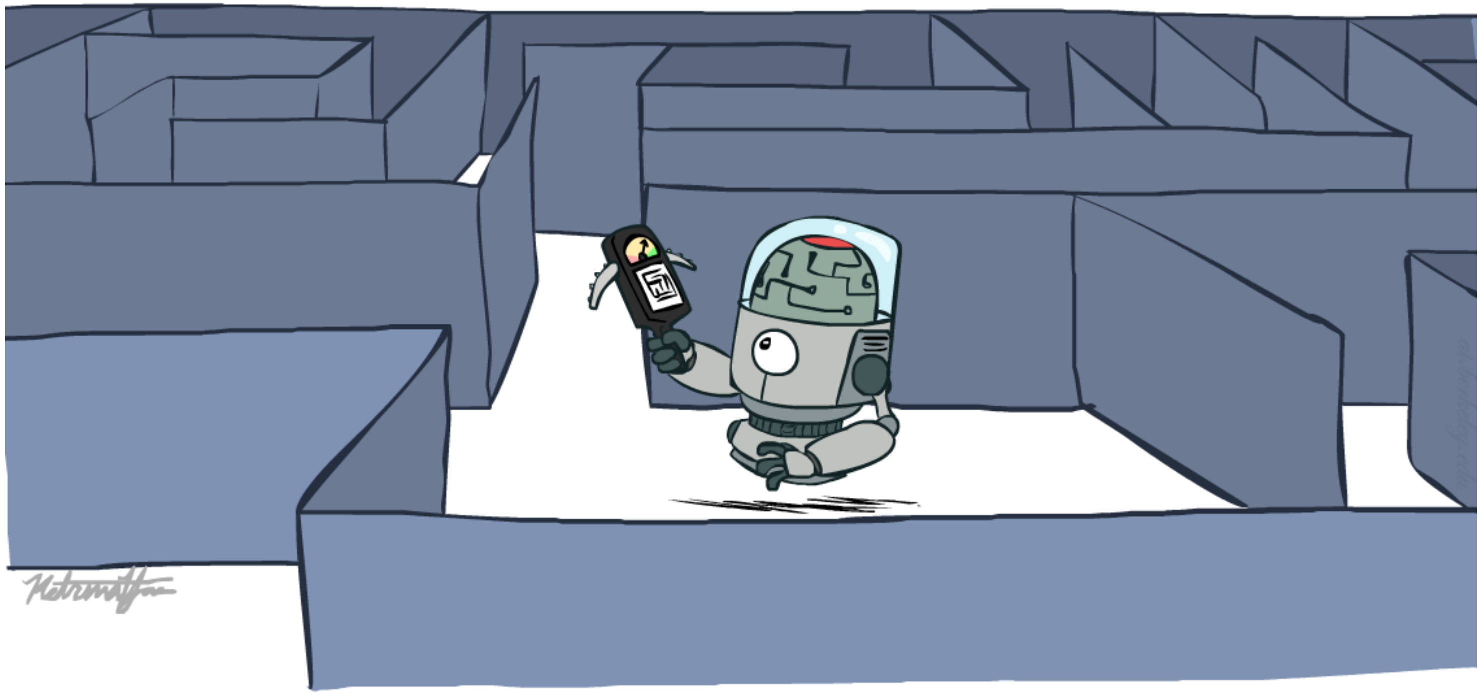
COE 4213564
Introduction to Artificial Intelligence
Informed Search

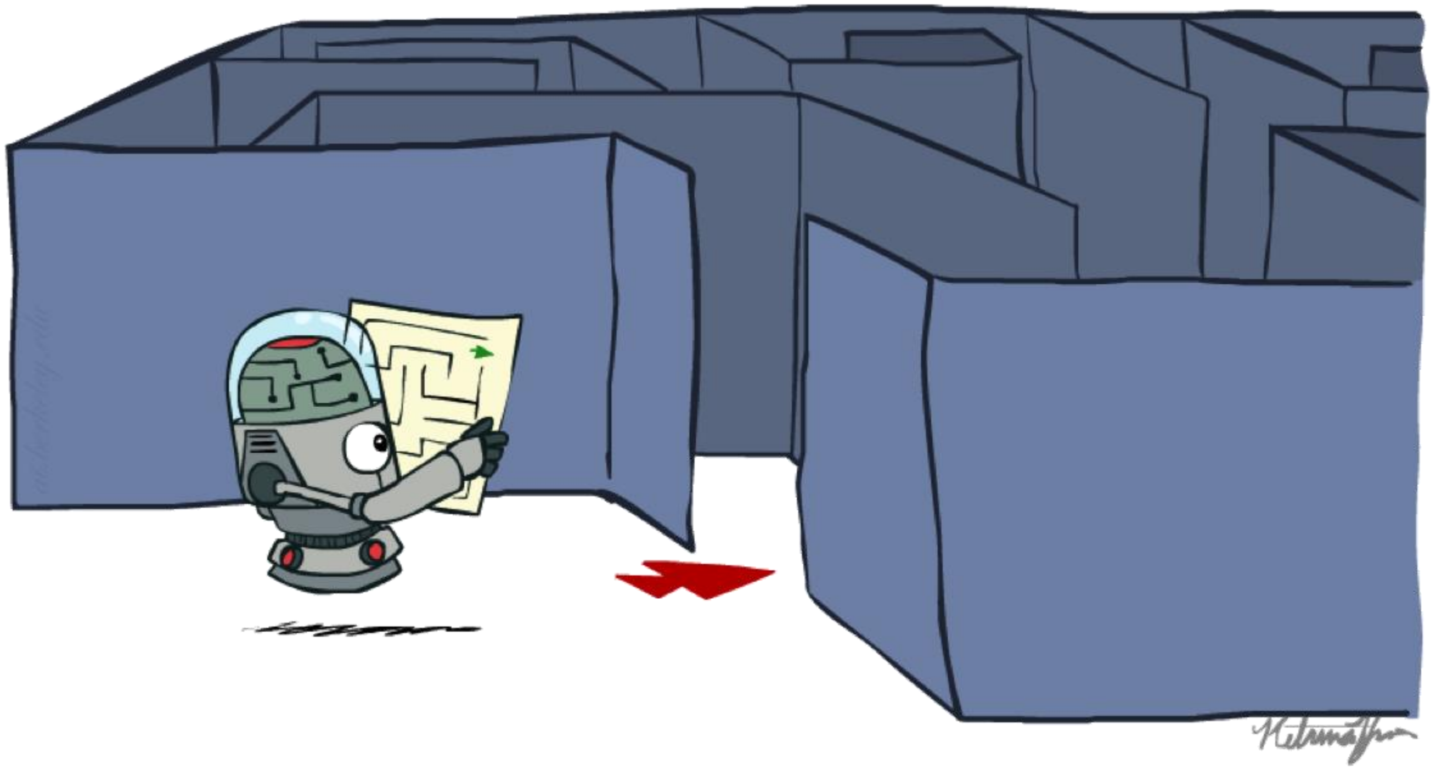


Many slides are adapted from CS 188 (http://ai.berkeley.edu), CIS 521, CS 221, CS182, CS4420.

Outline

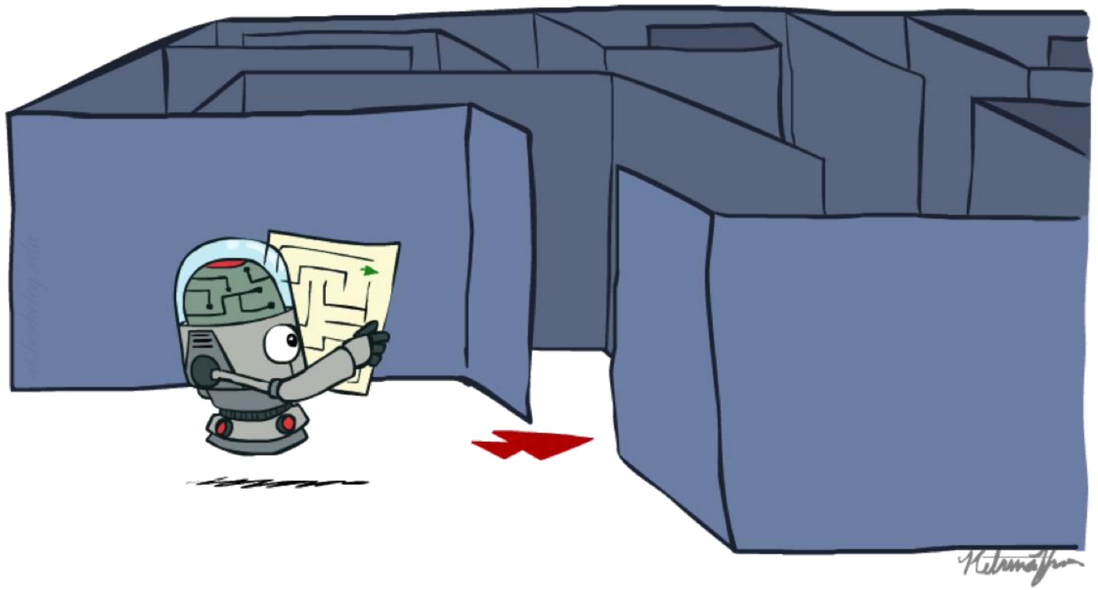
- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search
- Graph Search

Recap: Search



Recap: Search

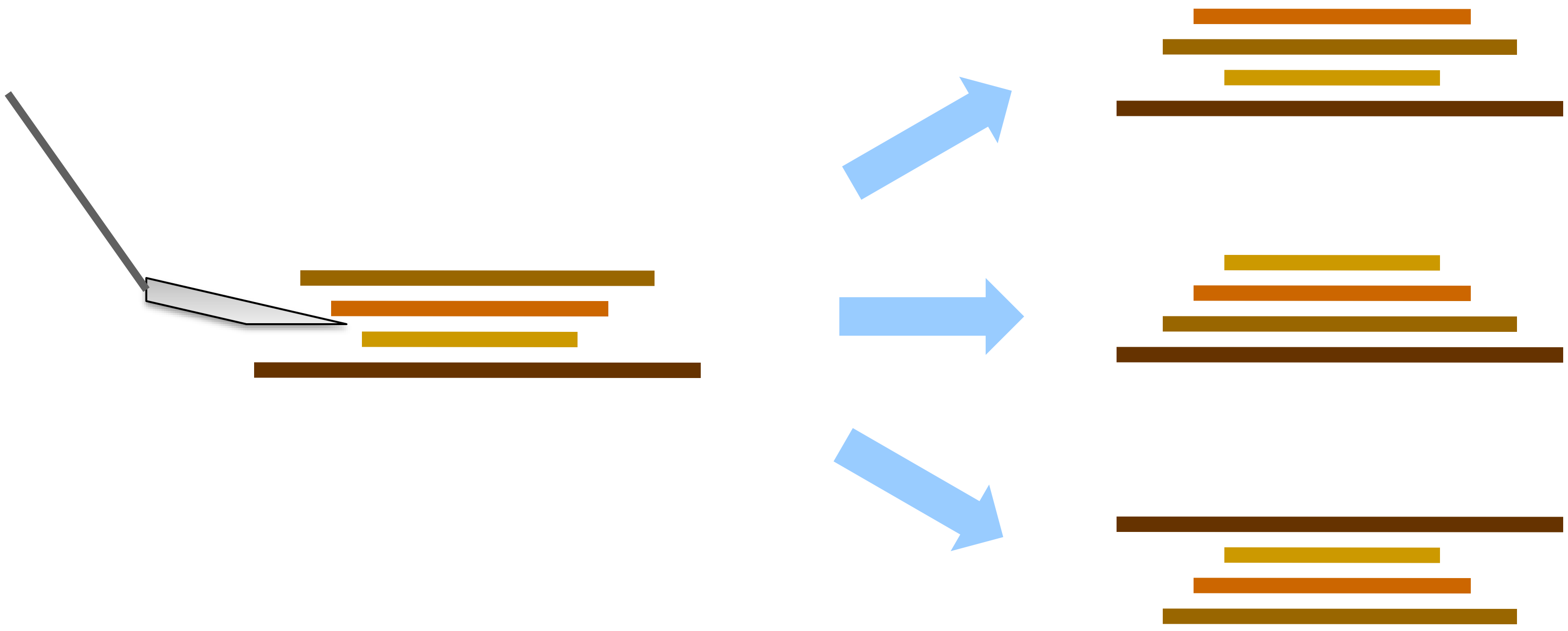
- Search problem:
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - Start state and goal test



- 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

Example: Pancake Problem



- Pancake Sorting Problem: We are given a stack of n pancakes, each of different size. Our goal is to sort this stack from smallest to largest (largest being on the bottom of the stack).
- The only thing we are allowed to do is to insert the spatula in between two pancakes (or between the bottom pancake and the plate), and flip over all the pancakes that are on top of the spatula.

Cost: Number of pancakes flipped

Example: Pancake Problem

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES

Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†

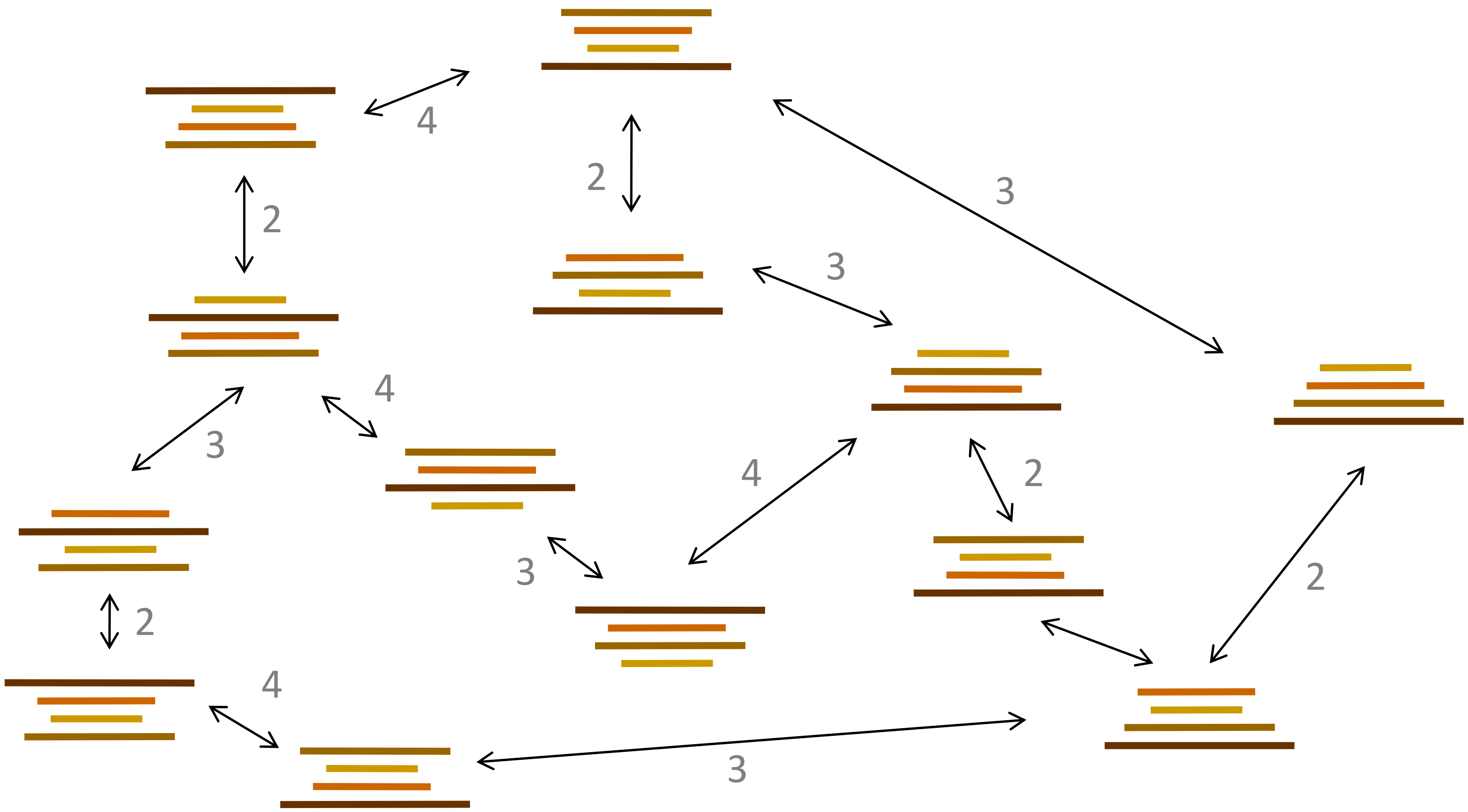
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

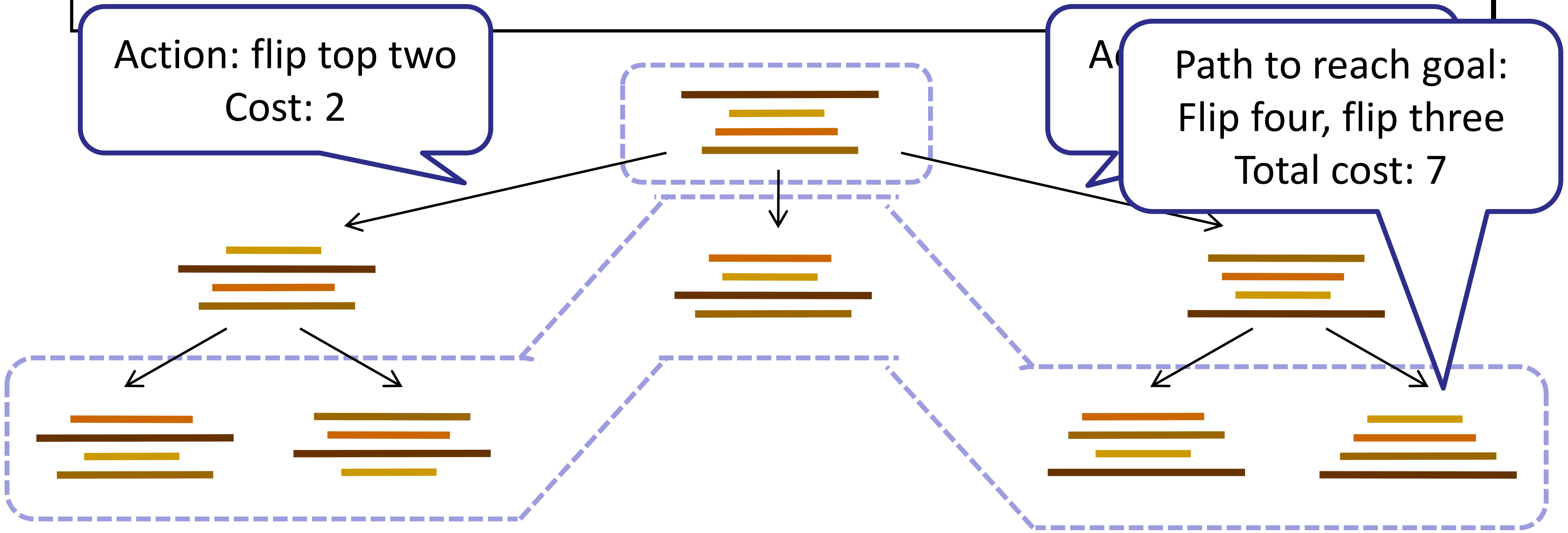
Example: Pancake Problem

State space graph with costs as weights



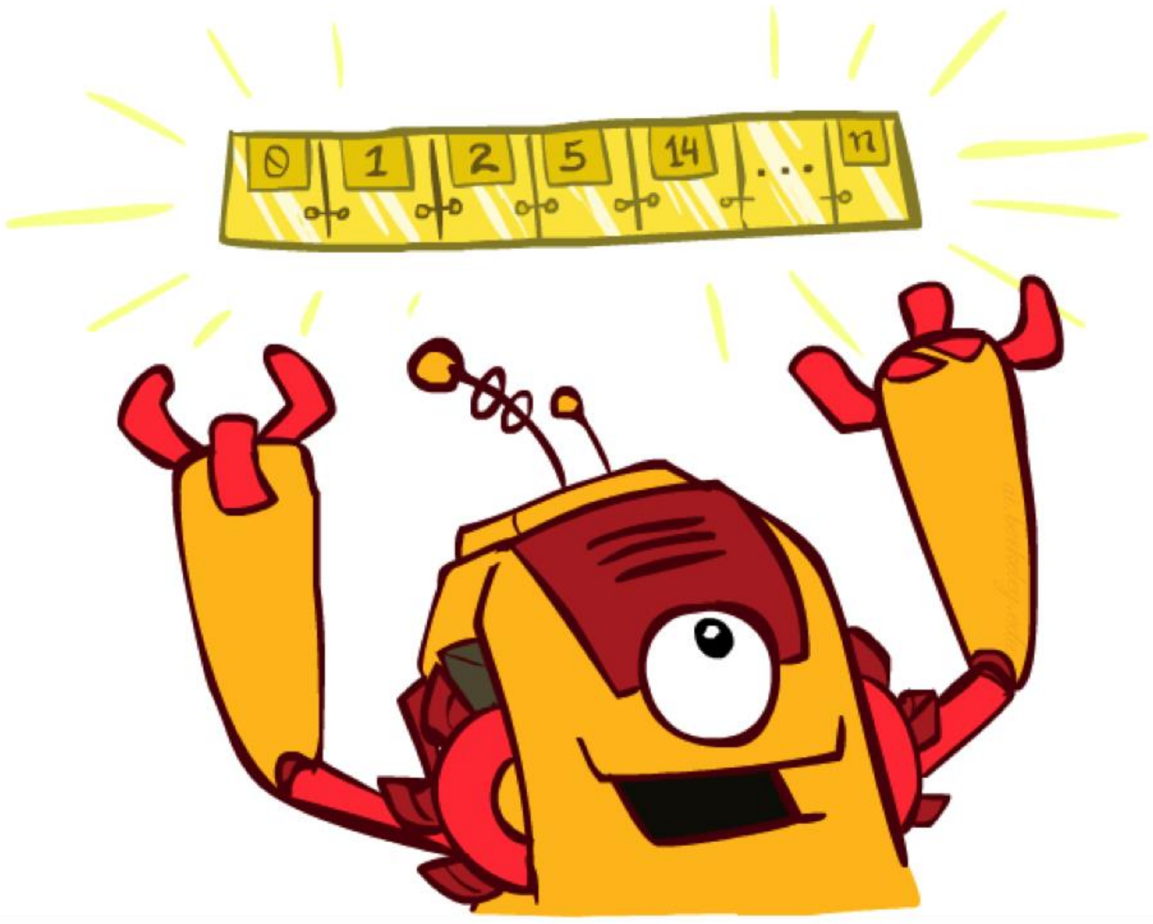
General Tree Search

```
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
```

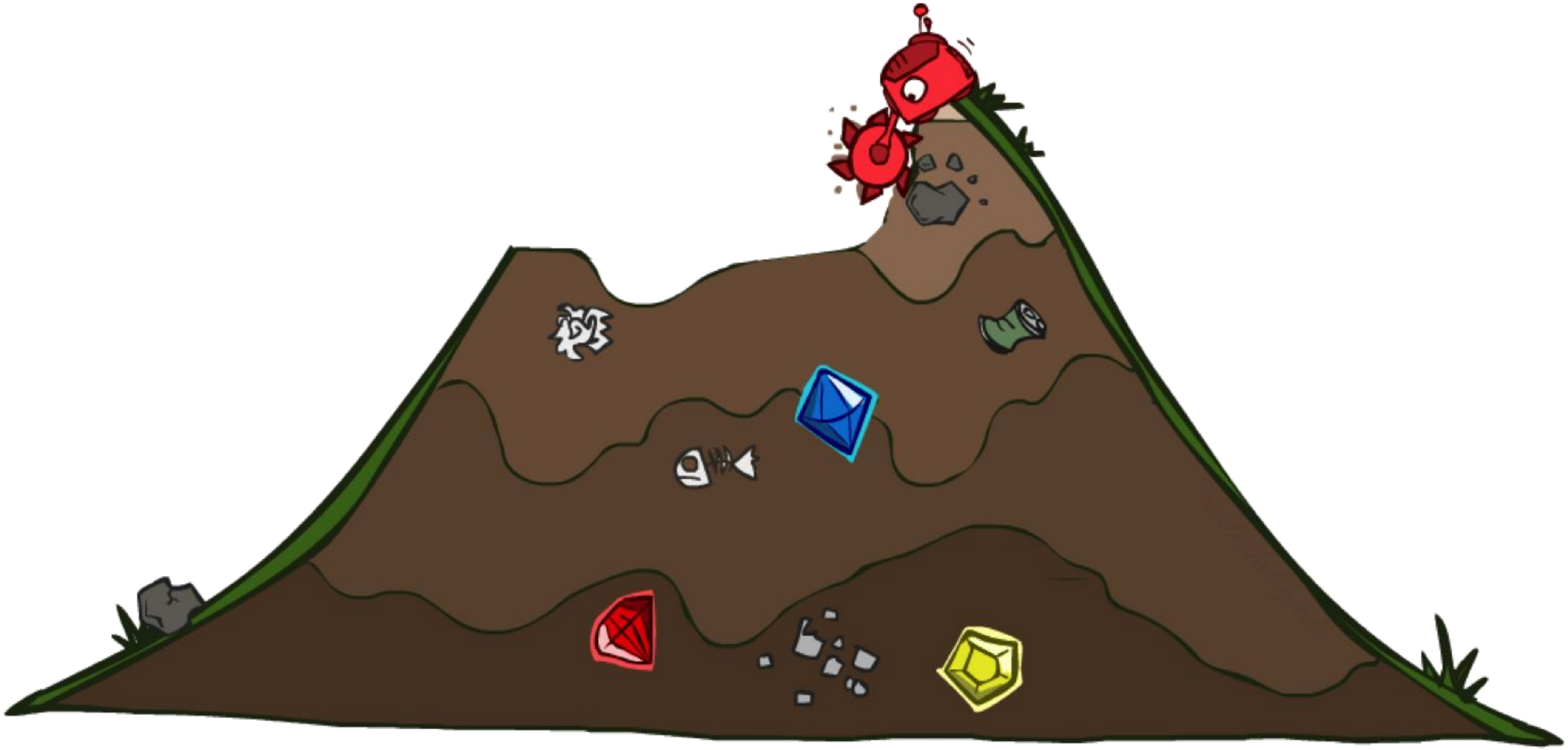


The One Queue

- 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



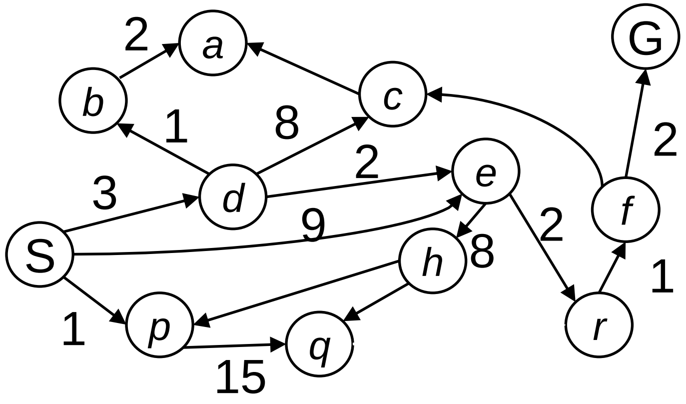
Uninformed Search



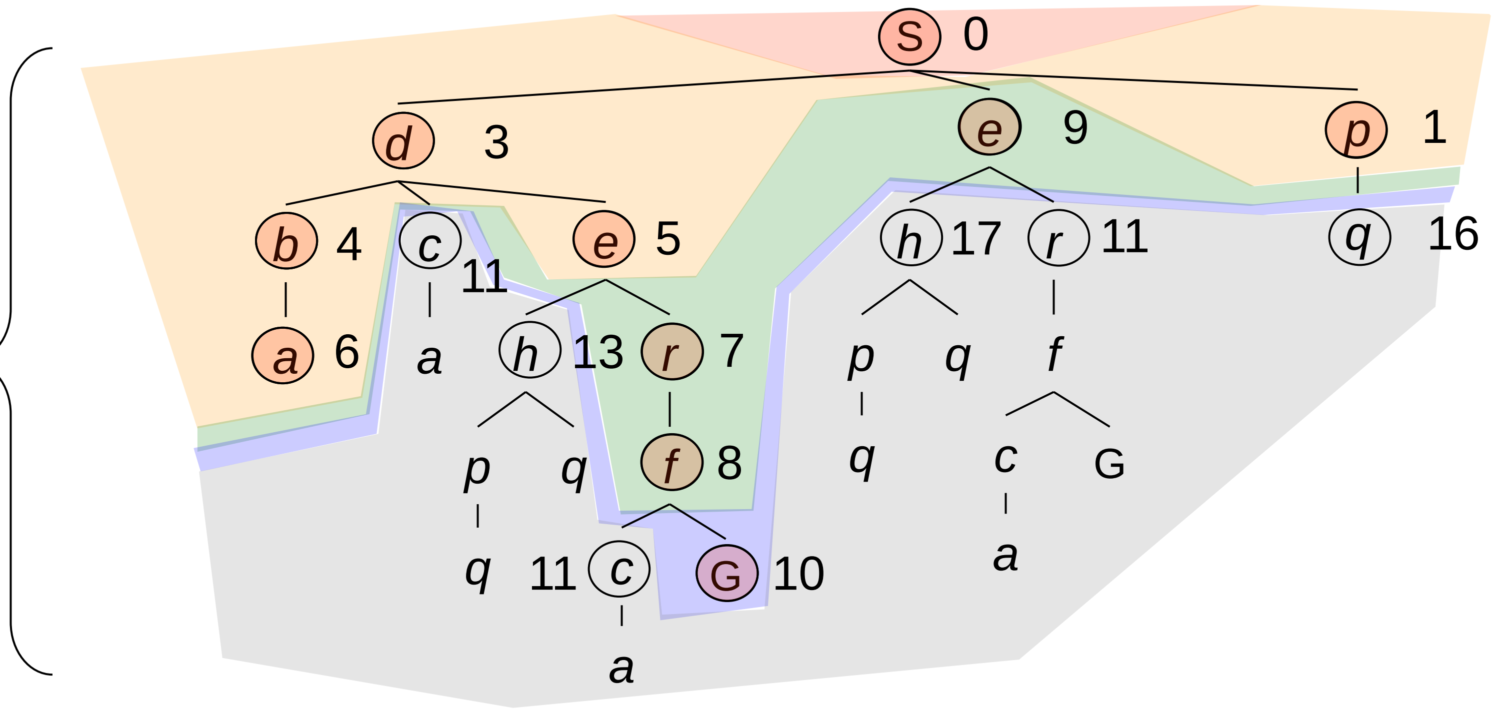
Uniform Cost Search

Strategy: expand a cheapest node first from the root:

Fringe is a priority queue (priority: cumulative cost)

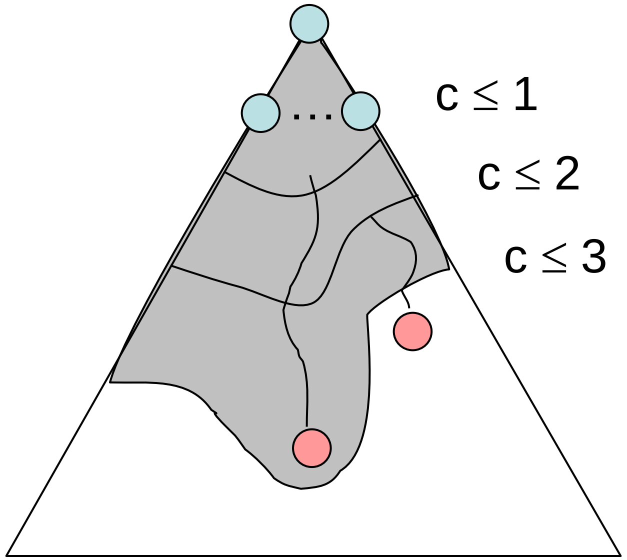


Cost contours



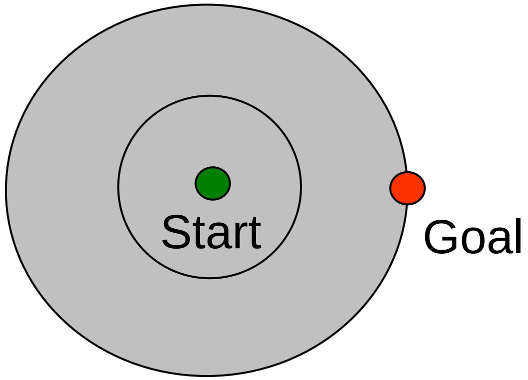
Uniform Cost Search

- Strategy: expand lowest path cost cost of the path from the root to the current node



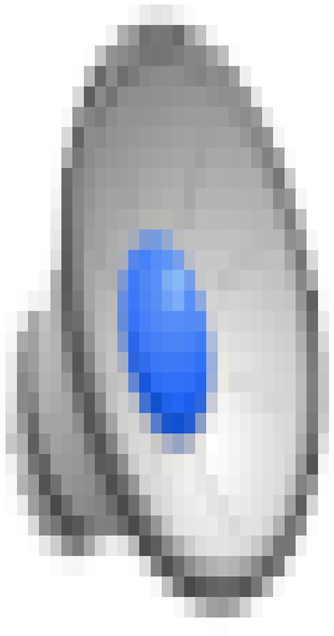
- The good: UCS is complete and optimal!

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

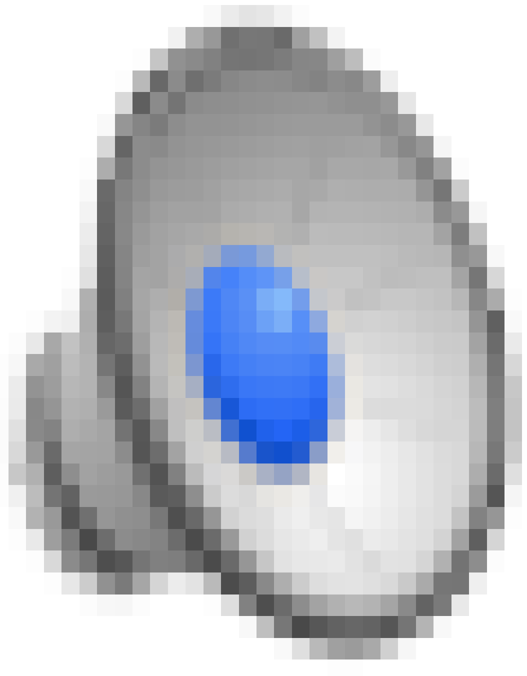


[Demo: contours UCS empty (L3D1)]
[Demo: contours UCS pacman small maze (L3D3)]

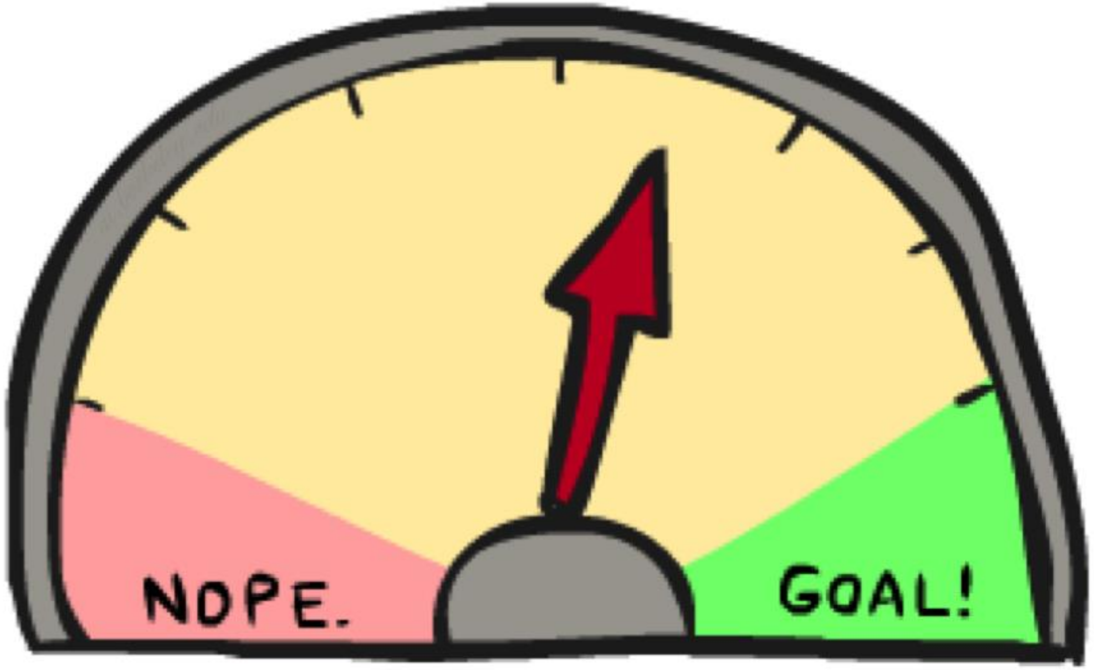
Video of Demo Contours UCS Empty



Video of Demo Contours UCS Pacman Small Maze



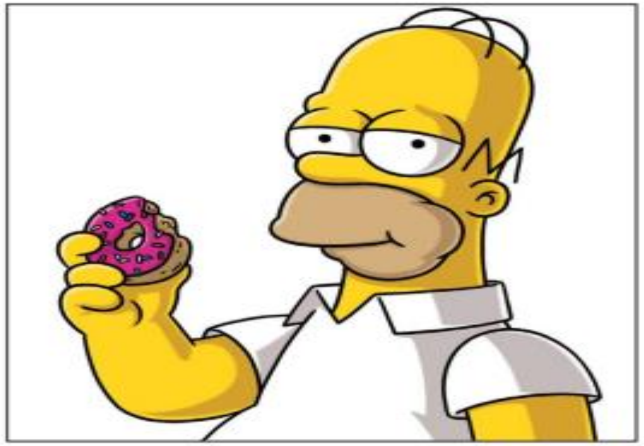
Informed Search



Informed Search Strategies

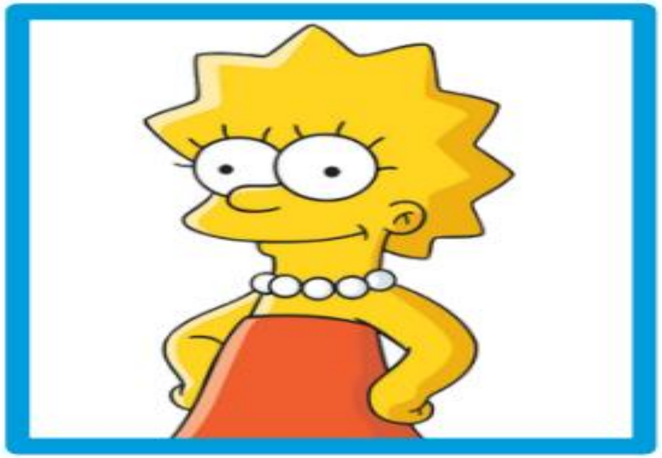
- Uninformed search strategies look for solutions by systematically generating new states and checking each of them against the goal
- This approach is very inefficient in most cases
- Most successor states are “obviously” a bad choice
- Such strategies do not know that because they have minimal problem-specific knowledge
- Informed search strategies exploit problem-specific knowledge as much as possible to drive the search
- They are almost always more efficient than uninformed searches and often also optimal

UNINFORMED VS. INFORMED



Uninformed

Can only generate successors and distinguish goals from non-goals



Informed

Strategies that know whether one non-goal is more promising than another

Informed Search Strategies

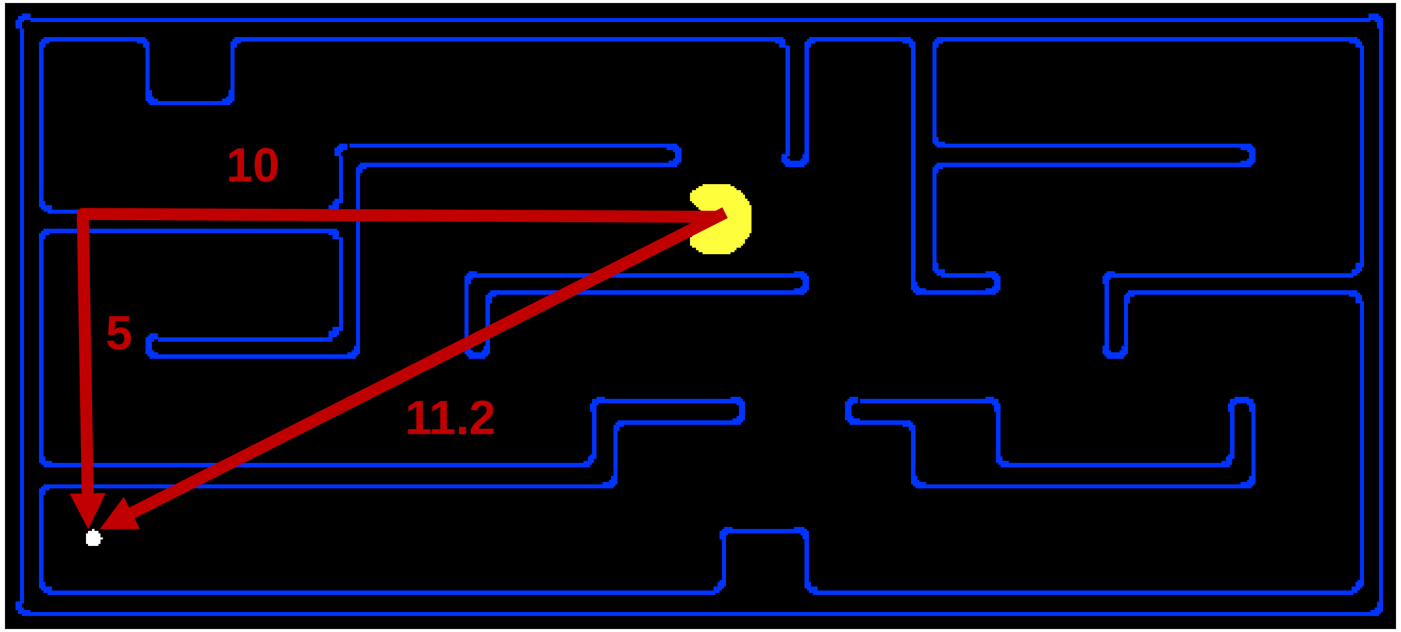
- Use the knowledge of the problem domain to build an evaluation function *h*
- For every node n in the search space, *h(n)* quantifies the desirability of expanding n in order to reach the goal
- Then use the desirability value of the nodes in the fringe to decide which node to expand next
- The evaluation function *h* is typically an imperfect measure of the goodness of the node
- i.e., the right choice of nodes is not always the one suggested by *h*
- The evaluation function is usually called heuristic function.

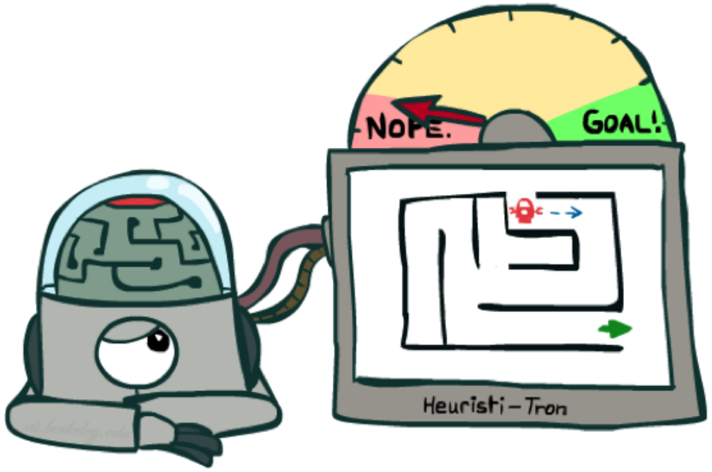
Heuristic

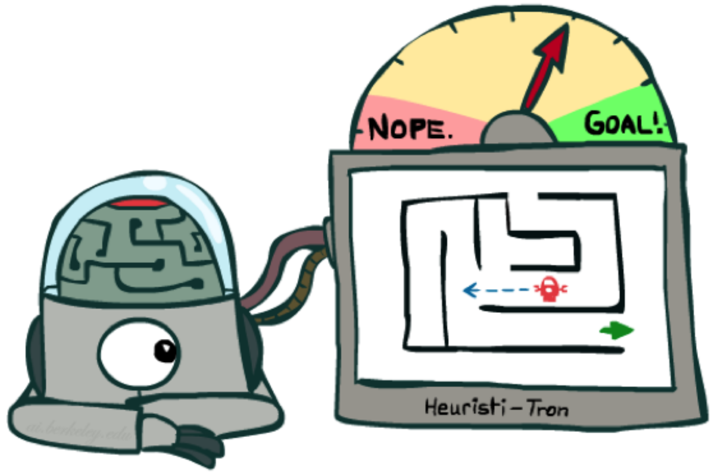
- Merriam-Webster's Online Dictionary
  - Heuristic (pron. \hyu-’ris-tik\): adj. [from Greek heuriskein to discover.] involving or serving as an aid to learning, discovery, or problemsolving by experimental and especially trial-and-error methods
- The Free On-line Dictionary of Computing
  - heuristic 1. A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. approximation algorithm.
- From WordNet (r) 1.6
  - heuristic adj 1: (computer science) relating to or using a heuristic rule 2: of or relating to a general formulation that serves to guide investigation [ant: algorithmic] n : a commonsense rule (or set of rules) intended to increase the probability of solving some problem [syn: heuristic rule, heuristic program]

Search Heuristics

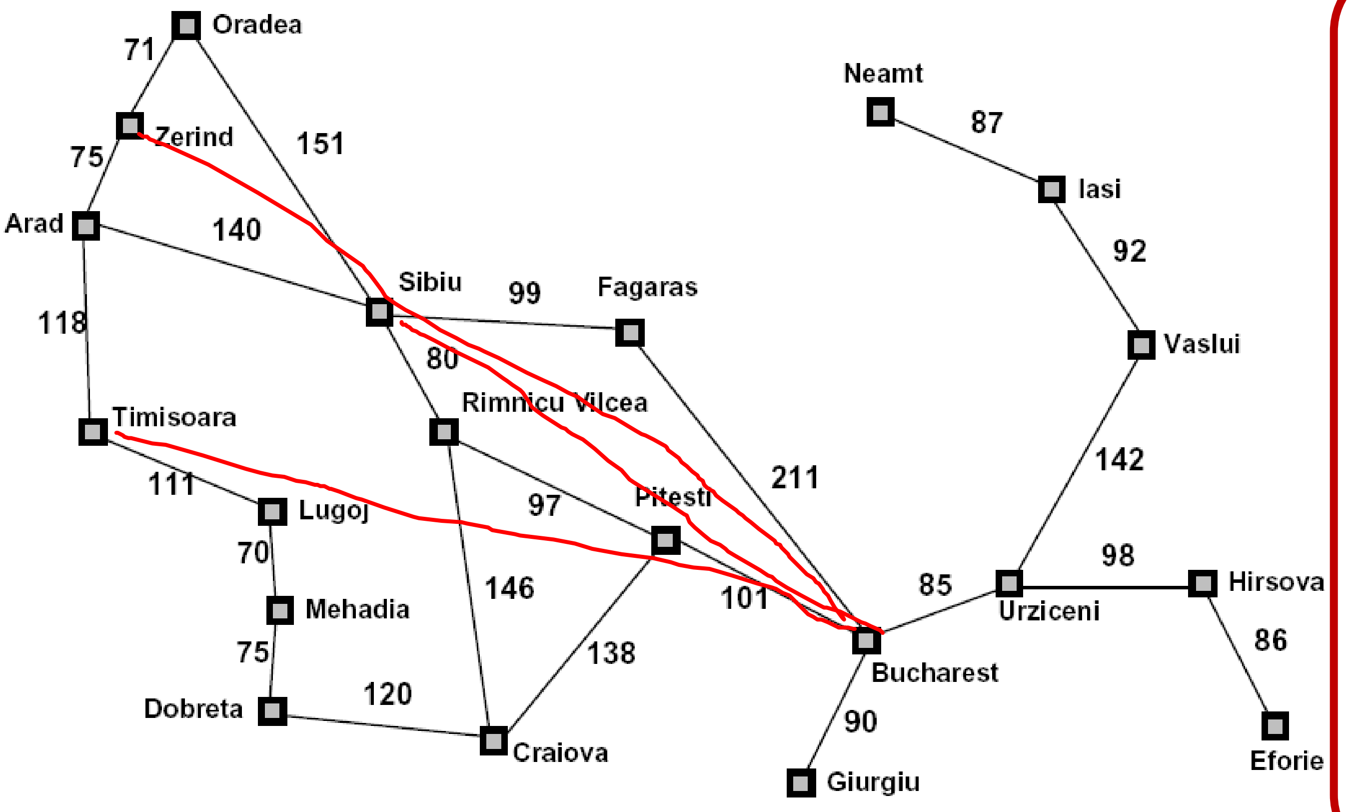
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing







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

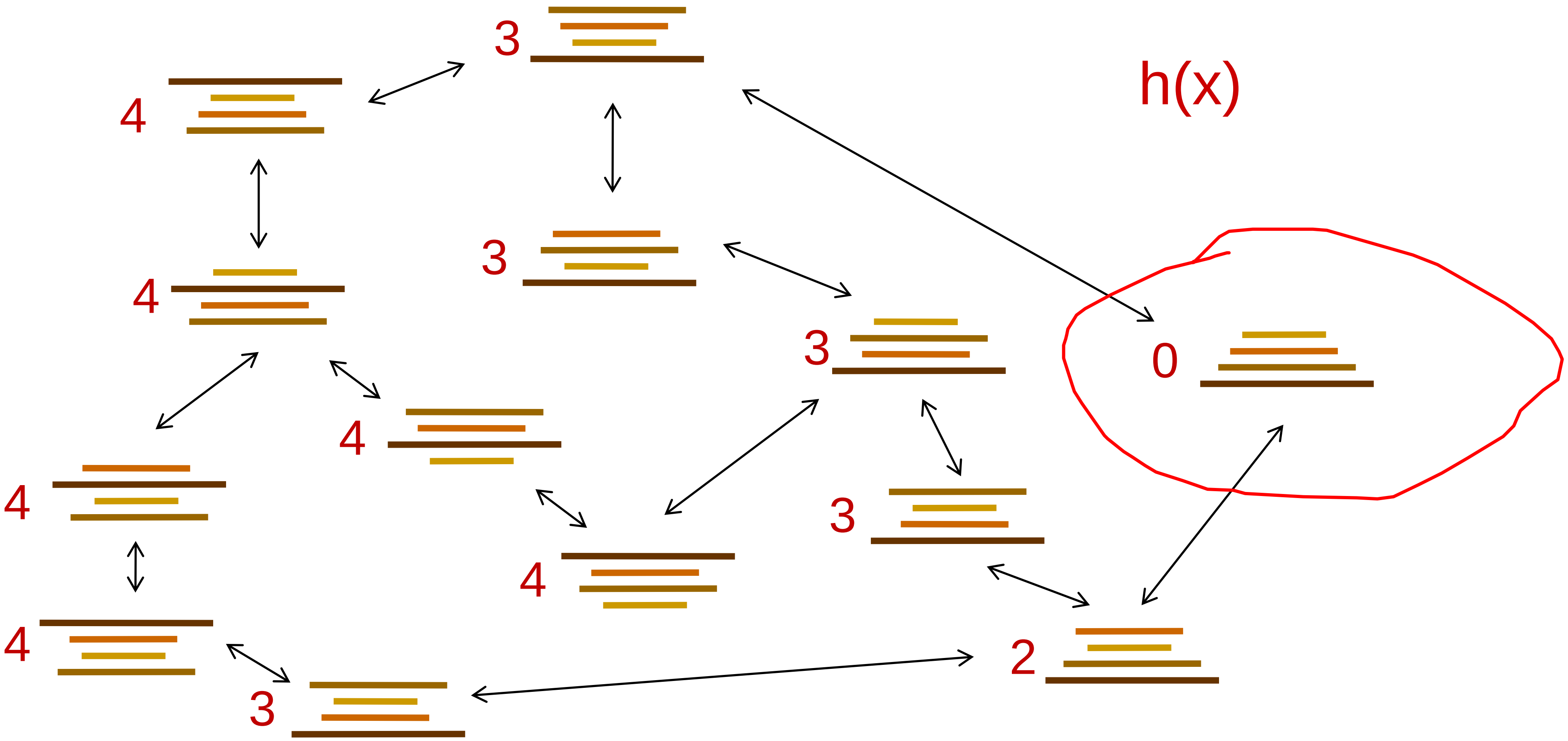
- in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points.

h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

h(x)

Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place



Heuristics for 8-puzzle

Misplaced Tiles Heuristic

(not including the blank)

Current State

3	2	8
4	5	6
7	1	

Goal State

1	2	3
4	5	6
7	8	

3	2	8
4	5	6
7	1	

3 tiles are not where they need to be

- Three tiles are misplaced (the 3, 8, and 1) so heuristic function evaluates to 3
- Heuristic says that it *thinks* a solution may be available in 3 or more moves
- Very rough estimate, but easy to calculate

h = 3

Heuristics for 8-puzzle

Manhattan Distance Heuristic

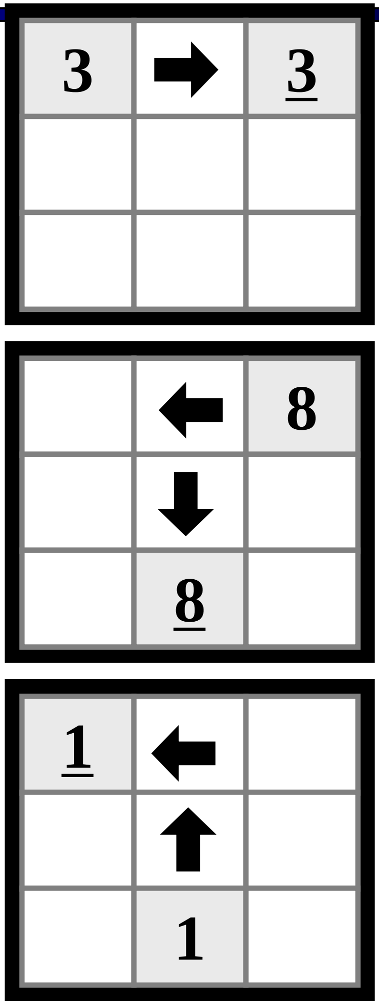
(not including the blank)

Current State

3	2	8
4	5	6
7	1	

Goal State

1	2	3
4	5	6
7	8	



2 steps

3 steps

3 steps

- The 3, 8, and 1 tiles misplaced by 2, 3, and 3 steps, so heuristic function evaluates to **8**
- Heuristic says that it *thinks* a solution may be available in 8 or more moves
- More accurate than the misplaced heuristic, but slightly more expensive to compute

h = 8

Best-First Search

- Idea: use an evaluation function estimating the desirability of each node
- Strategy: Always expand the most desirable unexpanded node
- Implementation: the fringe is a priority queue sorted in decreasing order of desirability
- Special cases:
  - Greedy search
  - A* search

Best-first Search Strategies

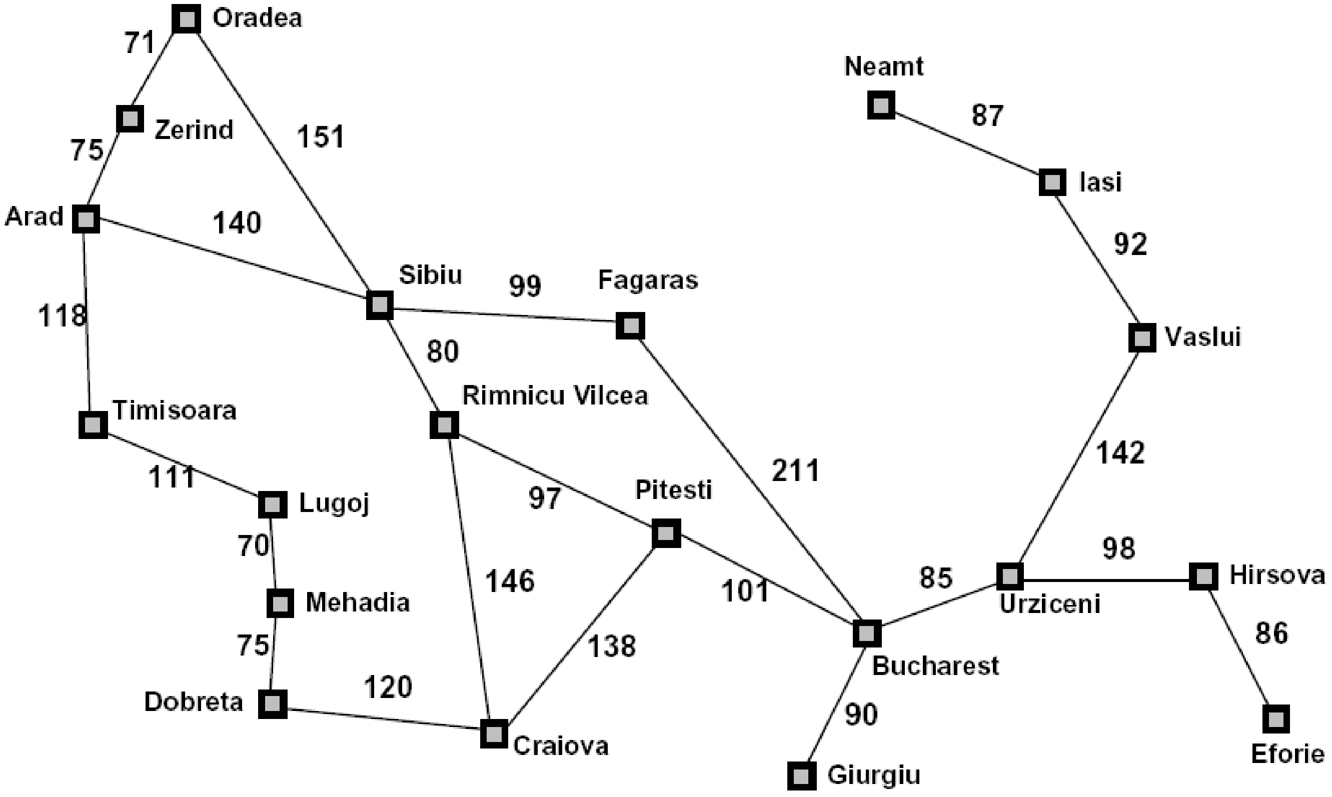
- Best-first is a family of search strategies, each with a different evaluation function
- Typically, strategies use estimates of the cost of reaching the goal and try to minimize it
- Uniform Search also tries to minimize a cost measure. Is it then a best-first search strategy?
  - Not in spirit, because the evaluation function should incorporate a cost estimate of going from the current state to the closest goal state

Greedy Search

Greedy best-first search

- Greedy best-first search is a form of best-first search that expands first the node with the lowest h(n) value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly.
- So the evaluation function $f(n) = h(n)$.
- Implementation: Order the nodes in fringe in decreasing order of desirability

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

$h_{SLD}(n)$

- Evaluation (heuristics) function h(n) = estimate cost of cheapest path from node n to closest goal.
- We use the **straight-line** distance heuristic here.
- E.g., h_{SLD} (n) = straight-line distance from n to Bucharest

- **Greedy search expands the node that appears to be closest to goal**

Route-finding in Romania

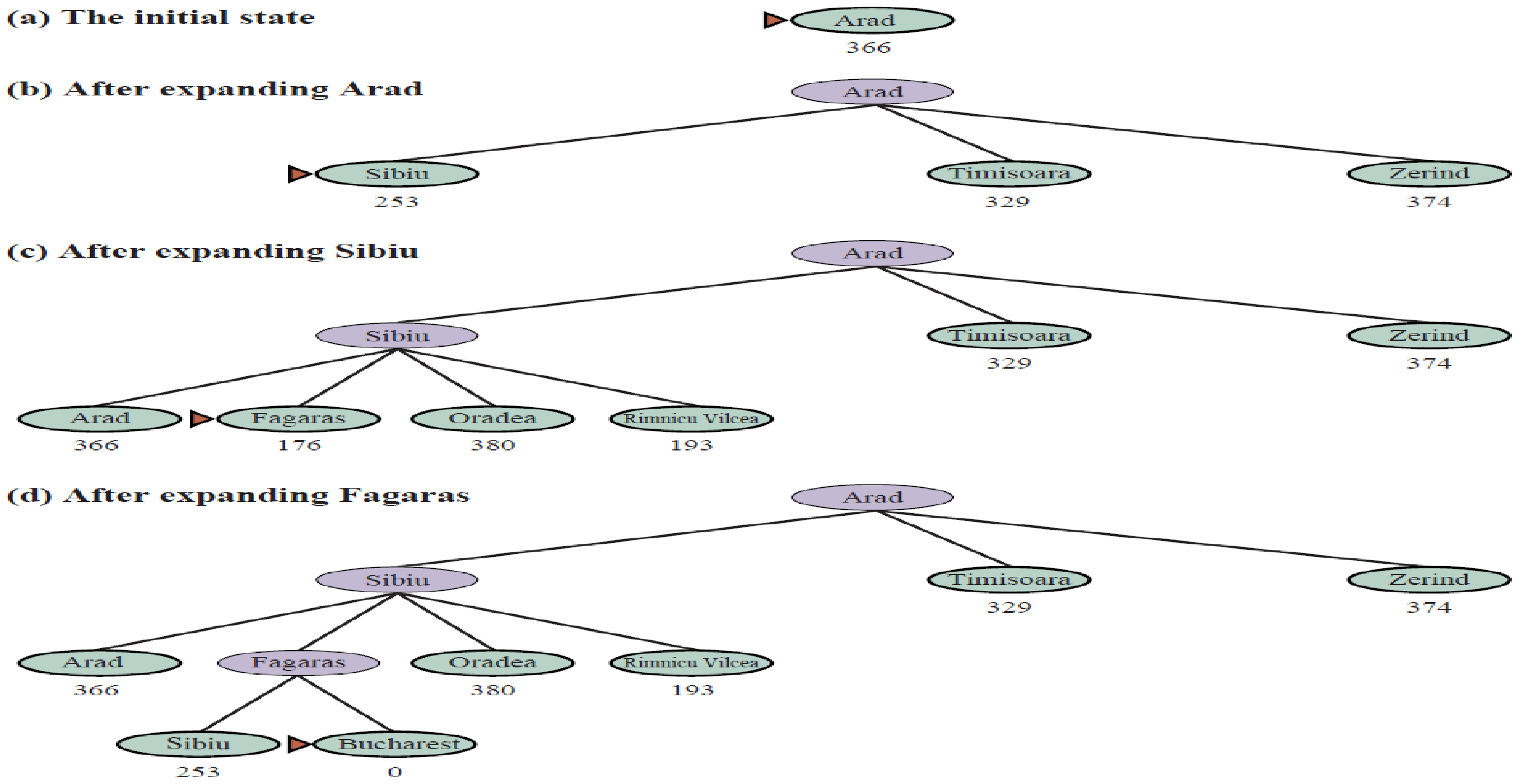
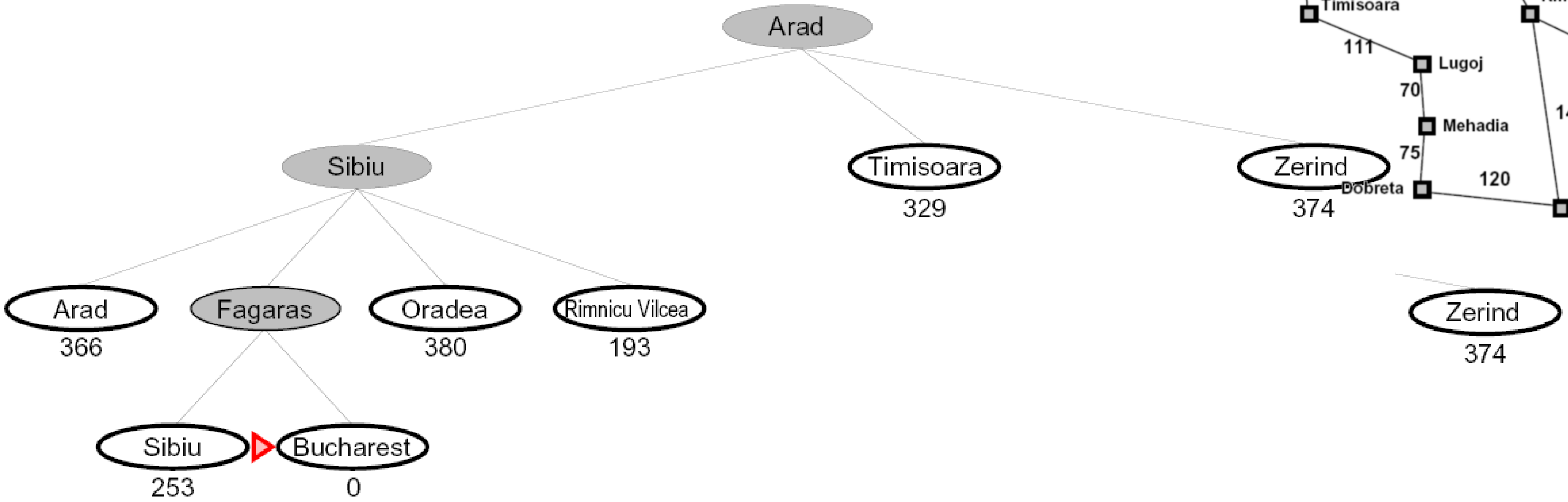


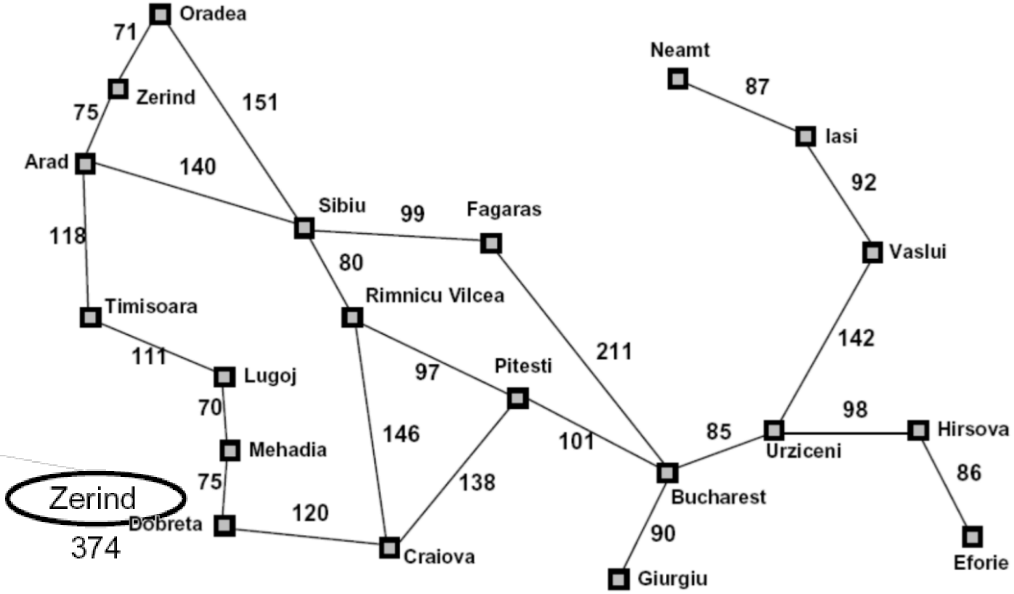
Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

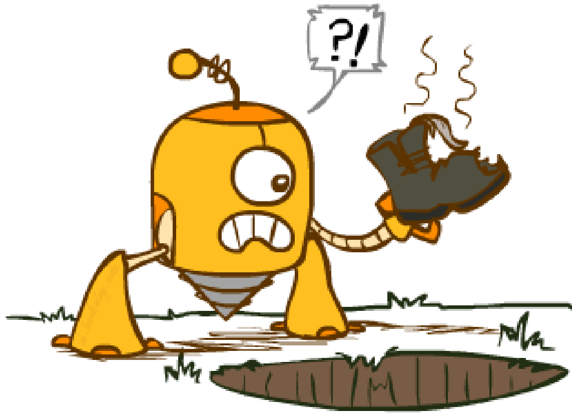
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Search

- Expand the node that seems closest…



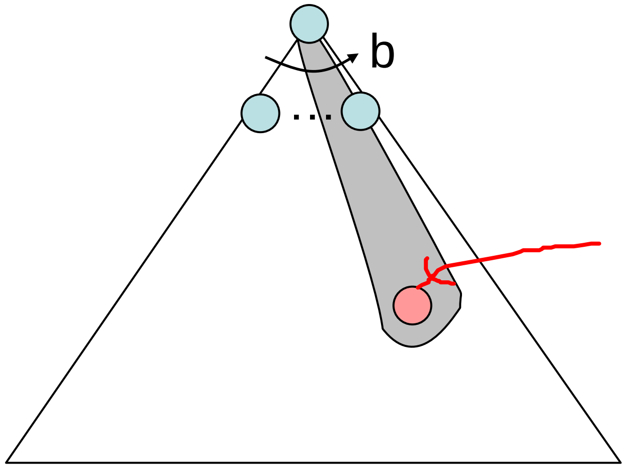




- What can go wrong?
  - For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.
  - It is not optimal, however.
  - The path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

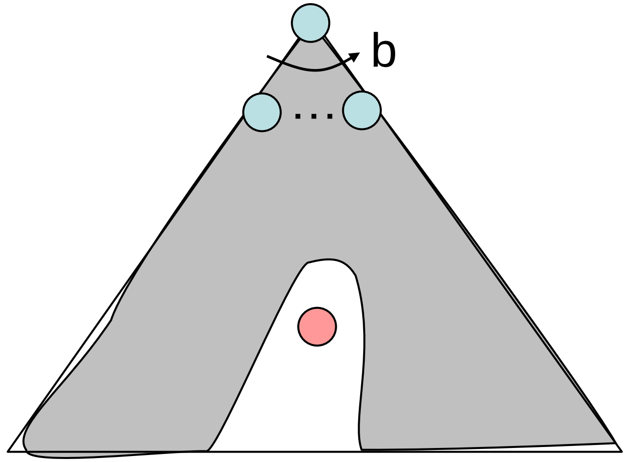
Greedy Search

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



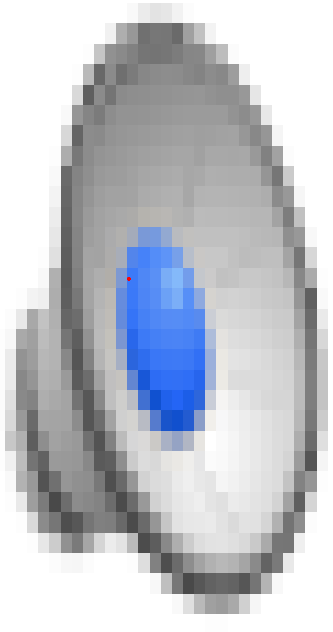
- A common case:
  - Best-first takes you straight to the (wrong) goal

- Worst-case: like a badly-guided DFS

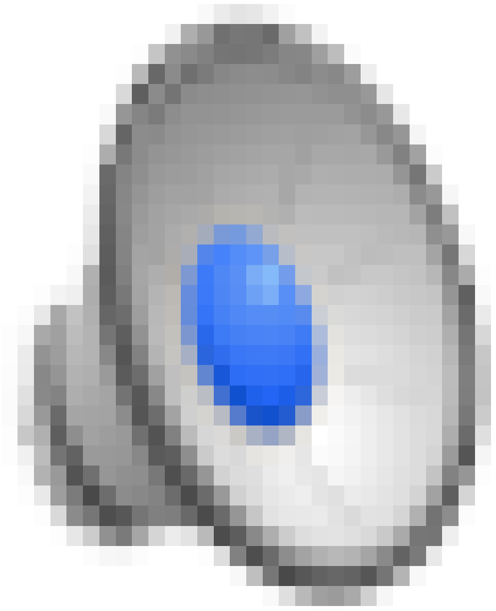


[Demo: contours greedy empty (L3D1)]
[Demo: contours greedy pacman small maze (L3D4)]

Video of Demo Contours Greedy (Empty)



Video of Demo Contours Greedy (Pacman Small Maze)



Properties of Greedy Best-First Search

Complete? Only in finite spaces with repeated-state checking
Otherwise, can get stuck in loops:
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Time complexity? $O(b^m)$ — may have to expand all nodes

Space complexity? $O(b^m)$ — may have to keep most nodes in memory

Optimal? No

- A good heuristic can nonetheless produce dramatic time/space improvements in practice.

A* Search

A* search

- The most common informed search algorithm is A* search (pronounced "A-star search"),
- A best-first search strategy that uses the evaluation function

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

- where
  - **g(n)** is the path cost from the initial state to node n, and
  - h(n) is the estimated cost of the shortest path from node n to a goal state,
- so we have

f (n) = estimated cost of the best path that continues from n to a goal.

Combining UCS and Greedy

Greedy Best-first search

- minimizes estimated cost $h(n)$ from current node n to goal
- is informed but almost always suboptimal and incomplete

Uniform cost search

- minimizes actual cost $g(n)$ to current node n
- is, in most cases, optimal and complete but uninformed

A* search

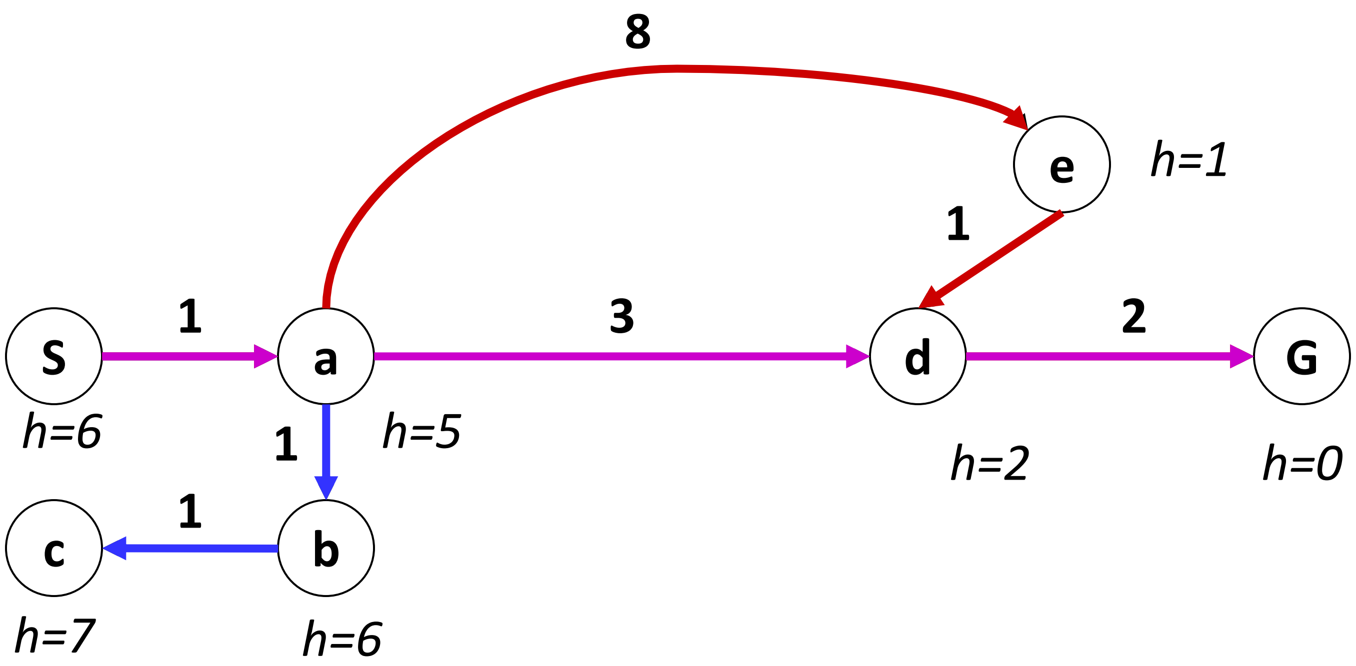
- combines the two by minimizing $f(n) = g(n) + h(n)$
- is, *under reasonable assumptions*, optimal and complete, and also informed

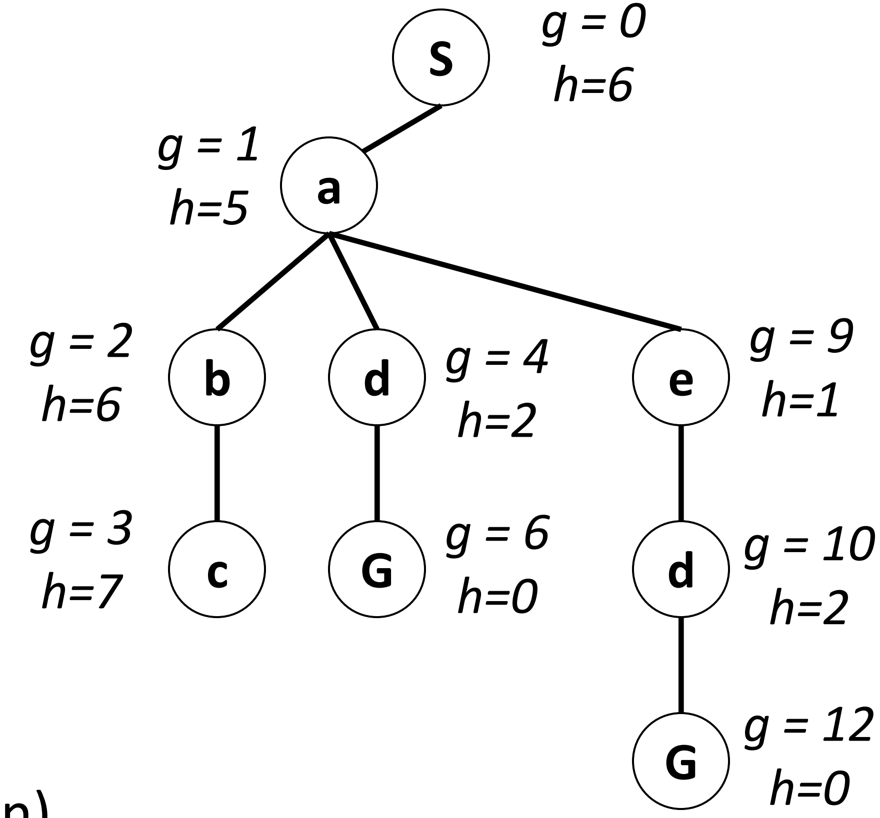
A* — A Better Best-First Strategy by combining UCS and Greedy

A* Search (turtle & rabbit analogy)

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)



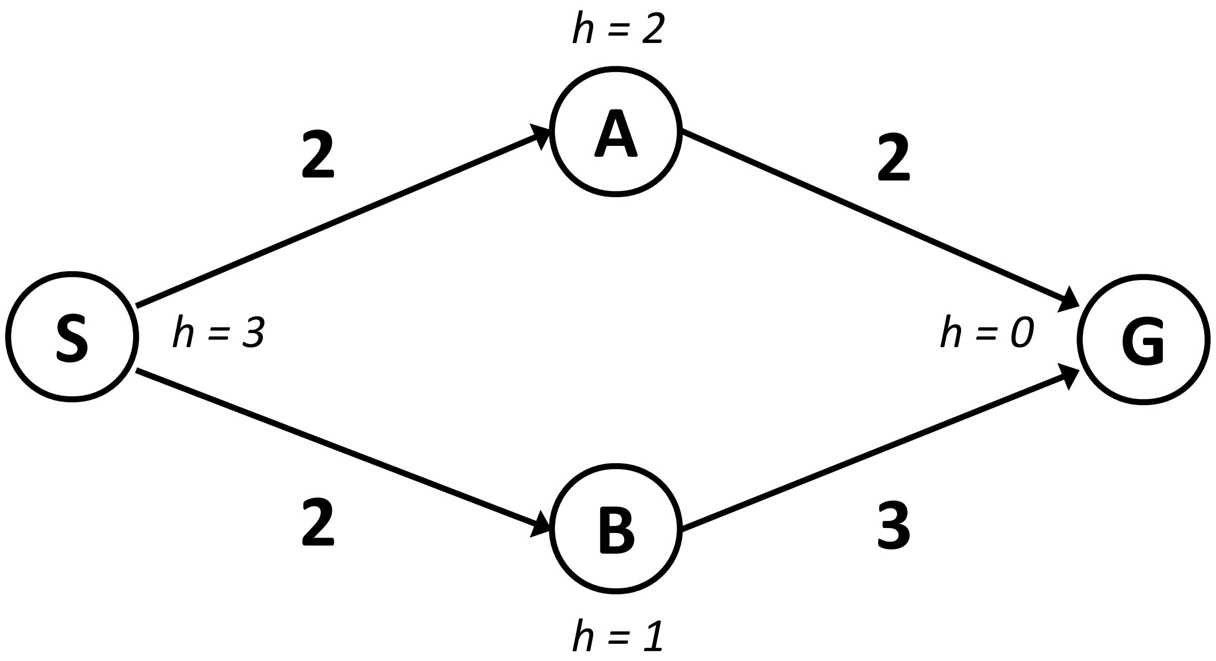


- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

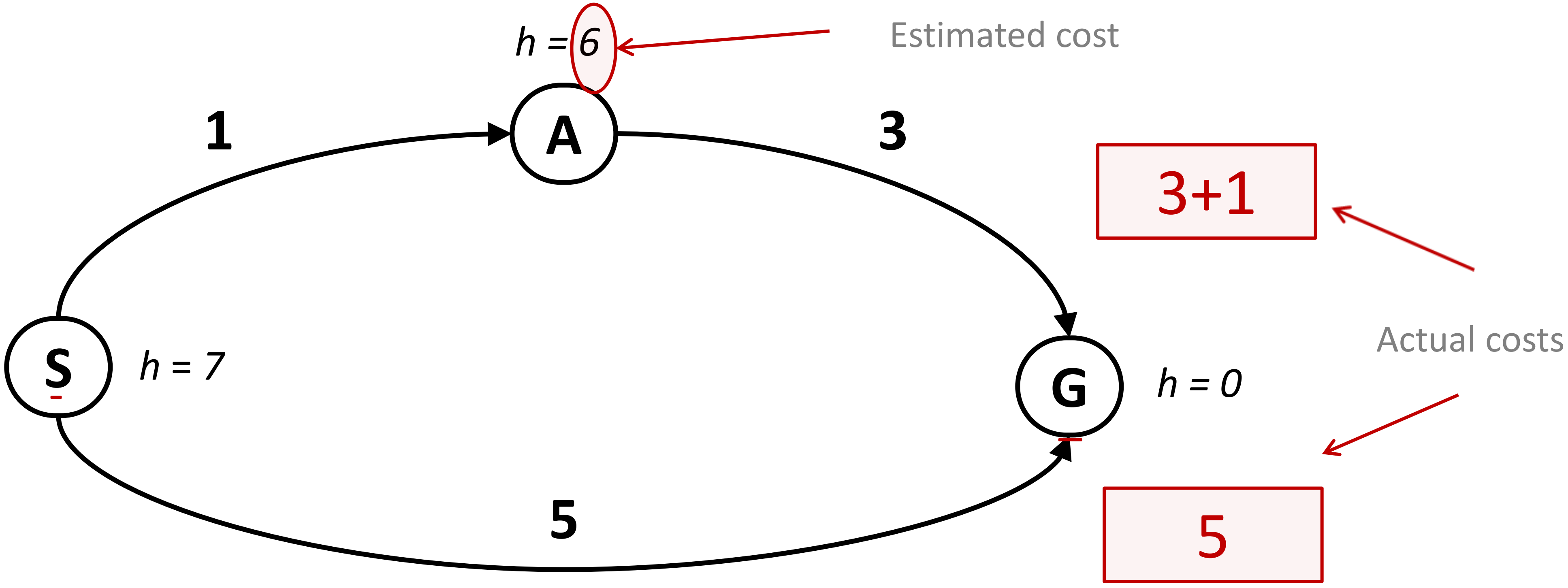
When should A* terminate?

- Should we stop when we enqueue a goal?



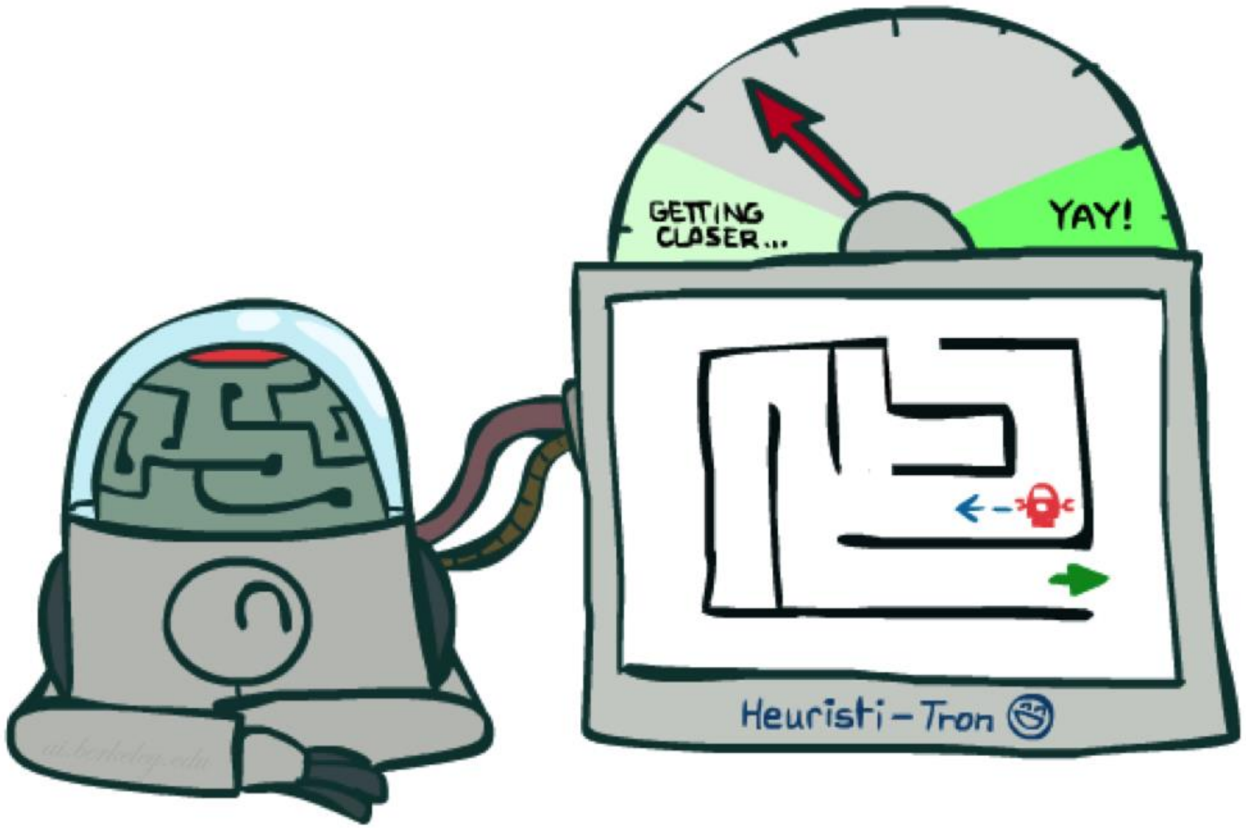
- No: only stop when we dequeue a goal

Is A* Optimal?

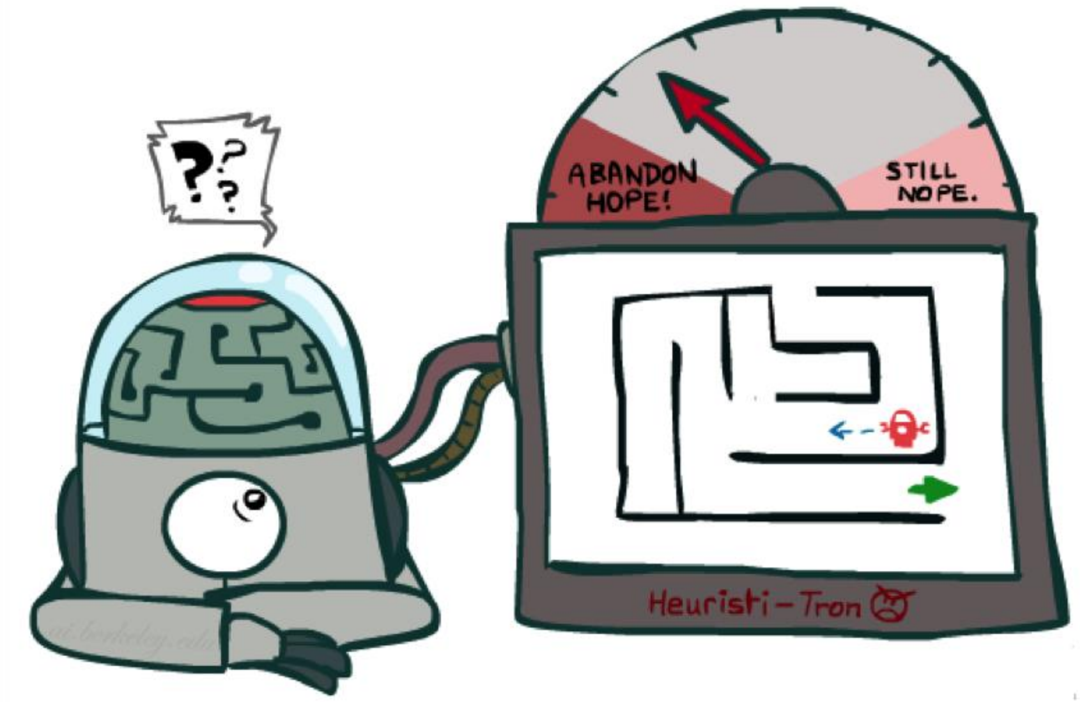


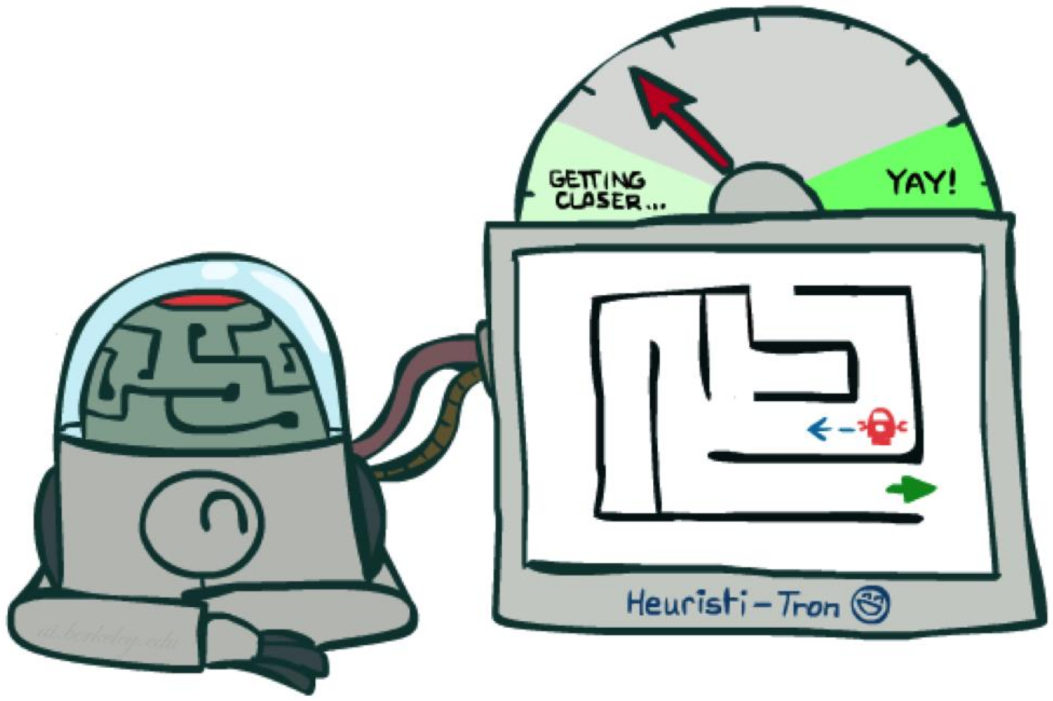
- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Admissible Heuristics



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

A* Search

Idea: avoid expanding paths that are already expensive

Evaluation function: $f(n) = g(n) + h(n)$

$$
\begin{aligned}
g(n) &= \text{cost so far to reach } n \\
h(n) &= \text{estimated cost to goal from } n \\
f(n) &= \text{estimated total cost of path through } n \text{ to goal}
\end{aligned}
$$

A* search should use an *admissible* heuristic:

for all n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n

E.g., $h_{\text{SLD}}(n)$ never overestimates the actual road distance

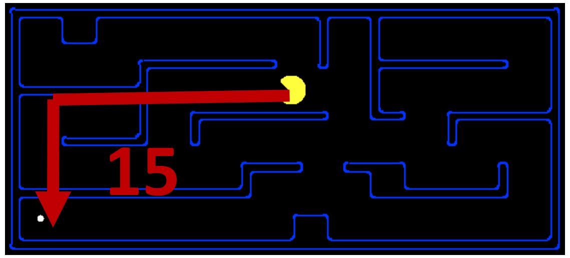
Admissible Heuristics

- 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:

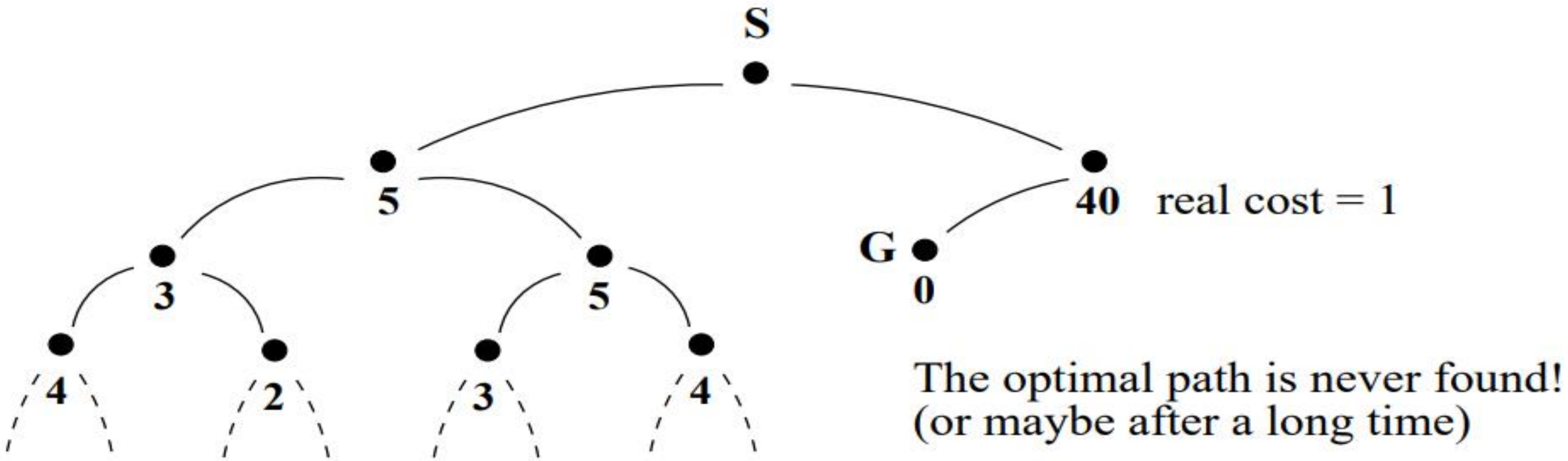


- Coming up with admissible heuristics is most of what's involved in using A* in practice.

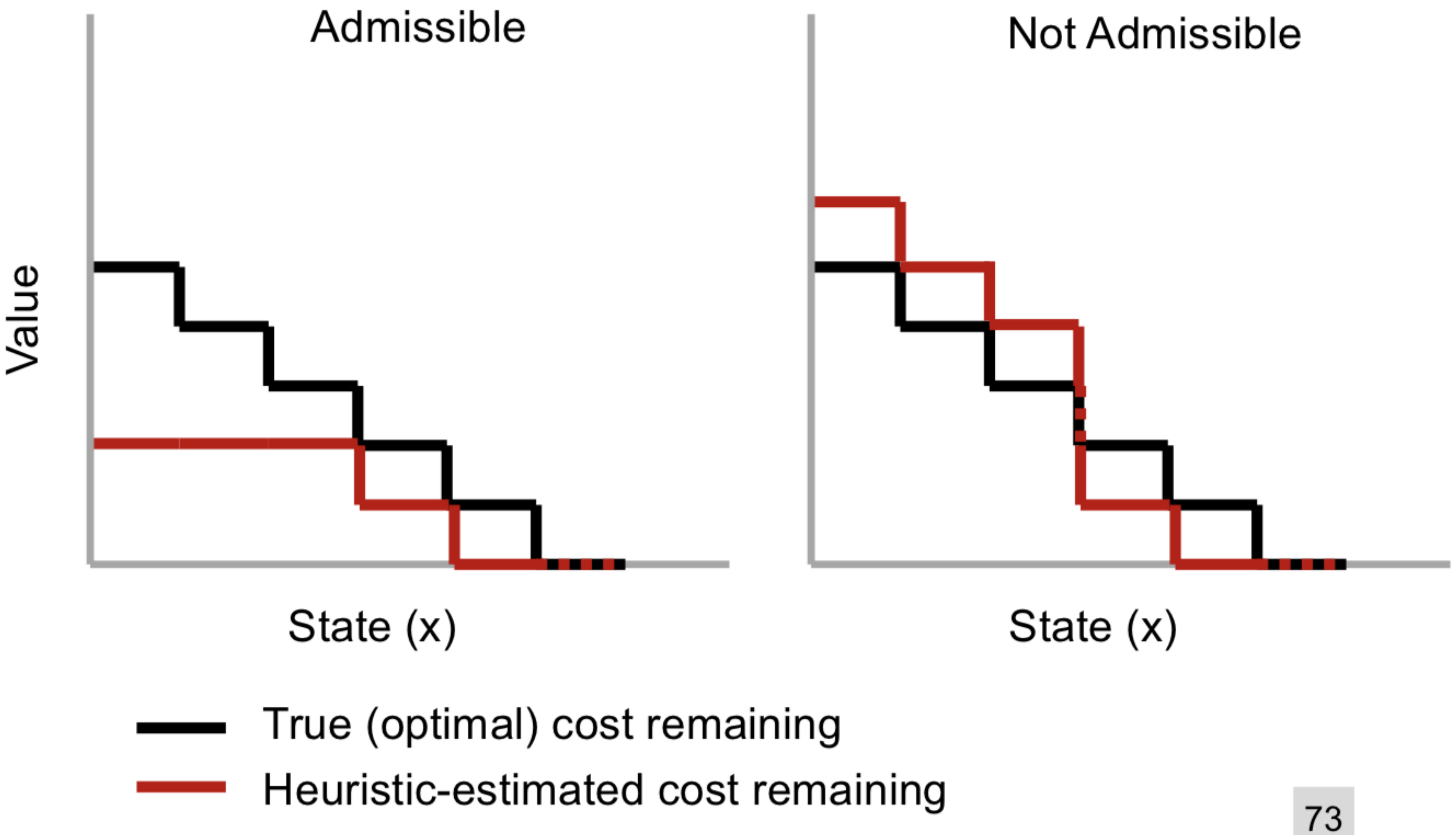
A* Search: Why an Admissible Heuristic

If h is admissible, $f(n)$ never overestimates the actual cost of the best solution through n

Overestimates are dangerous

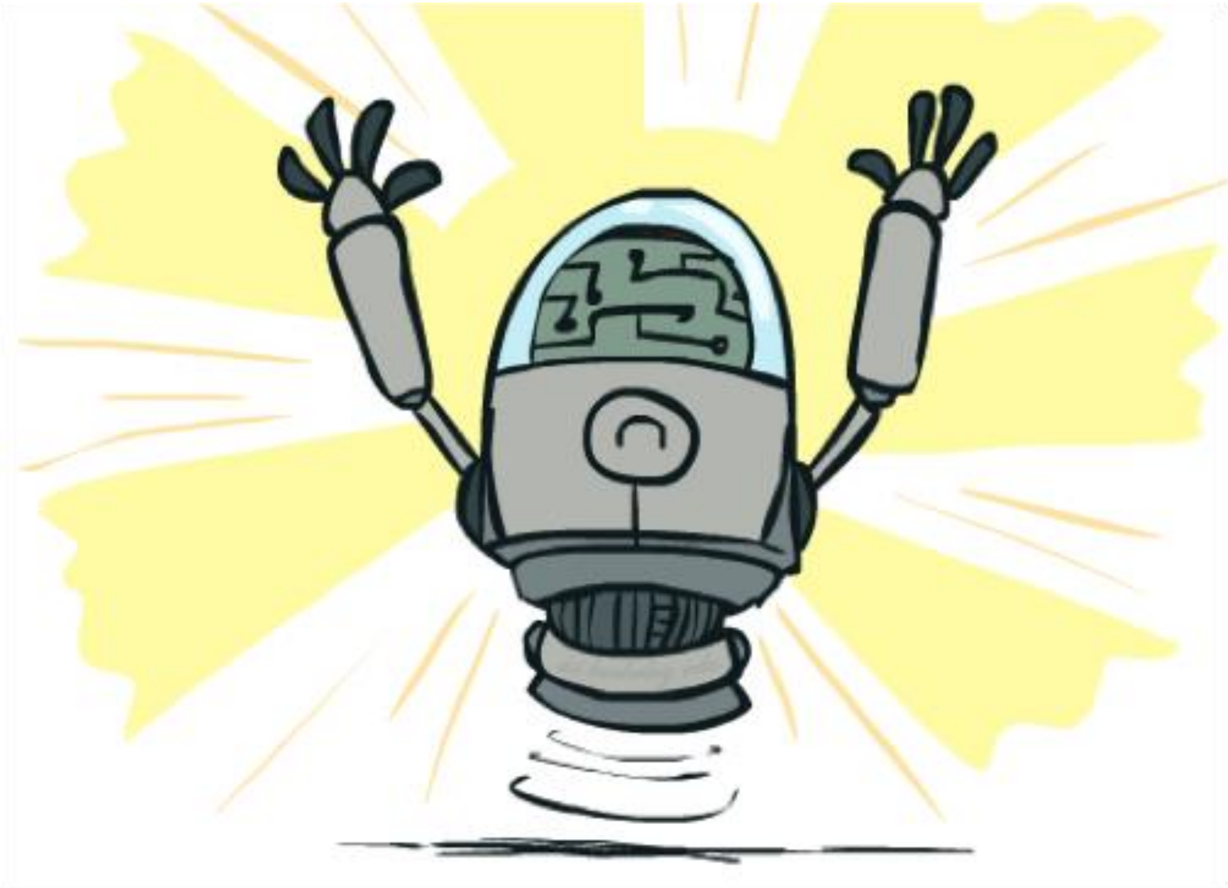


Admissible Heuristics



Slide credit: Travis Mandel

Optimality of A* Tree Search



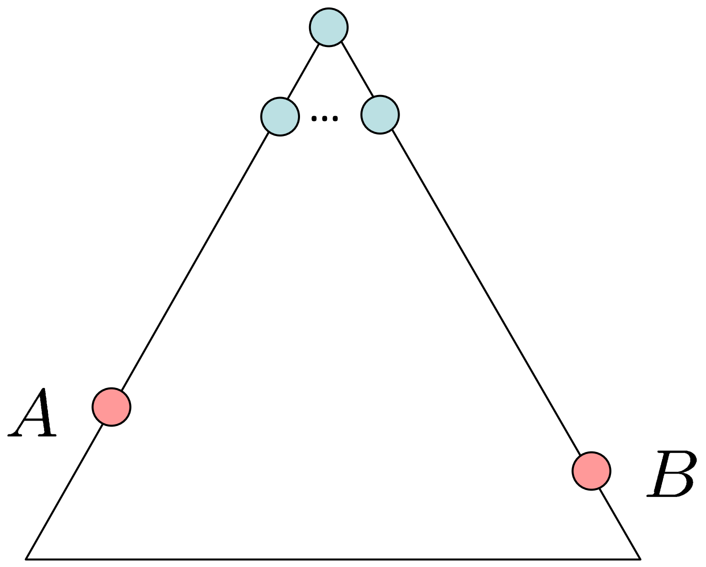
Proof : Optimality of A* Tree Search

Assume:

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

Claim:

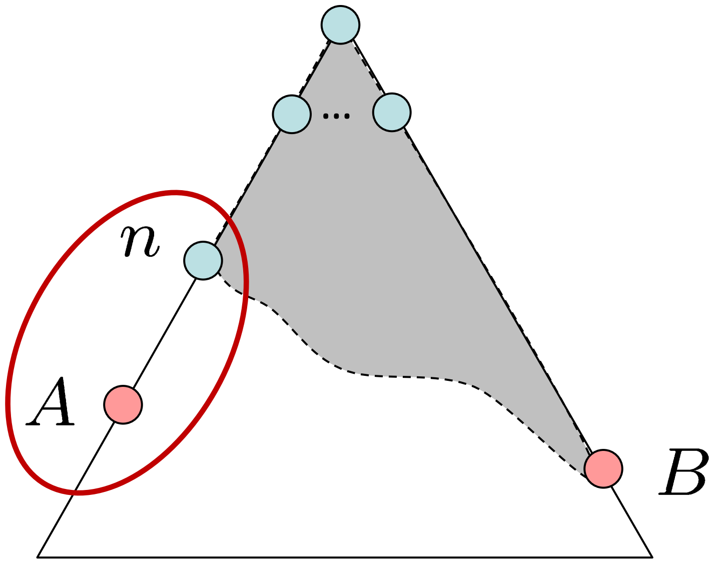
- A will exit the fringe before B



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)

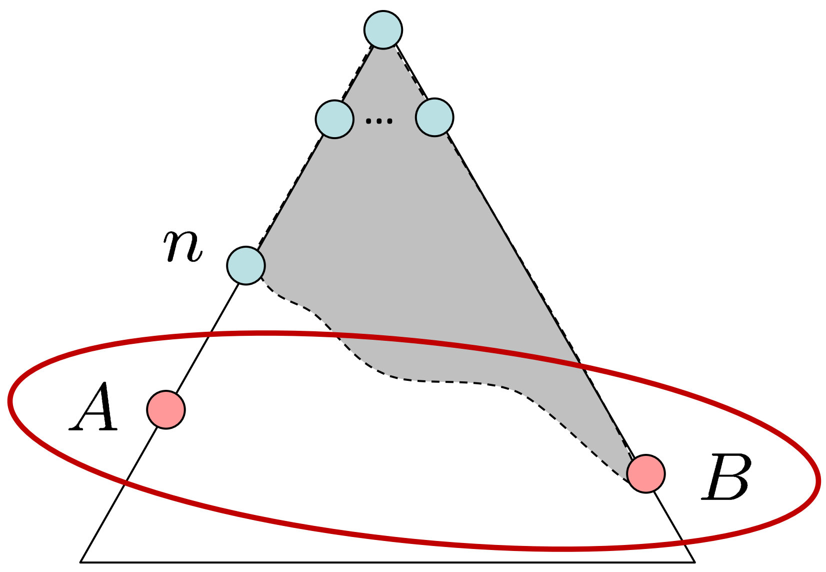


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

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)



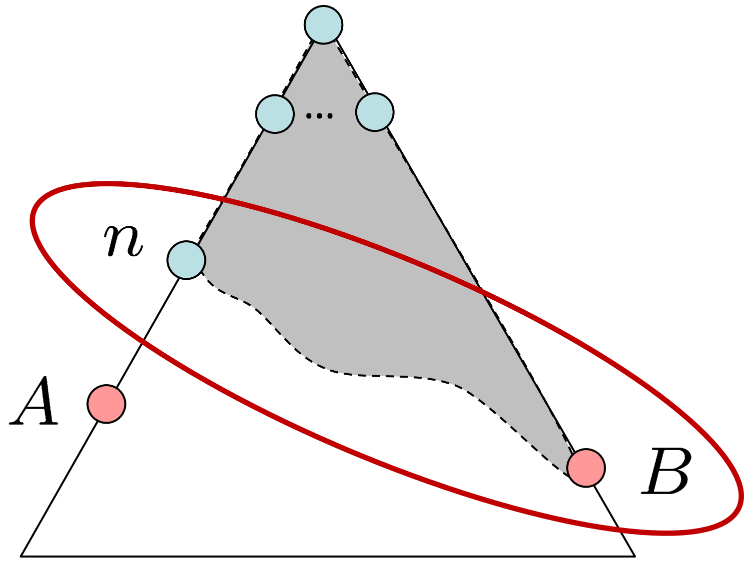
$$g(A) < g(B)$$ B is suboptimal

$$f(A) < f(B)$$ 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

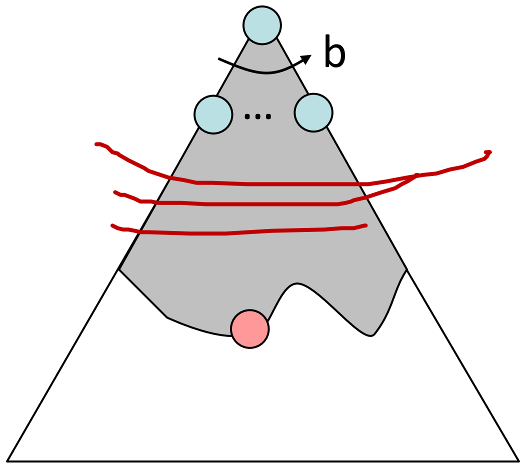


$$f(n) \leq f(A) < f(B)$$

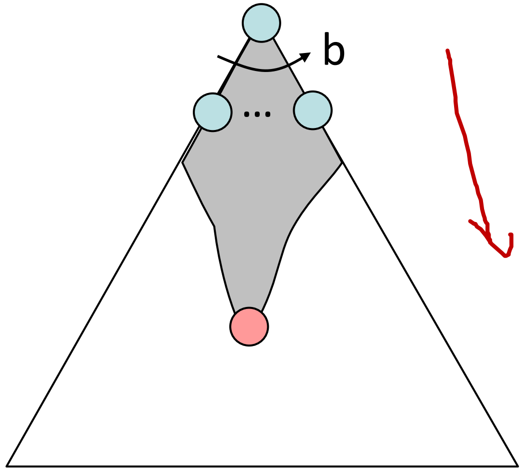
Properties of A*

Properties of A*

Uniform-Cost

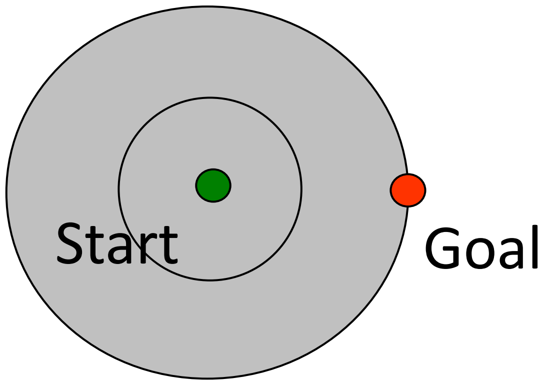


A*

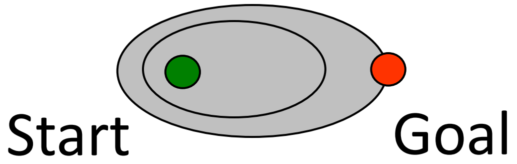


UCS vs A* Contours

- Uniform-cost expands equally in all "directions"



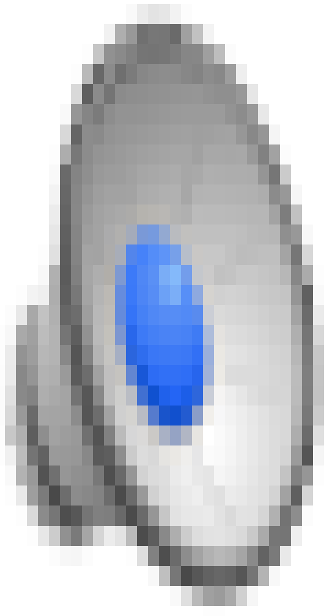
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



[Demo: contours UCS / greedy / A* empty (L3D1)]
[Demo: contours A* pacman small maze (L3D5)]

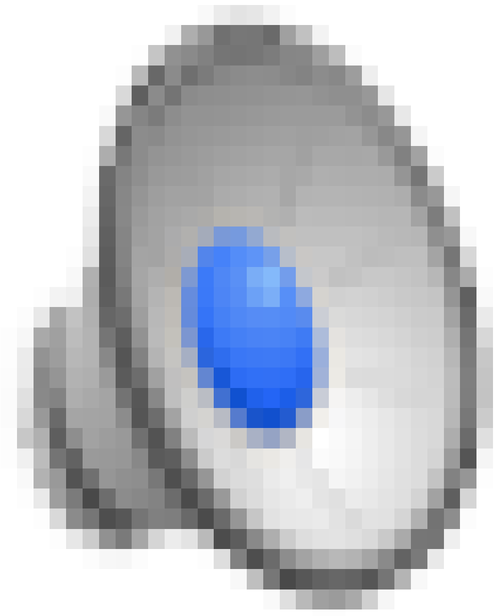
Video of Demo Contours (Empty) -- UCS

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*

Properties of A*

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

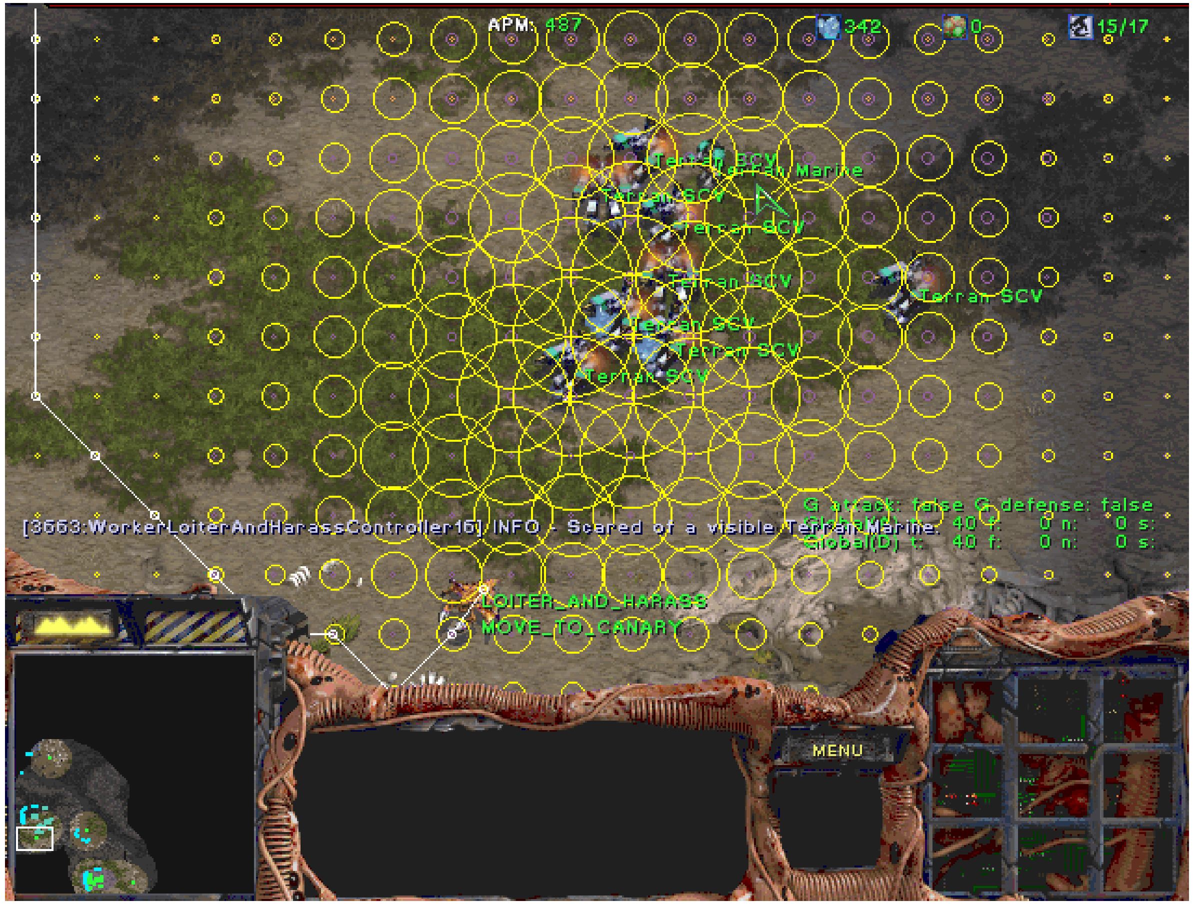
Time complexity? $O(b^{\epsilon d})$ where
$$\begin{aligned} \epsilon &= |h(n_0) - h^*(n_0)| \\ n_0 &= \text{start state} \\ h^* &= \text{actual cost to goal state} \end{aligned}$$

Subexponential only in uncommon case where $\epsilon \leq O(\log h^*(n_0))$

Space complexity? $O(b^m)$, as in Greedy Best-First — may end up with all nodes in memory

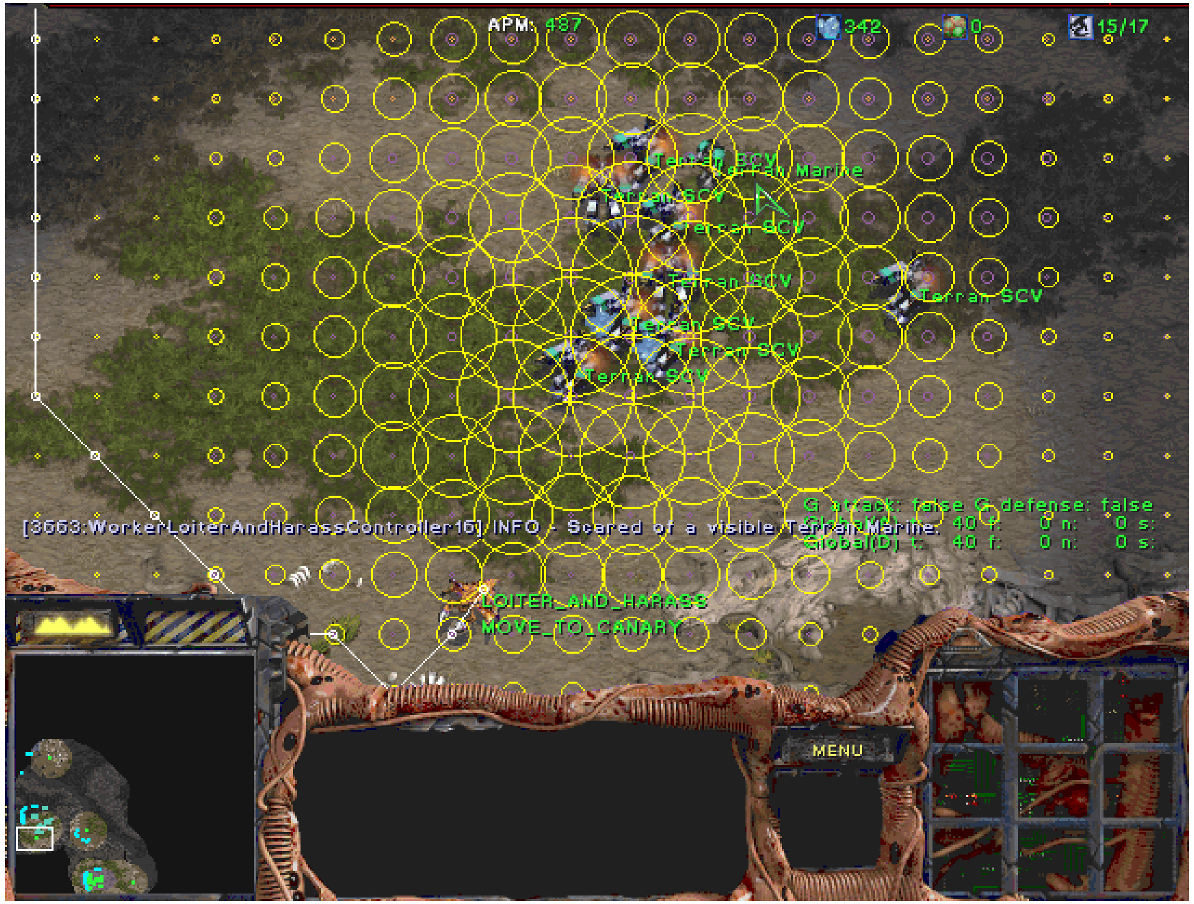
Optimal? Yes if h is admissible (and standard assumptions hold) — cannot expand f_{i+1} until f_i is finished

A* Applications



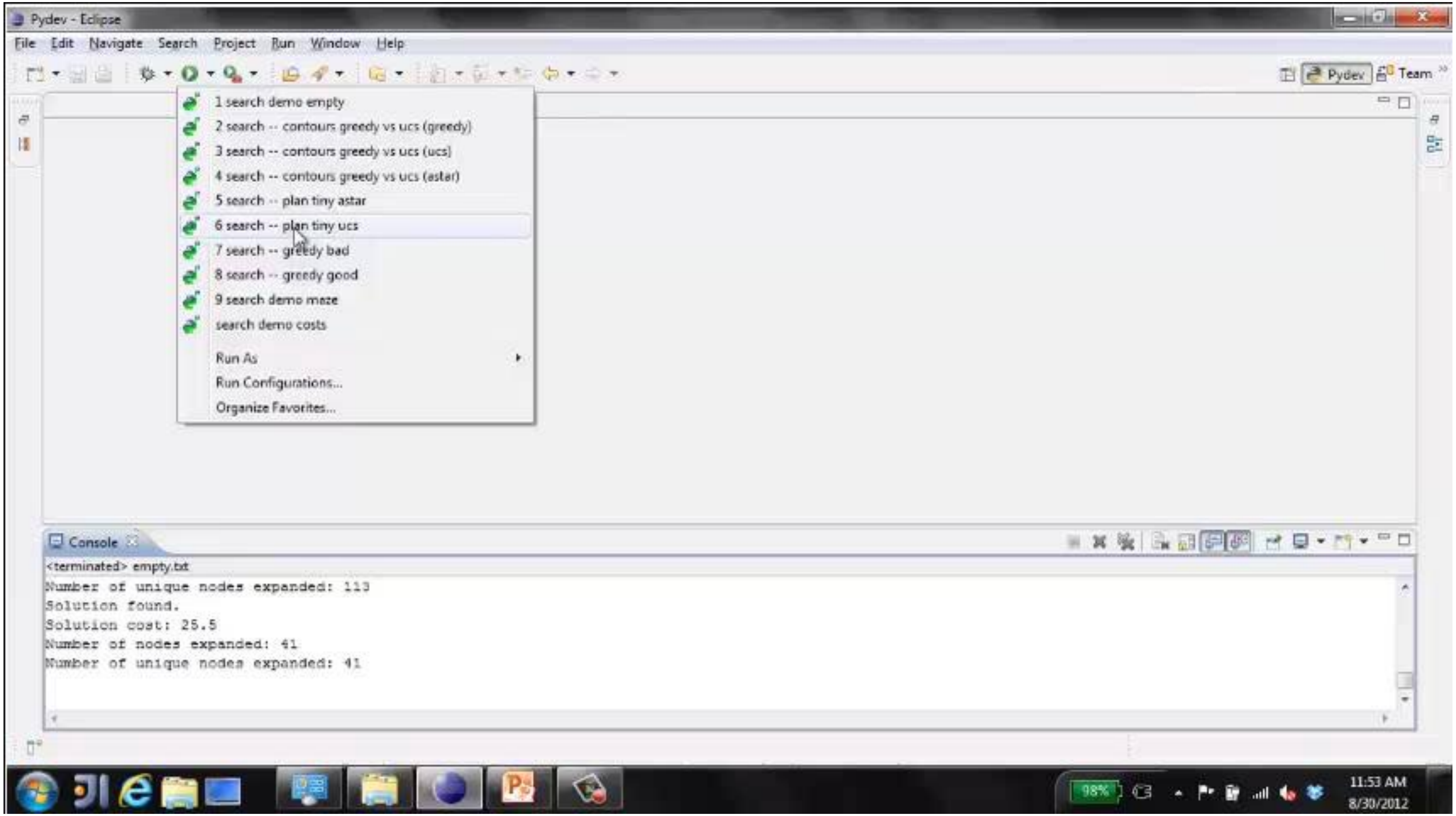
A* Applications

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

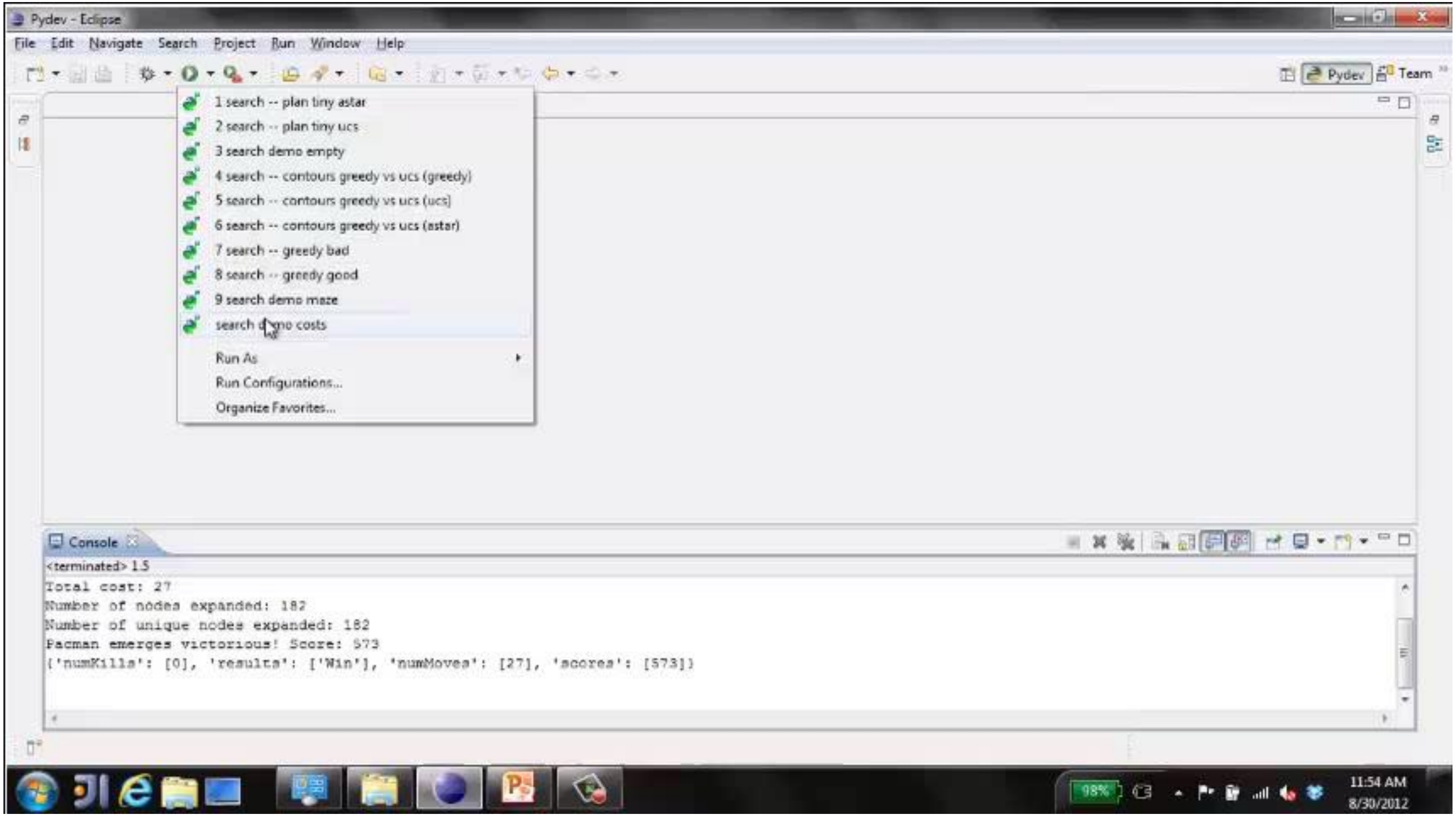


[Demo: UCS / A* pacman tiny maze (L3D6,L3D7)]
[Demo: guess algorithm Empty Shallow/Deep (L3D8)]

Video of Demo Pacman (Tiny Maze) – UCS / A*



Video of Demo Empty Water Shallow/Deep – Guess Algorithm



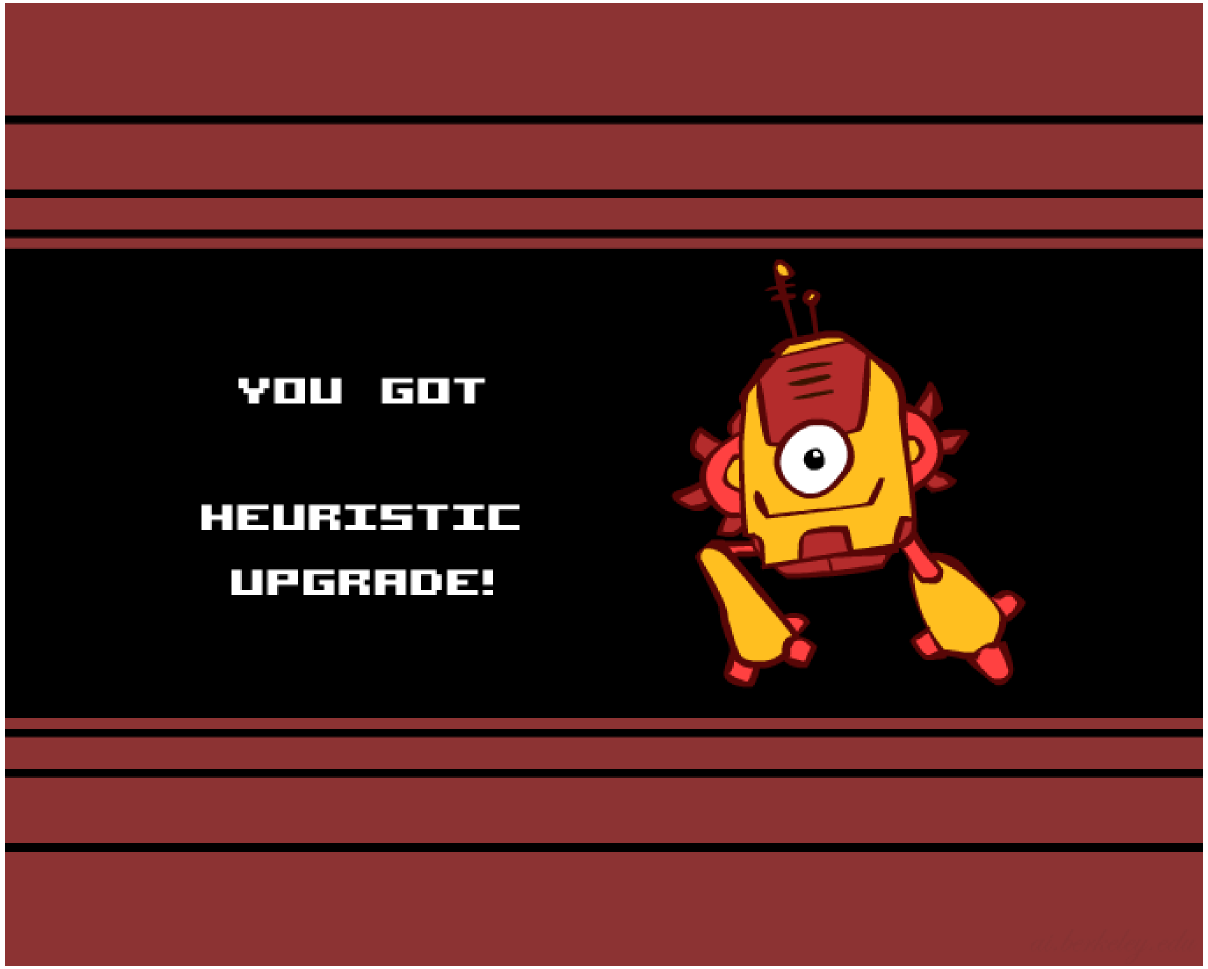
Beyond A*

A* generally runs out of memory before it runs out of time

Other best-first strategies keep the good properties on A* while trying to reduce memory consumption:

- Recursive Best-First search (RBFS)
- Iterative Deepening A* (IDA*)
- Memory-bounded A* (MA*)
- Simple Memory-bounded A* (SMA*)

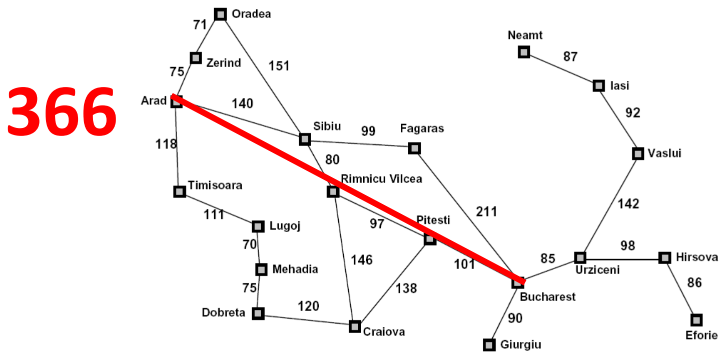
Creating Heuristics

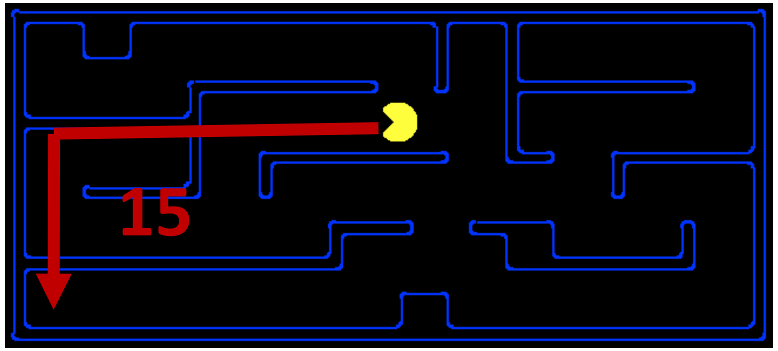


Creating Admissible Heuristics

- 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

Devising Heuristic Functions

A relaxed problem is a version of a search problem with less restrictions on the applicability of the next-state operators

Example: n-puzzle

- original: "A tile can move from position p to position q if p is adjacent to q and q is empty"
- relaxed-1: "A tile can move from p to q if p is adjacent to q"
- relaxed-2: "A tile can move from p to q if q is empty"
- relaxed-3: "A tile can move from p to q"

The exact solution cost of a relaxed problem is often a good (admissible) heuristics for the original problem

Key point: the optimal solution cost of the relaxed problem is no greater than the optimal solution cost of the original problem

Relaxed Problems: Example

Traveling salesperson problem

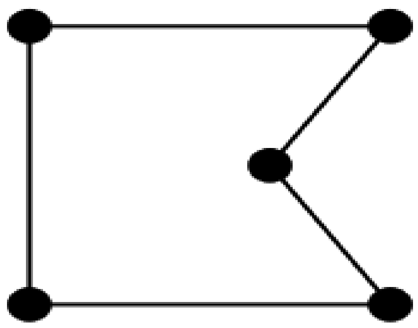
Original problem: Find the shortest tour visiting n cities exactly once

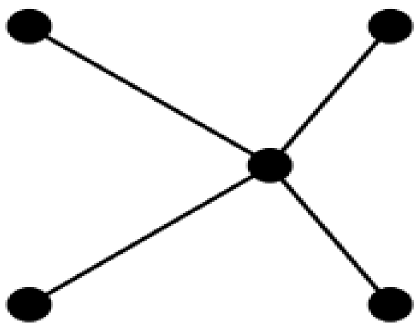
Complexity: NP-complete

Relaxed problem: Find a tree with the smallest cost that connects the n cities (minimum spanning tree)

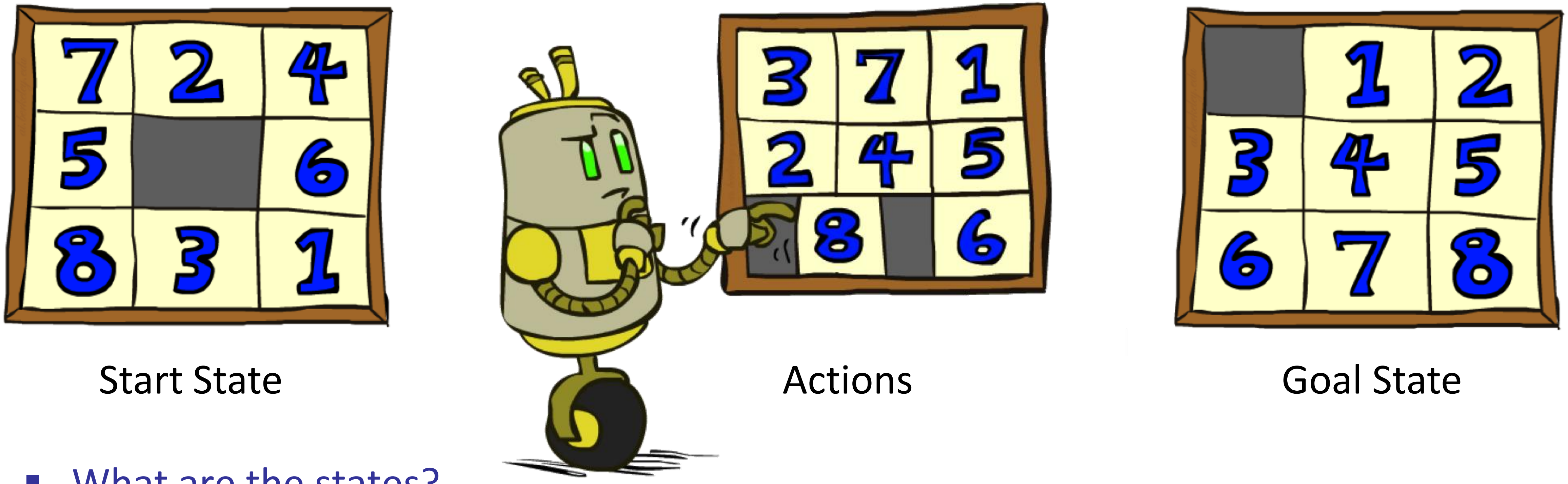
Complexity: $O(n^2)$

Cost of tree is a lower bound on the shortest (open) tour





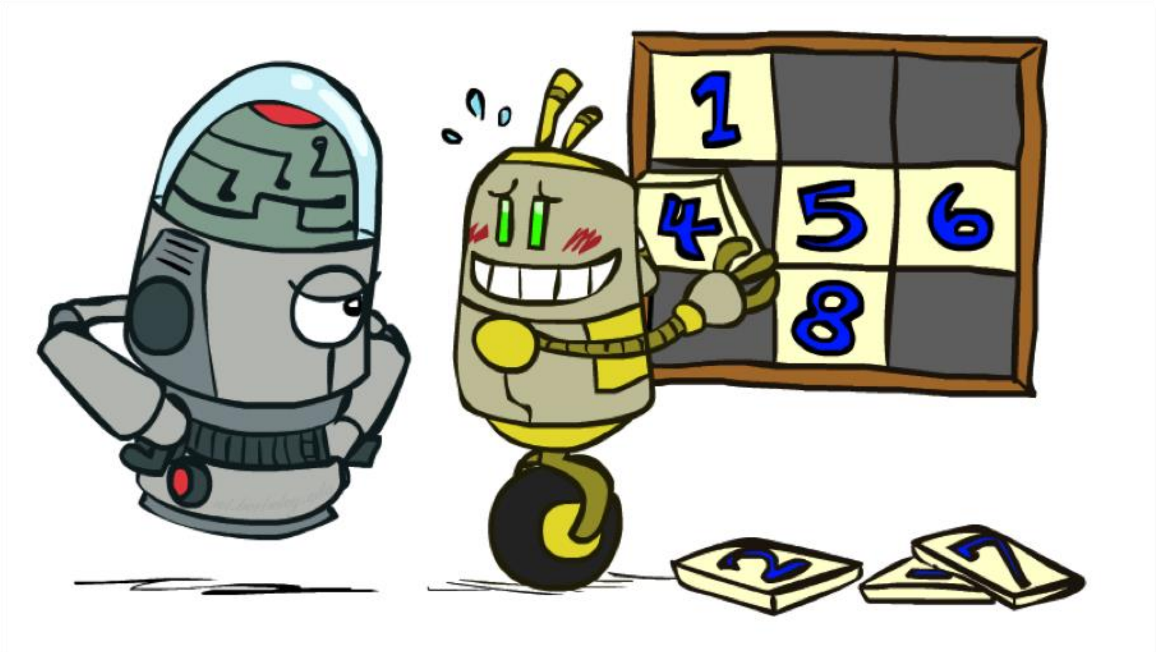
Example: 8 Puzzle

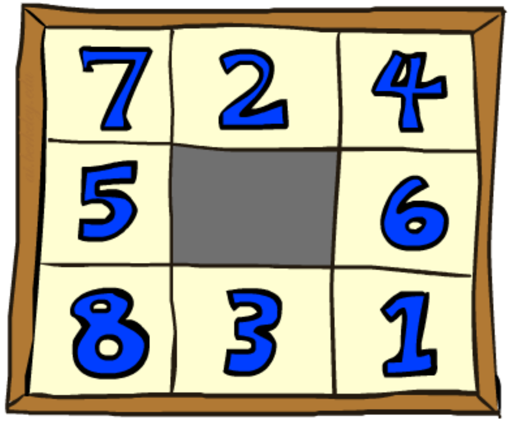


- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

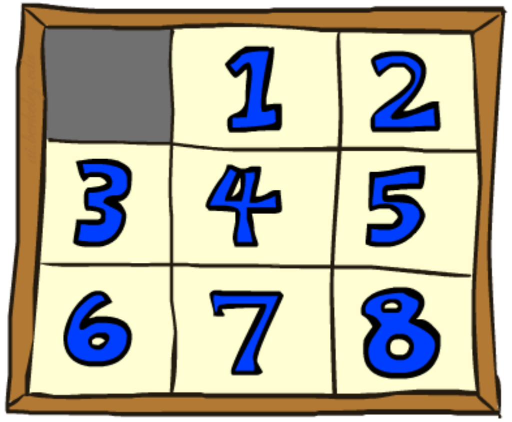
8 Puzzle I

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





Start State



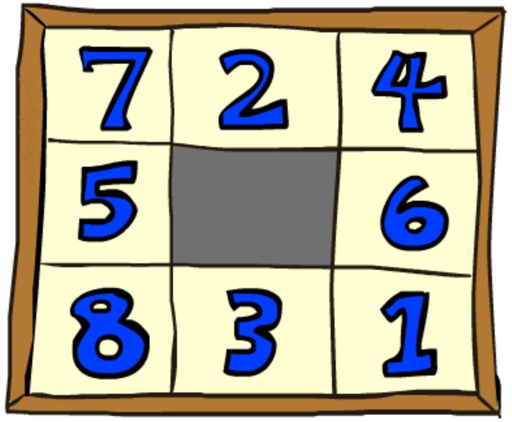
Goal State

	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

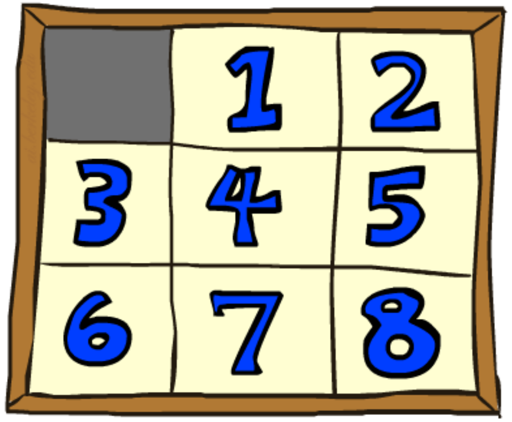
Statistics from Andrew Moore

8 Puzzle II

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



Start State

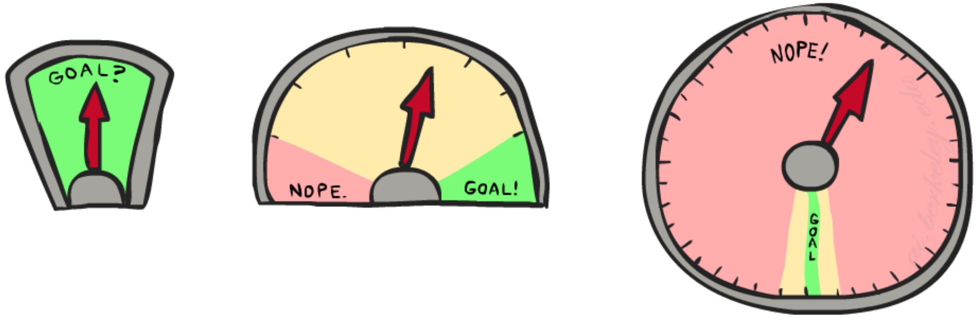


Goal State

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

8 Puzzle 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

Semi-Lattice of Heuristics

Trivial Heuristics, Dominance

A heuristic function h_2 *dominates* a heuristic function h_1 for a problem P if $h_2(n) \geq h_1(n)$ for all nodes n in P's space

Ex.: the 8-puzzle

h_2 = *total Manhattan distance* dominates
h_1 = *number of misplaced tiles*

With A*, if h_2 is admissible and dominates h_1, then it is always better for search: A* will never expand more nodes with h_2 than with h_1

What if neither of h_1, h_2 dominates the other?
If both h_1, h_2 are admissible, use $h(n) = \max(h_1(n), h_2(n))$

Trivial Heuristics, Dominance

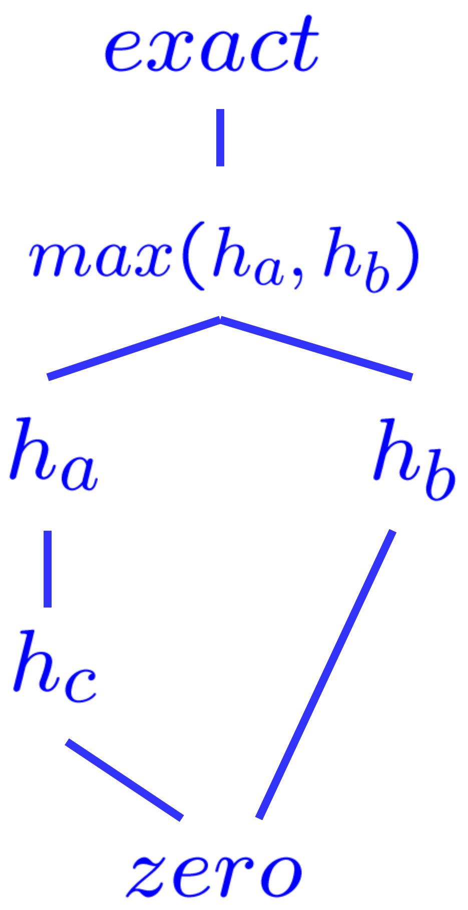
- Dominance: $h_a \geq h_c$ if

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

- 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



Effectiveness of Heuristic Functions

Let

- h be a heuristic function h for A*
- N the total number of nodes expanded by one A* search with h
- d the depth of the found solution

The effective branching Factor (EBF) of h is the value b^* that solves the equation

$$x^d + x^{d-1} + \cdots + x^2 + x + 1 - N = 0$$

(the branching factor of a uniform tree with N nodes and depth d)

A heuristics h for A* is effective in practice if its average EBF is close to 1

Note: If h_1 dominates h_2, then EFB(h_2) ≤ EFB(h_1)

Dominance and EFB: The 8-puzzle

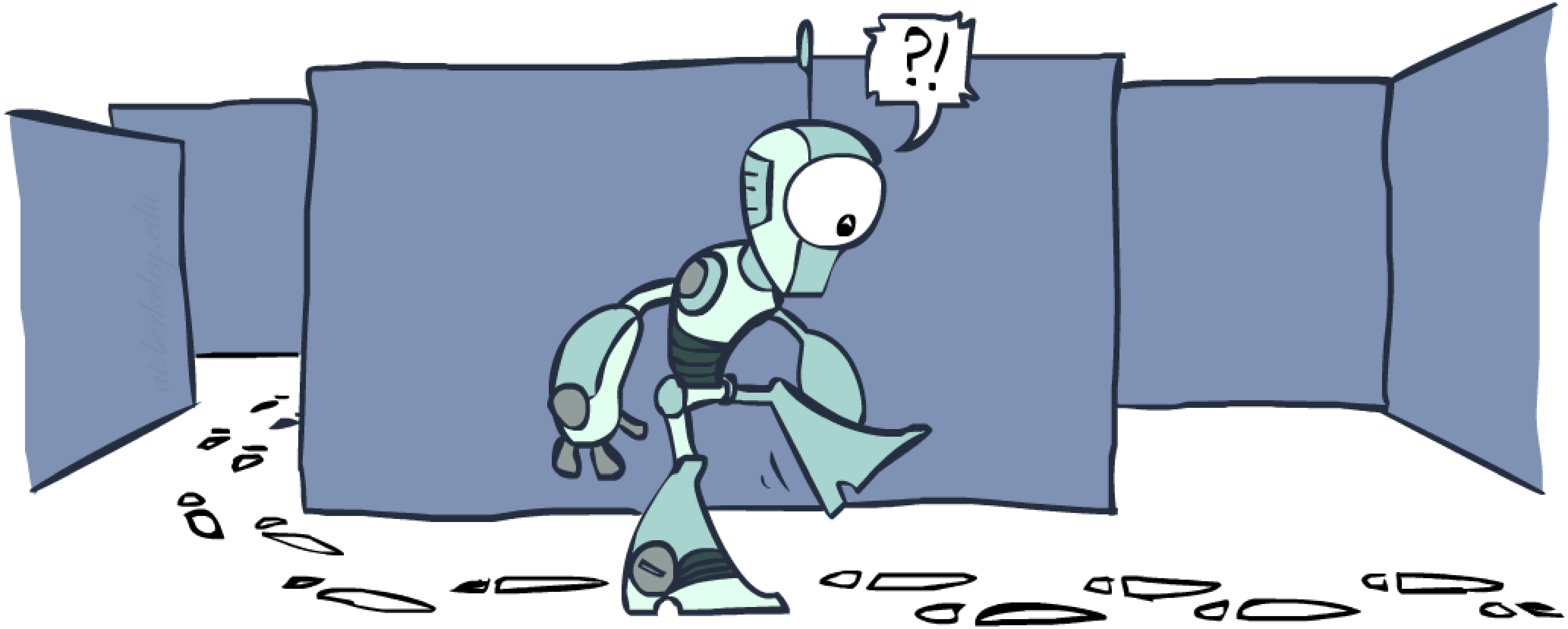
	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A^* with h_1 (misplaced tiles), and A^* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

Devising Heuristic Functions Automatically

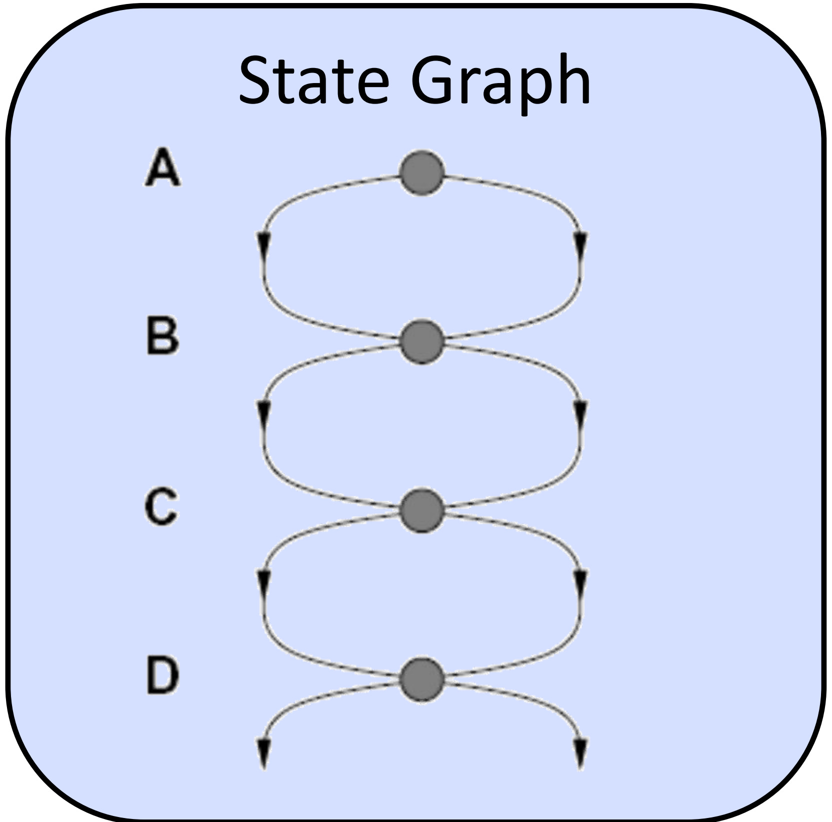
- Relaxation of formally described problems:
  - A problem with fewer restrictions on the actions is called a relaxed problem. The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- Pattern databases:
  - Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem. The idea behind pattern databases is to store these exact solution costs for every possible Pattern database subproblem instance. Then we compute an admissible heuristic h_{DB} for each state encountered during a search simply by looking up the corresponding subproblem configuration in the database.
- Learning :
  - An alternative is to learn from experience. “Experience” here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides an example (goal, path) pair. From these examples, a learning algorithm can be used to construct a function h that can (with luck) approximate the true path cost for other states that arise during search.

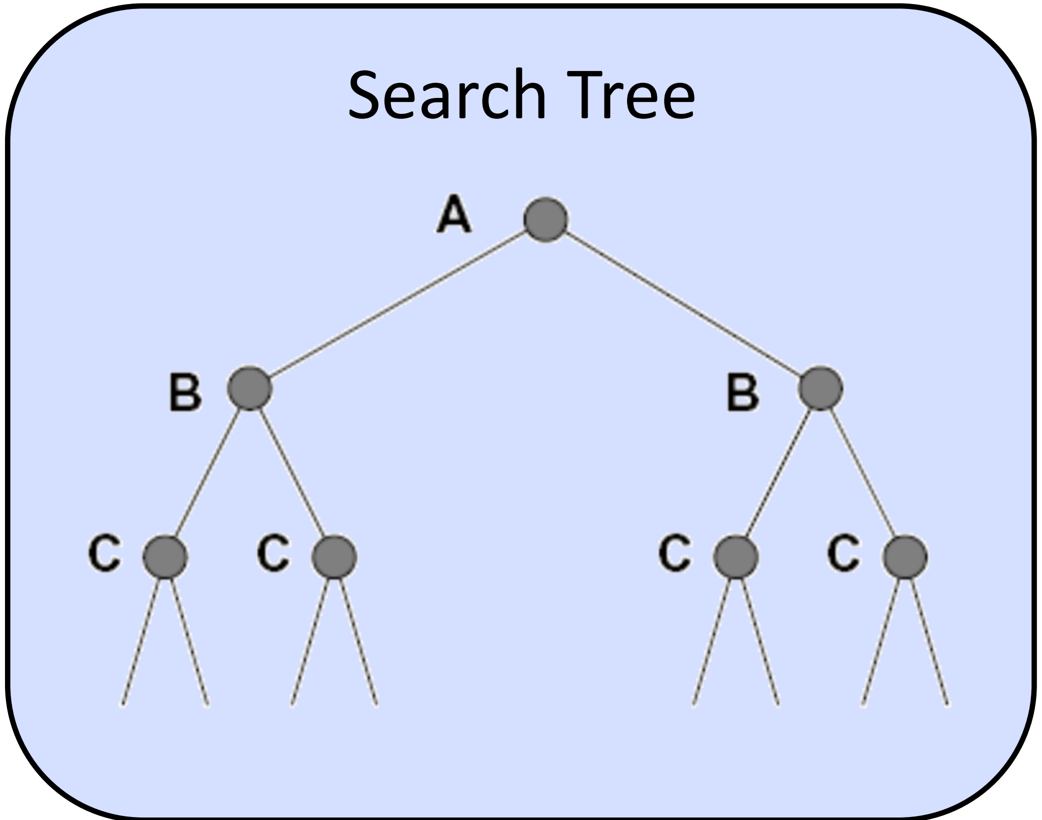
Graph Search



Tree Search: Extra Work!

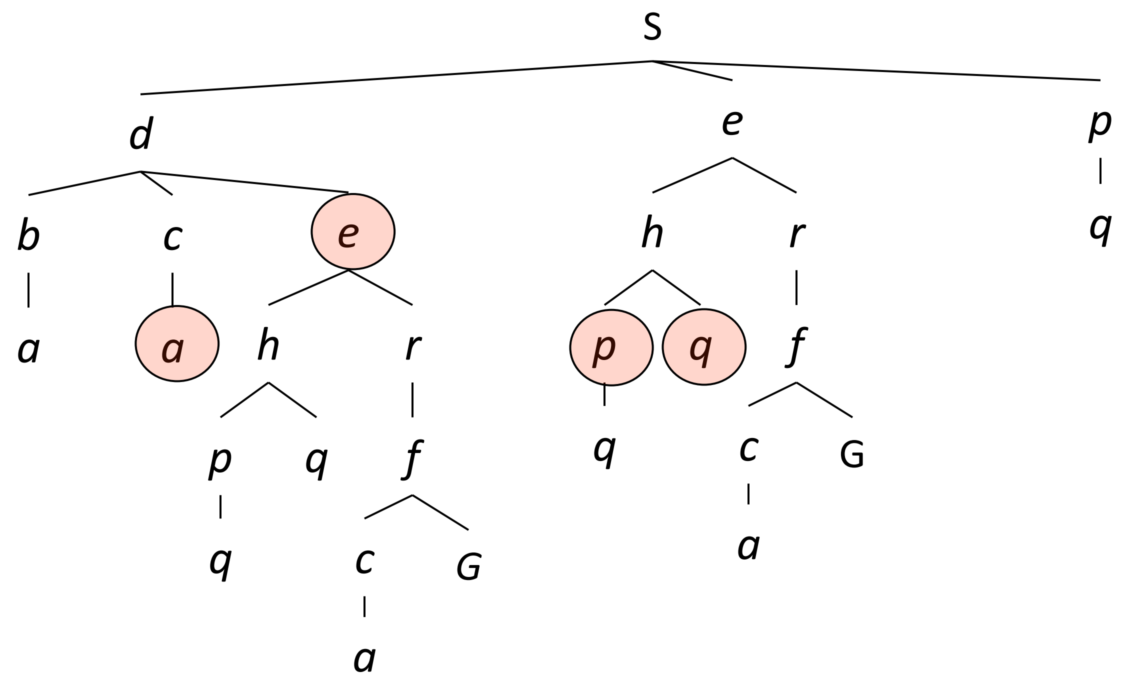
- Failure to detect repeated states can cause exponentially more work.
- We call a search algorithm a graph search if it checks for redundant paths and a tree-like search if it does not check.
- The BEST-FIRST-SEARCH algorithm in Figure 3.7 is a graph search algorithm; if we remove all references to reached we get a treelike search that uses less memory but will examine redundant paths to the same state, and thus will run slower.





Graph Search

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

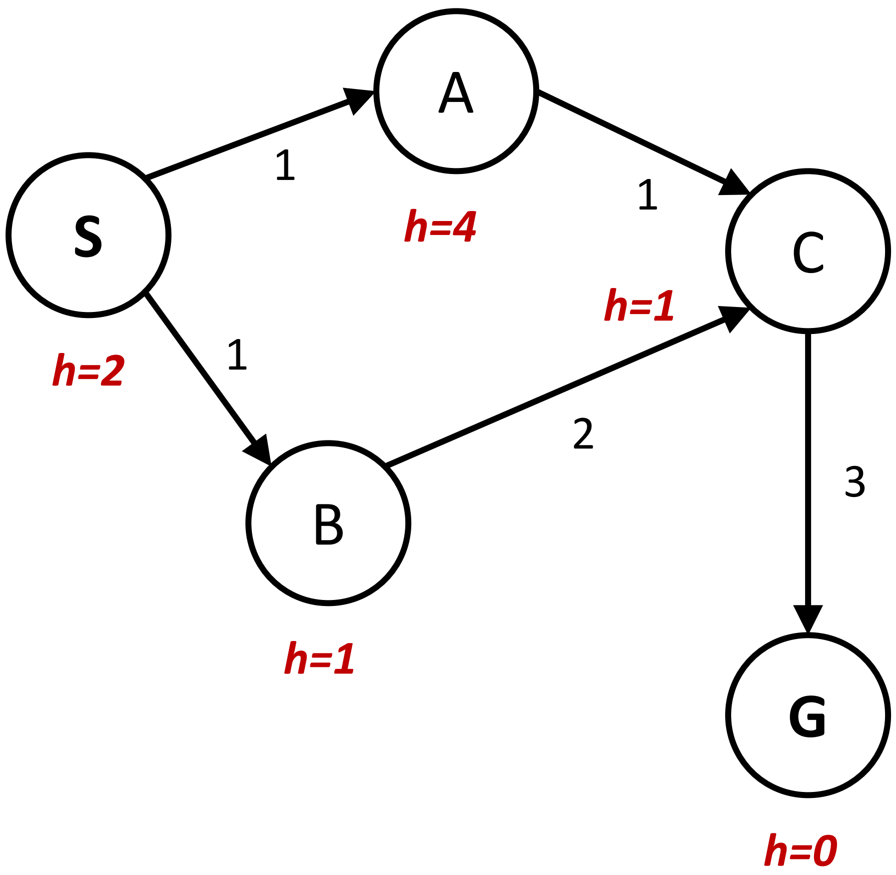


Graph Search

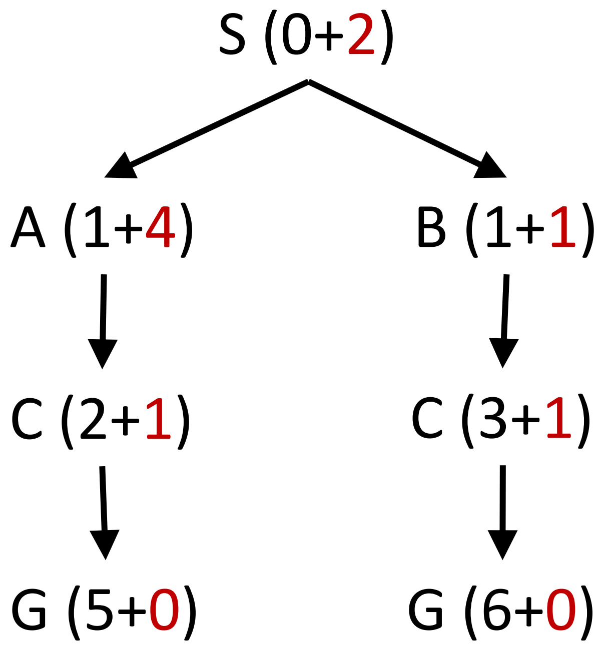
- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: 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



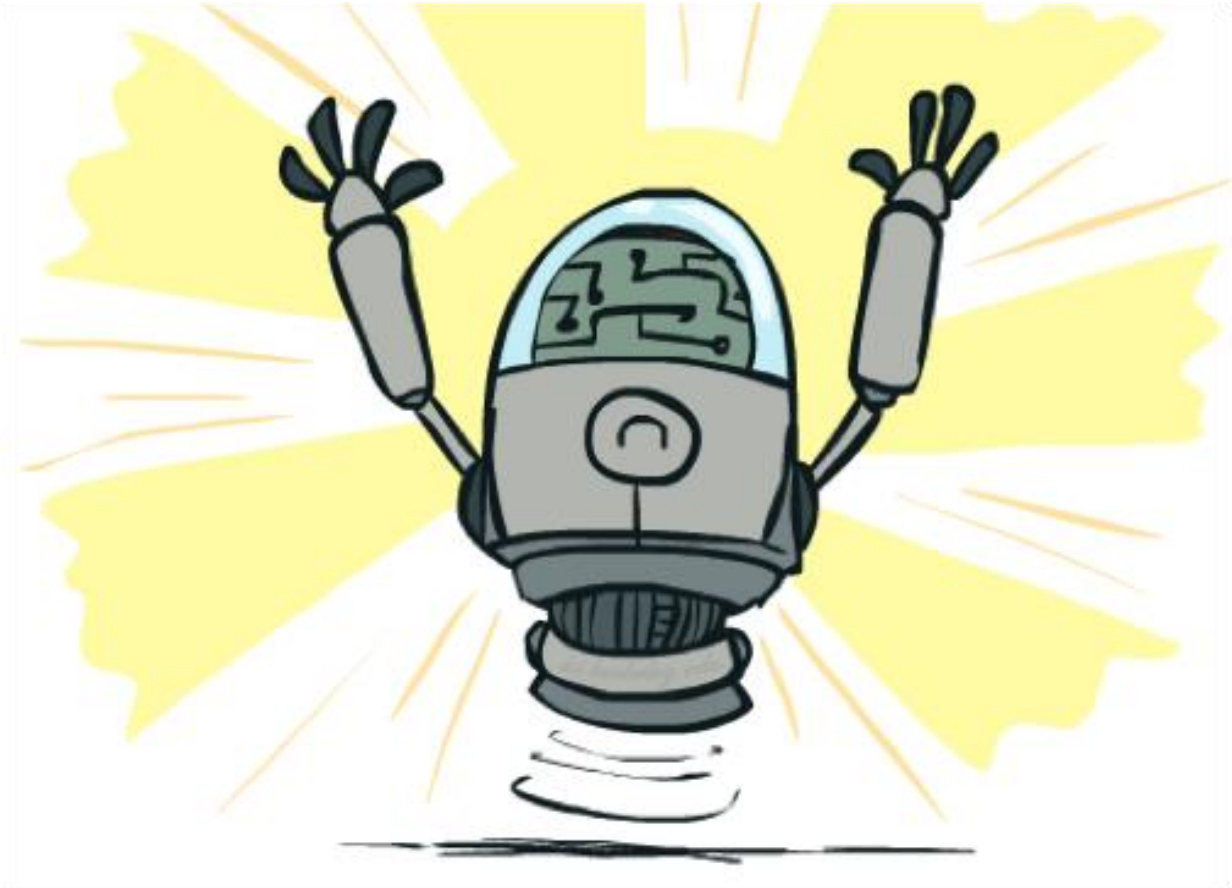
Tree Search Pseudo-Code

```
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

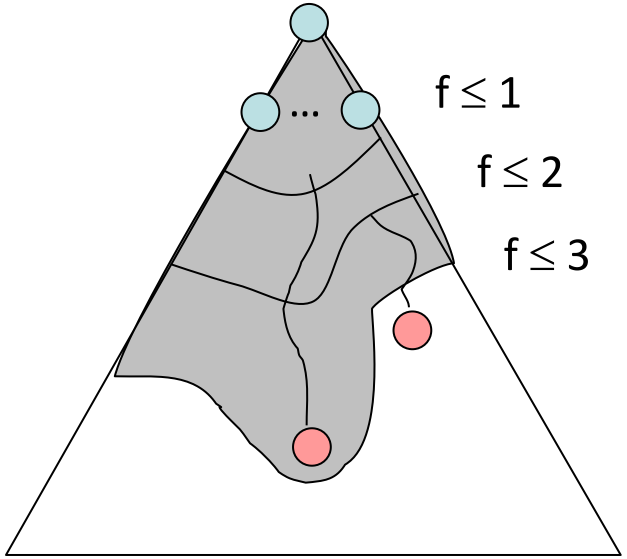
```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then 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
```

Optimality of A* Graph Search



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



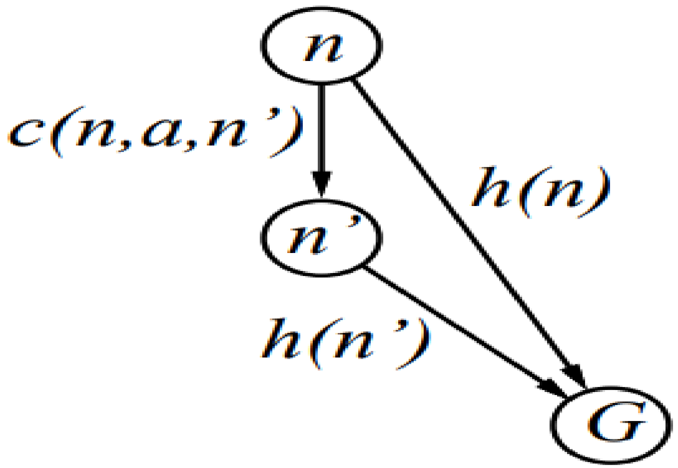
Consistent (or Monotonicity) Heuristics

A heuristic is *consistent* if

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

If f is consistent, we have

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



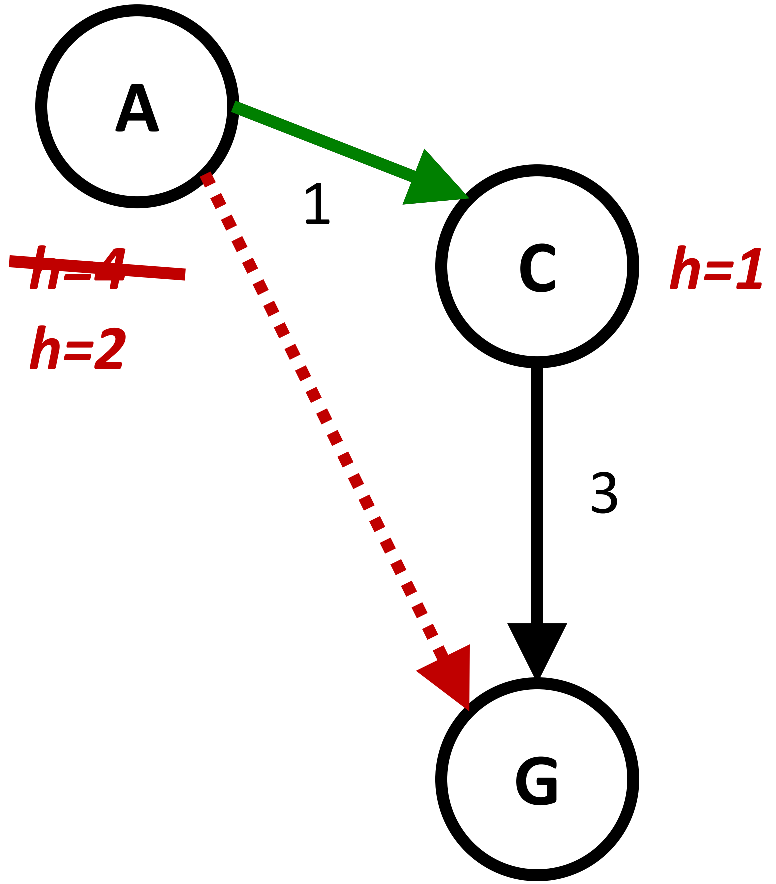
I.e., $f(n)$ is nondecreasing along any path

Note:

- Consistent $\Rightarrow$ admissible
- Most admissible heuristics are also consistent

c(n,a,n'): the cost of applying action *a* in state *n* to arrive at state *n'*.

Consistency of Heuristics



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

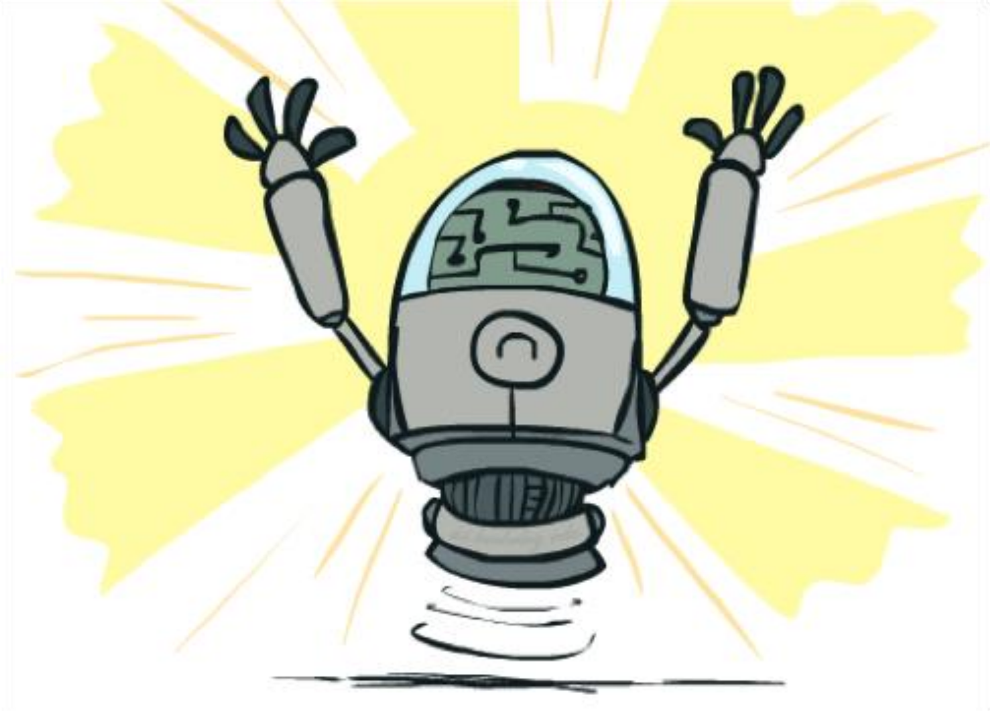
    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    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

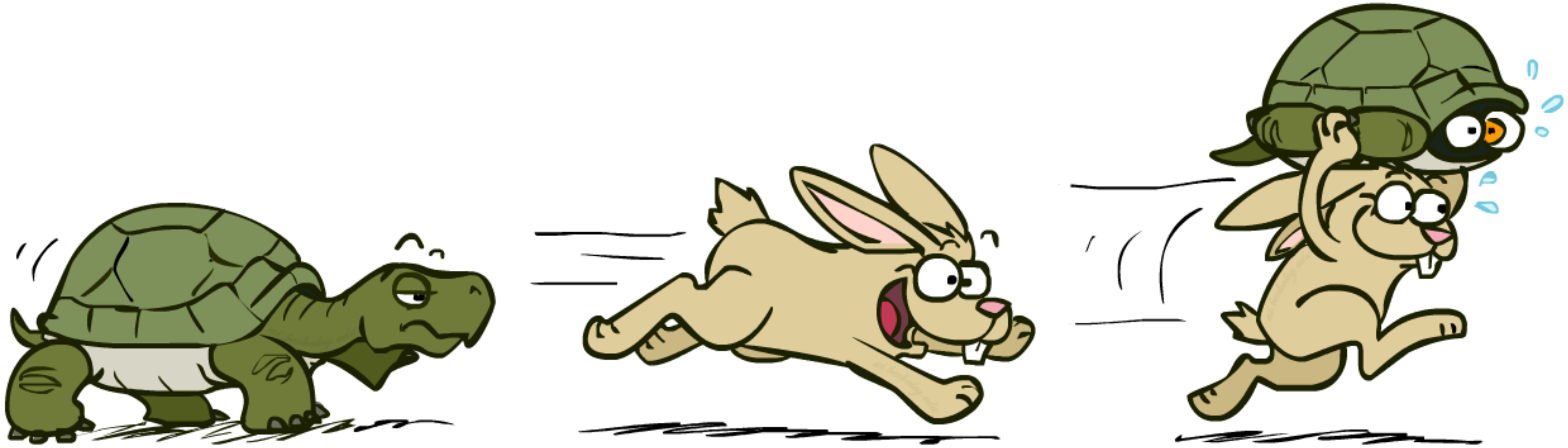
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially 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



- http://qiao.github.io/PathFinding.js/visual/