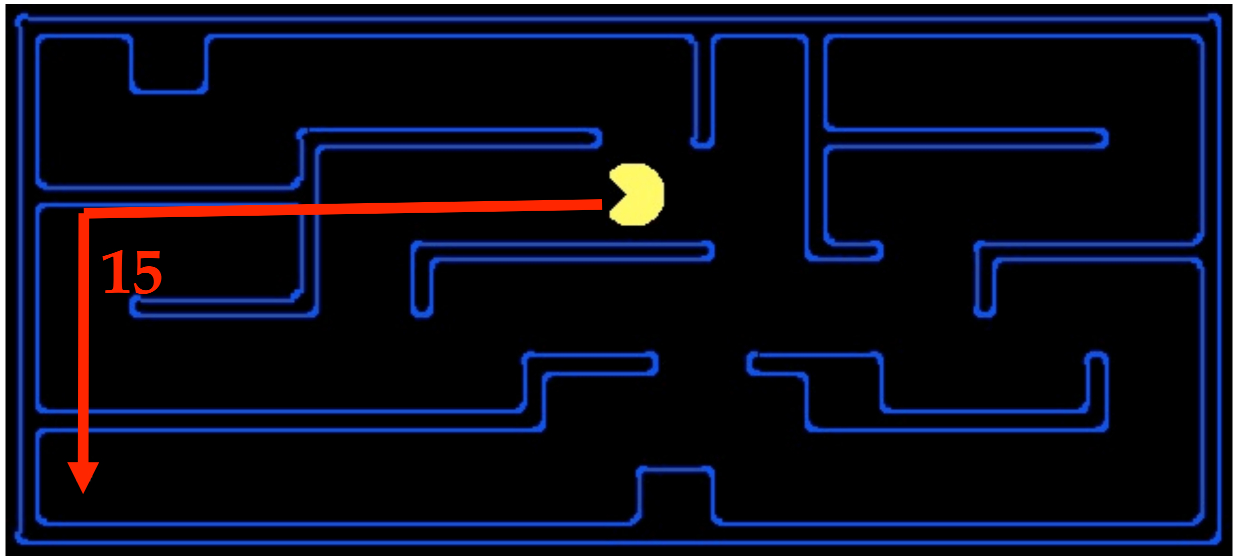


Creating Admissible Heuristics

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

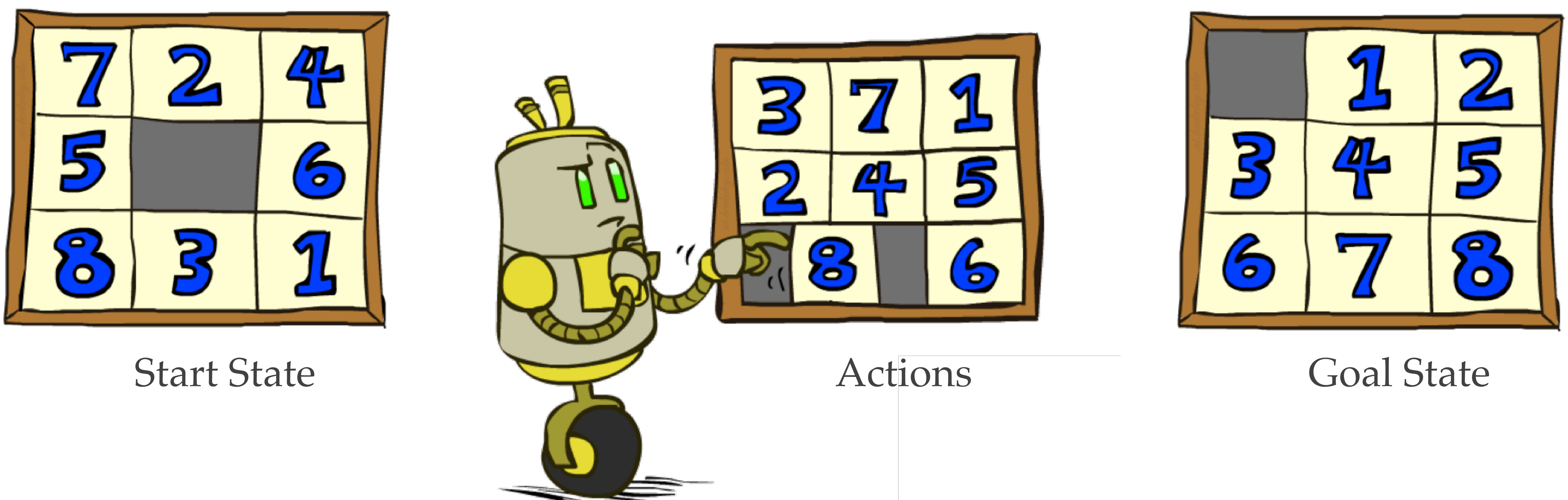
- Often, admissible heuristics are solutions to relaxed problems, where new actions are available





- Inadmissible heuristics are often useful too

Example: 8-Puzzle

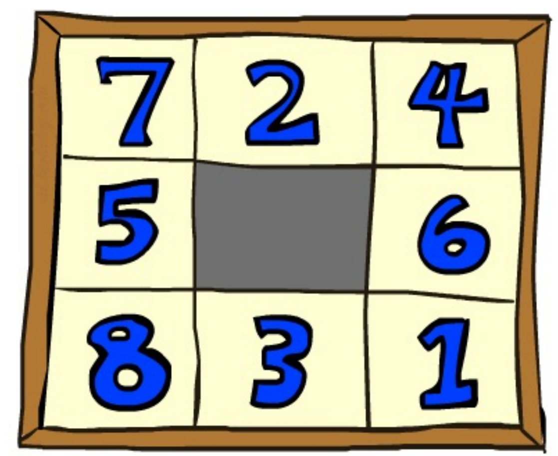


Start State Actions Goal State

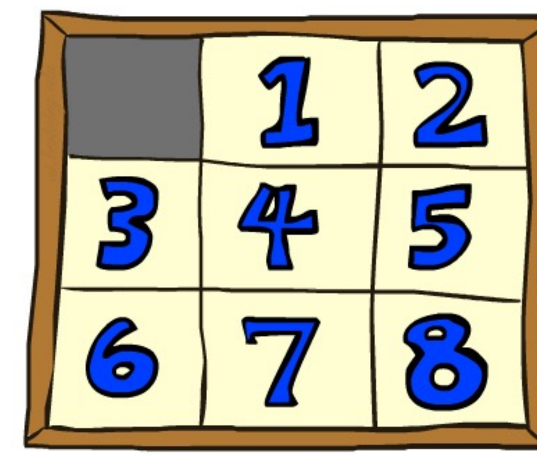
- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

8-Puzzle I

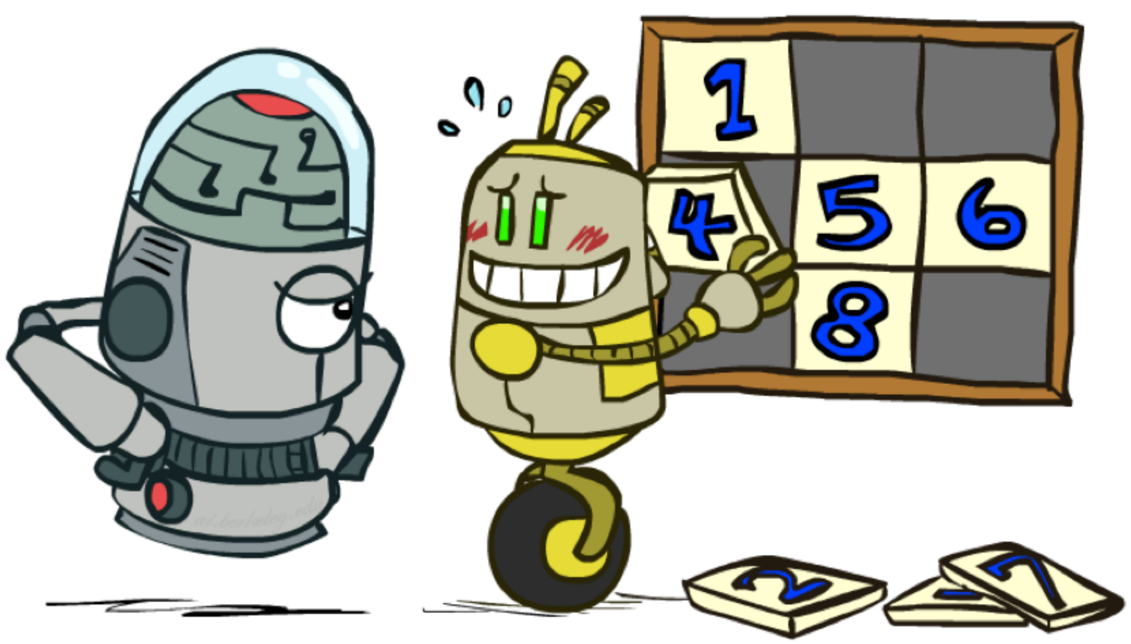
- Heuristic: Number of tiles misplaced
- h(start) = 8
- Why is it admissible?
- This is a relaxed-problem heuristic



Start State



Goal State

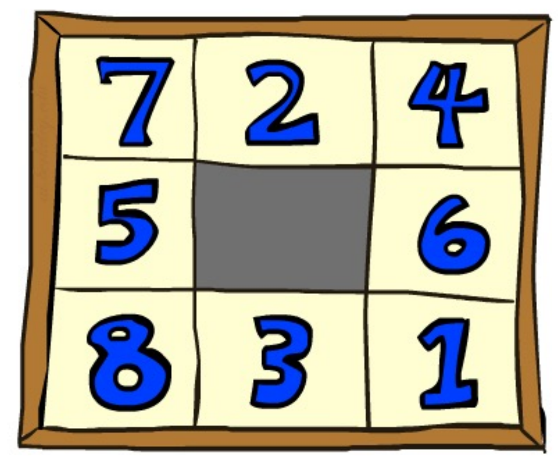


	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

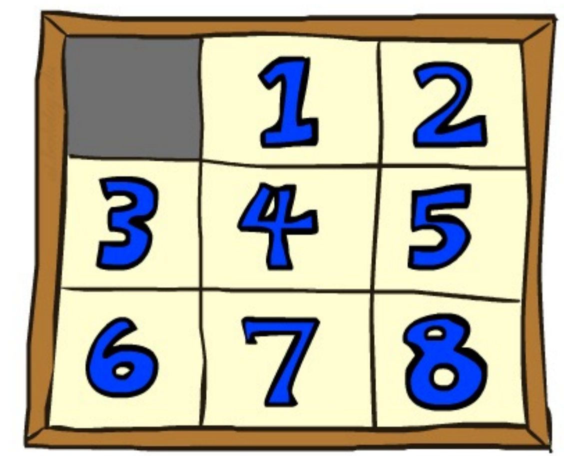
Statistics from Andrew Moore

8-Puzzle II

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



Start State

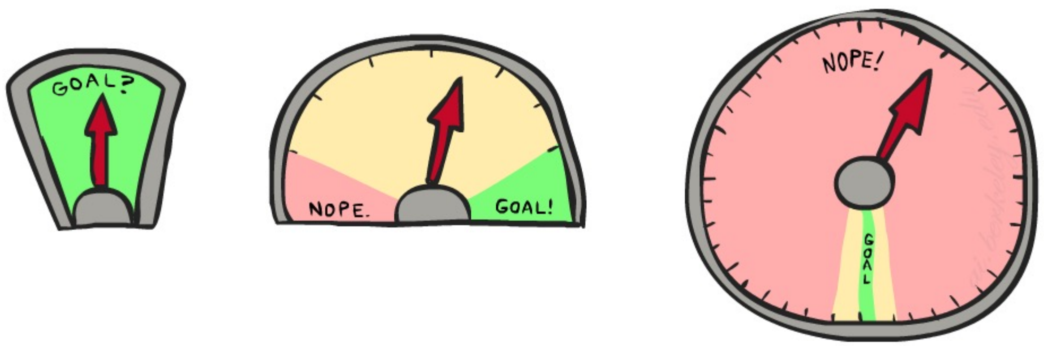


Goal State

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

8-Puzzle III

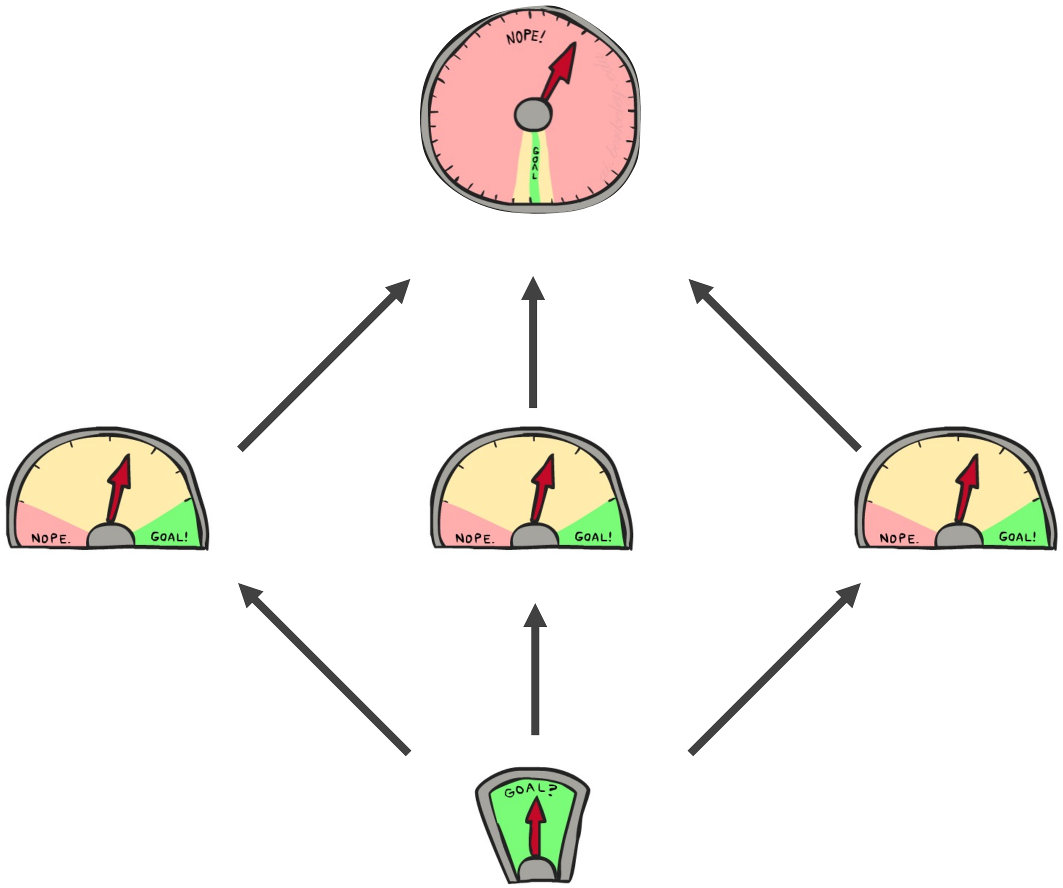
- How about using the actual cost as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



- 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

*Informed Search: A**

Semi-Lattice of Heuristics



Trivial Heuristics, Dominance

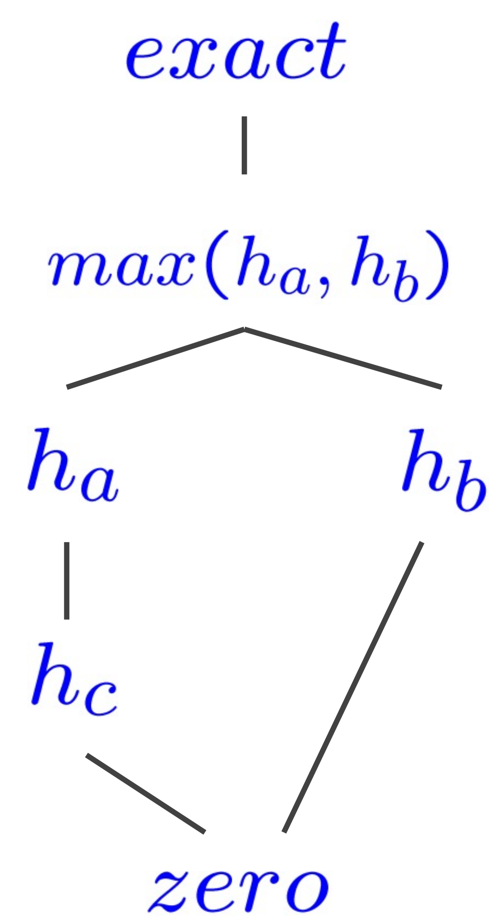
- Dominance: $h_a \geq h_c$ if

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

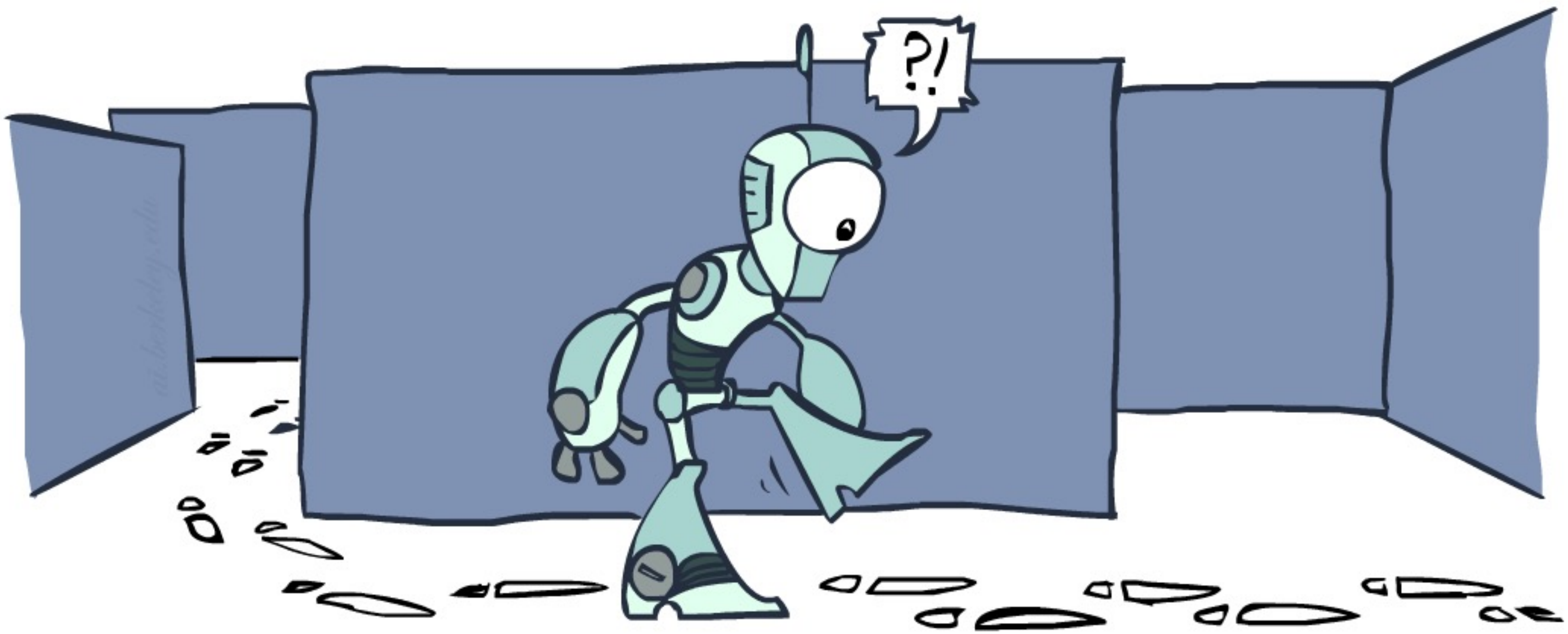
- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible

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

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

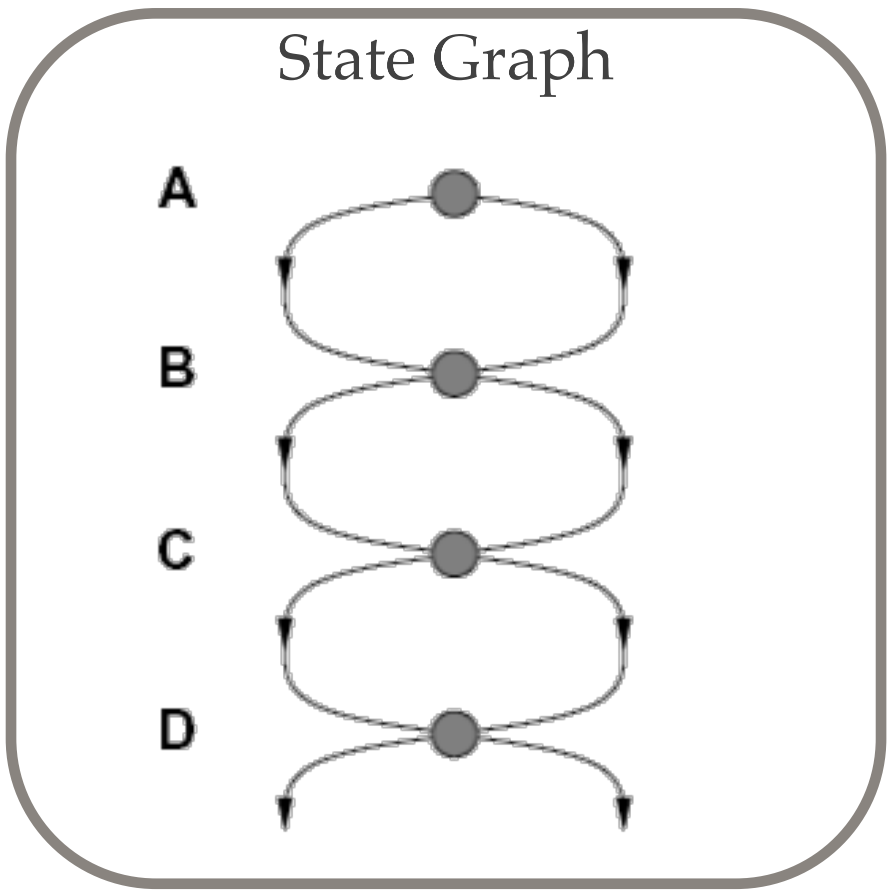


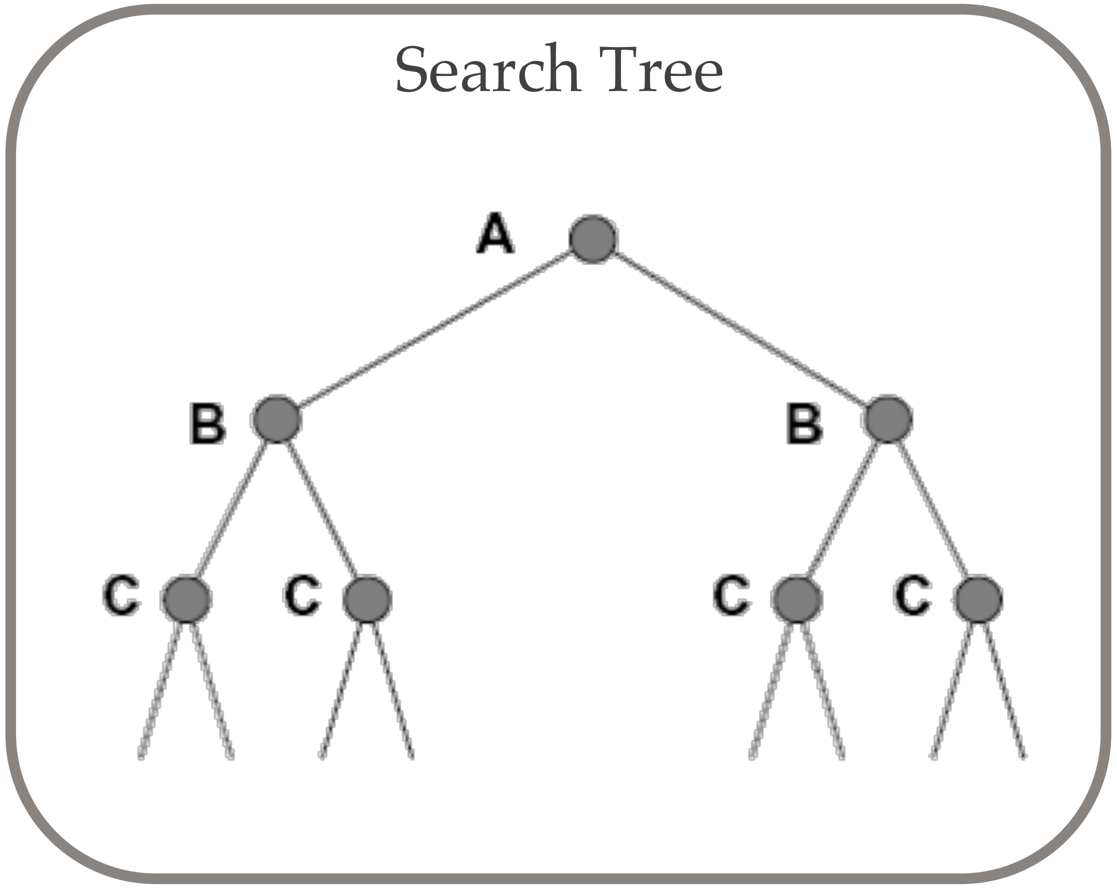
Graph Search



Tree Search: Extra Work!

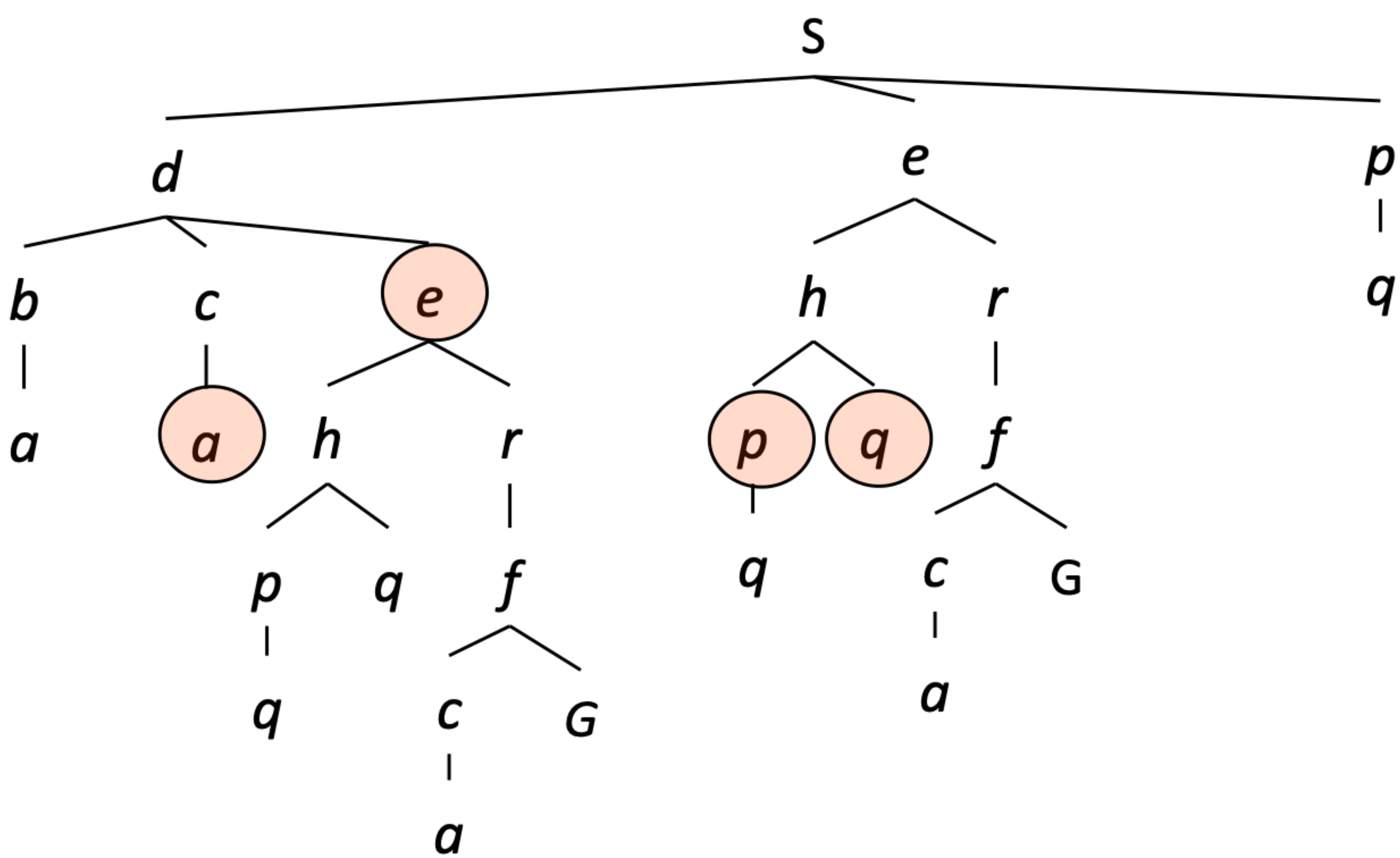
- Failure to detect repeated states can cause exponentially more work.





Graph Search

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)

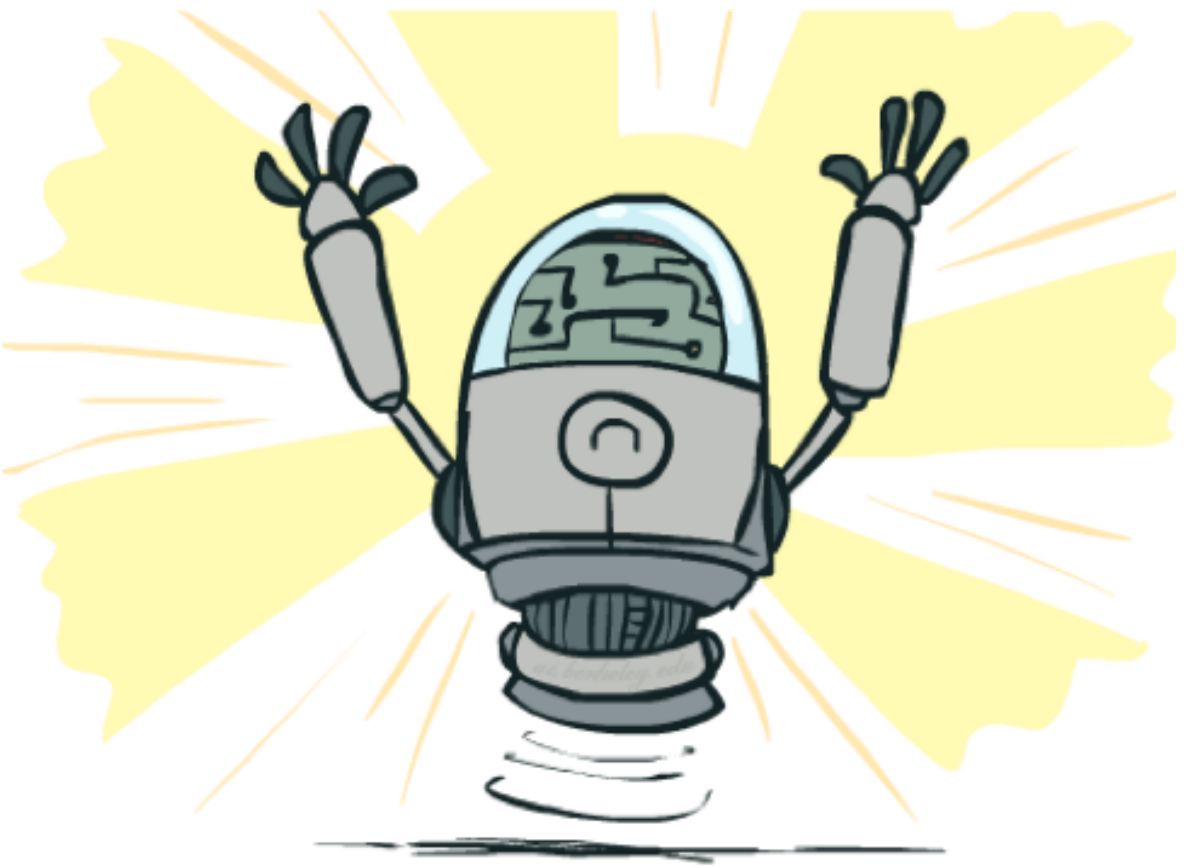


Graph Search

- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

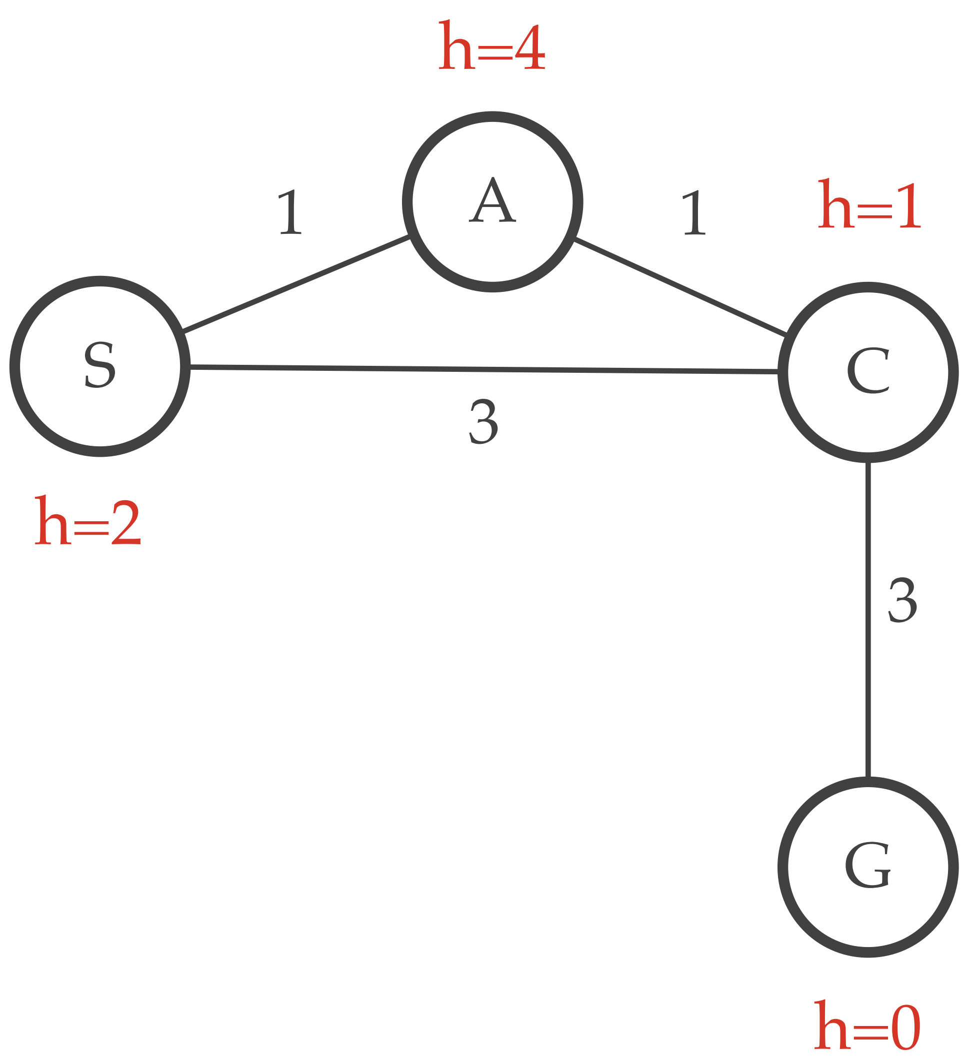
Graph Search

Optimality of A* Graph Search

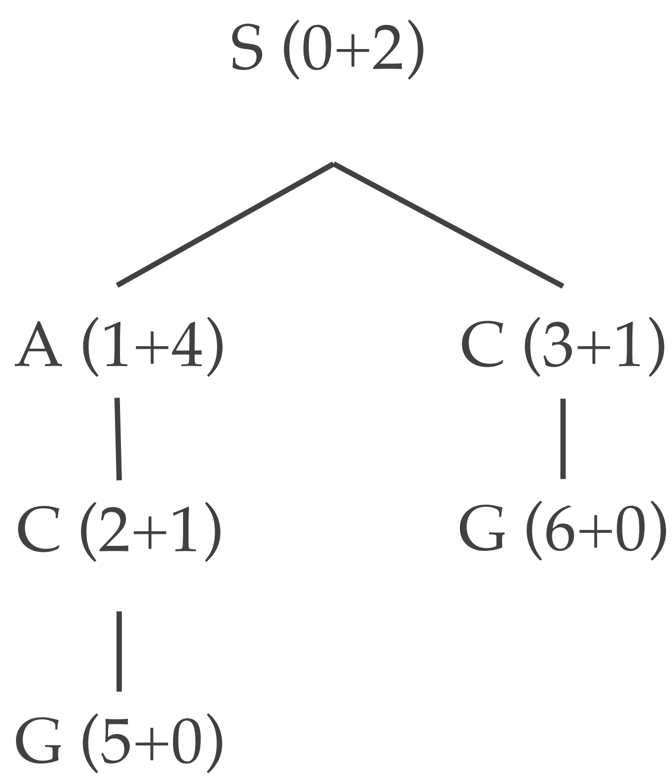


A* Tree Search

State space graph

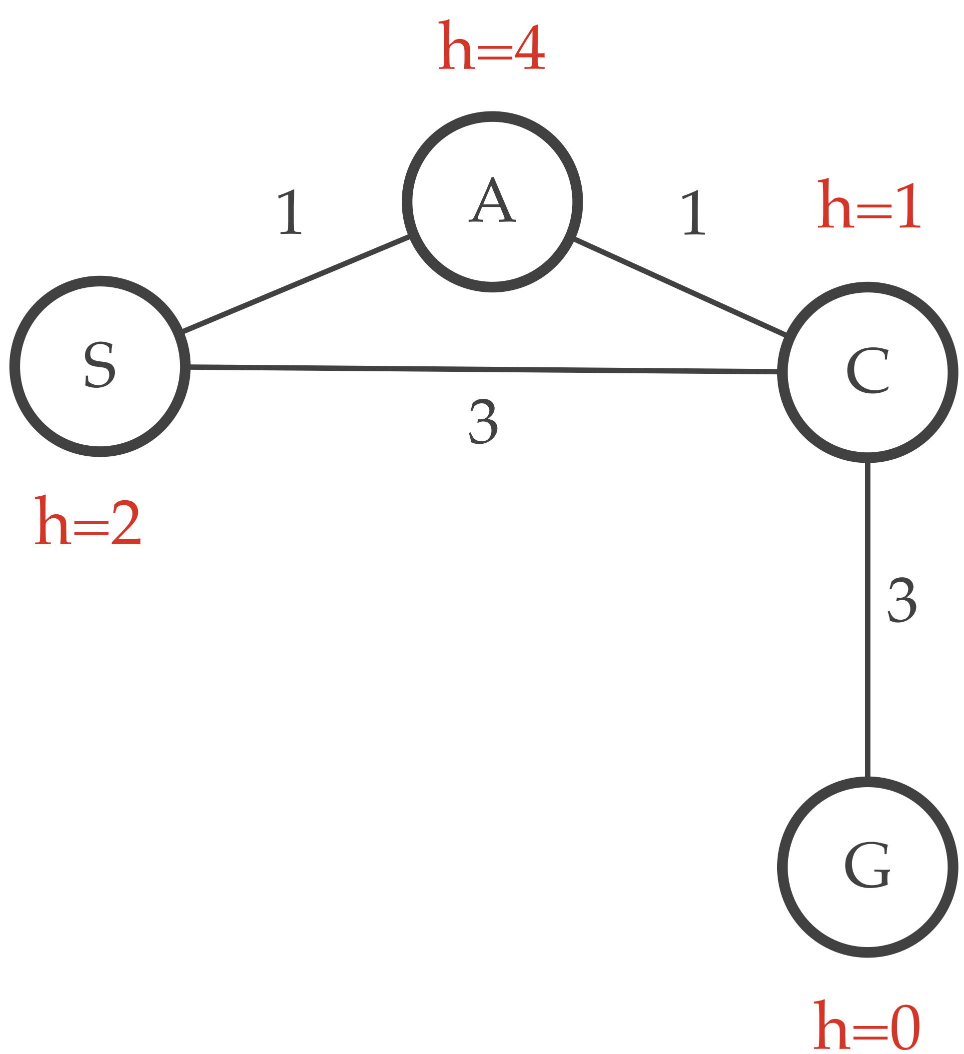


Search tree



Quiz: A* Graph Search

What paths does A* graph search consider during its search?



A) S, S-A, S-C, S-C-G

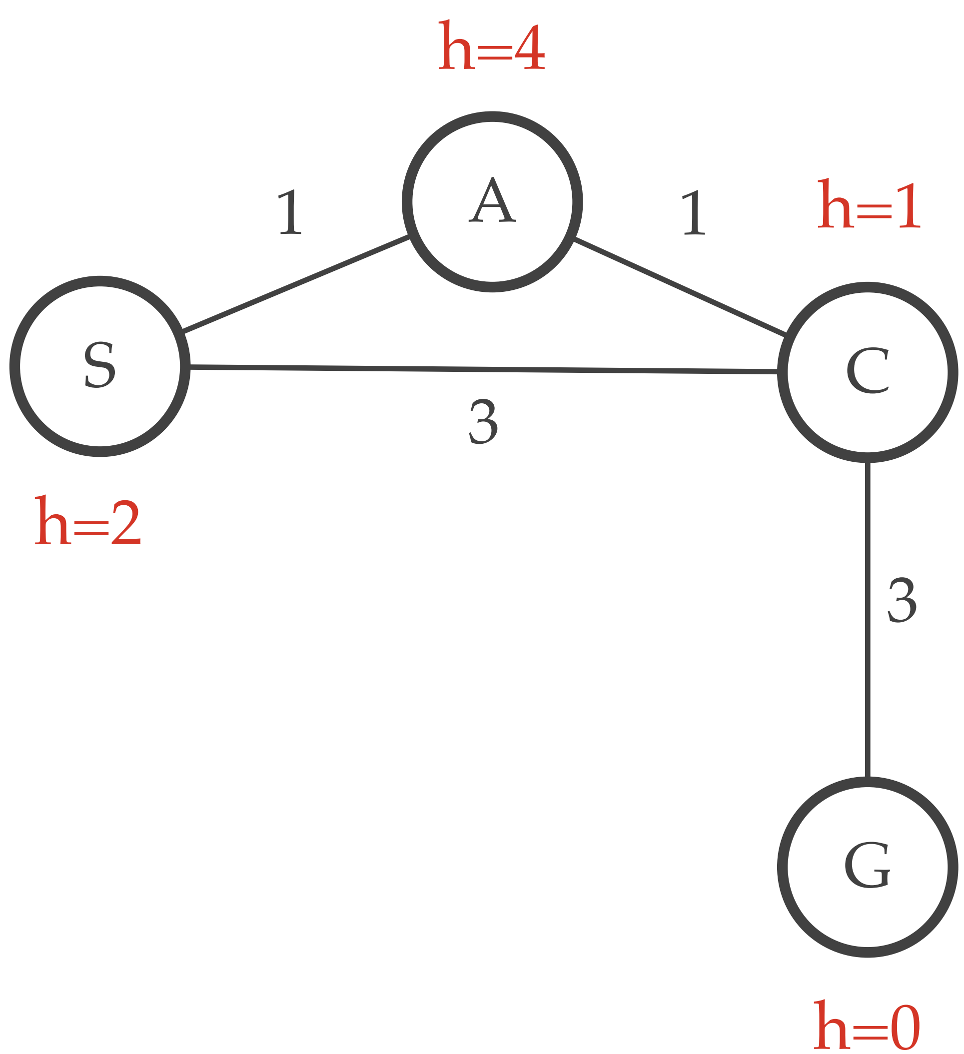
B) S, S-A, S-C, S-A-C, S-C-G

C) S, S-A, S-A-C, S-A-C-G

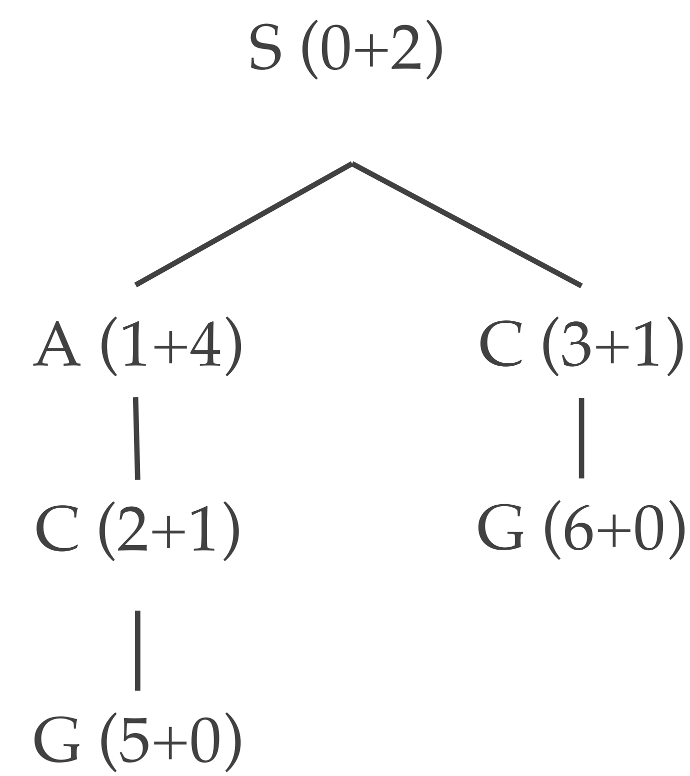
D) S, S-A, S-C, S-A-C, S-A-C-G

A* Graph Search Gone Wrong?

State space graph



Search tree



- Check if state already explored
- Revisit if cheaper but requires recalculating descendants

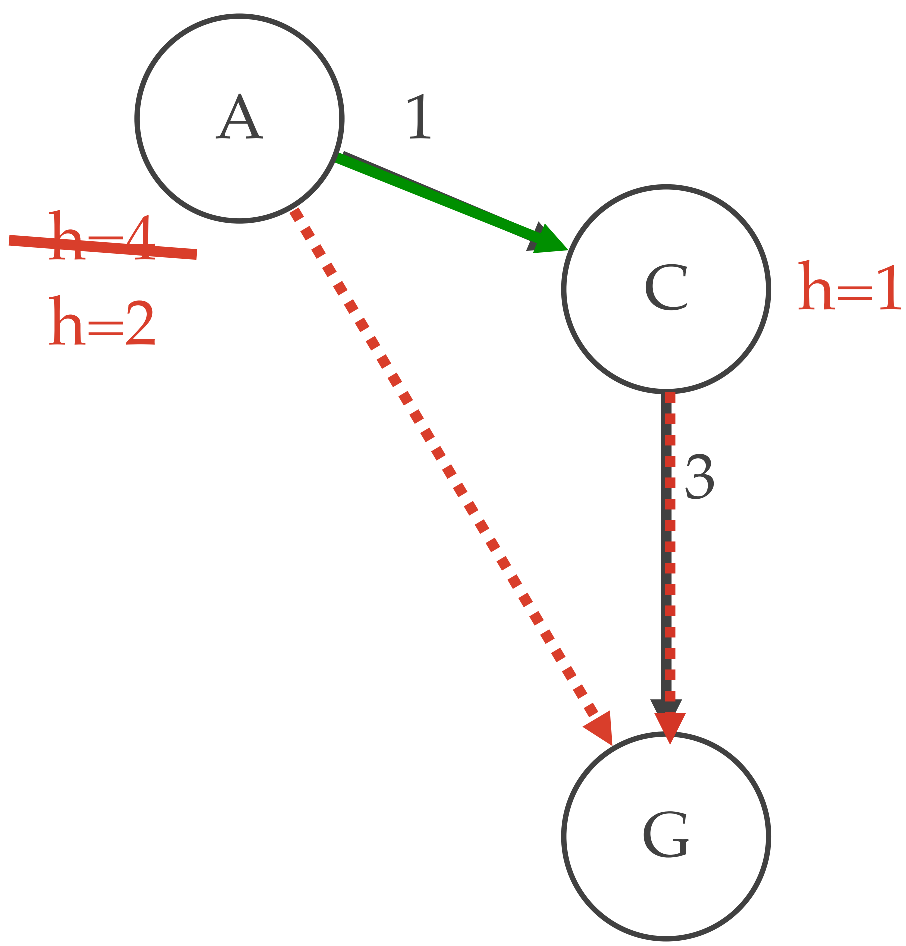
Consistency of Heuristics

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

    h(A) ≤ actual cost from A to G
  - Consistency:
    - triangular inequality

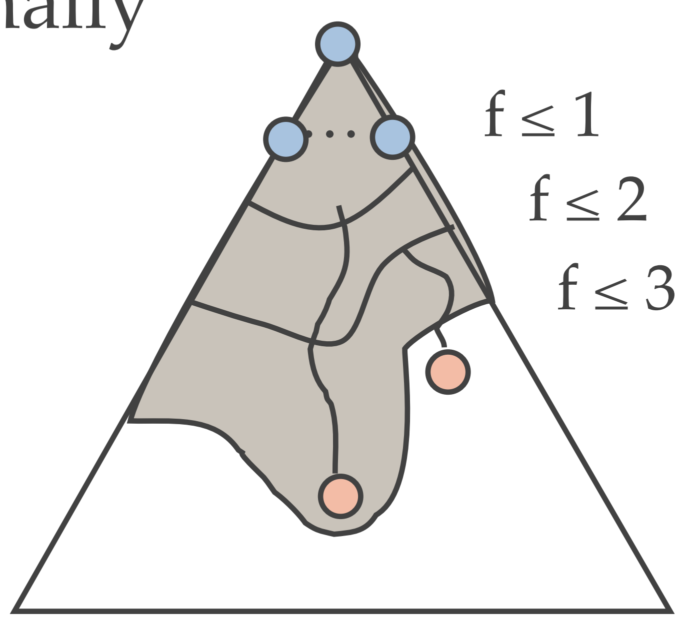
    h(A) ≤ cost(A to C) + h(C)
    - heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)
- **Consequences of consistency:**
  - The f value along a path never decreases
  - A* graph search is optimal



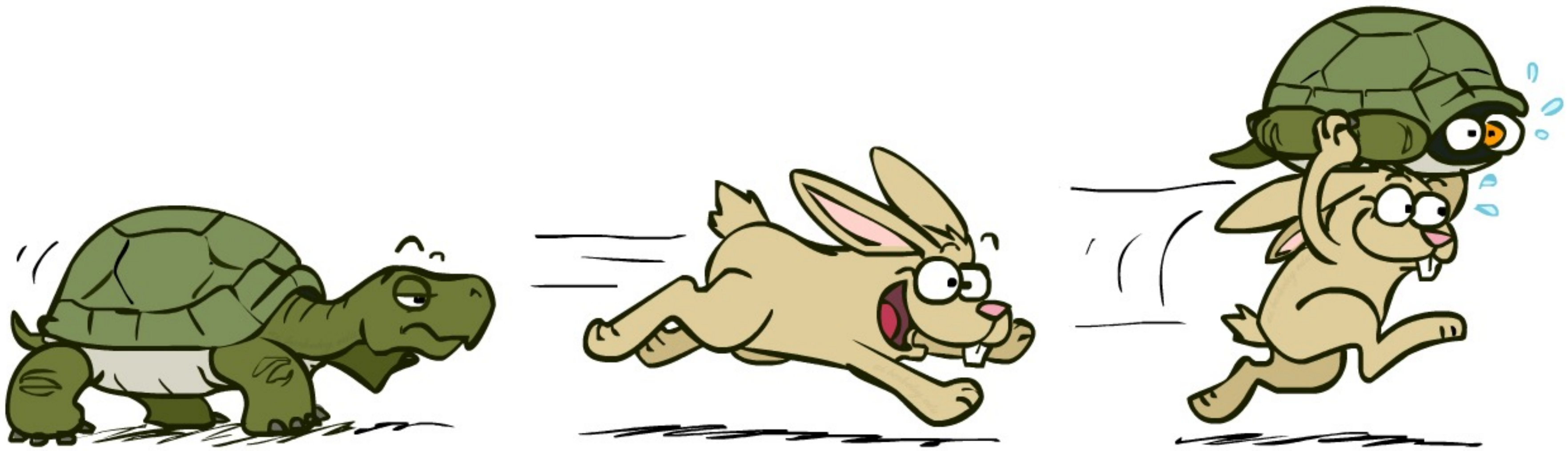
Optimality of A* Graph Search

- Sketch: 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 n, nodes that reach n optimally are expanded before nodes that reach n suboptimally
  - Result: A* graph search is optimal



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

Tree Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Graph Search Pseudo-Code

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```