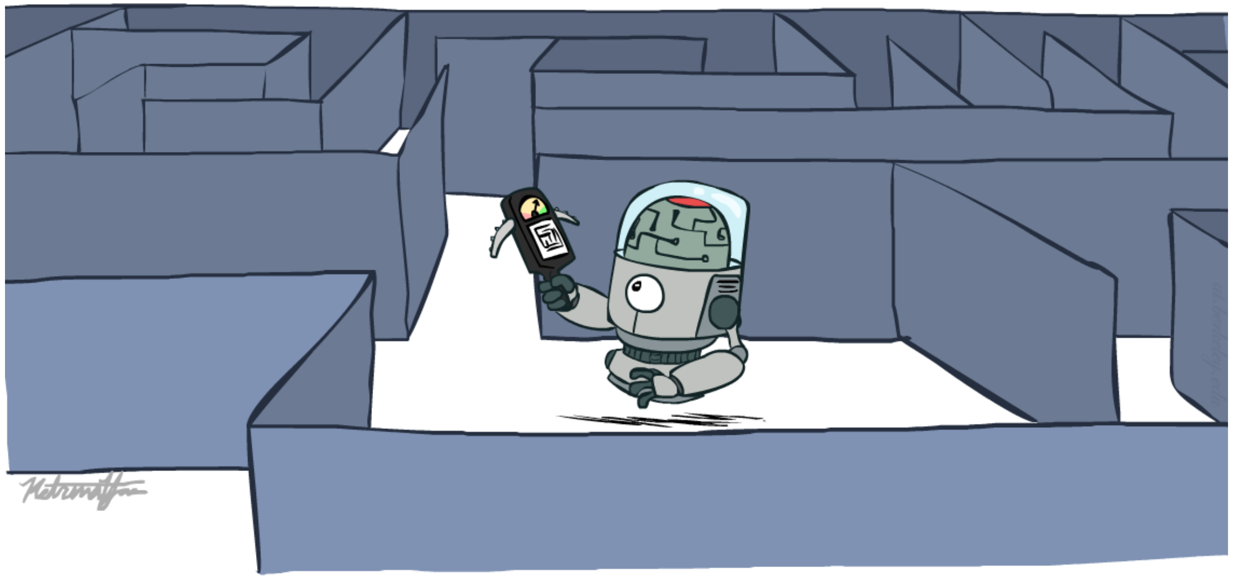
CS 4100 // artificial intelligence

instructor: byron wallace



Search II

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

Questions before we begin?

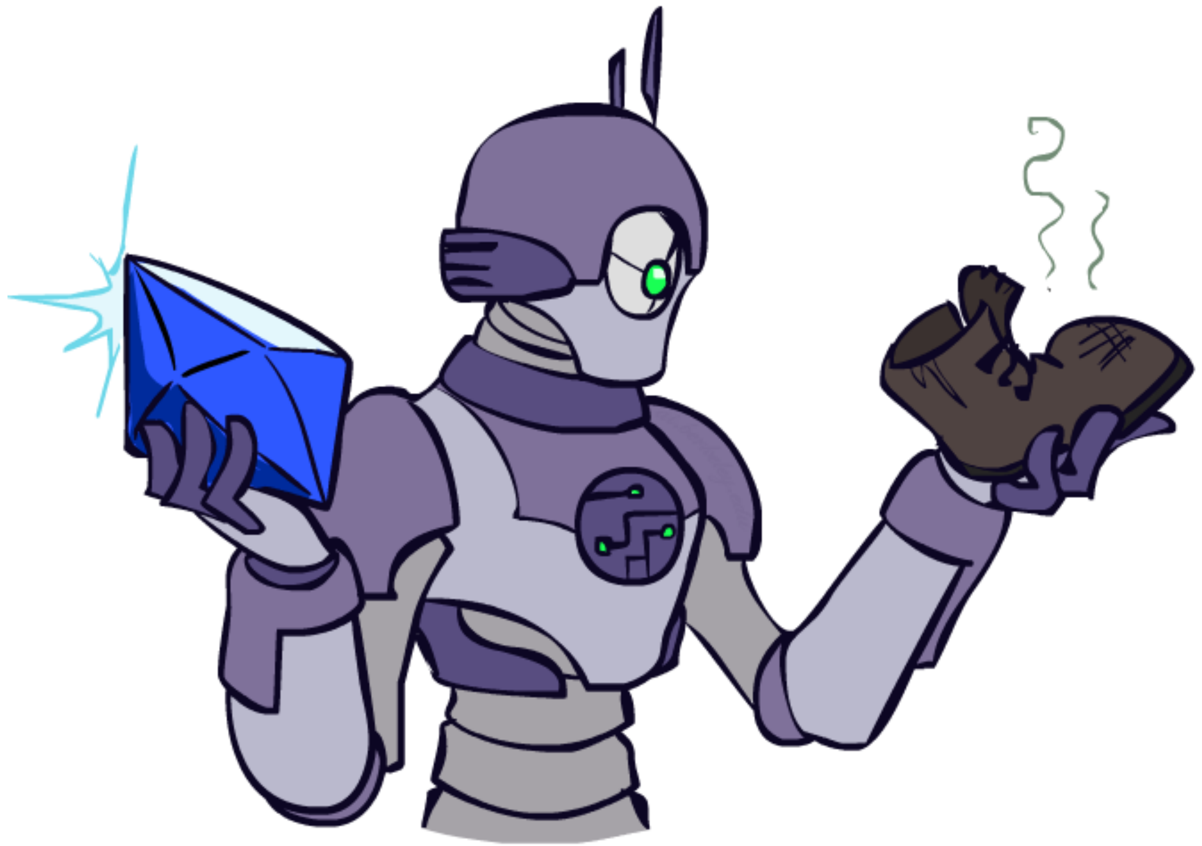
- On HW or anything else?

Today

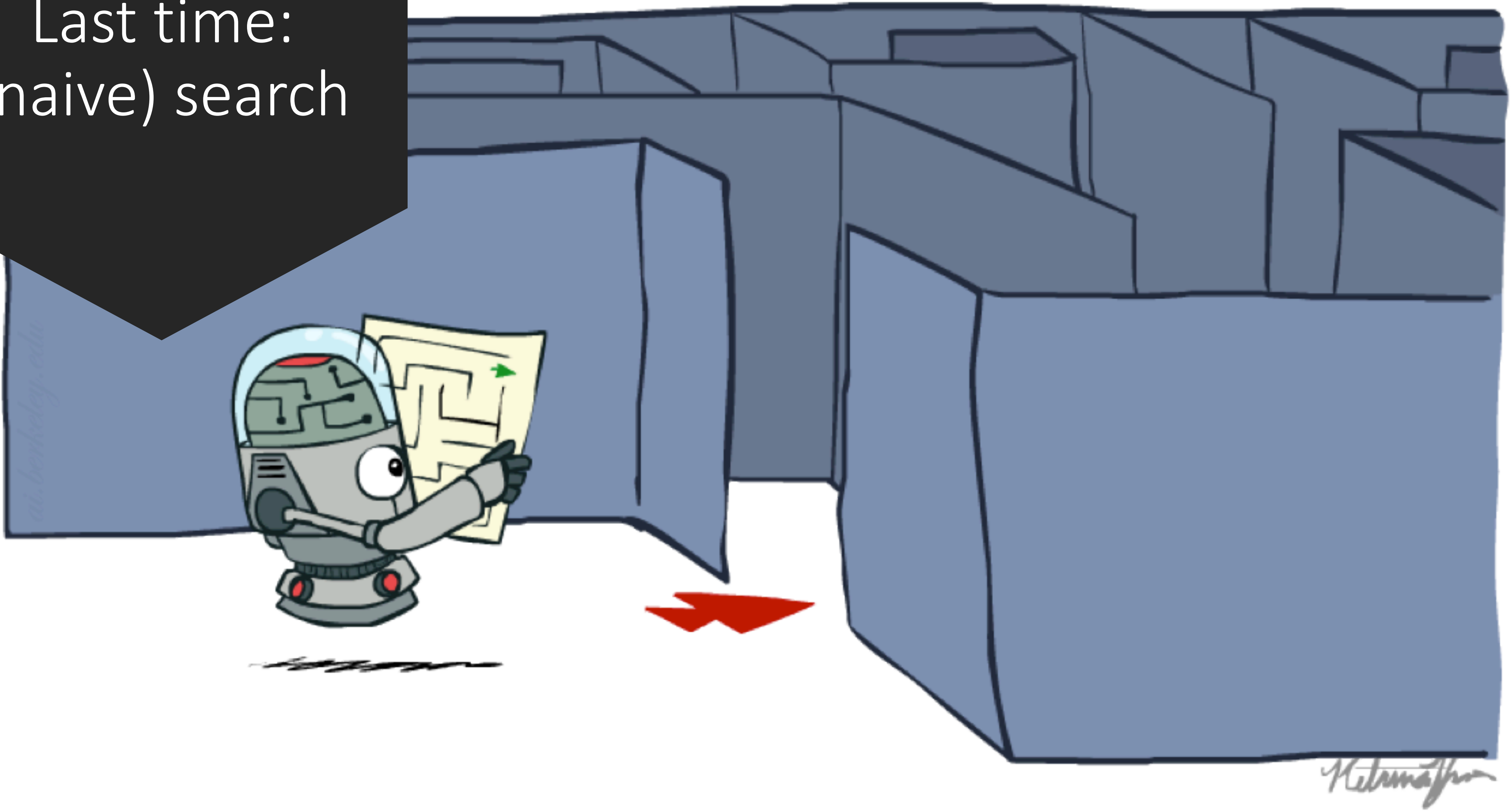
Informed Search

- Heuristics
- Greedy Search
- A* Search

Graph Search



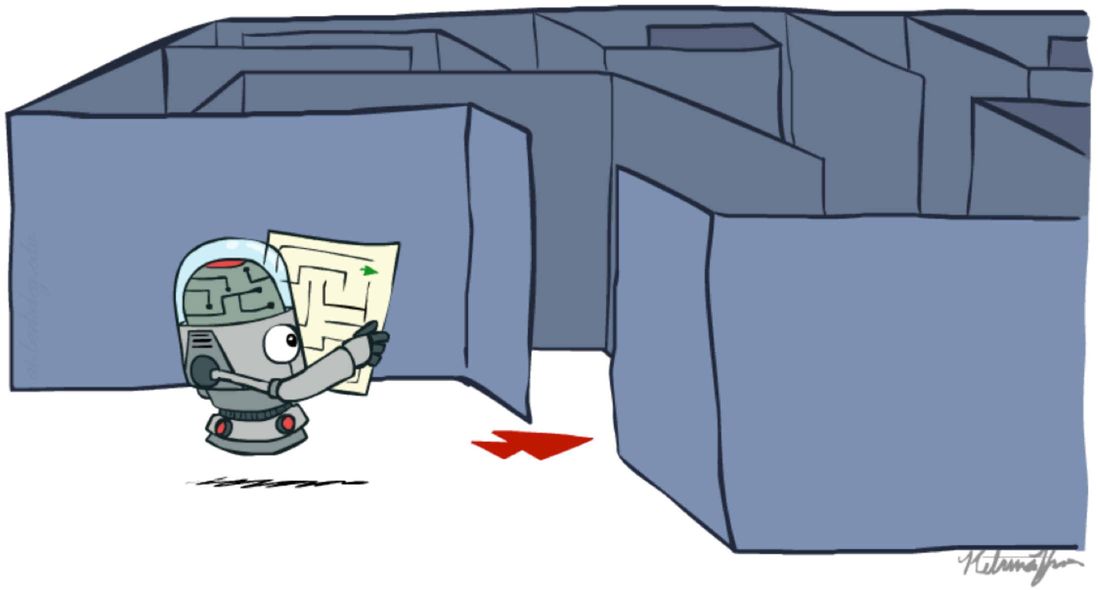
Last time: (naive) search



Recap

Search problem

- States (configurations of the world)
- Actions and costs
- Successor function (world dynamics)
- Start state and goal test



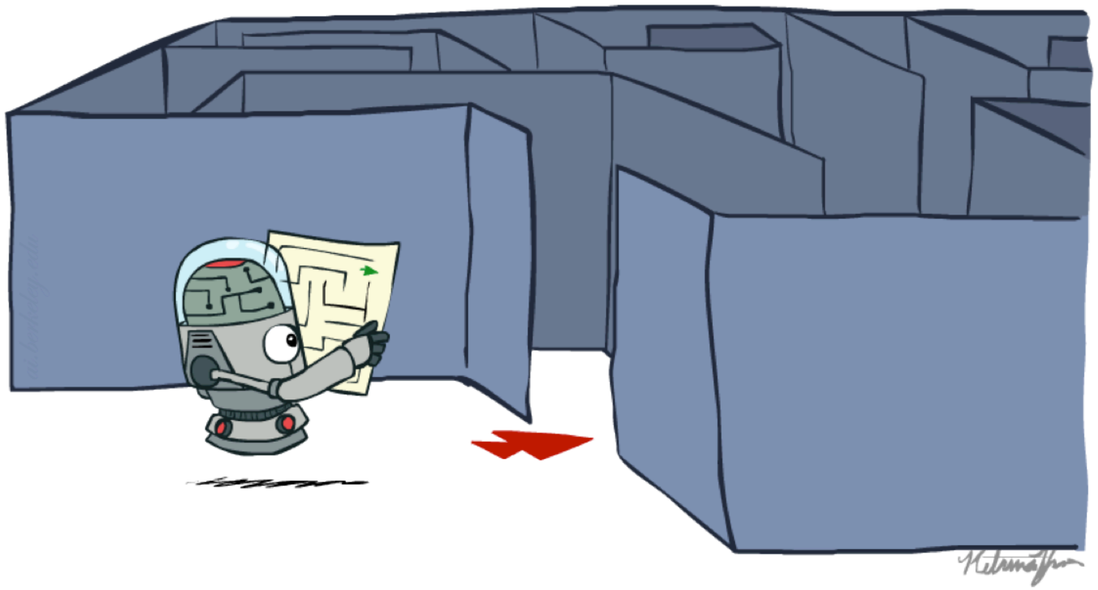
Recap

Search problem

- States (configurations of the world)
- Actions and costs
- Successor function (world dynamics)
- Start state and goal test

Search tree

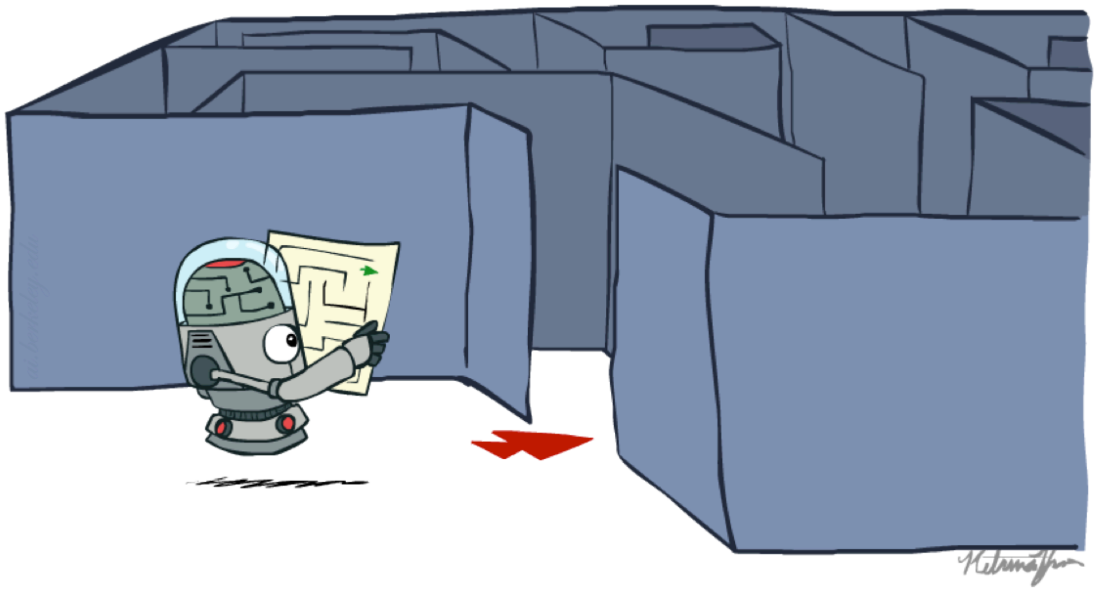
- Nodes: represent plans for reaching states
- Plans have costs (sum of action costs)



Recap

Search problem

- States (configurations of the world)
- Actions and costs
- Successor function (world dynamics)
- Start state and goal test



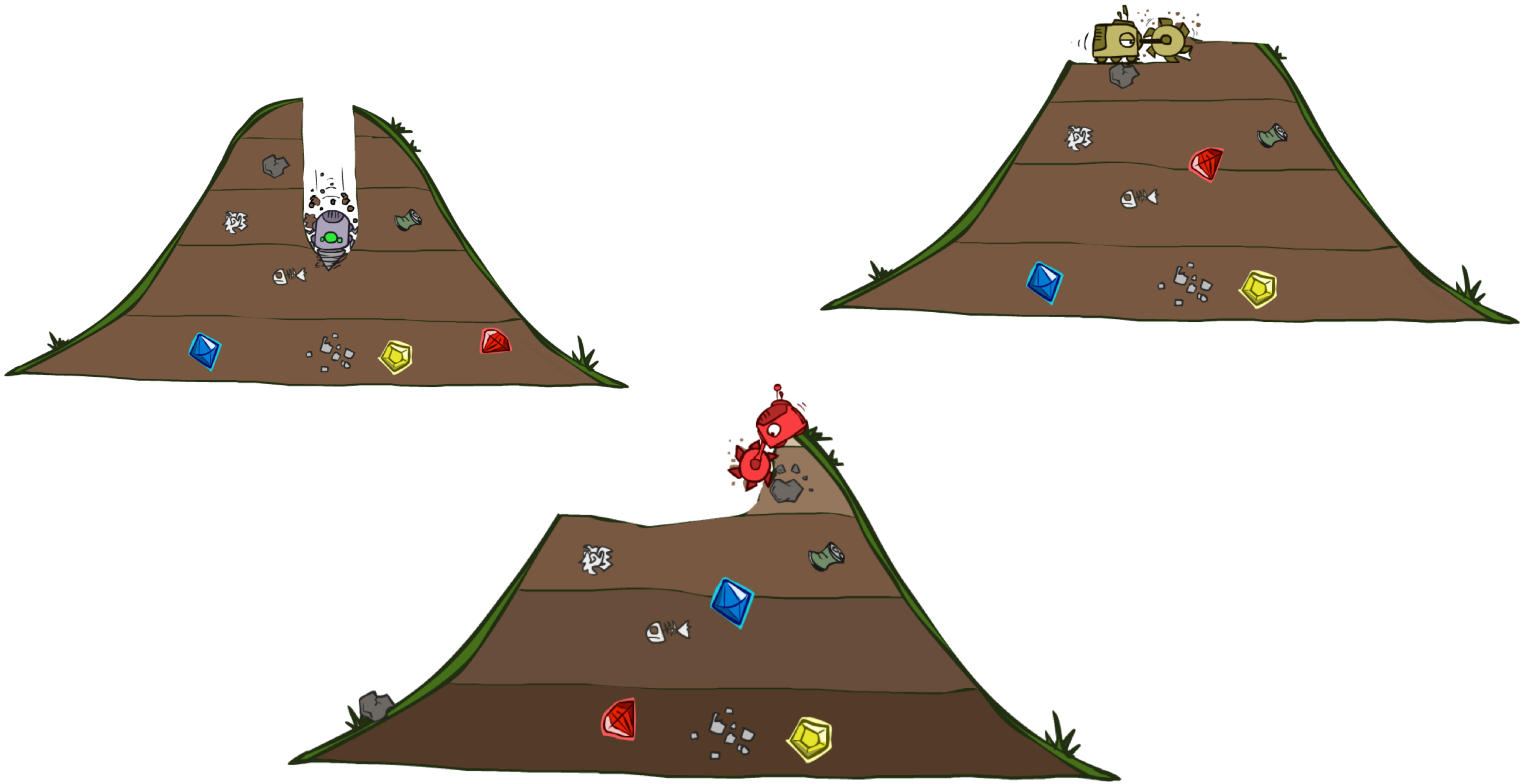
Search tree

- 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

DFS, BFS, UCS



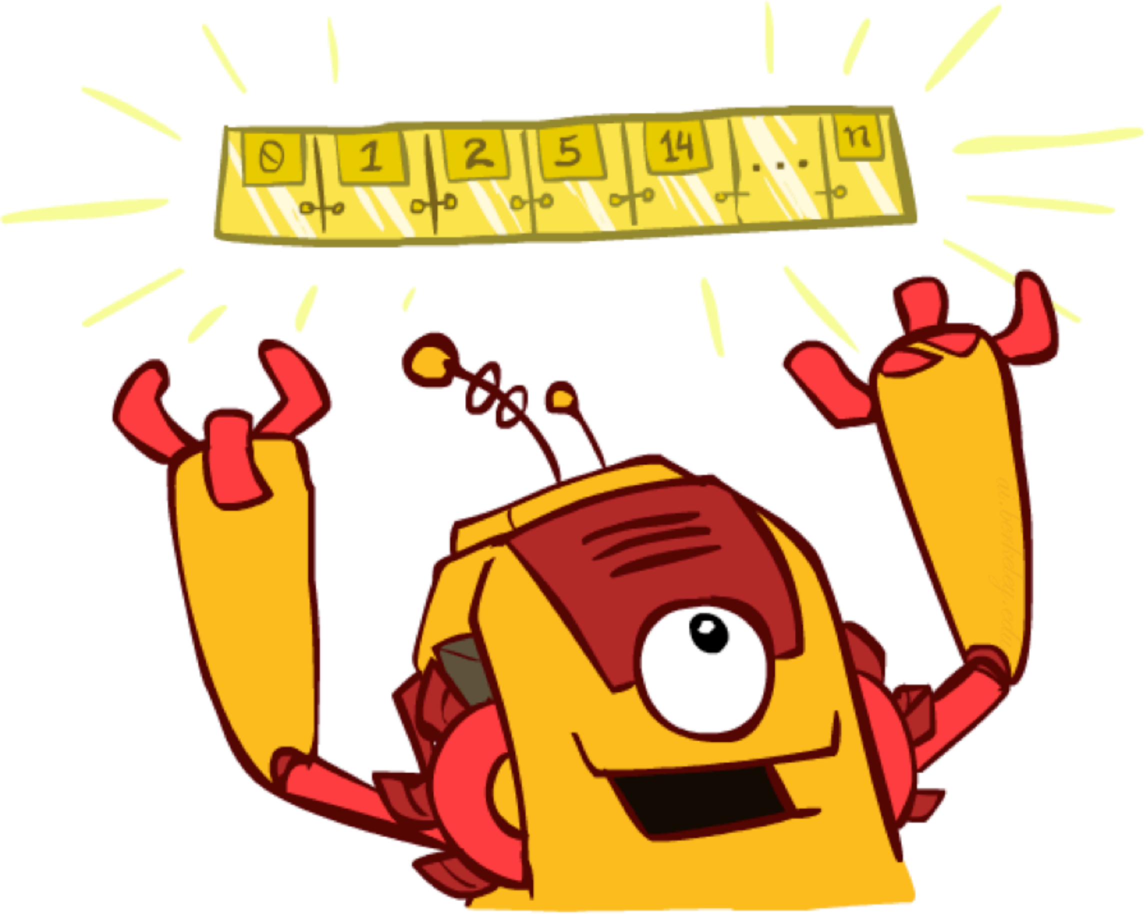
General tree search algorithm

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

The one queue

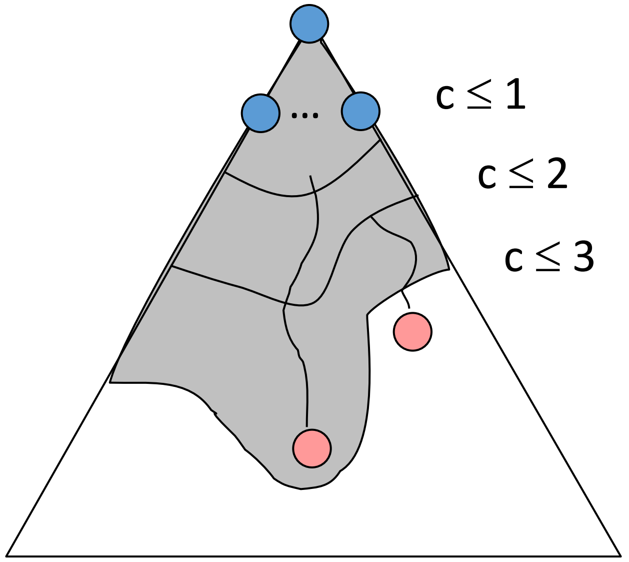
These search algorithms are the *same* except for fringe strategies

- 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



Uniform Cost Search

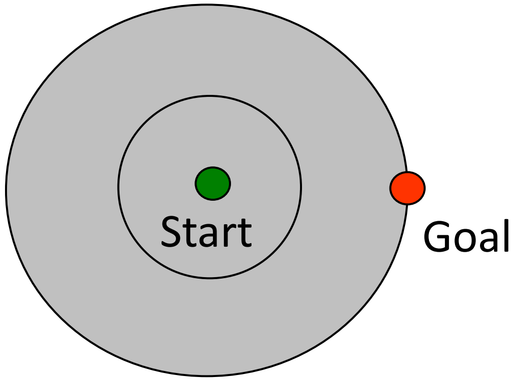
Strategy: expand lowest path cost



The good: UCS is complete and optimal!

The bad:

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



UCS

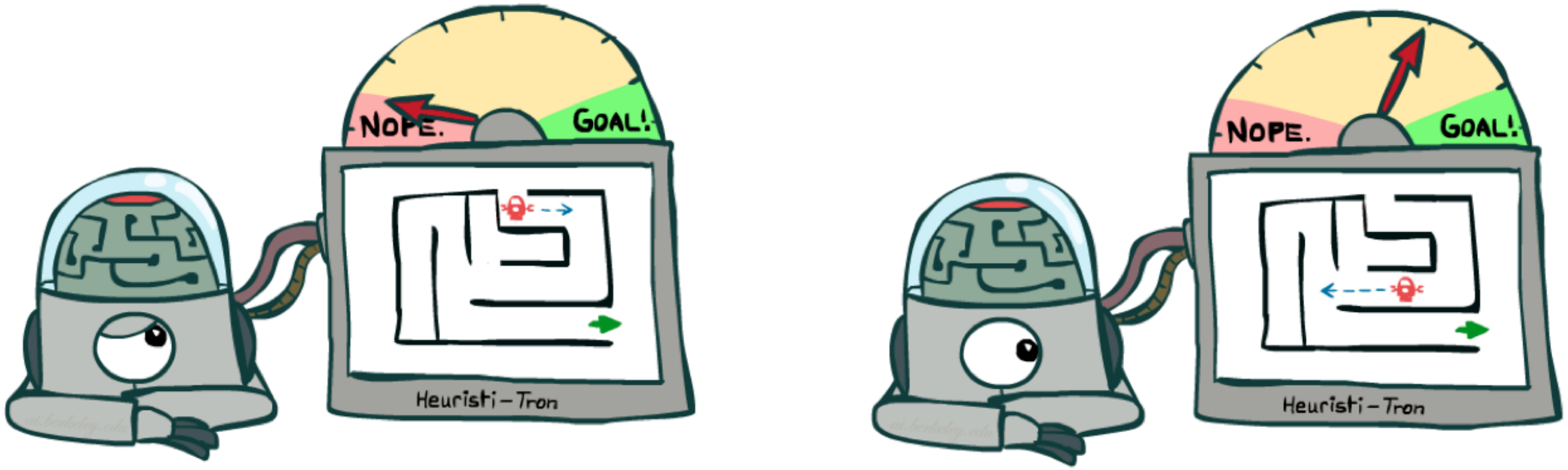
UCS: PacMan

Can we do better?

This is the motivation behind *informed search*, which uses problem-specific knowledge to try and find solutions more efficiently

Search heuristics

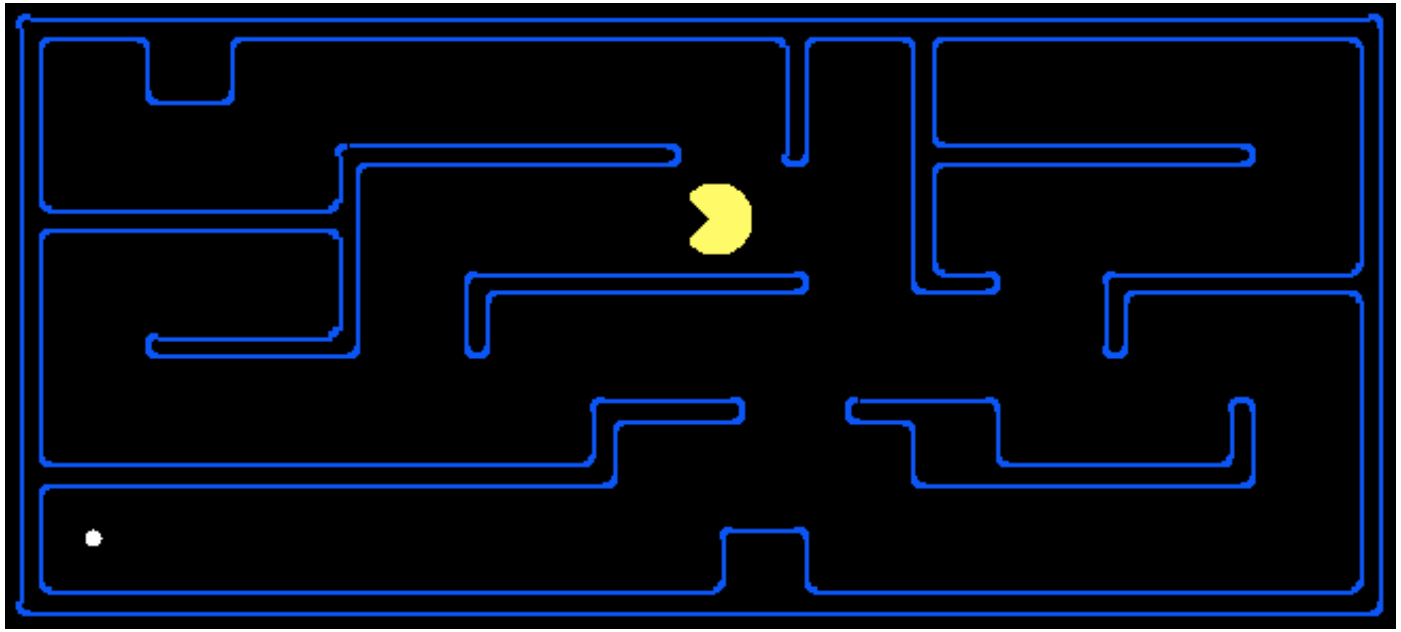
- Key addition for informed search
- A trick that tells us how far from our goal we are from a given state
- Specifically: a *function* mapping from *states* to *reals* that encode proximity to goal

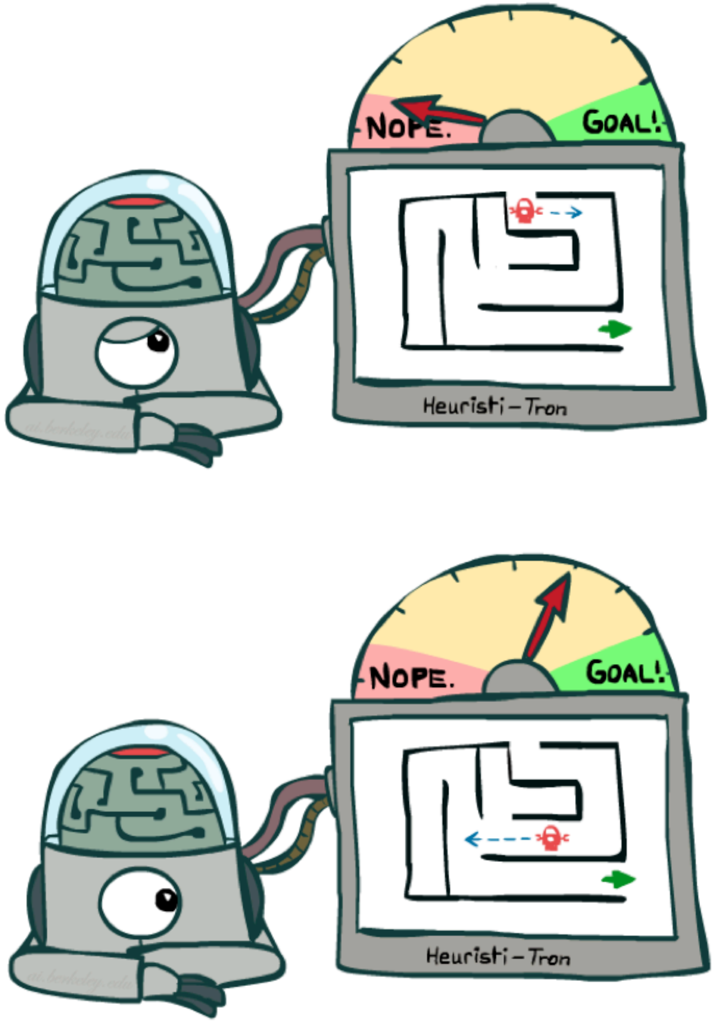


Search heuristics

A *heuristic* is

- A *function* that estimates how close a state is to a goal
- Designed for a particular search problem
- What might we use for PacMan (e.g., for pathing)?

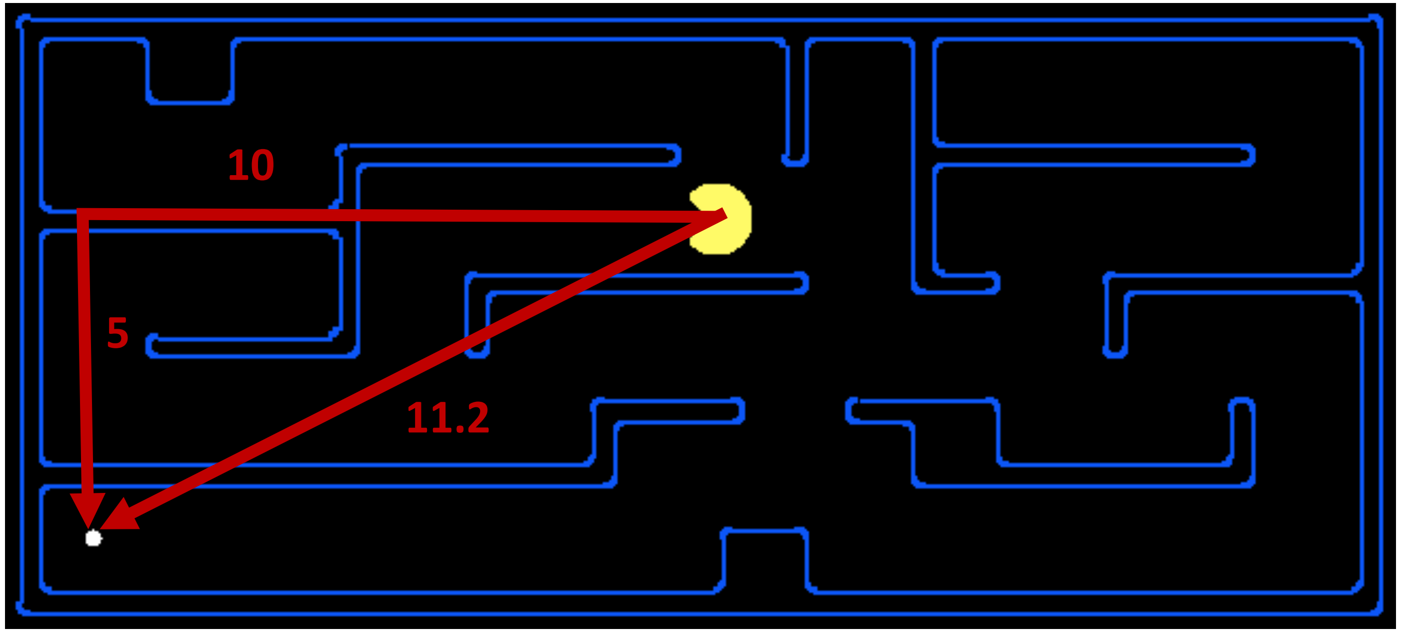


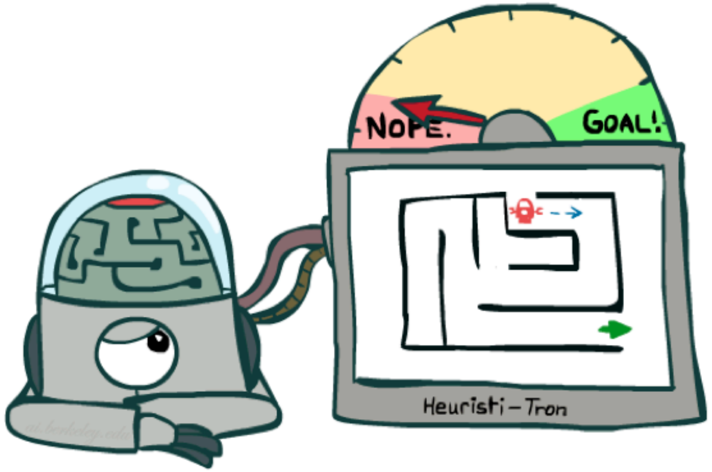


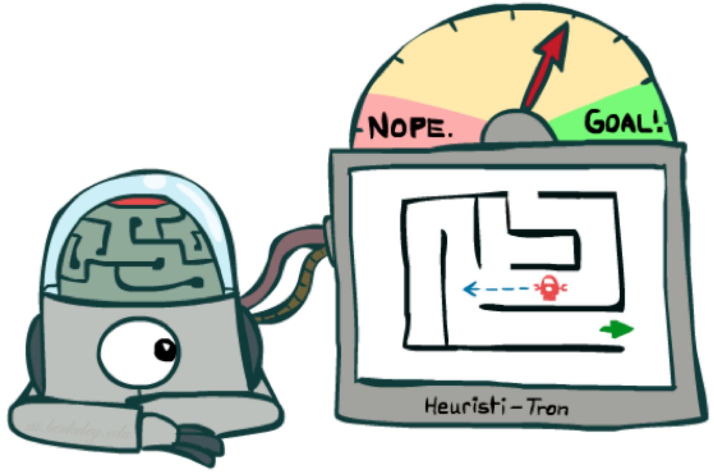
Search heuristics

A *heuristic* is

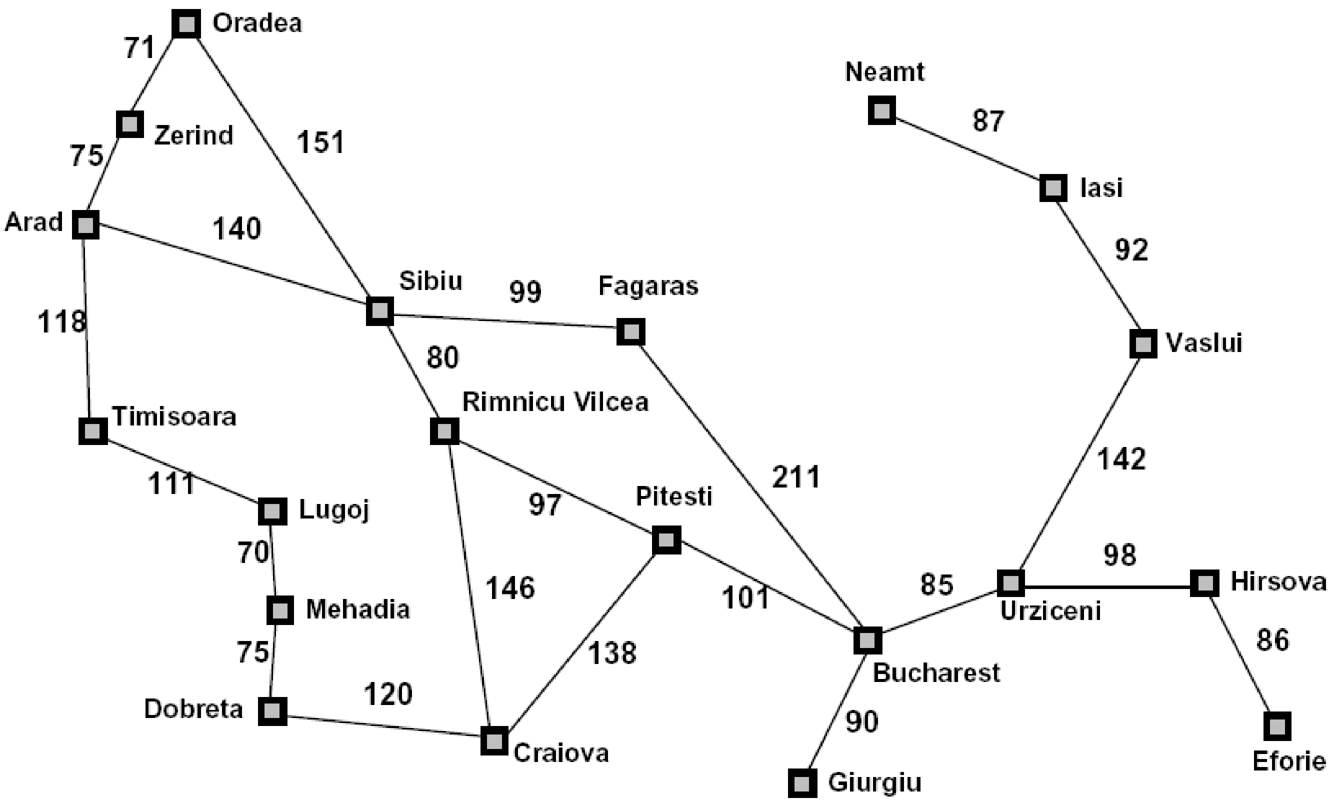
- A *function* that estimates how close a state is to a goal
- Designed for a particular search problem
- What might we use for PacMan (e.g., for pathing)? Manhattan distance, Euclidean distance







Example: heuristic function



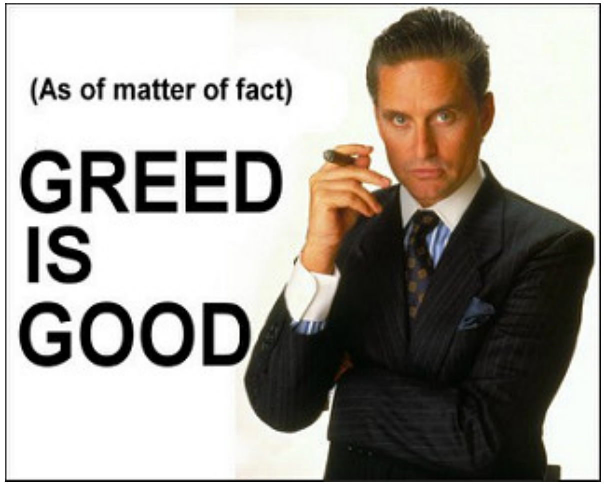
Straight−line distance to Bucharest

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

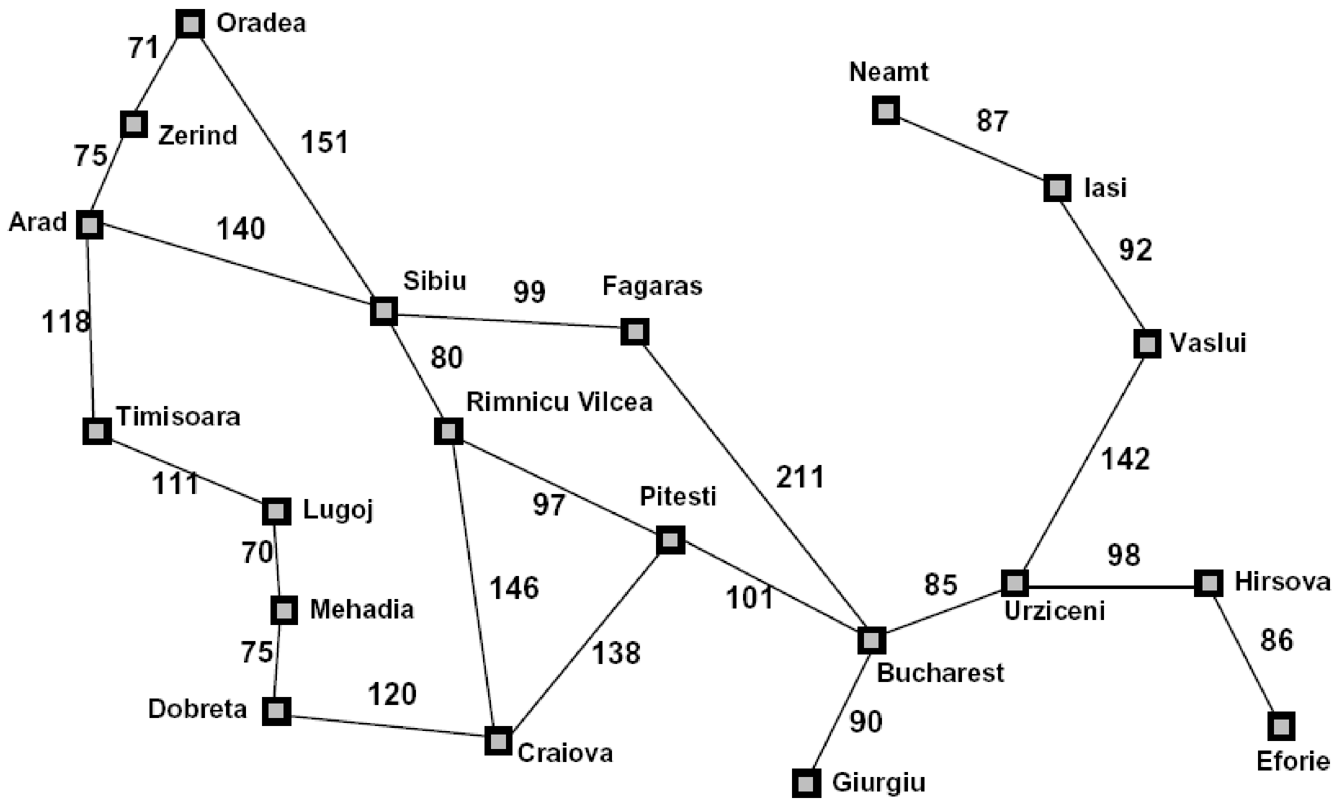
h(x)

Great, but… what do we do with these things?

Greedy search



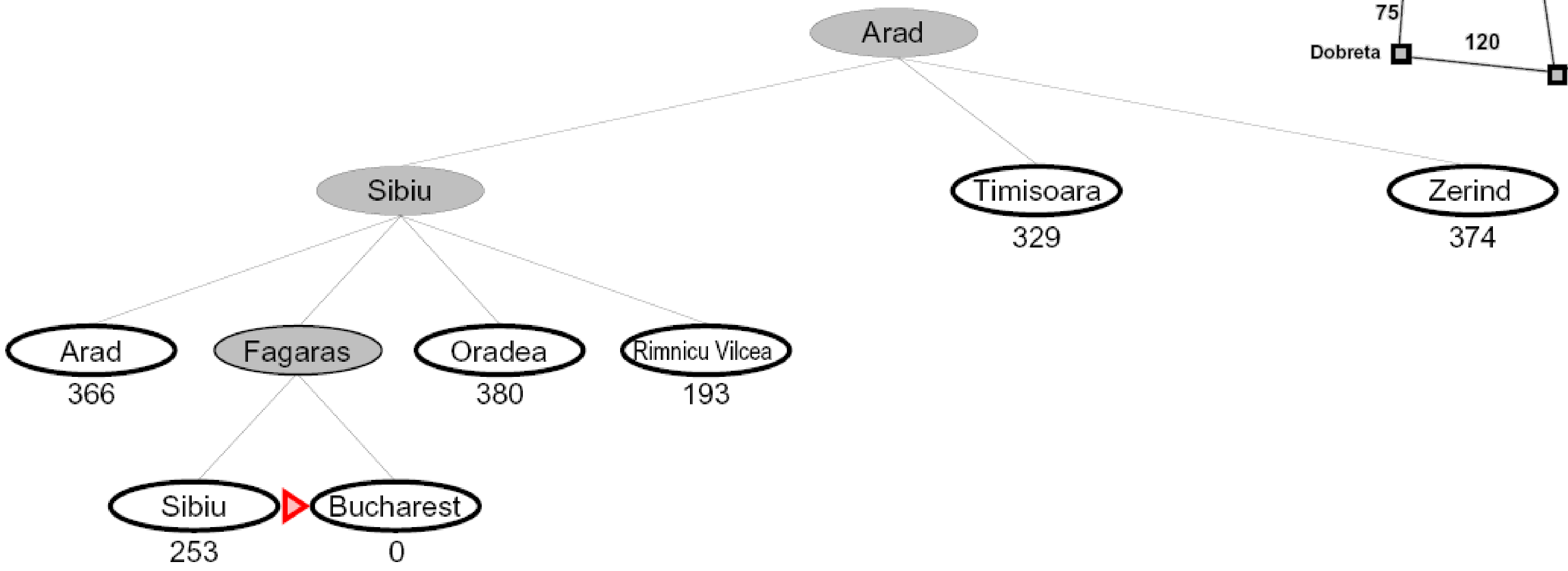
Example: heuristic function

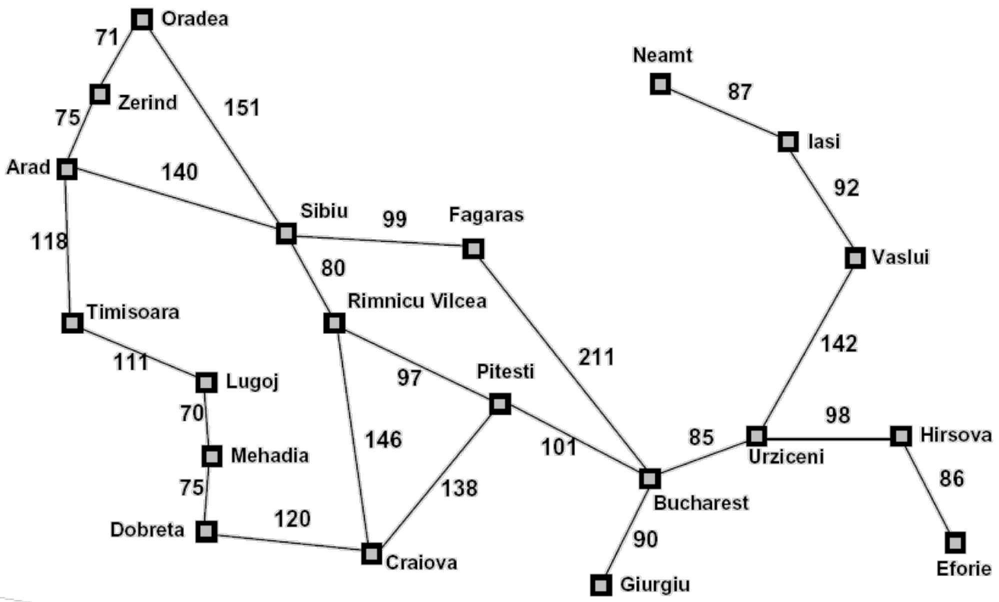


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Expand the node that seems closest…



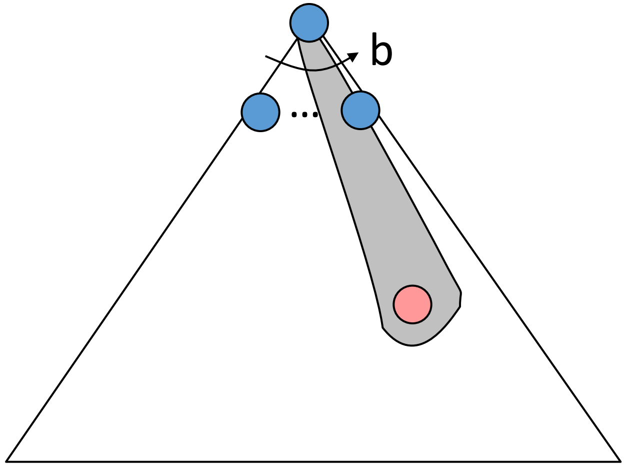


What can go wrong?

Greedy search

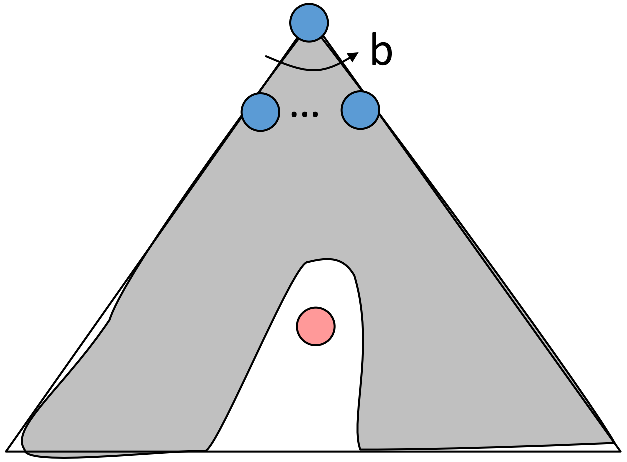
Strategy: expand a node that you think is closest to a goal state

- 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

Demo of Greedy

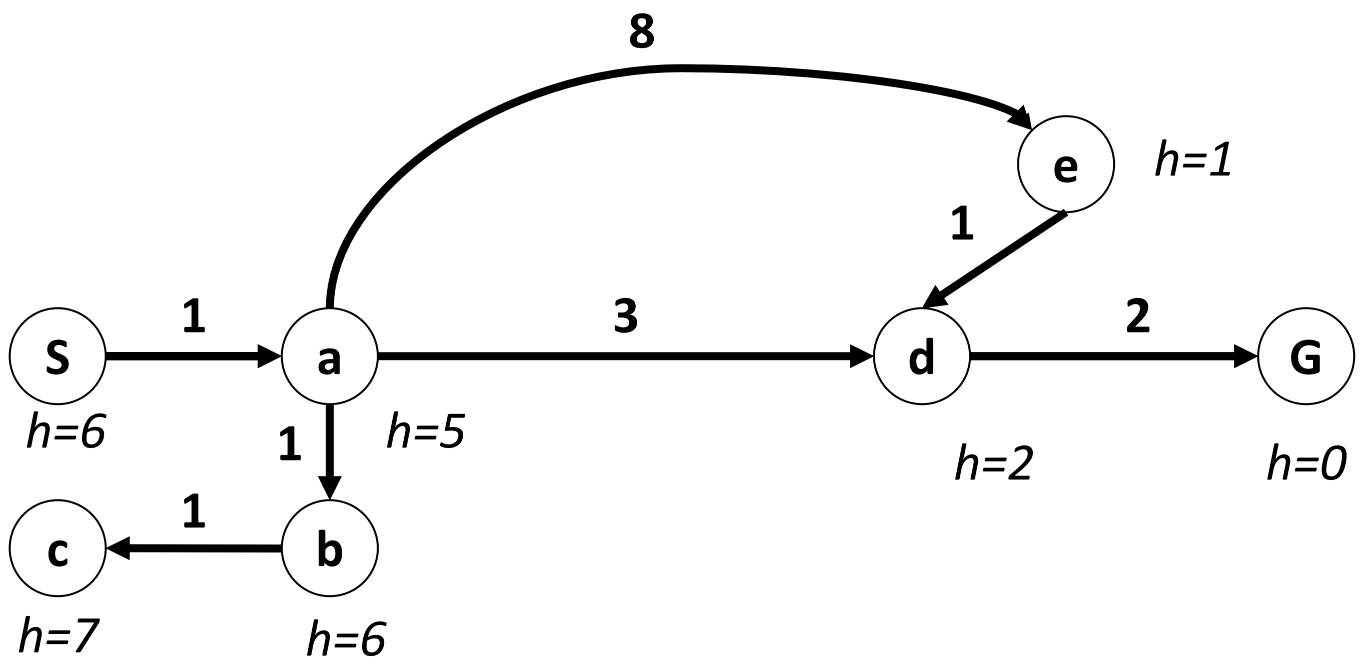
Demo of Greedy: PacMan

Greedy ***is only as good as your heuristic***

A* search

Combining UCS and Greedy

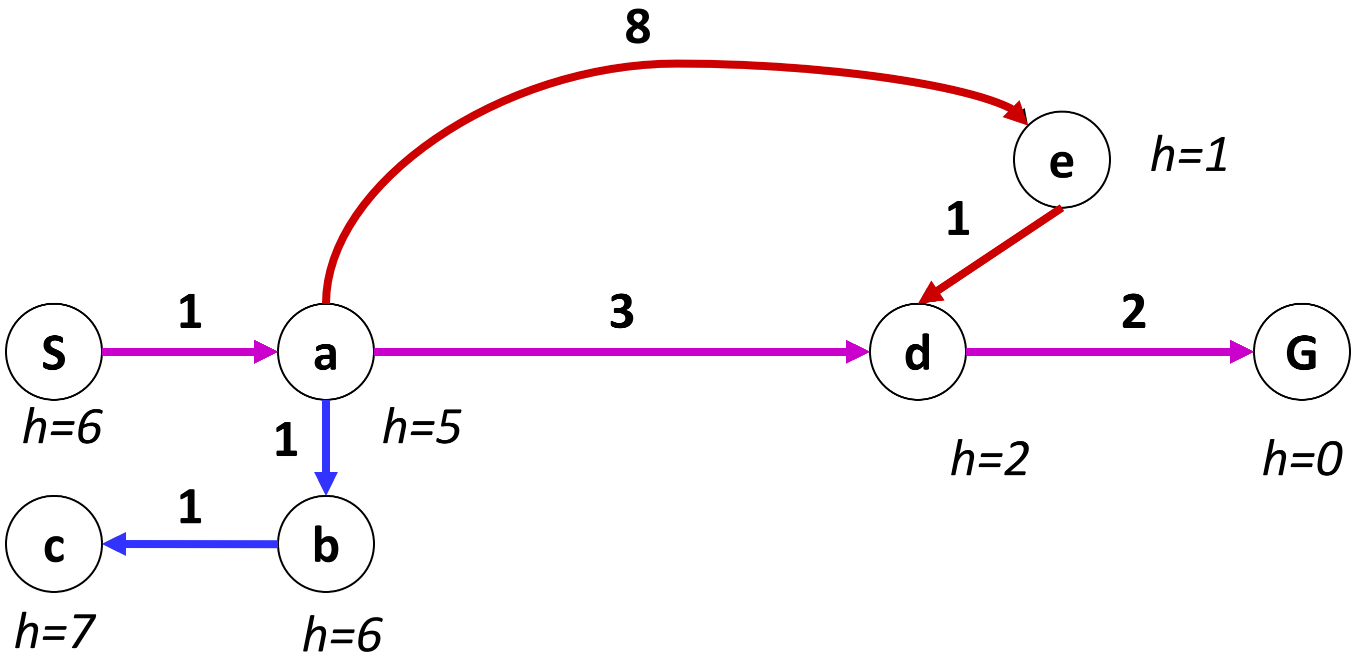
- Uniform-cost orders by (cumulative) path cost, or backward cost g(n)
- Greedy orders by goal proximity, or forward cost h(n)

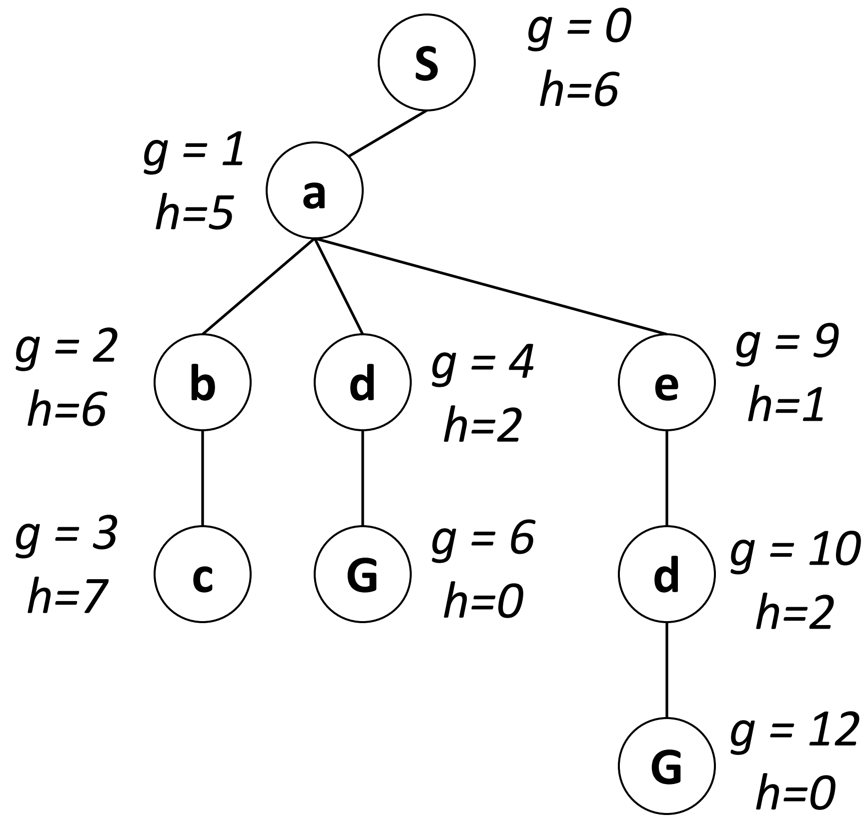


Example: Teg Grenager

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

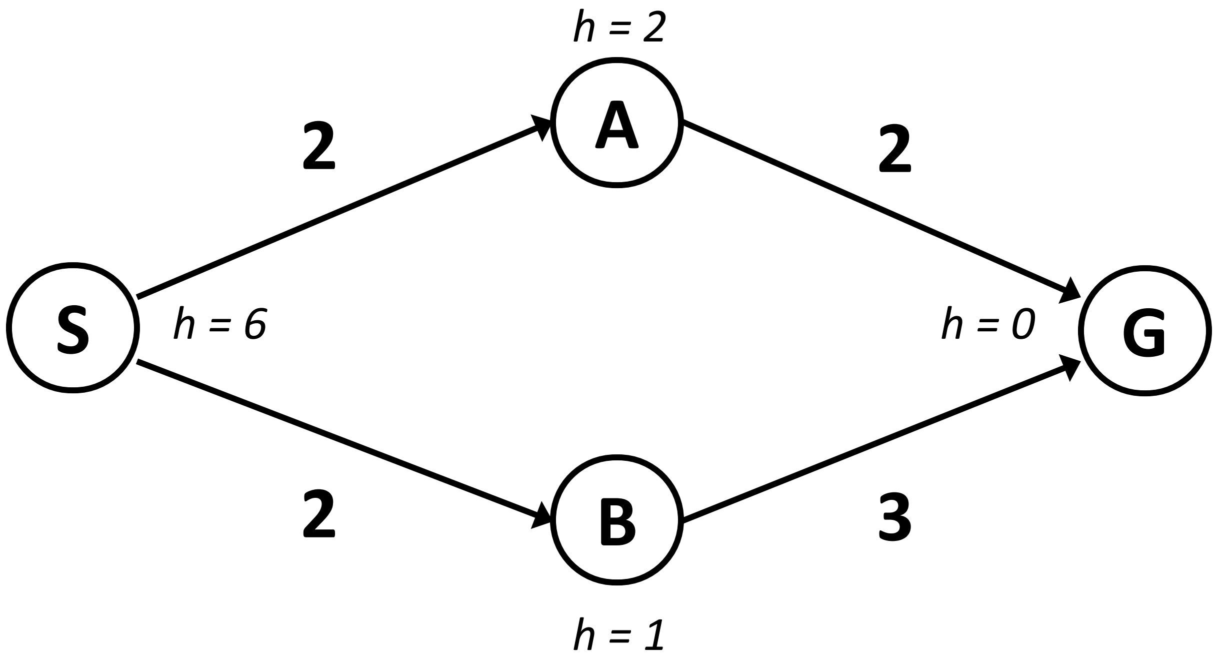
A*, in sum

Order node expansion in order of minimal f(n), where

$$f(n) = g(n) + h(n)$$

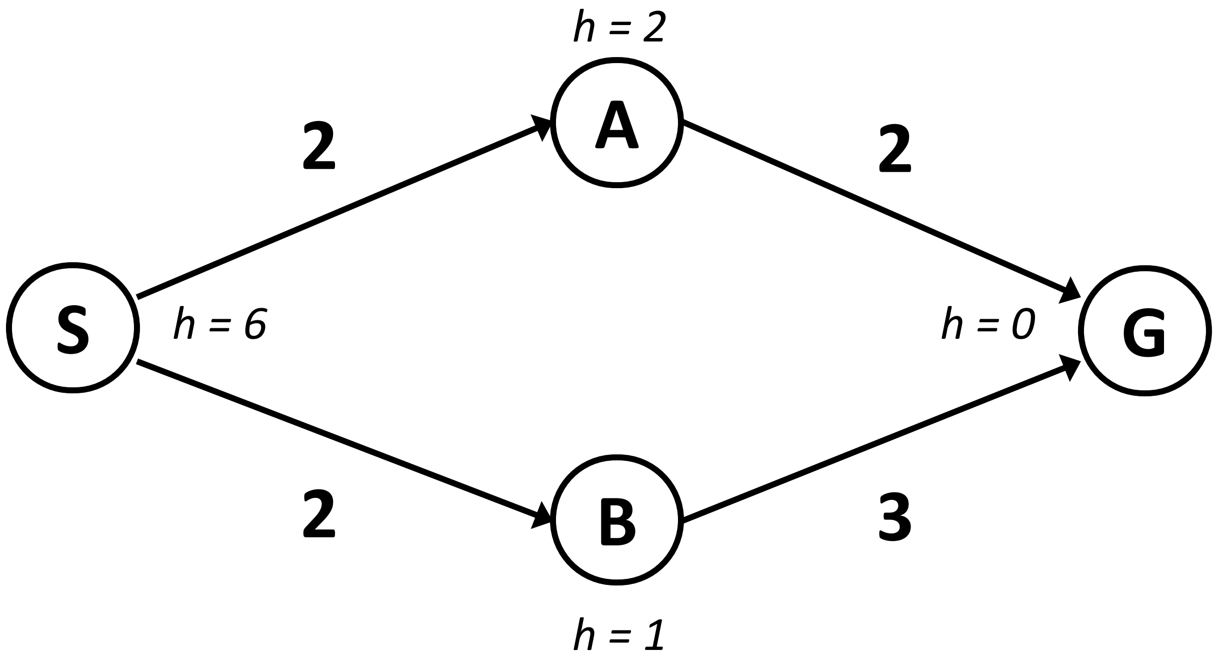
And g(n) is cost of path so far; h(n) is estimate (via heuristic function) of the remaining cost to goal

A note on enqueuing and heuristics



Let's run A*.

A note on enqueuing and heuristics

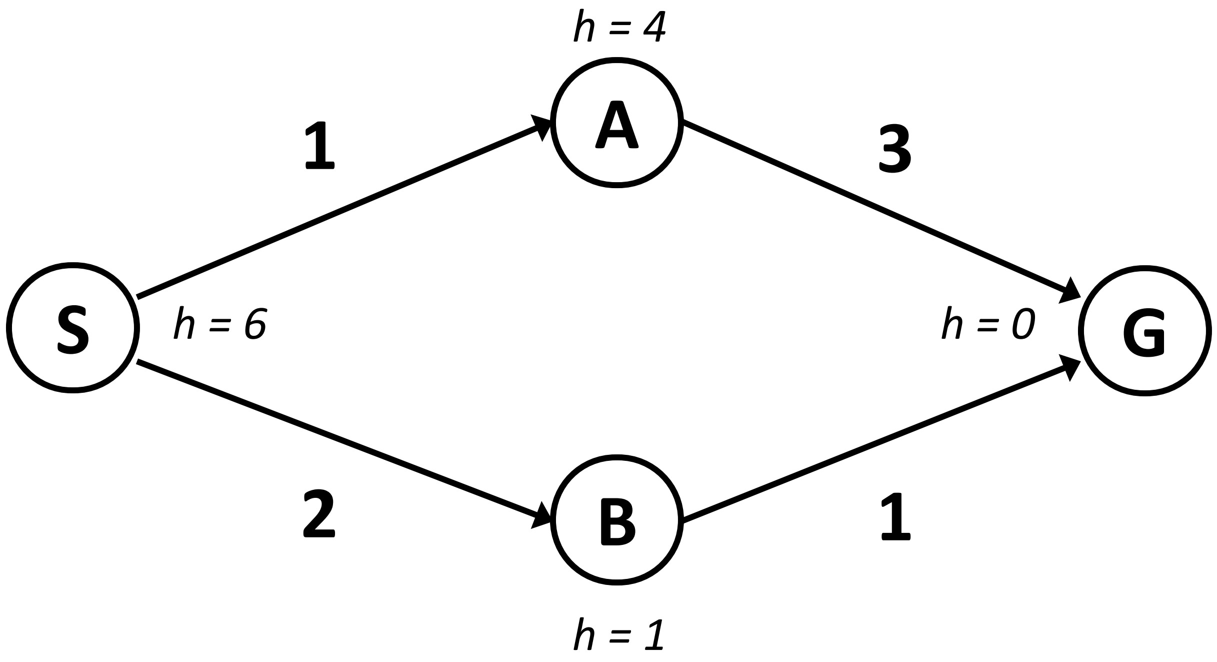


Let's run A*.

So we found the goal but the path there was suboptimal! What happened?

Important! stop when you *dequeue* a goal state; *not* when you enqueue it!

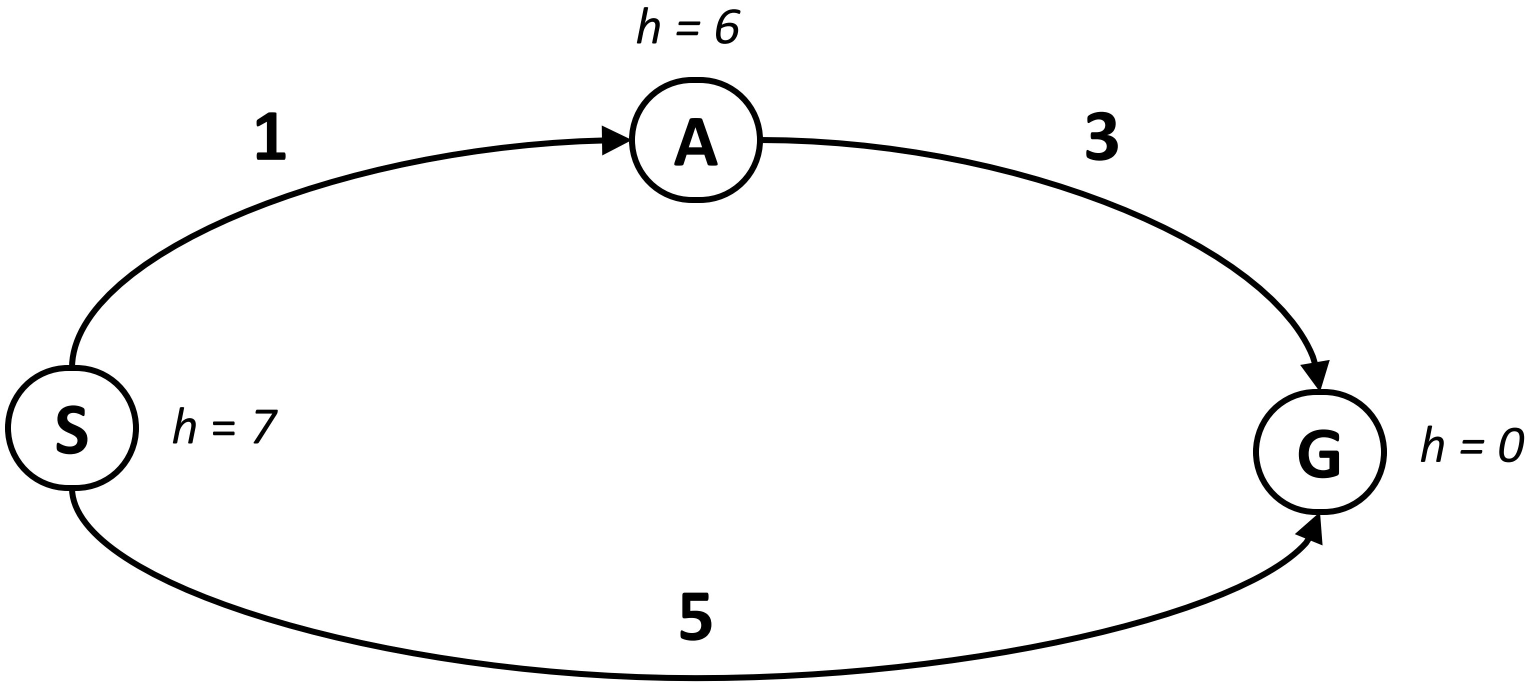
Exercise (you may work in small groups; include all names *legibly* on hand-in)



Starting from *S*, produce the set of nodes expanded to reach goal under:

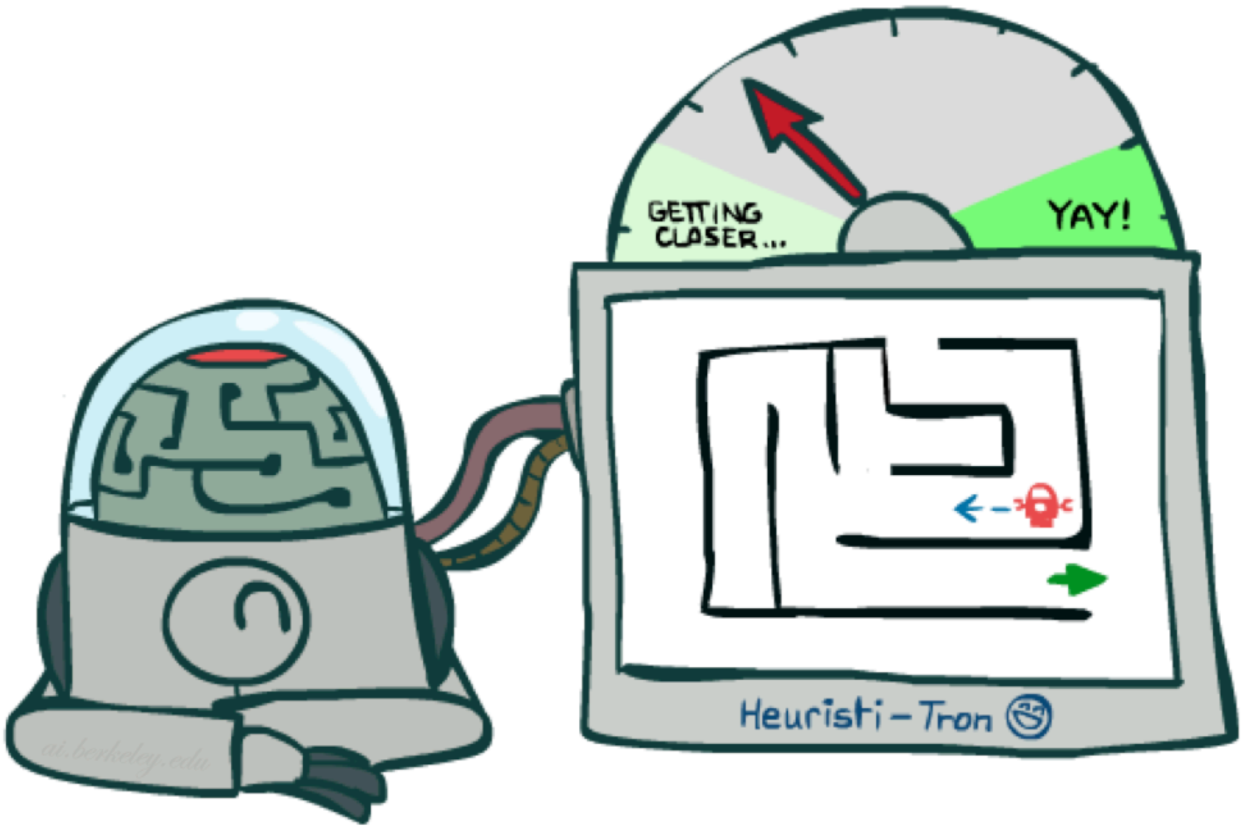
1. DFS
2. UCS
3. A* -- for A*, include a table with g(n), h(n) and their sum, f(n)

Is A* optimal?



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

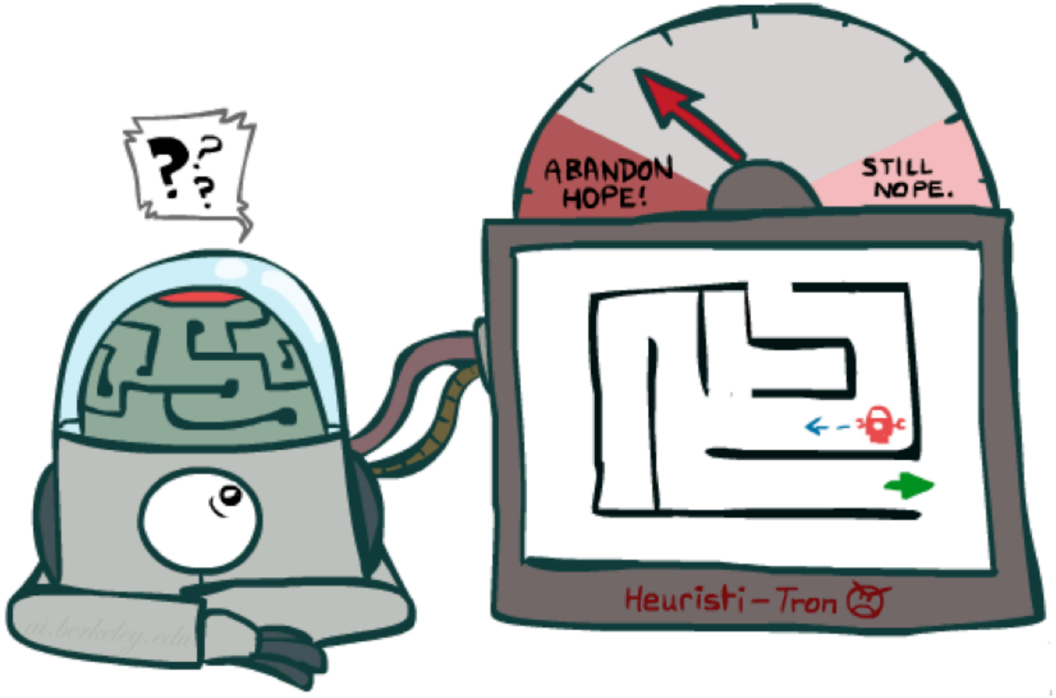
Admissible heuristics

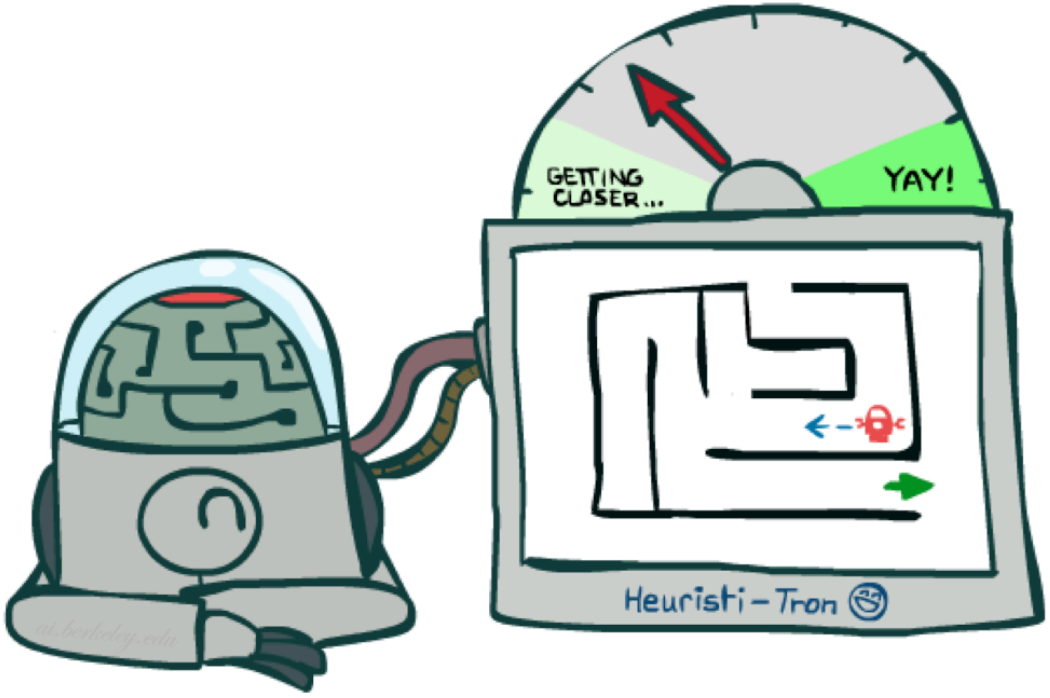


Heuristic functions must be optimistic to be admissible.

Otherwise, a bad heuristic will prevent you from exploring possibly good areas of the 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, formally

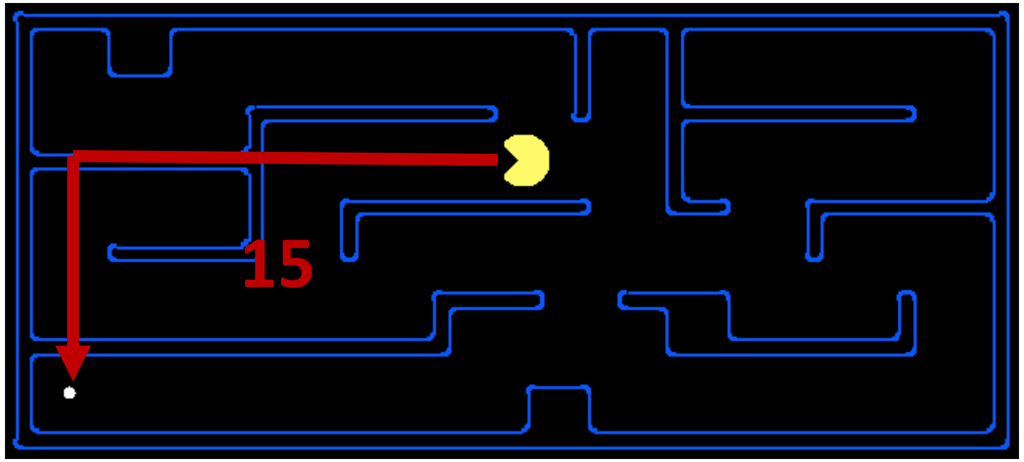
A heuristic h is *admissible* (optimistic) if:

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

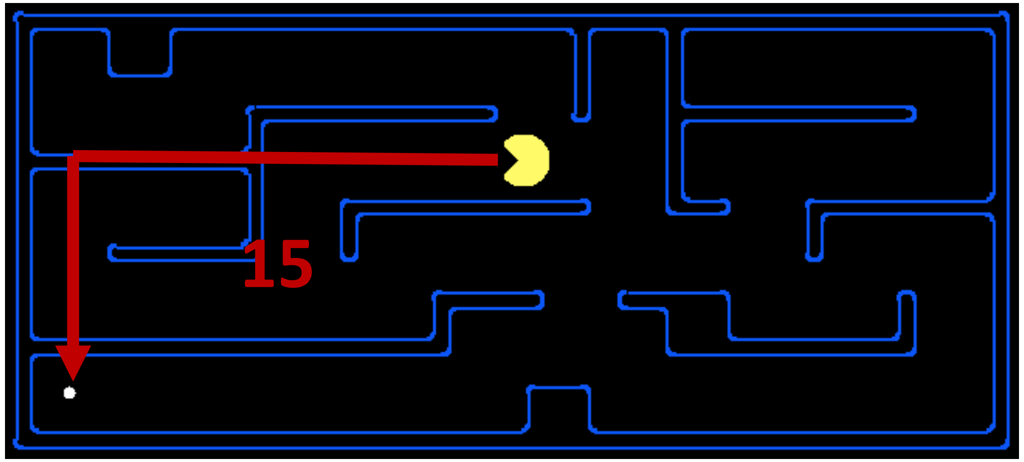
where $h^*(n)$ is the true cost to a nearest goal.

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

Manhattan distance for PacMan pathing admissible?



Manhattan distance for PacMan pathing admissible?



Q: would Euclidean distance be admissible? Would it be better or worse here?

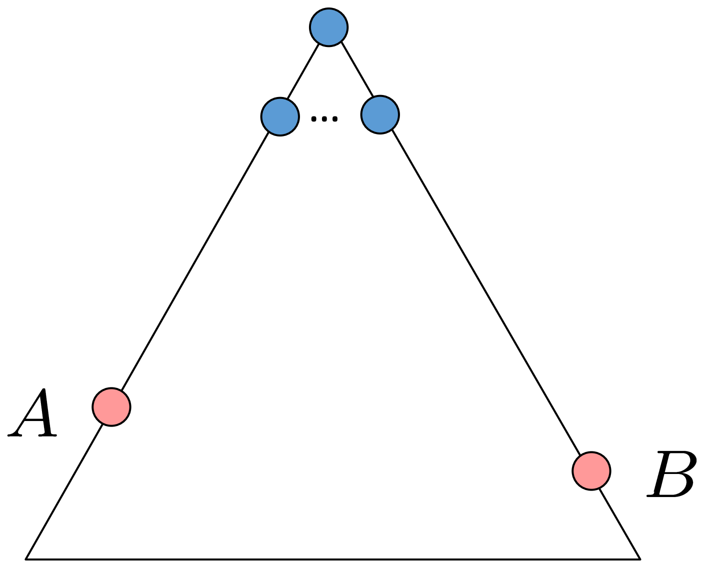
Questions on A* before we continue?

In which A earns its *.
(On the optimality of A*)

Optimality of A* Tree Search

Assume:

1. A is an optimal goal node
2. B is a suboptimal goal node
3. h is admissible



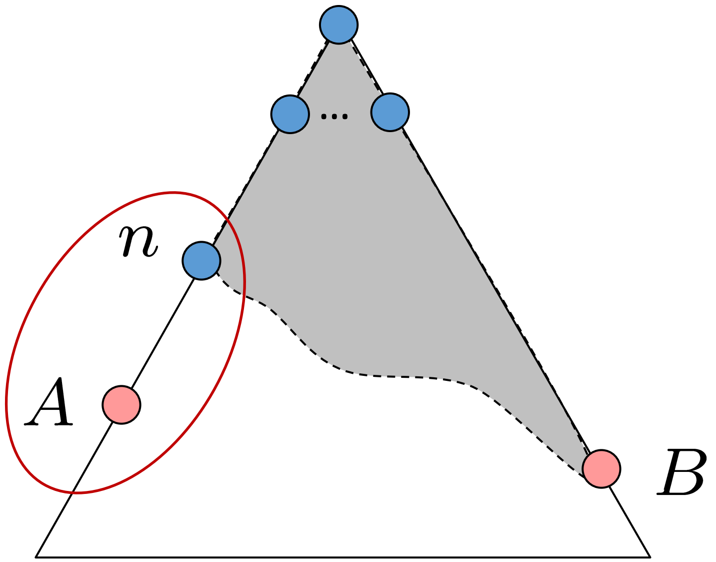
Claim: ***A will exit the fringe before B.***

Note: this would imply general optimality.

Optimality of A* Tree Search

Proof:

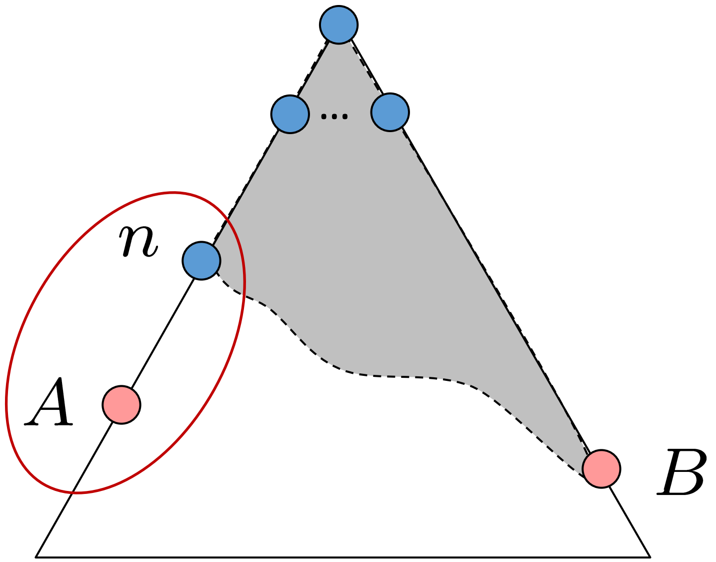
- Imagine B is on the fringe
- Some ancestor n of A is on the fringe, too (maybe A!)
- Claim: ***n will be expanded before B***



Optimality of A* Tree Search

Proof:

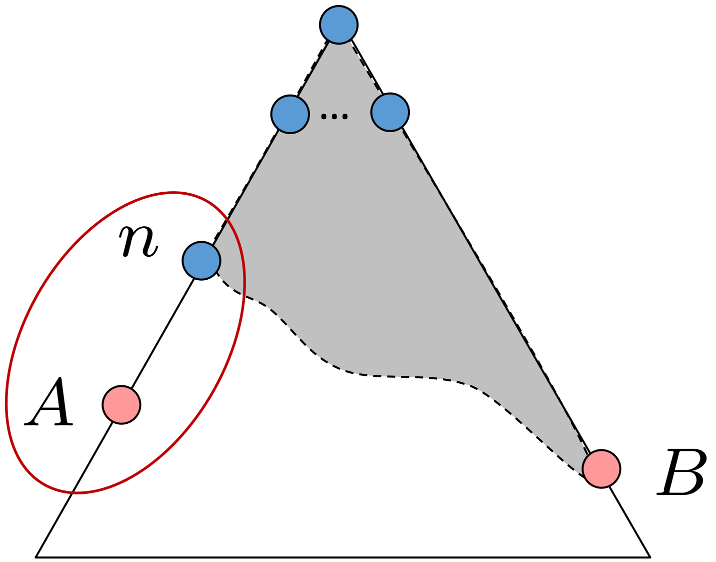
- Imagine B is on the fringe
- Some ancestor n of A is on the fringe, too (maybe A!)
- Claim: ***n will be expanded before B***
    1. f(n) is less than or equal to f(A)



Optimality of A* Tree Search

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 than or equal to f(A)

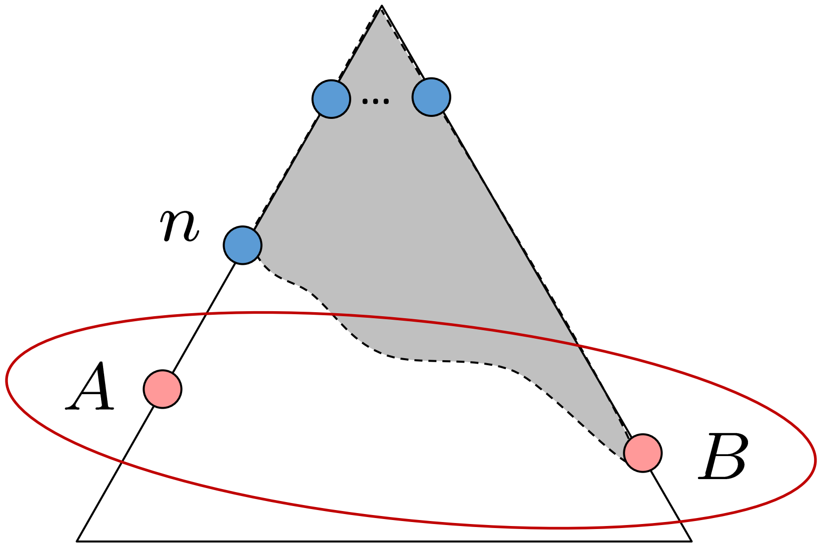


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

Optimality of A* Tree Search

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 than or equal to f(A)
  2. f(A) is less than f(B)



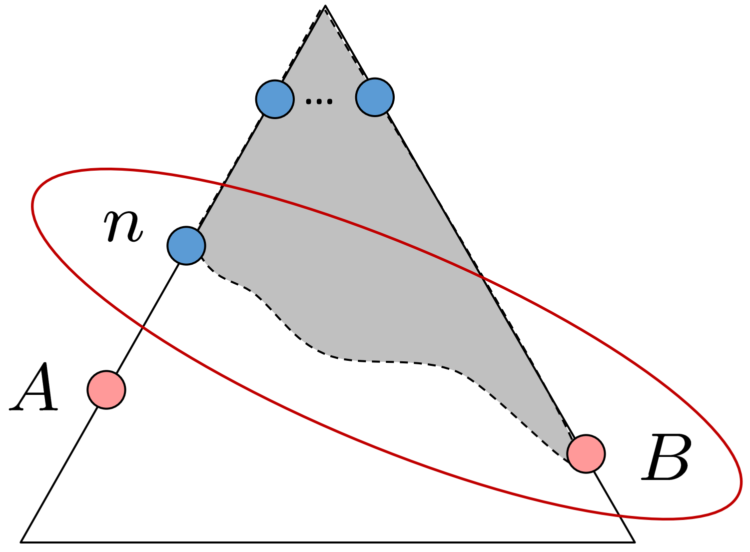
$g(A) < g(B)$ B is suboptimal

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

Optimality of A* Tree Search

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

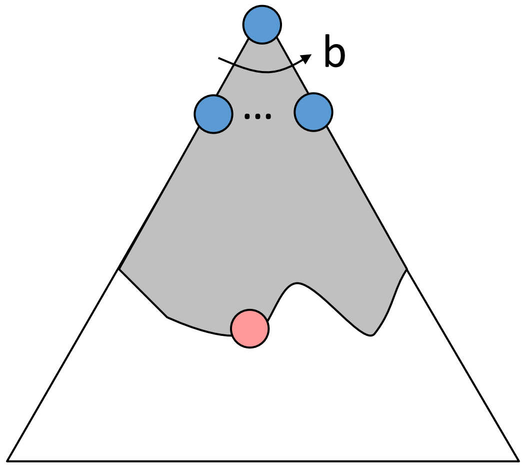


$$f(n) \leq f(A) < f(B)$$

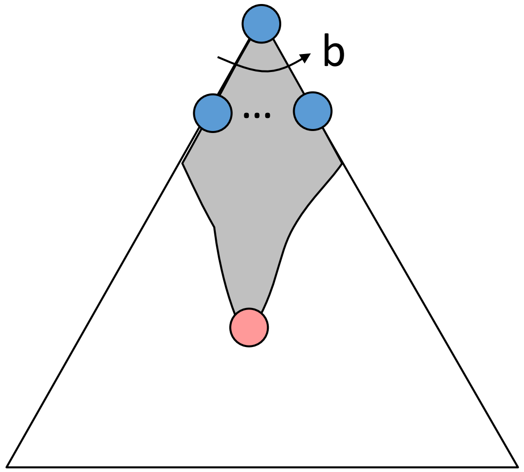
Punchline: A* is optimal, due to admissibility of h

UCS v A*

Uniform-Cost

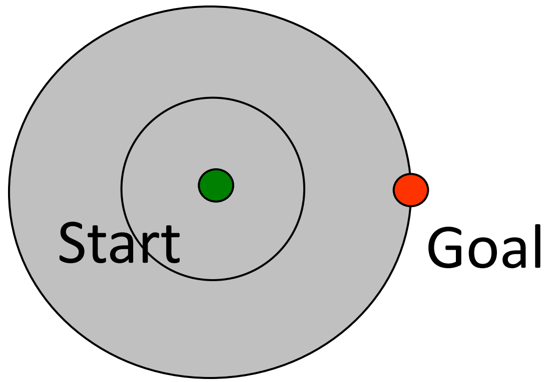


A*

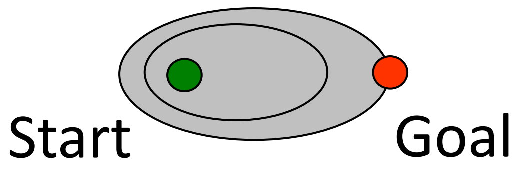


UCS vs A* Contours

- Uniform-cost expands equally in all “directions”



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

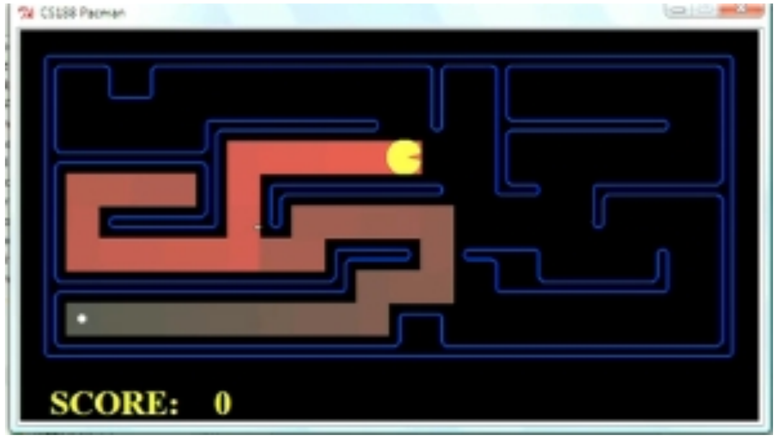


Video of Demo Contours: UCS

Video of Demo Contours: Greedy

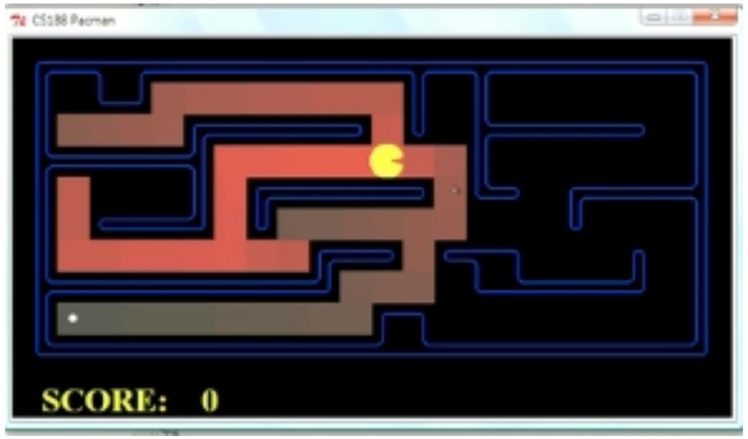
Video of Demo Contours: A*

Video of Demo Contours, PacMan: A*



Greedy

Uniform Cost



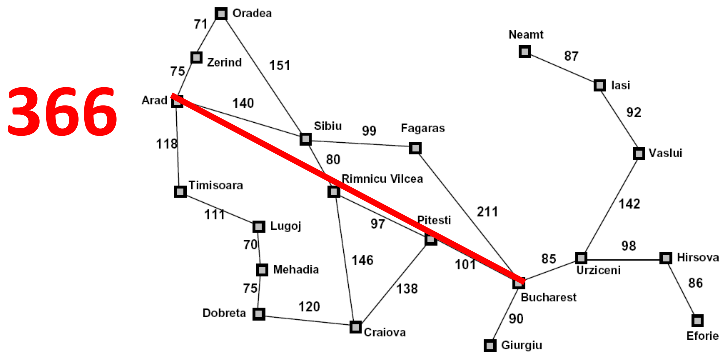
A*

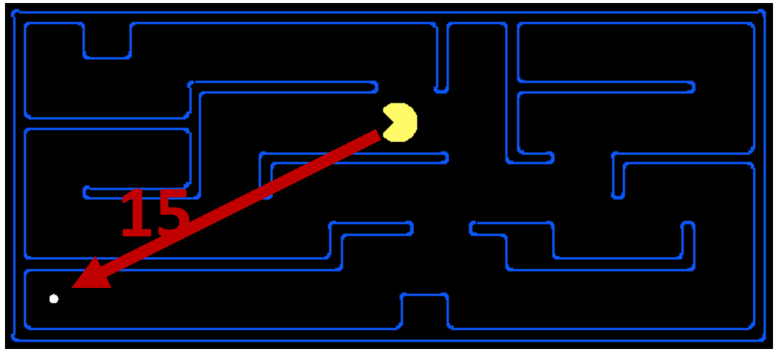
Designing heuristics

Creating admissible heuristics

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

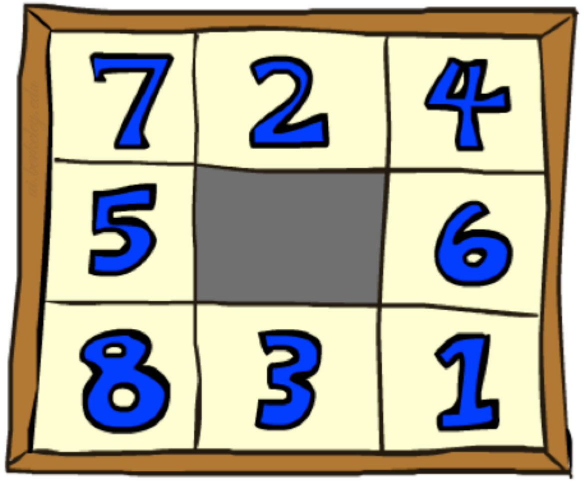
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available



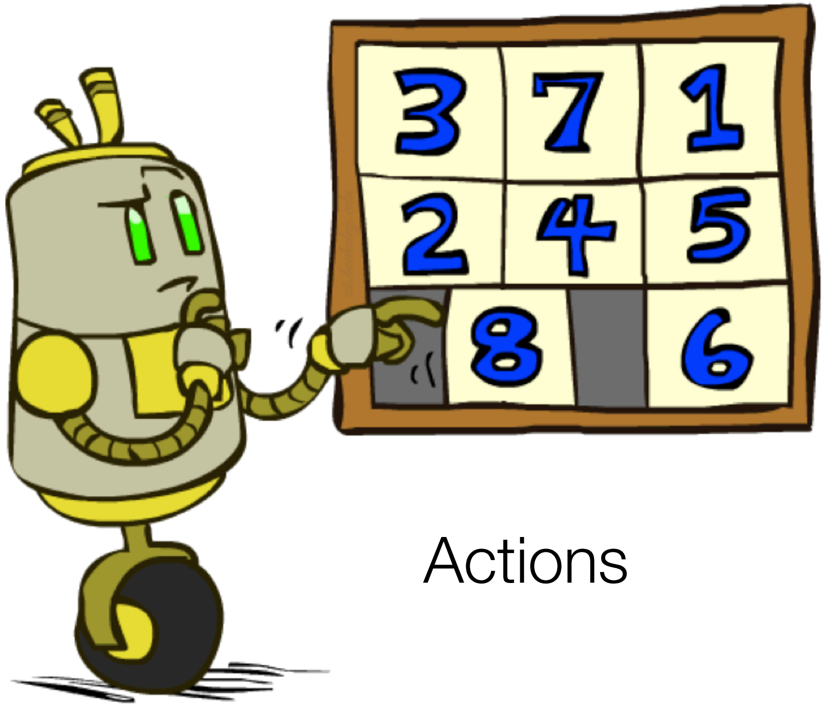


- Inadmissible heuristics are often useful too

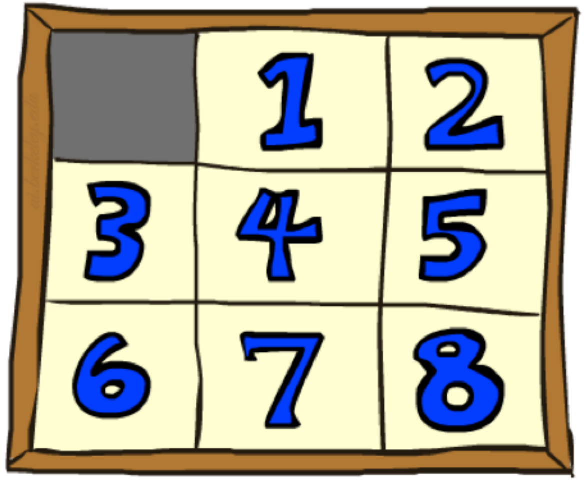
Example: 8 Puzzle



Start State



Actions

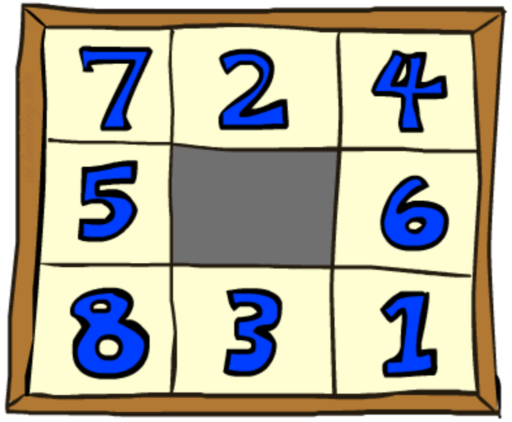


Goal State

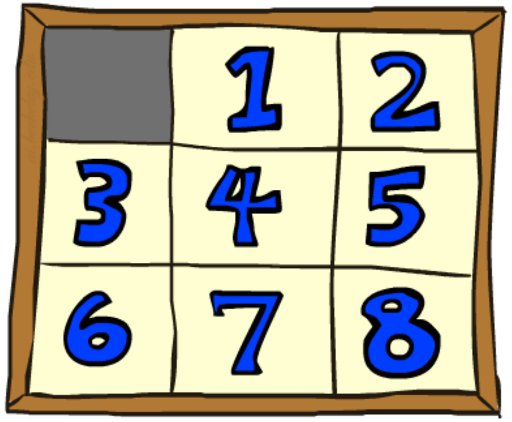
- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

8 Puzzle I

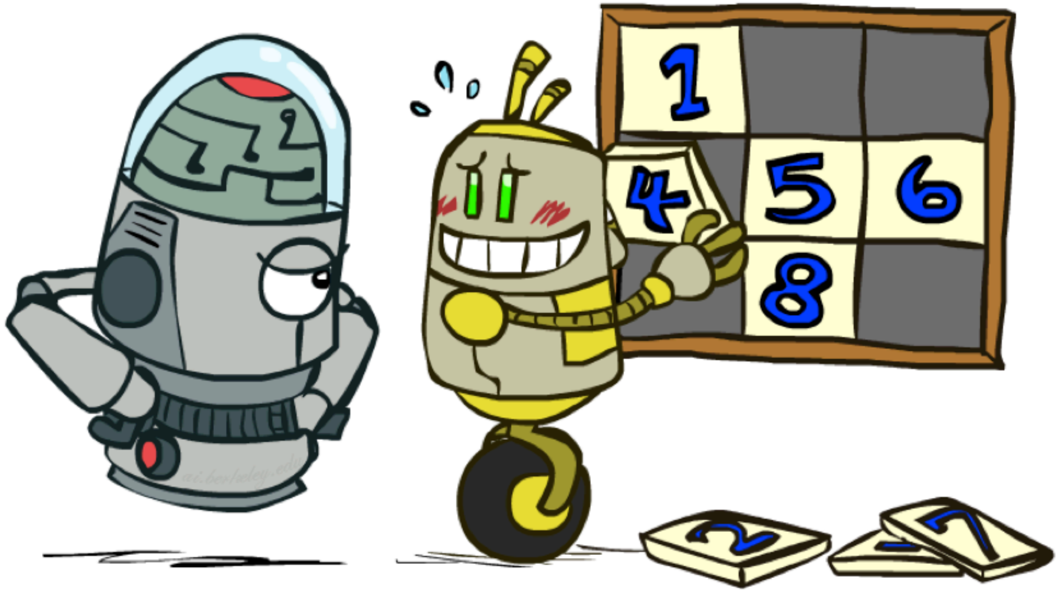
- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic



Start State



Goal State

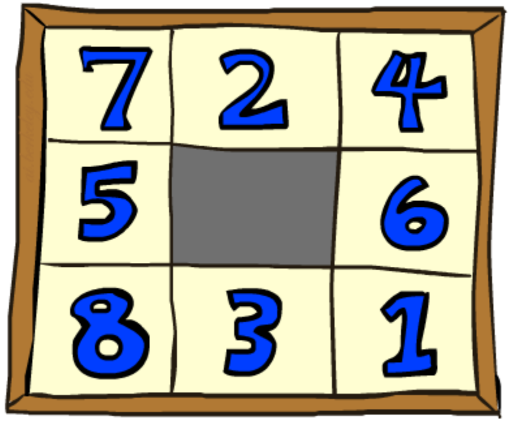


	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

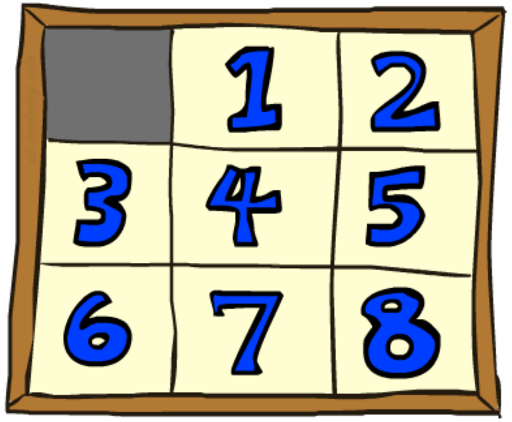
Statistics from Andrew Moore

8 Puzzle II

- 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



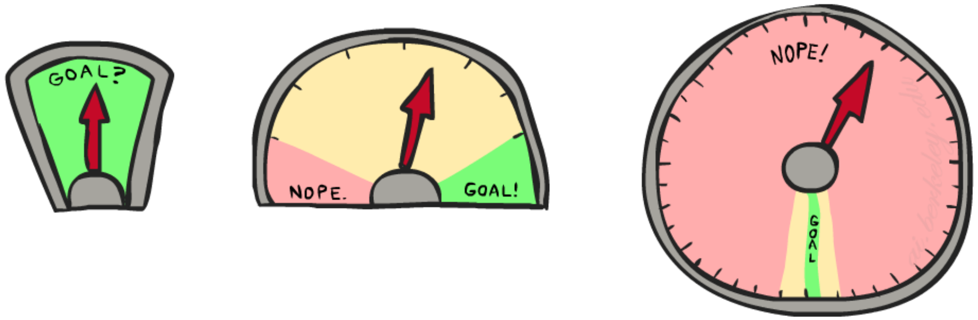
Goal State

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

8 Puzzle II

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

Trivial heuristics, dominance

Dominance: $h_a \geq h_c$ if

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

Heuristics form a semi-lattice:

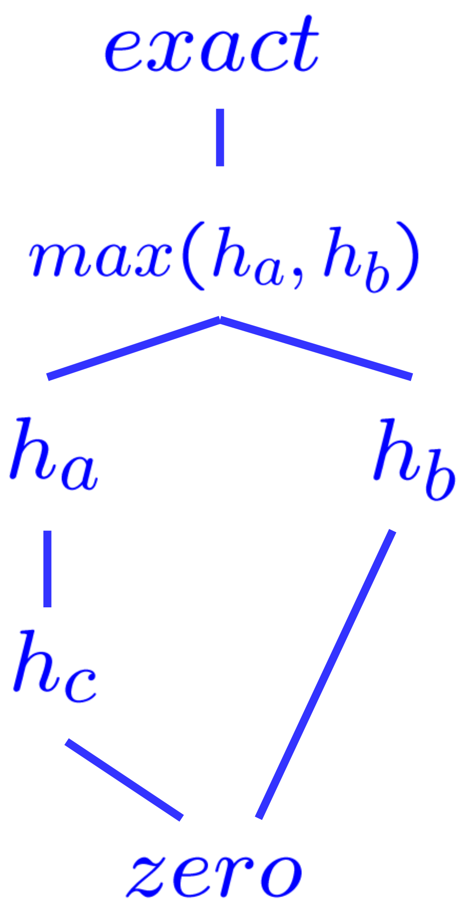
Max of admissible heuristics is admissible

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

Trivial heuristics

Bottom of lattice is the zero heuristic (what does this give us?)

Top of lattice is the exact heuristic



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

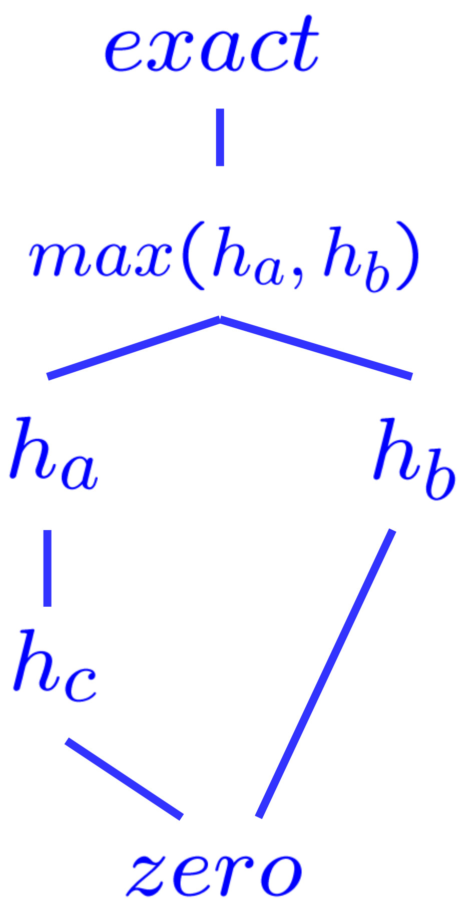
$$h(n) = max(h_a(n), h_b(n))$$

Trivial heuristics

Bottom of lattice is the zero heuristic

Top of lattice is the exact heuristic

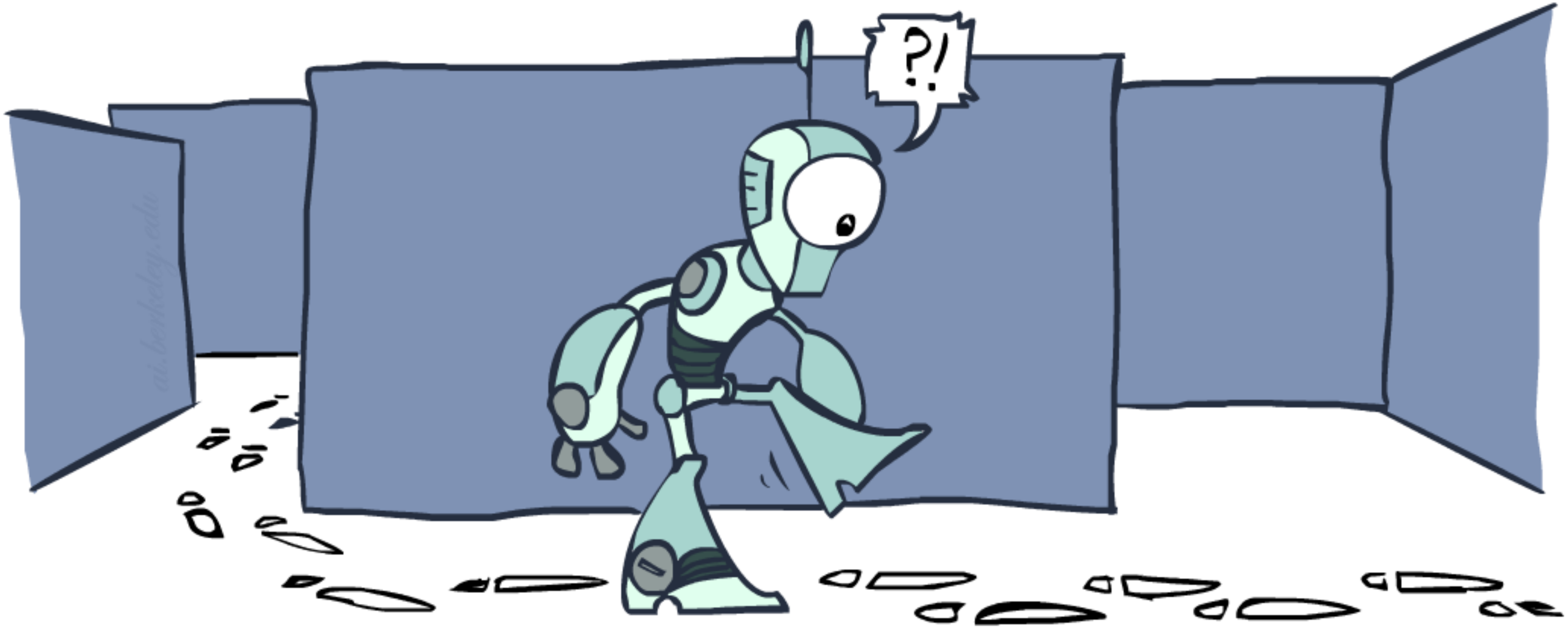
Q: what happens if we use h(n) = 0 for all n?



Learning heuristics

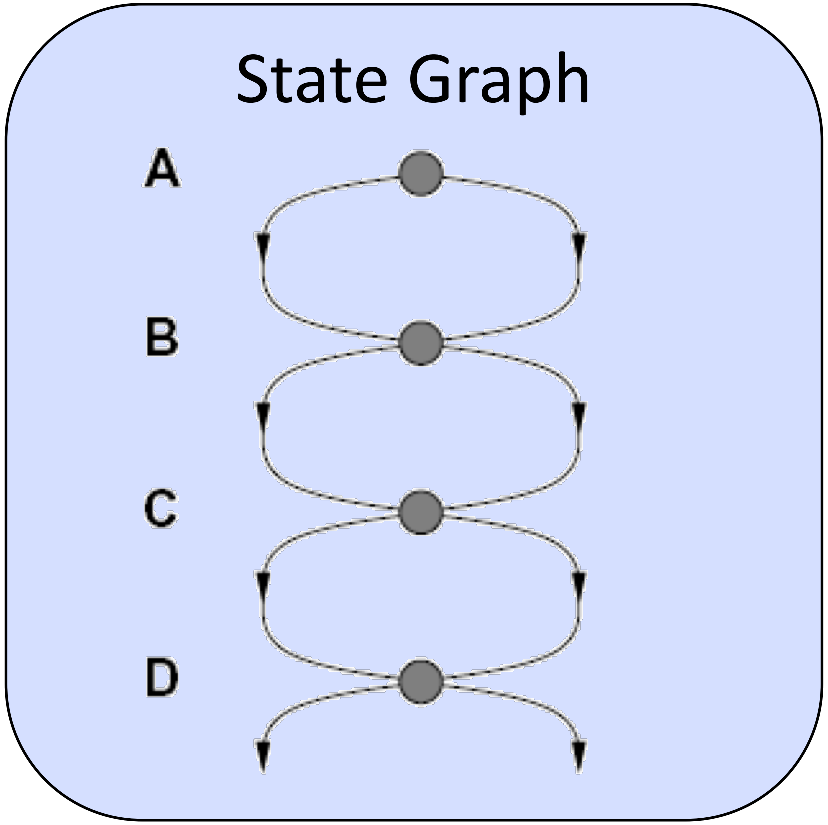
- Rather than hand-crafting heuristics, what if we let the machine *learn* a heuristic function?
- We'll come back to this once we cover machine learning

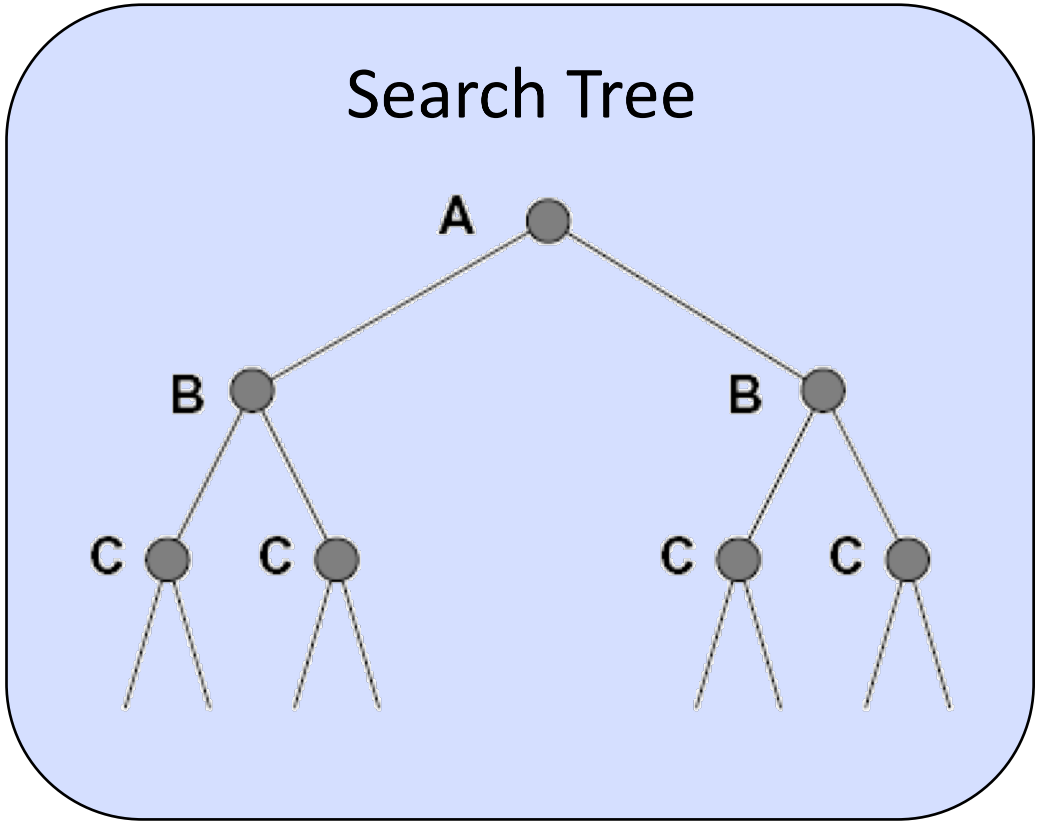
Graph search: don't retrace steps



Tree search: extra work!

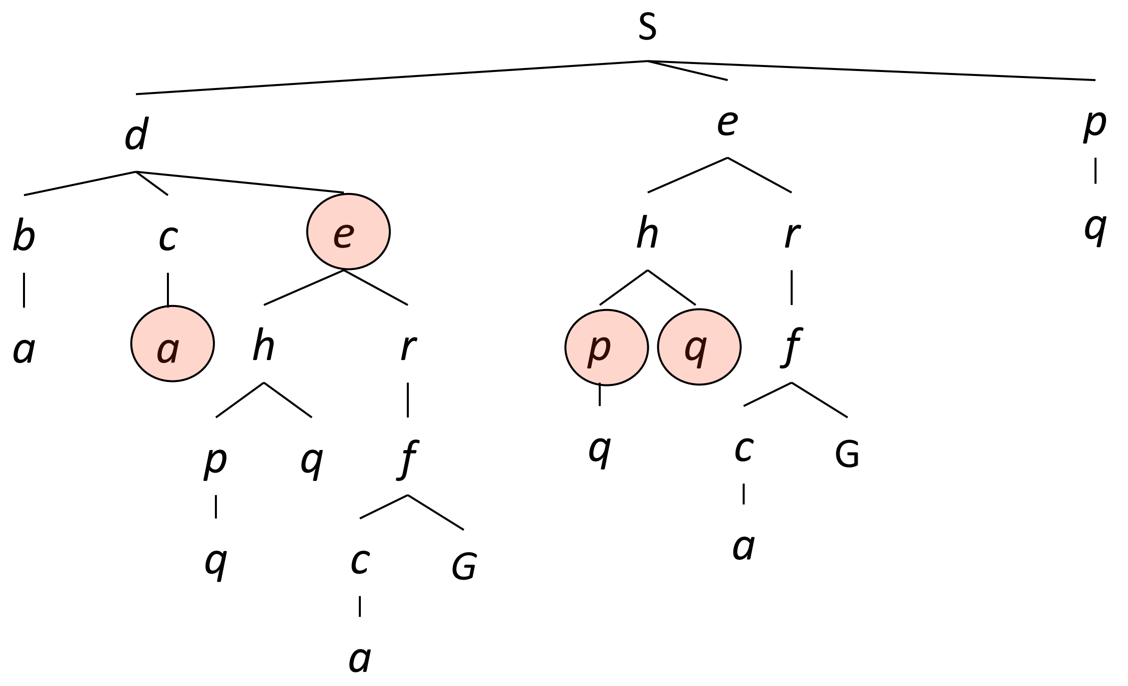
Failure to detect repeated states can cause exponentially more work.





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

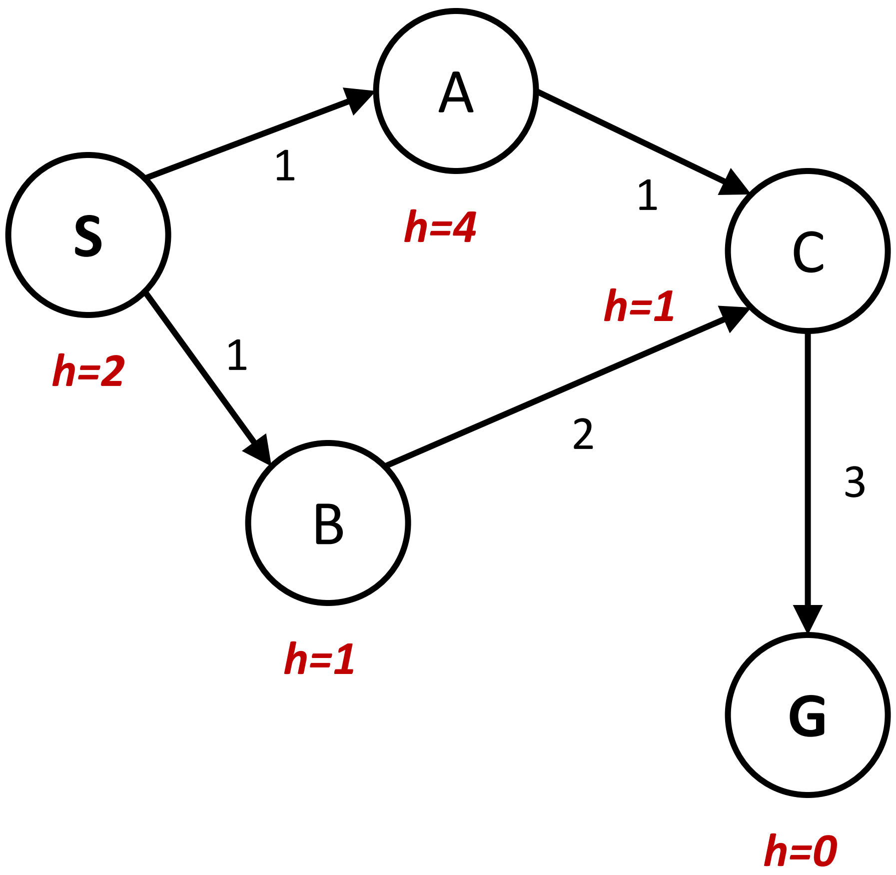
Important: **store the closed set as a set**, not a list

Can graph search wreck completeness? Why/why not?

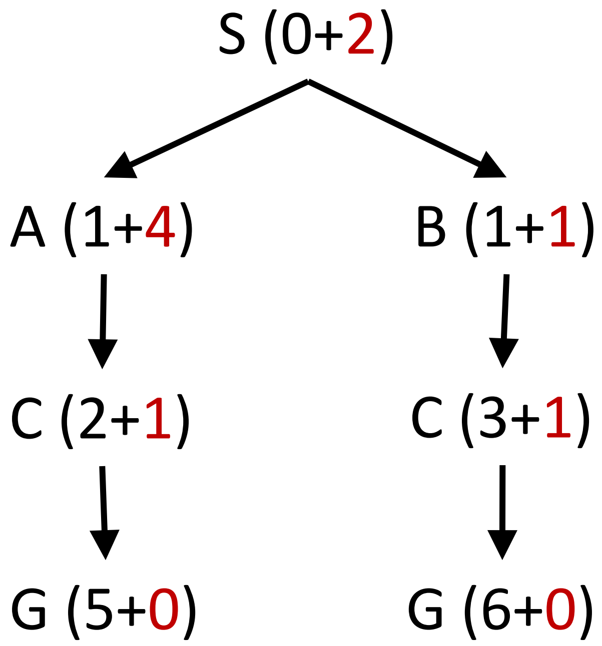
How about optimality?

A* Graph search gone wrong?

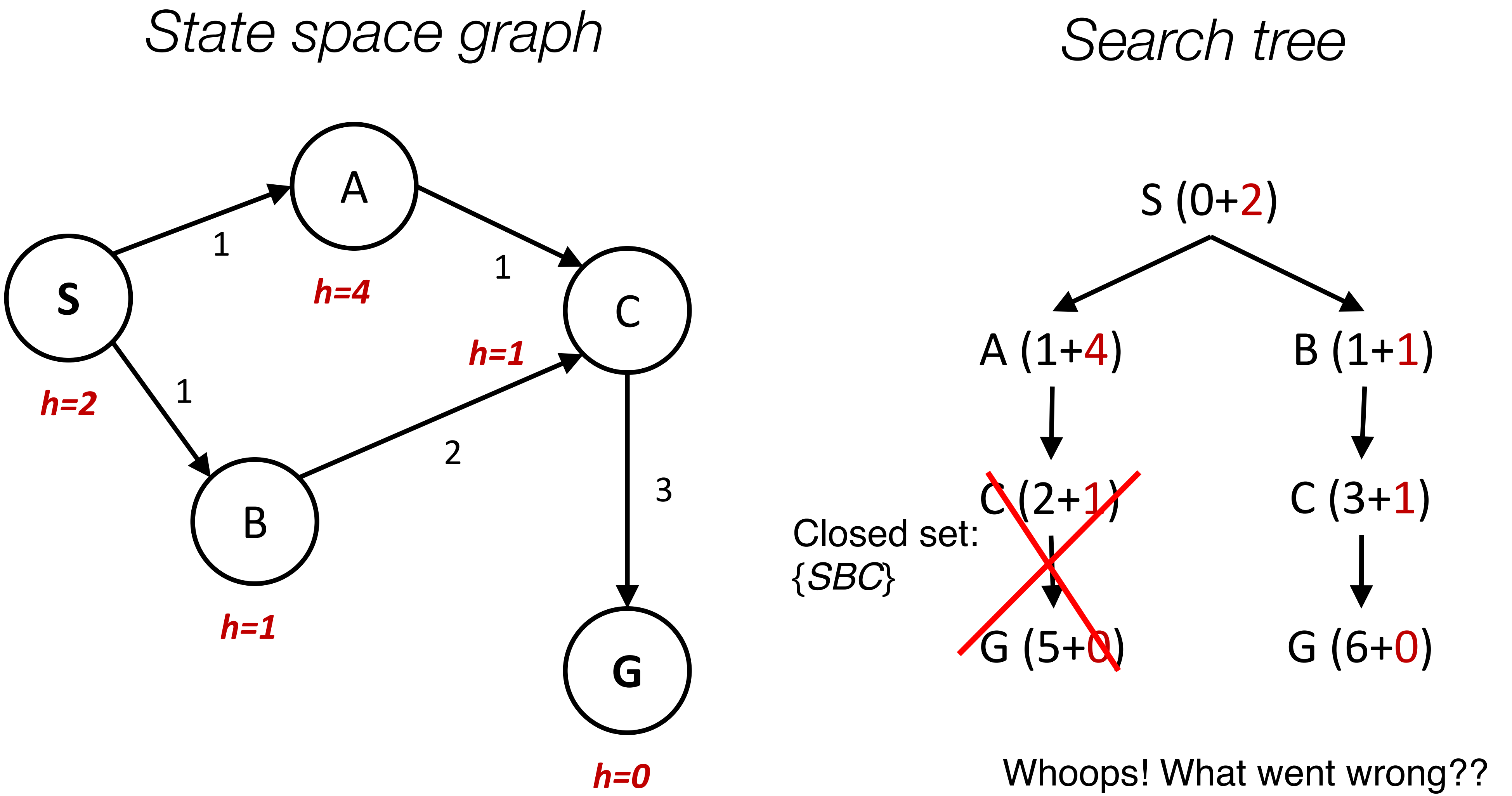
State space graph



Search tree

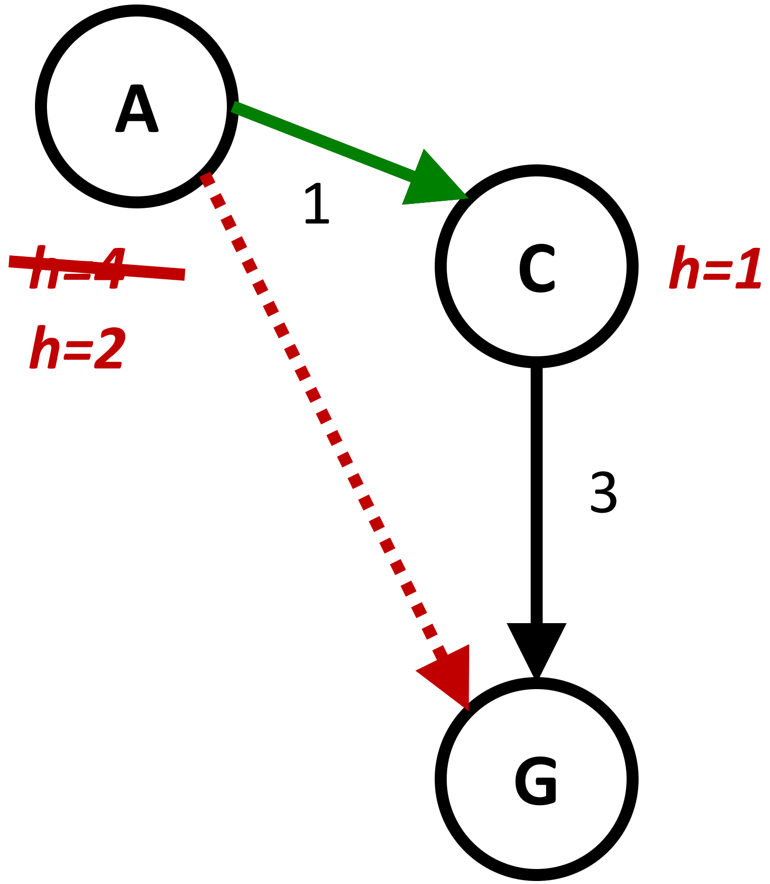


A* Graph search gone wrong?



Whoops! What went wrong??

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

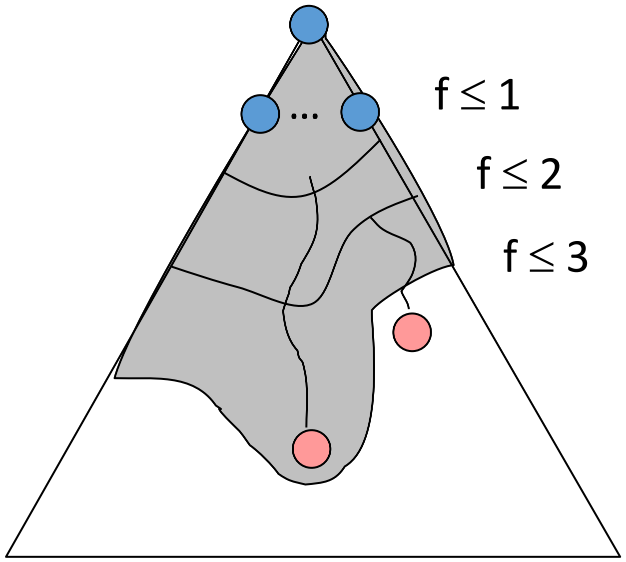
h(A) ≤ cost(A to C) + h(C)

A* graph search is optimal

Optimality of A* graph search

Sketch: 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

Tree search:

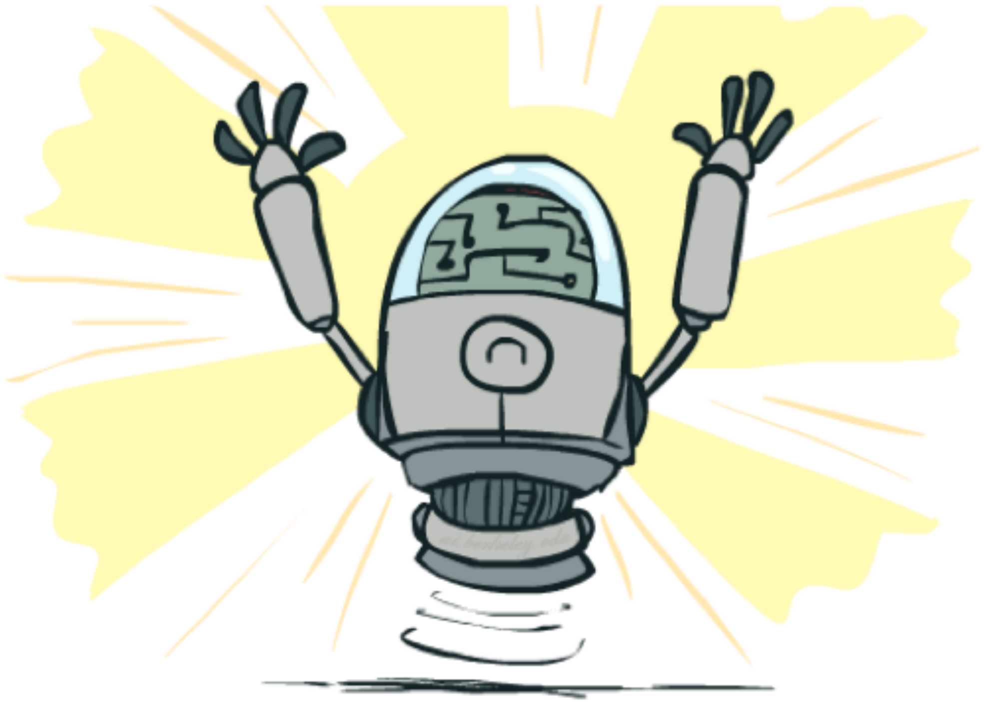
- A* is optimal if heuristic is admissible
- UCS is a special case (h = 0)

Graph search:

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

Consistency implies admissibility

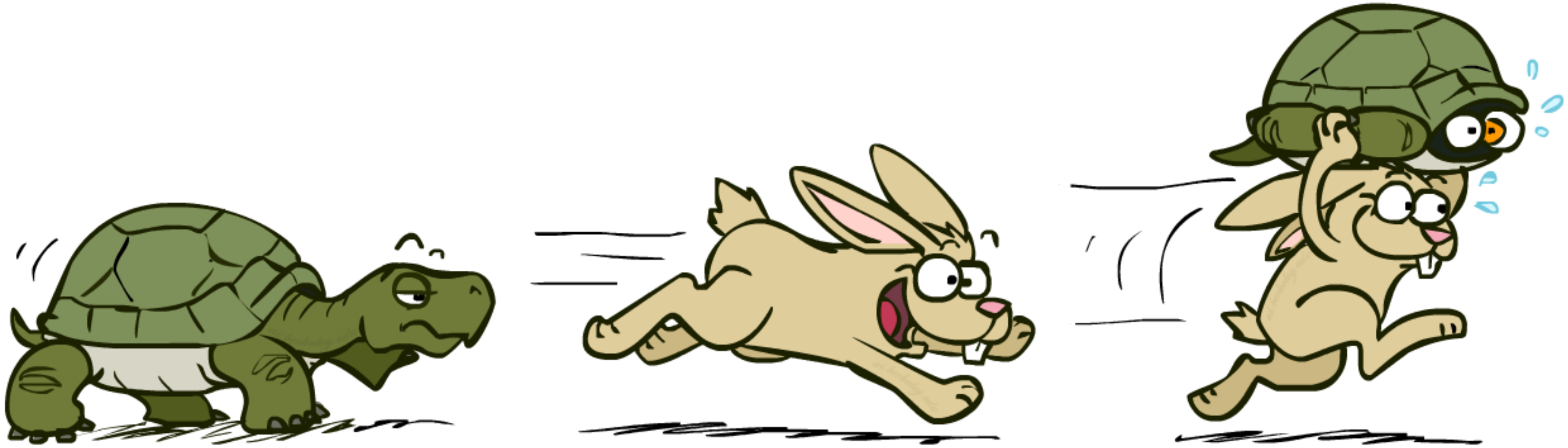
In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



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



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
```

That’s all for today.

Up next time: Beyond “classical” search – dealing with constraints and stochastic environments

Be sure to make progress on the homeworks!