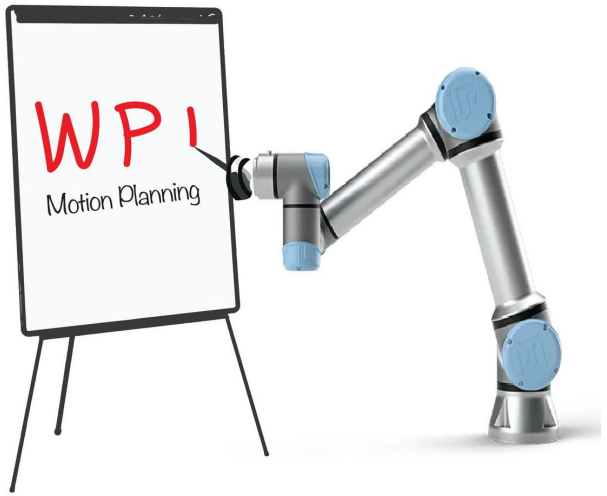
RBE550
Motion Planning
Discrete Search & Roadmaps



Constantinos Chamzas
www.cchamzas.com
www.elpislab.org

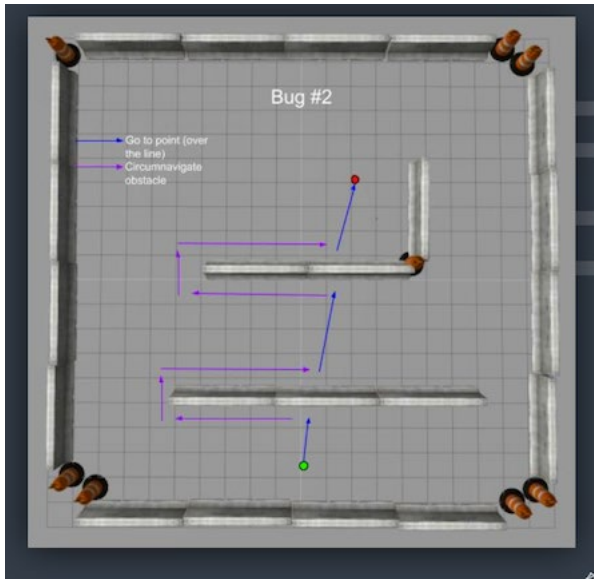
Disclaimer and Acknowledgments

The slides are a compilation of work based on notes and slides from Constantinos Chamzas, Lydia Kavraki, Zak Kingston, Howie Choset, David Hsu, Greg Hager, Mark Moll, G. Ayorkor Mills-Tetty, Hyungpil Moon, Zack Dodds, Nancy Amato, Steven Lavalle, Seth Hutchinson, George Kantor, Dieter Fox, Vincent Lee-Shue Jr., Prasad Narendra Atkar, Kevin Tantiseviand, Bernice Ma, David Conner, Morteza Lahijanian, Erion Plaku, and students taking comp450/comp550 at Rice University.

The Search slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.

Last time

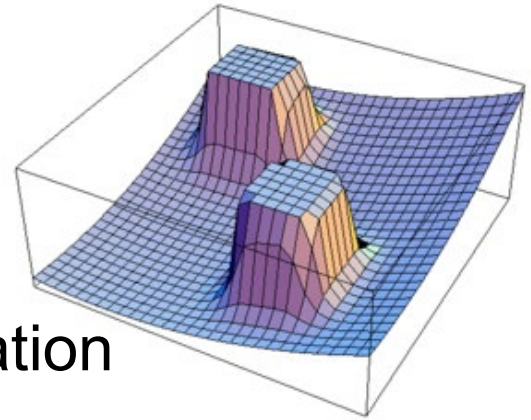
Bug Algorithms



Pros:
- Simple to implement
- Complete
- Continuous

Cons:
- 2D robots
- Non Optimal Motions

Potential Fields

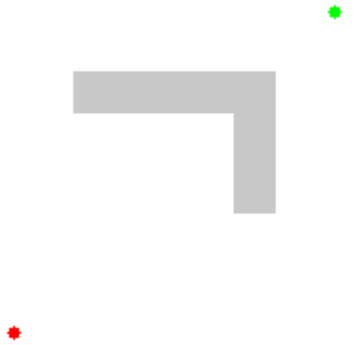


Pros:
- Beyond 2D
- Smooth Motions
- Continuous
- Efficient Computation

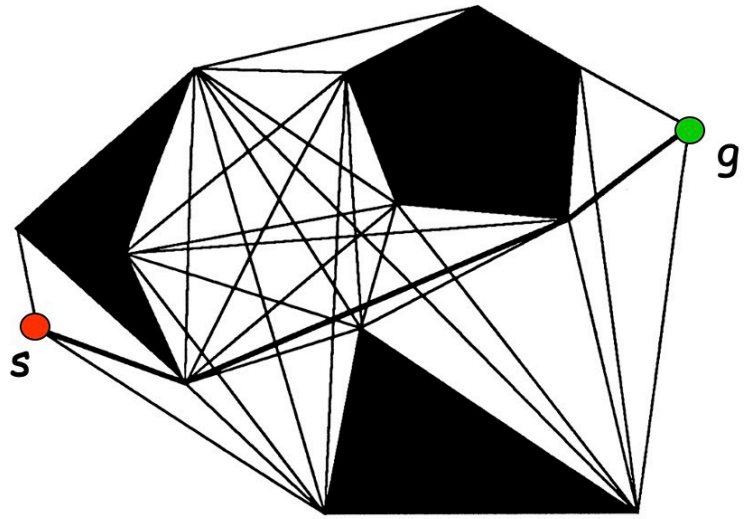
Cons:
- Stuck in Local Minima
- Not-Complete
- Non-Optimal Motions
- Challenging distance calculation

Today's Overview

Discrete Search

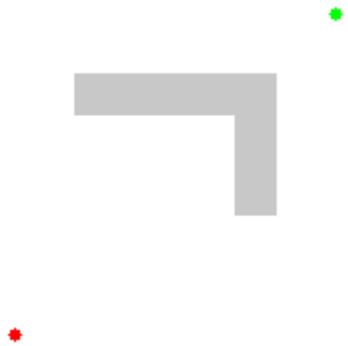


Roadmaps



Discrete Search Vs Bugs and Potentials

Discrete Search



Pros:
- Optimal
- Complete
- Efficient

Cons:
- Discrete C-Space

Bug Algorithms

Pros:
- Simple to implement
- Complete
- Continuous C-Space

Cons:
- 2D robots
- Non-Optimal Motions

Potential Fields

Pros:
- Beyond 2D
- Smooth Motions
- Continuous C-Space
- Efficient Computation

Cons:
- Stuck in Local Minima
- Not-Complete
- Non-Optimal Motions

How to use Discrete Search for Motion Planning?

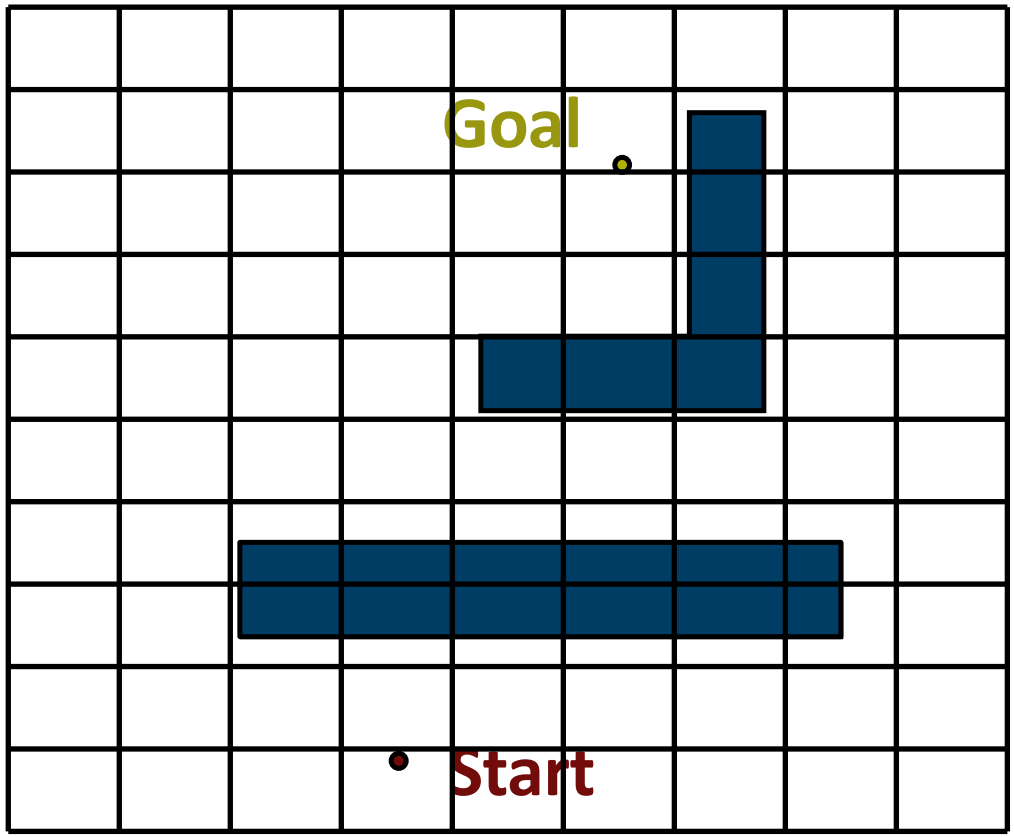
- **Motion Planning Definition (Point Robot)**
- **Workspace**: The environment in which the robot operates
  - Notation: W
  - Generally, either 2- or 3-Dimensional Euclidean space,
  - For 2-D point robot $W = \mathbb{R}^2$
    - Obstacle i in workspace W is denoted by WO_i
    - Free workspace: $W_{\text{free}} = W \setminus \cup_i WO_i$
- **Configuration Space:** a complete specification of the robot's state (informal definition)
  - Notation: Q or X or C
  - Dimension generally depends on the robot
  - For point robot $Q = W = \mathbb{R}^2$ and WE
  - Collision-free space $Q_{\text{free}} = W_{\text{free}}$, and $q_{start}, q_{goal} \in Q_{free}$
- **Robot path from q_{start} to q_{goal} :** A continuous curve $\sigma(t)$ such that $\{\sigma(t) \in Q_{free} \mid \sigma(0) = \text{q}_{\text{start}}, \sigma(1) = \text{q}_{\text{goal}}\}$

We need to discretize the C-Space

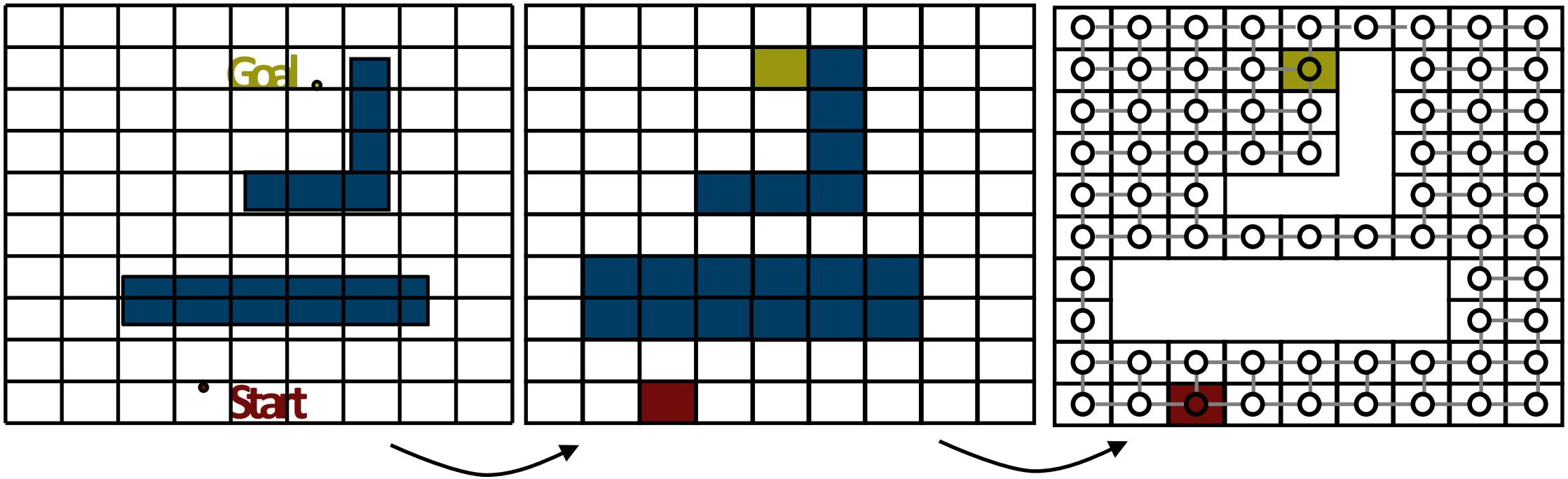
Discretizing the Configuration Space (2D Point Robot)

In general it is challenging problem

- For now we will just use a grid on $\mathbb{R}^2$



Simple Grid Discretization for (2D Point Robot)



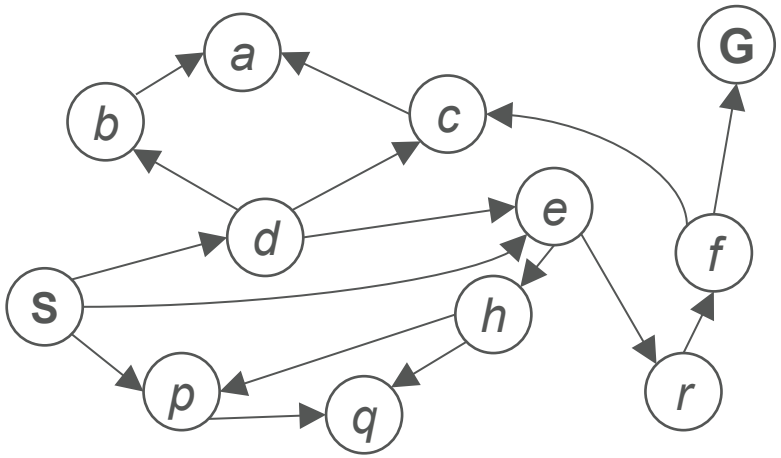
Map each grid-cell to free or occupied

Free cells become graph nodes
L_1-norm neighbors have edges

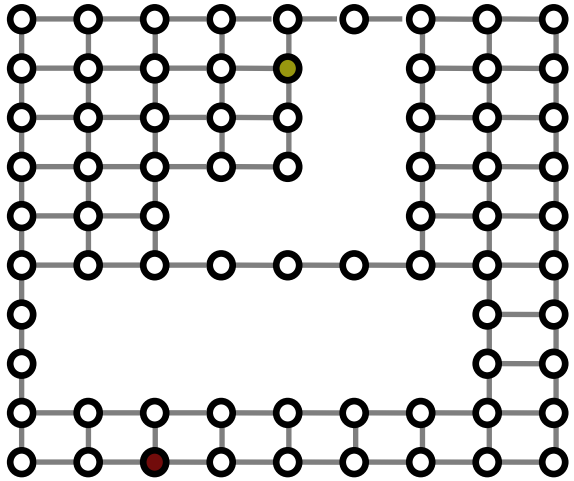
We have arrived at a Graph

Search Graph: A mathematical representation of a search problem

- Nodes are (abstracted) world configurations
- Edges represent possible actions
- The goal is a set of goal nodes (maybe only one)

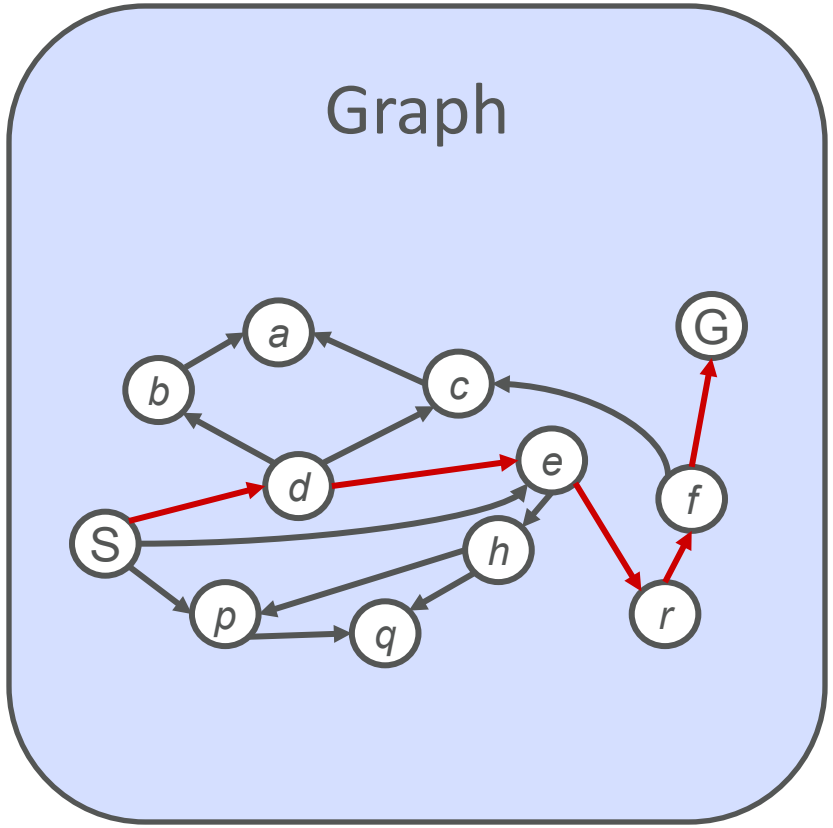


A graph for a small search problem

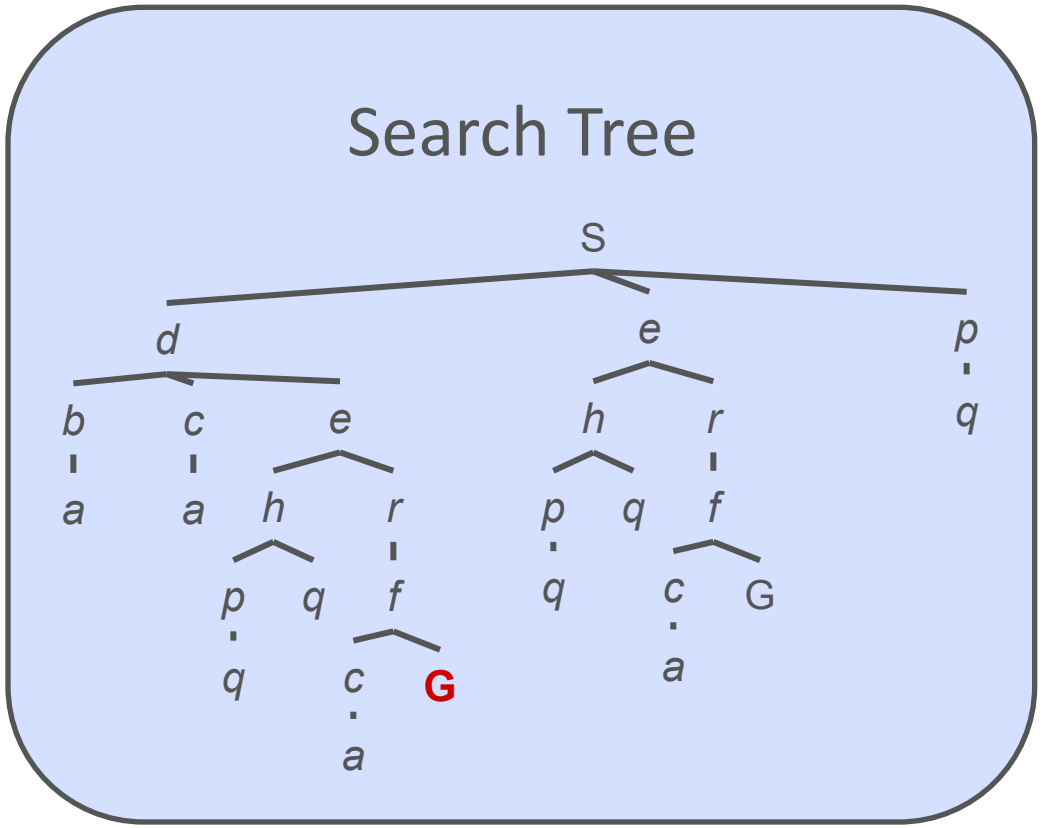


Graph representation for the point 2D robot

Searching a Graph with a search Tree

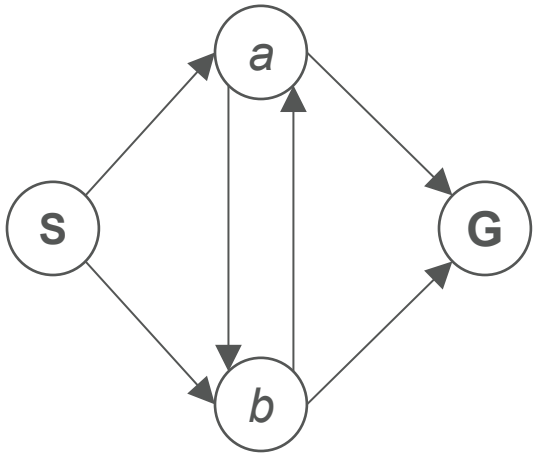


Each NODE in in the search tree is an entire PATH in the state space graph.



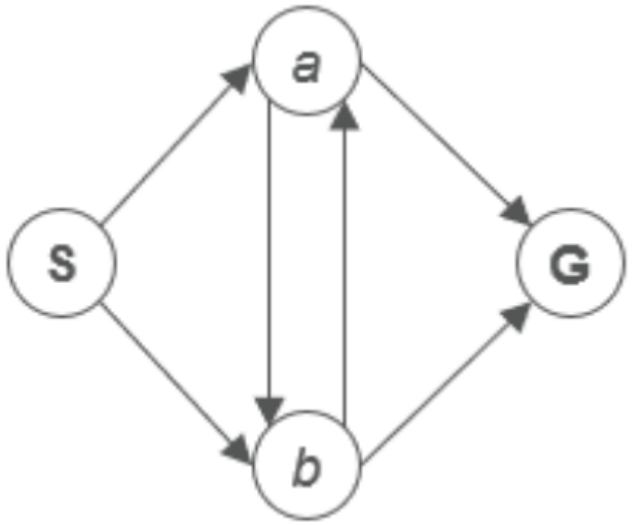
Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



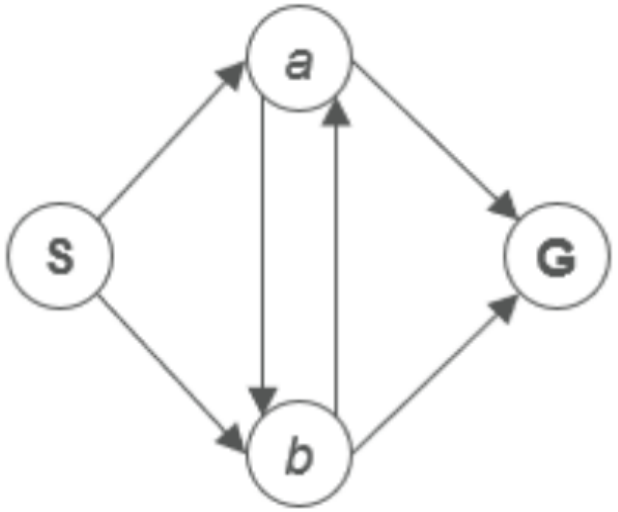
- How big is the search tree (from S)?
- In other words how many paths exist from S to G?

How big is the search tree from S (how many leaf nodes does it have)



- 4
- 6
- 10
- ∞
- None of the above

How big is the search tree from S (how many leaf nodes does it have)



4
0%

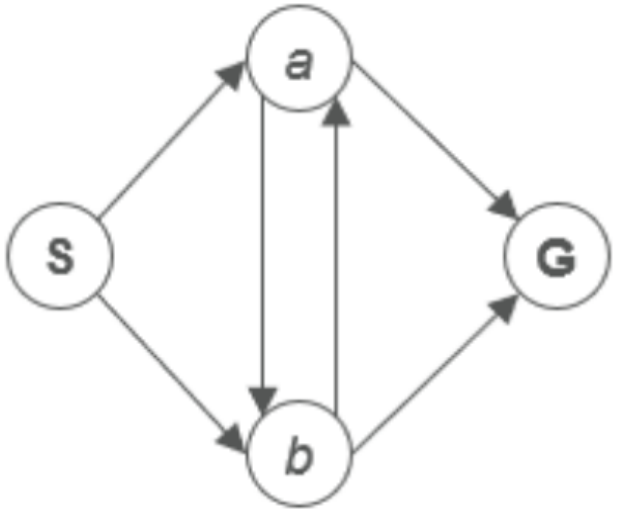
6
0%

10
0%

∞
0%

None of the above
0%

How big is the search tree from S (how many leaf nodes does it have)



4 0%

6 0%

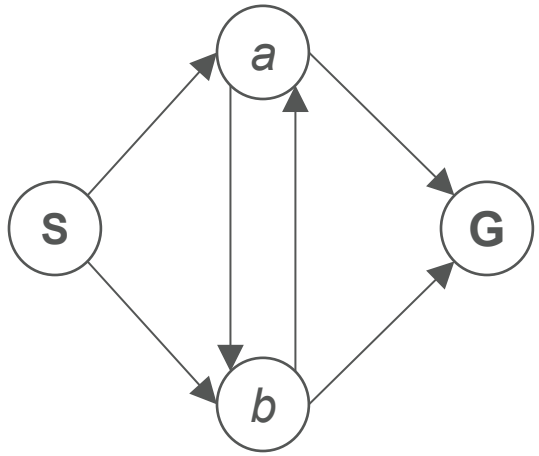
10 0%

∞ 0%

None of the above 0%

Quiz: State Space Graphs vs. Search Trees

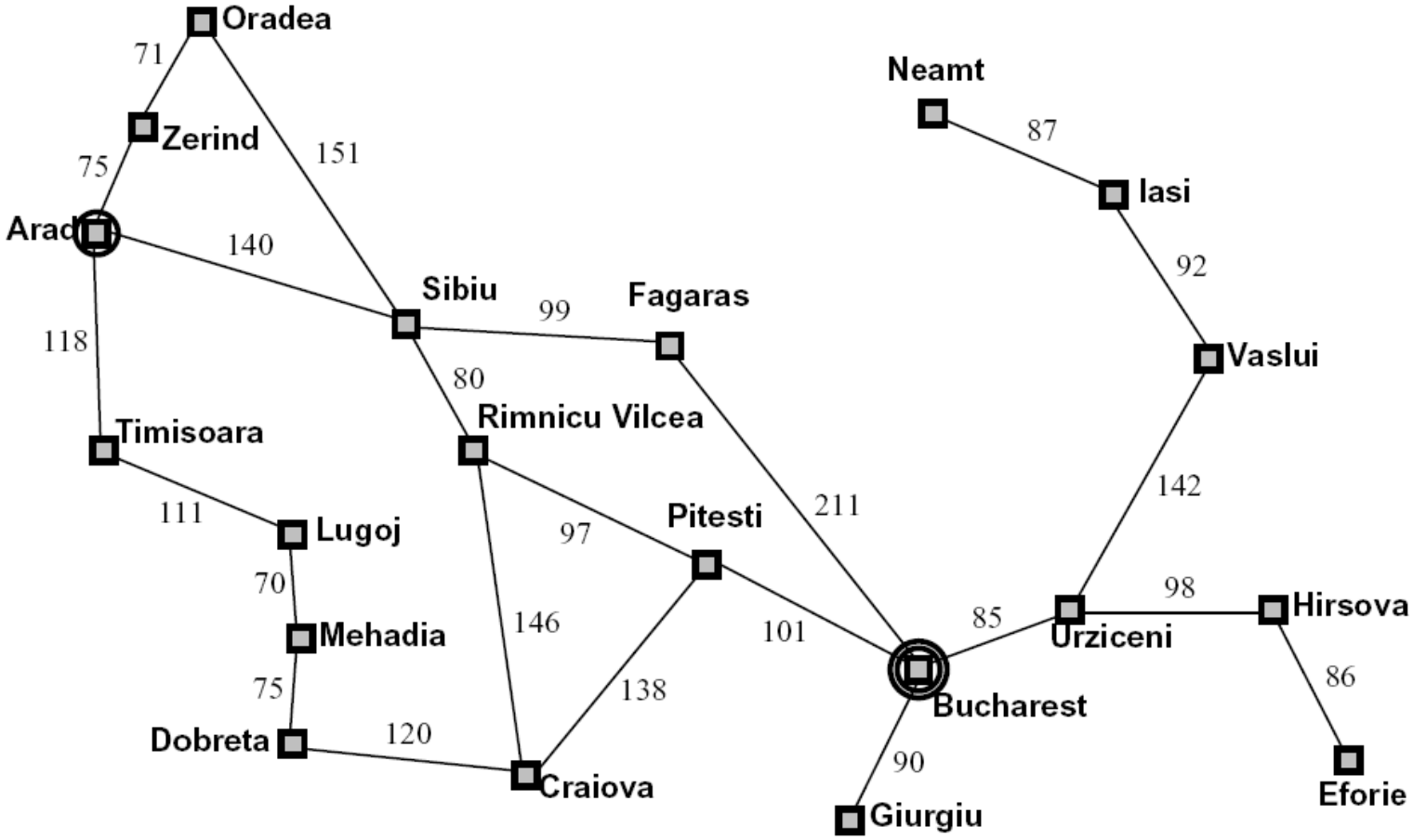
Consider this 4-state graph:



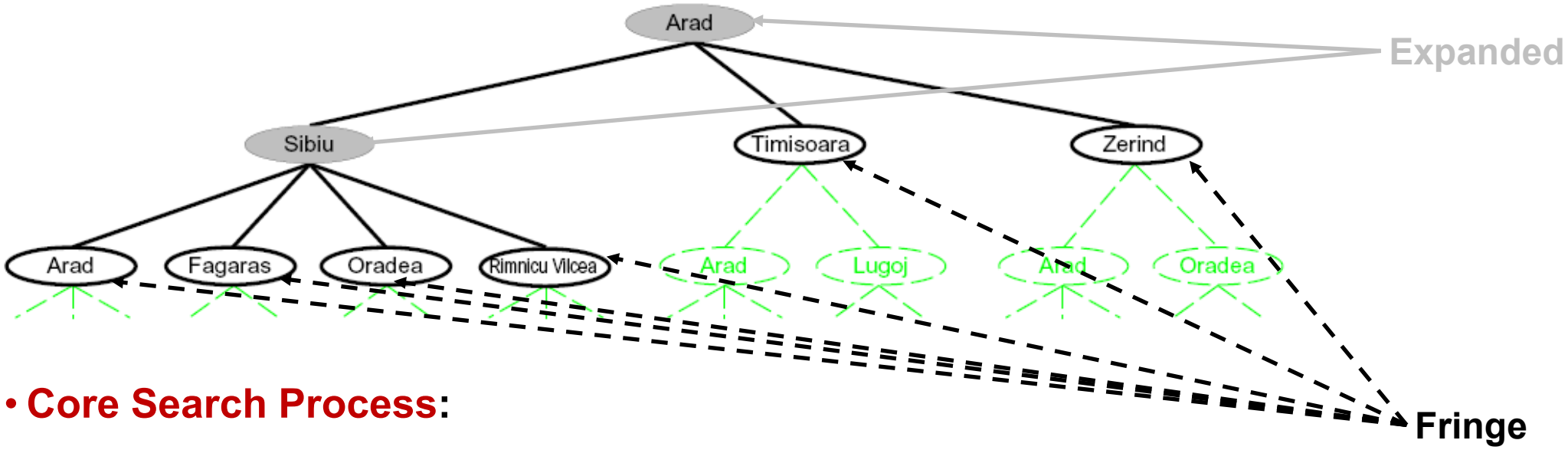
How big is its search tree (from S)?
In other words how many paths exist from S to G?

Important: Lots of repeated structure in the search tree!

Search Example: Romania



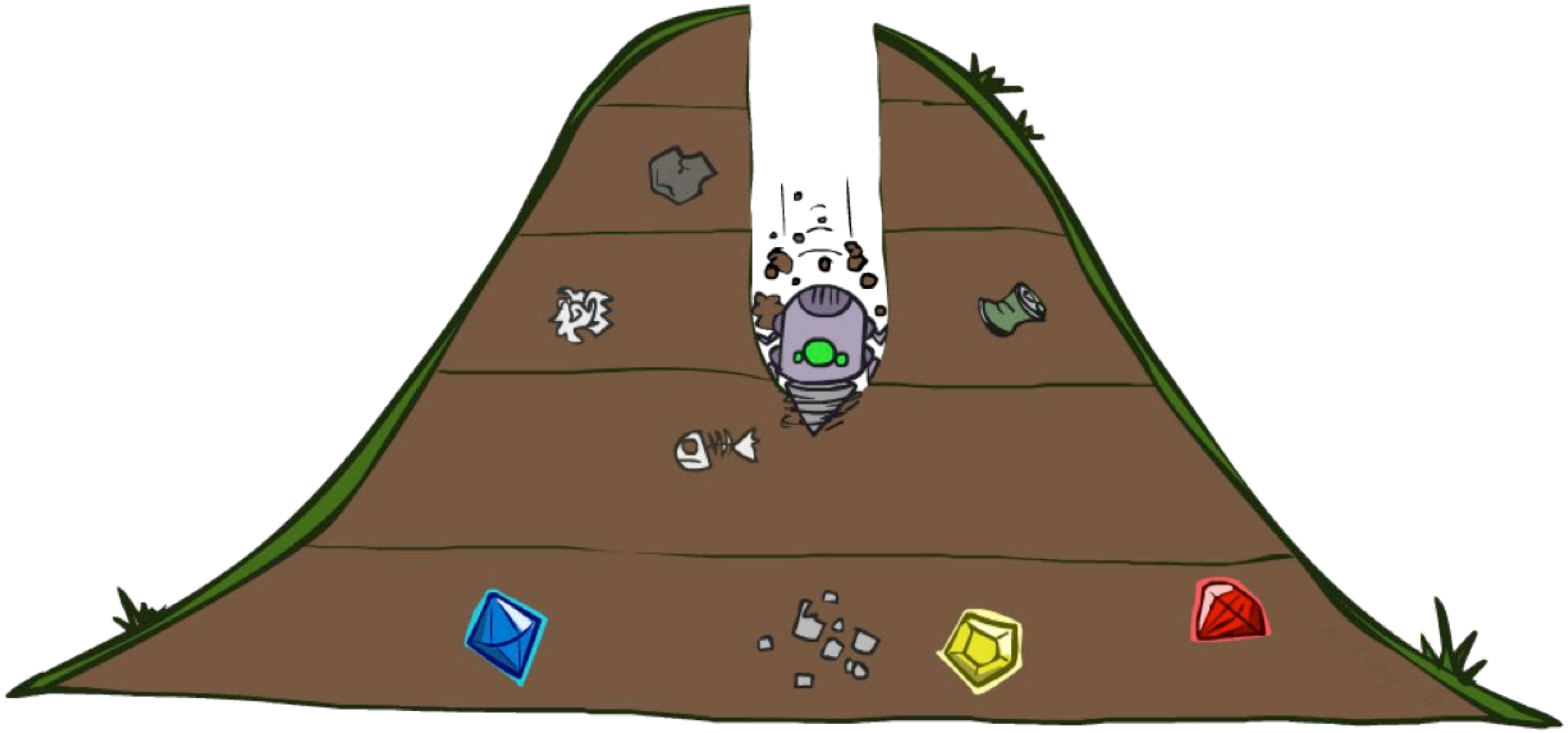
Searching with a Search Tree



- **Core Search Process**:
  - Expand out potential plans (tree nodes)
  - Maintain a **fringe** of partial plans under consideration
  - Try to expand as few tree nodes as possible

Main question: which fringe nodes to explore first?

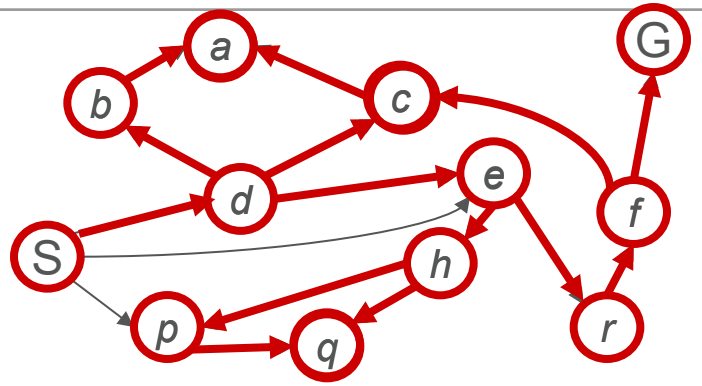
Depth-First Search

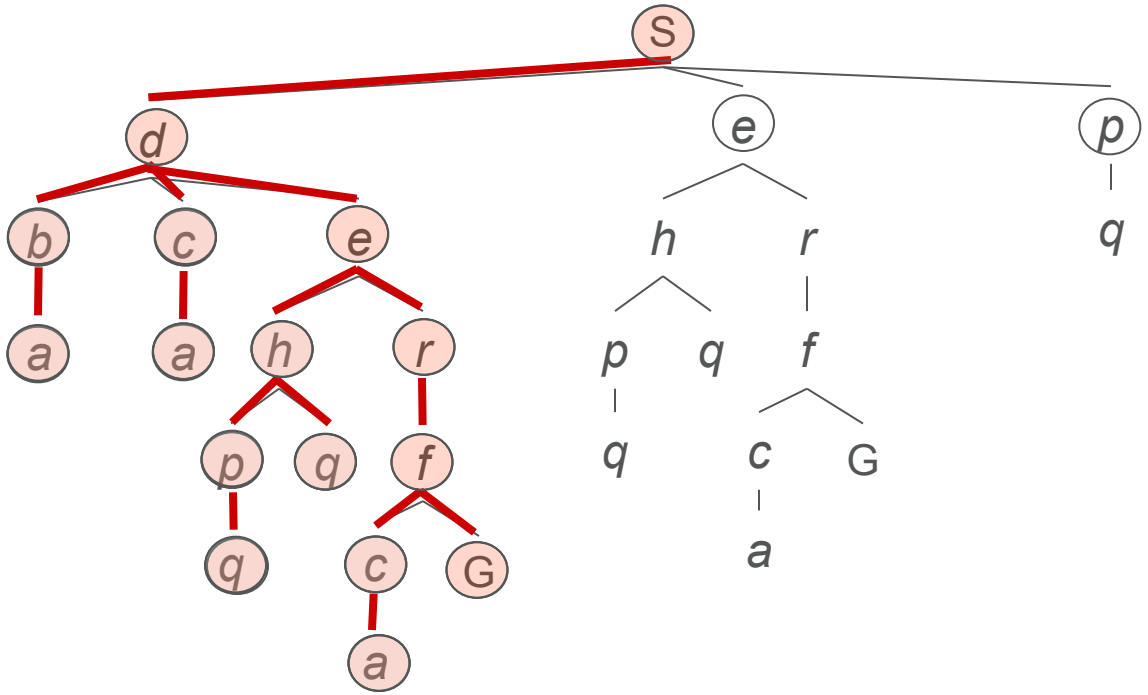


Depth-First Search

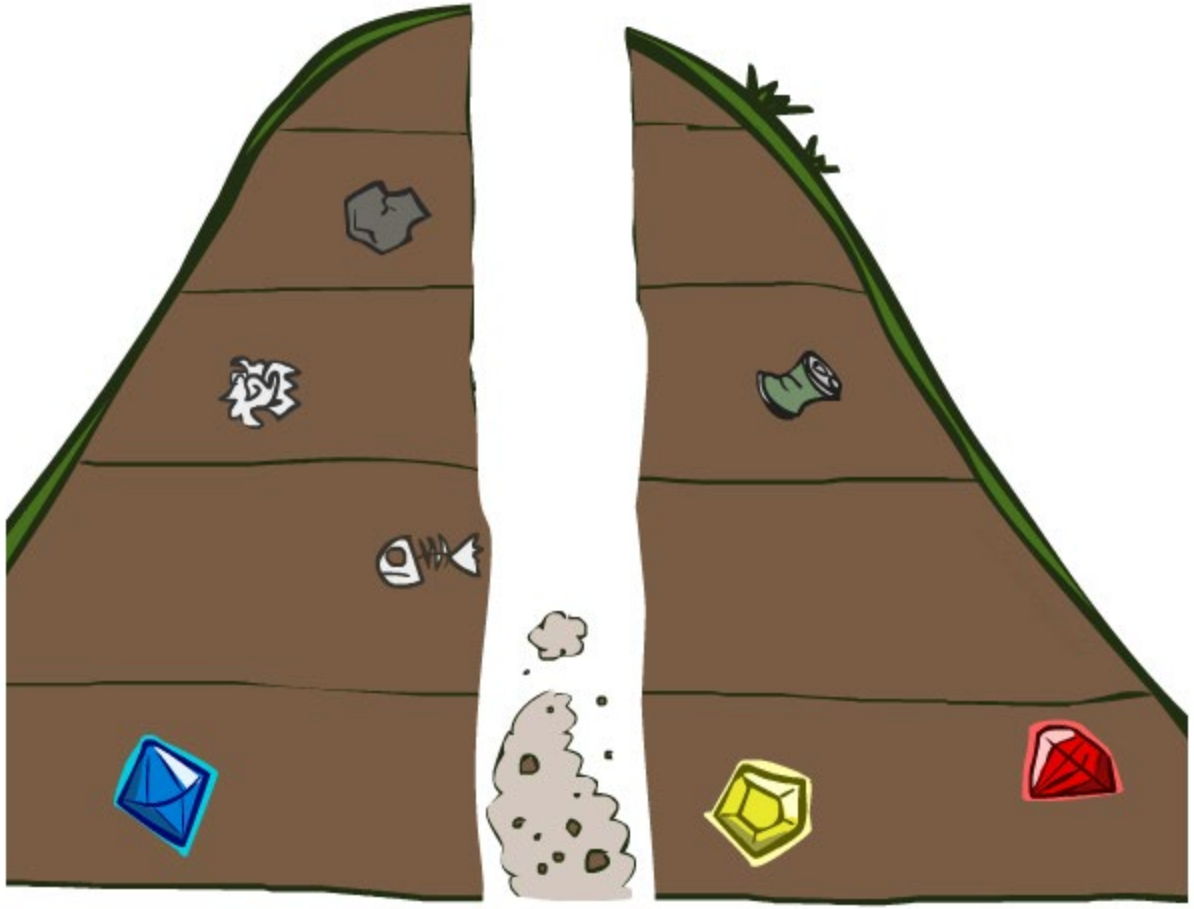
Strategy: expand the deepest node first

Implementation: Fringe is a LIFO stack



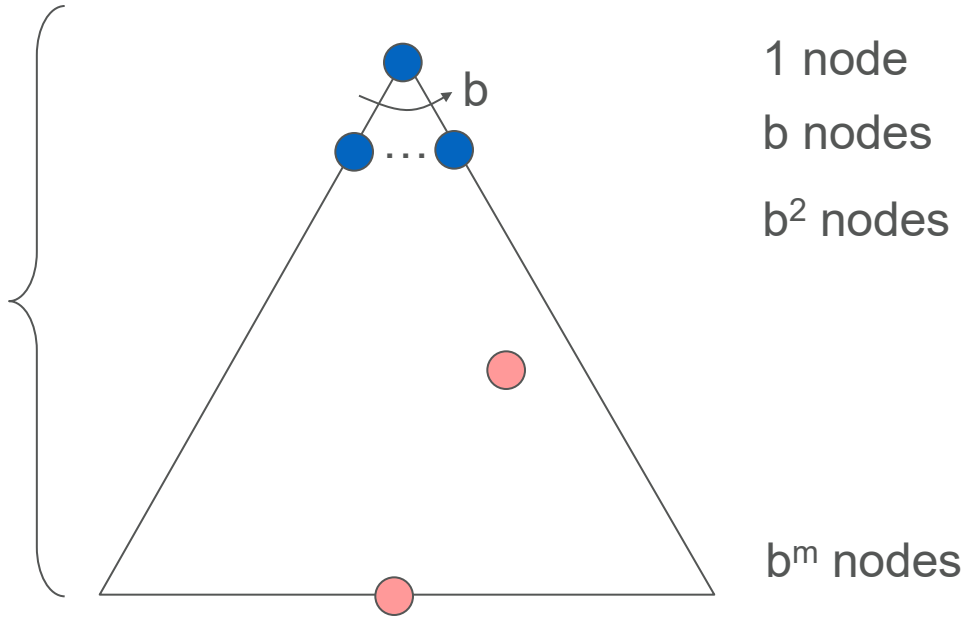


Search Algorithm Properties



Search Algorithm Properties

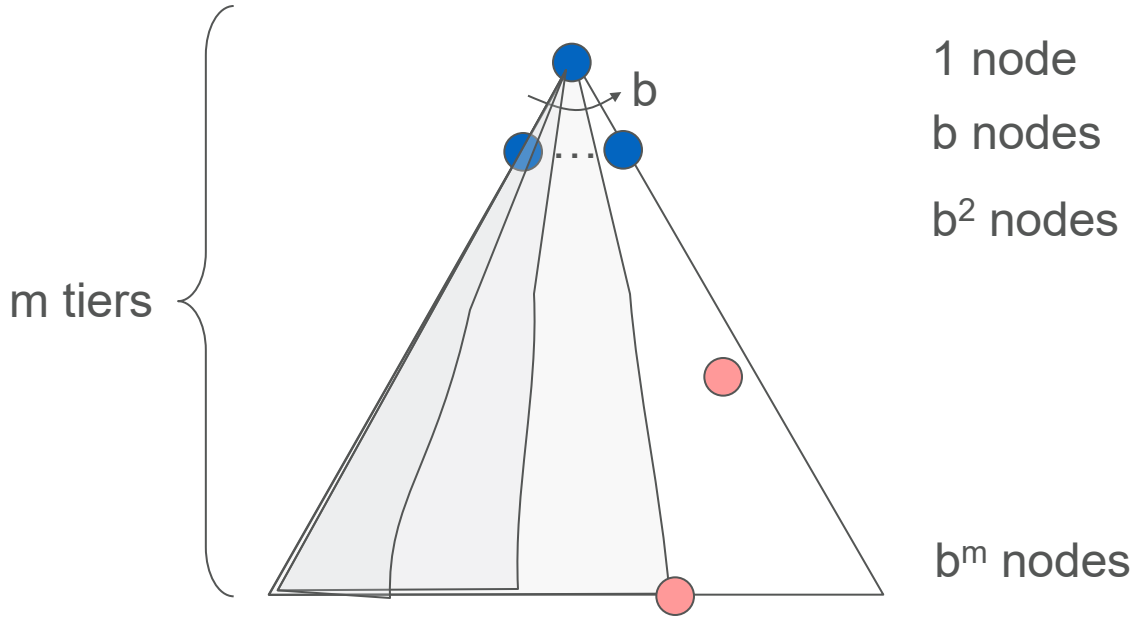
- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths
- Number of nodes in entire tree?



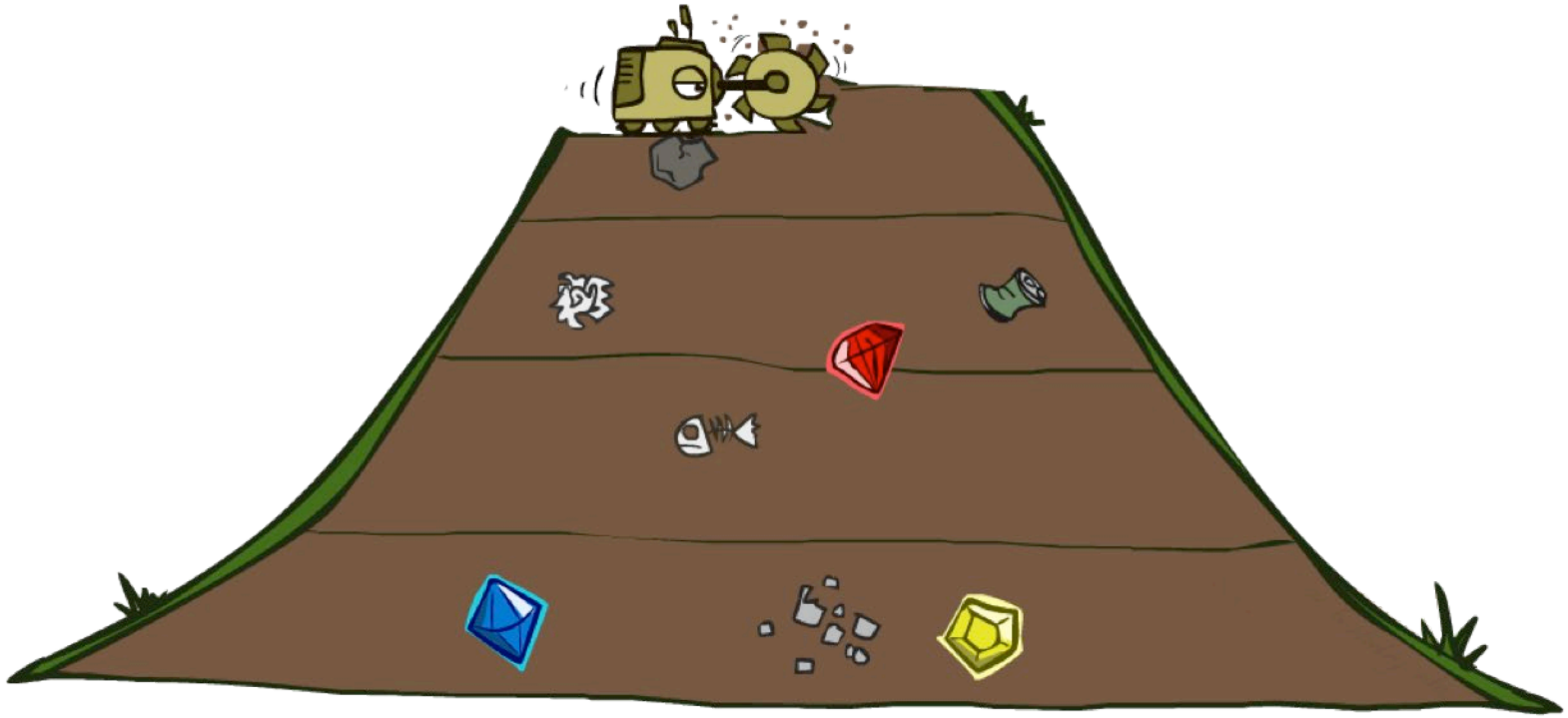
- $1 + b + b^2 + \ldots. b^m = \frac{b^{m+1}-1}{b-1} = O(b^m)$

Depth-First Search (DFS) Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



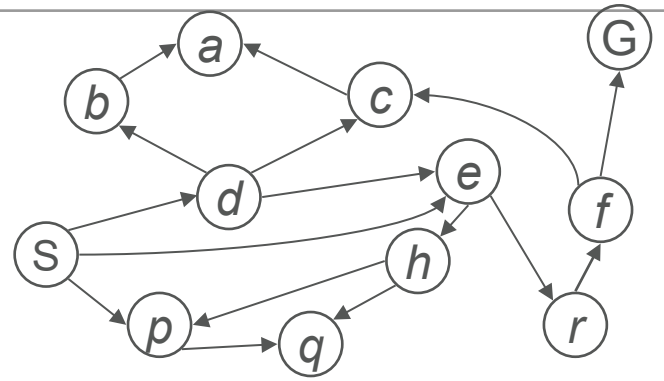
Breadth-First Search

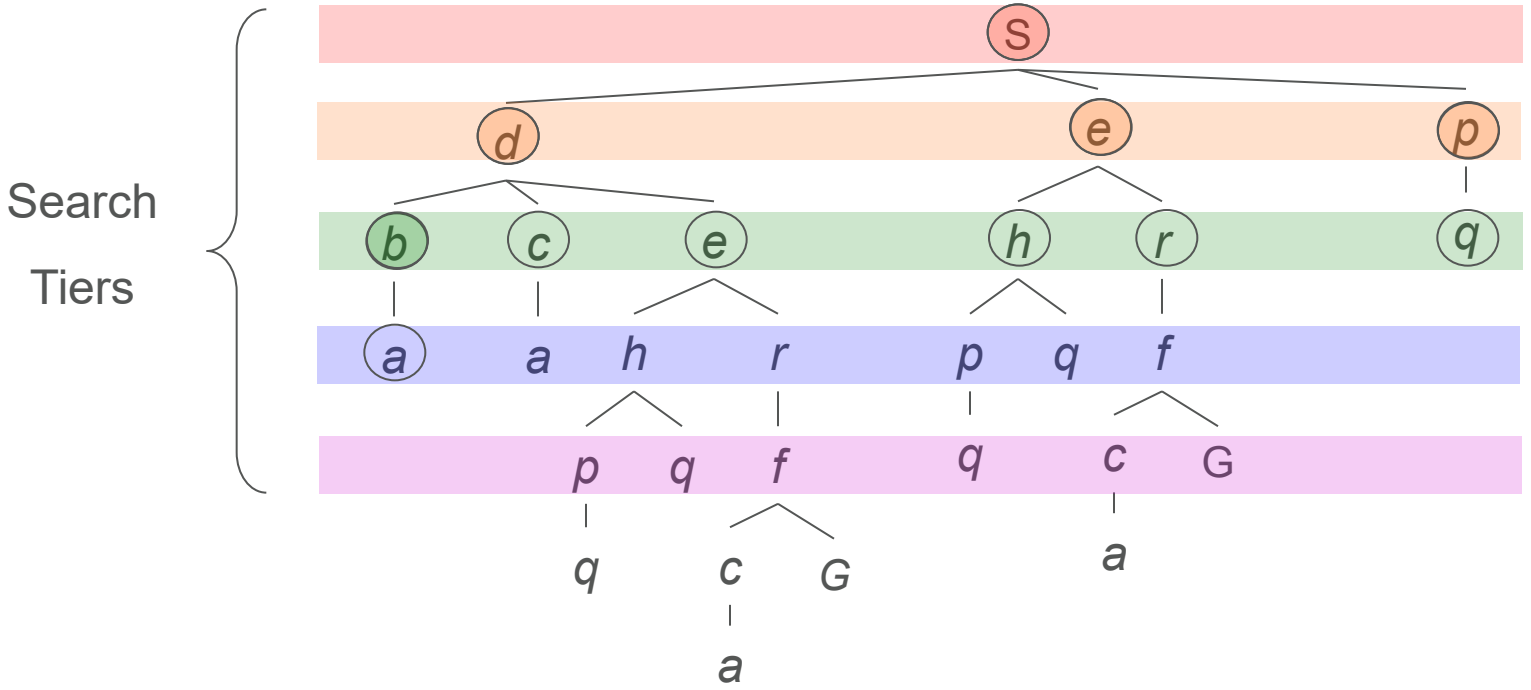


Breadth-First Search

Strategy: expand a shallowest node first

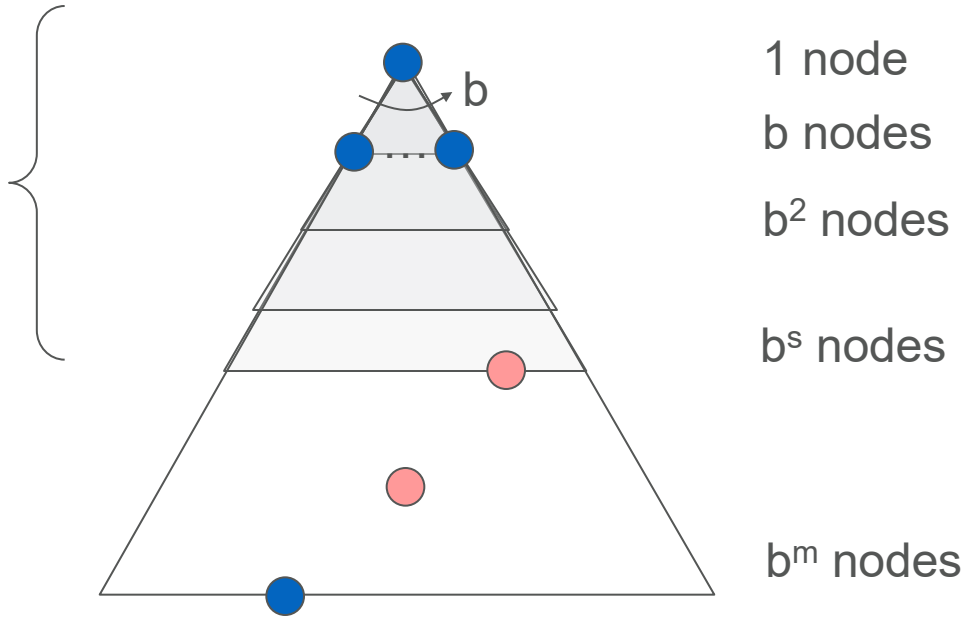
Implementation: Fringe is a FIFO queue



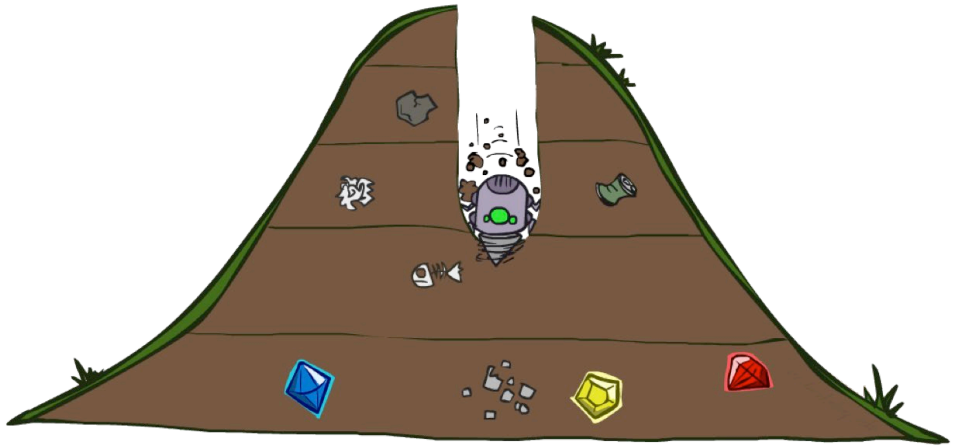


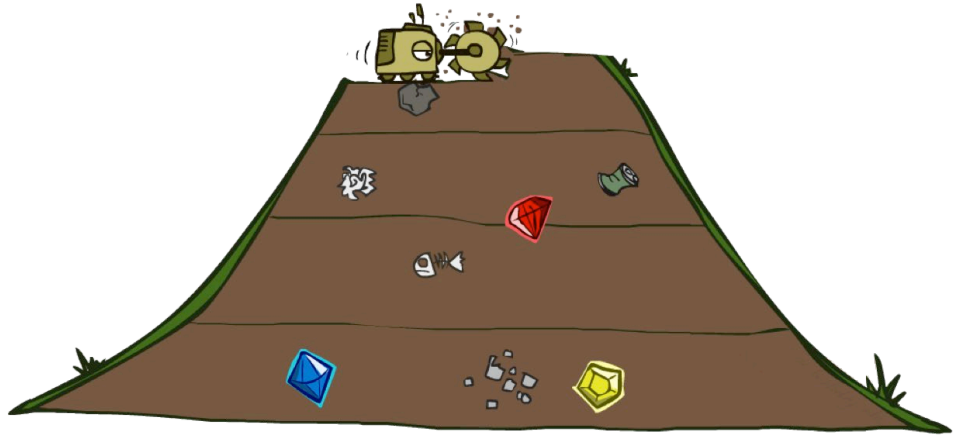
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



Quiz: DFS vs BFS

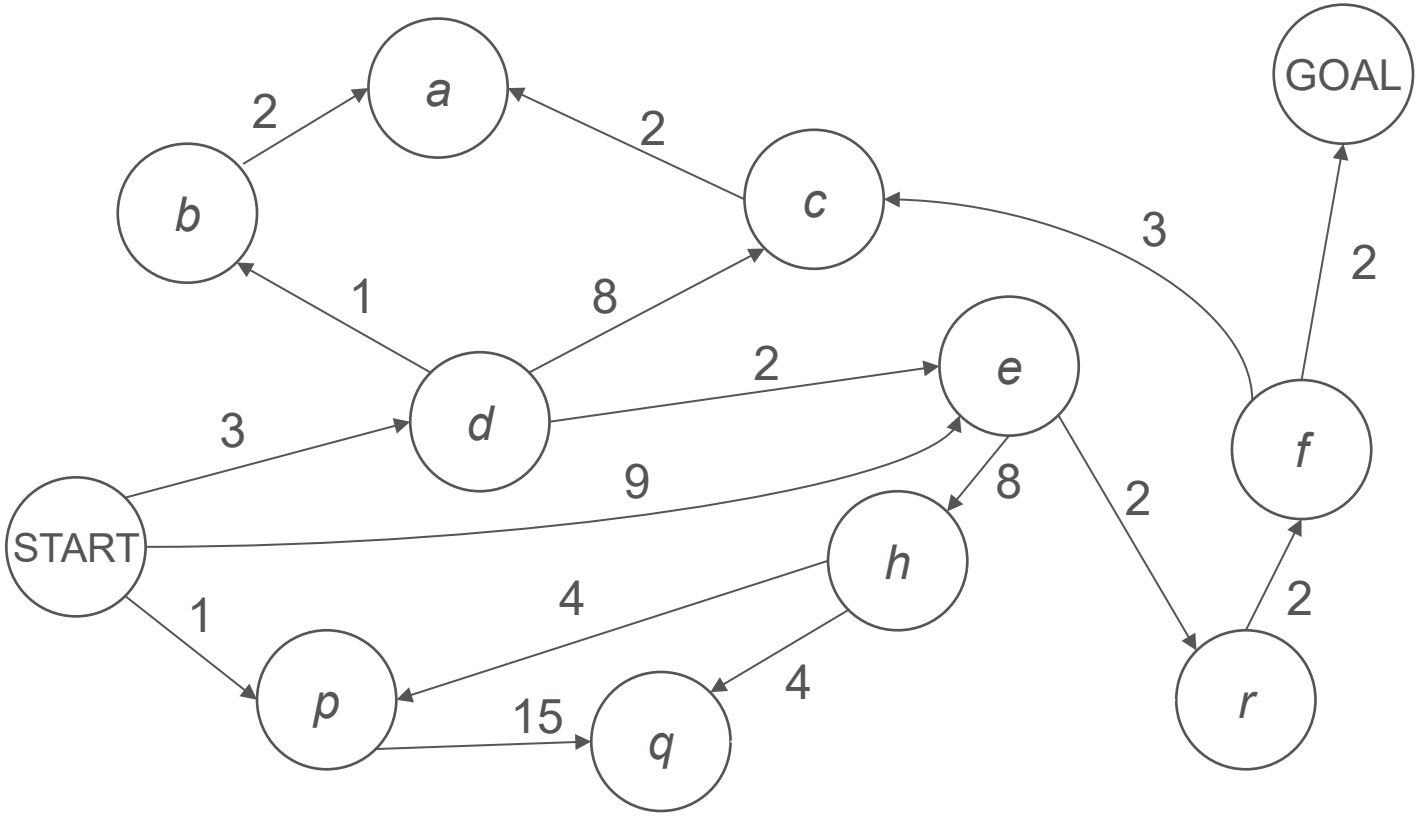




Video of Demo Maze Water DFS/BFS (part 1)

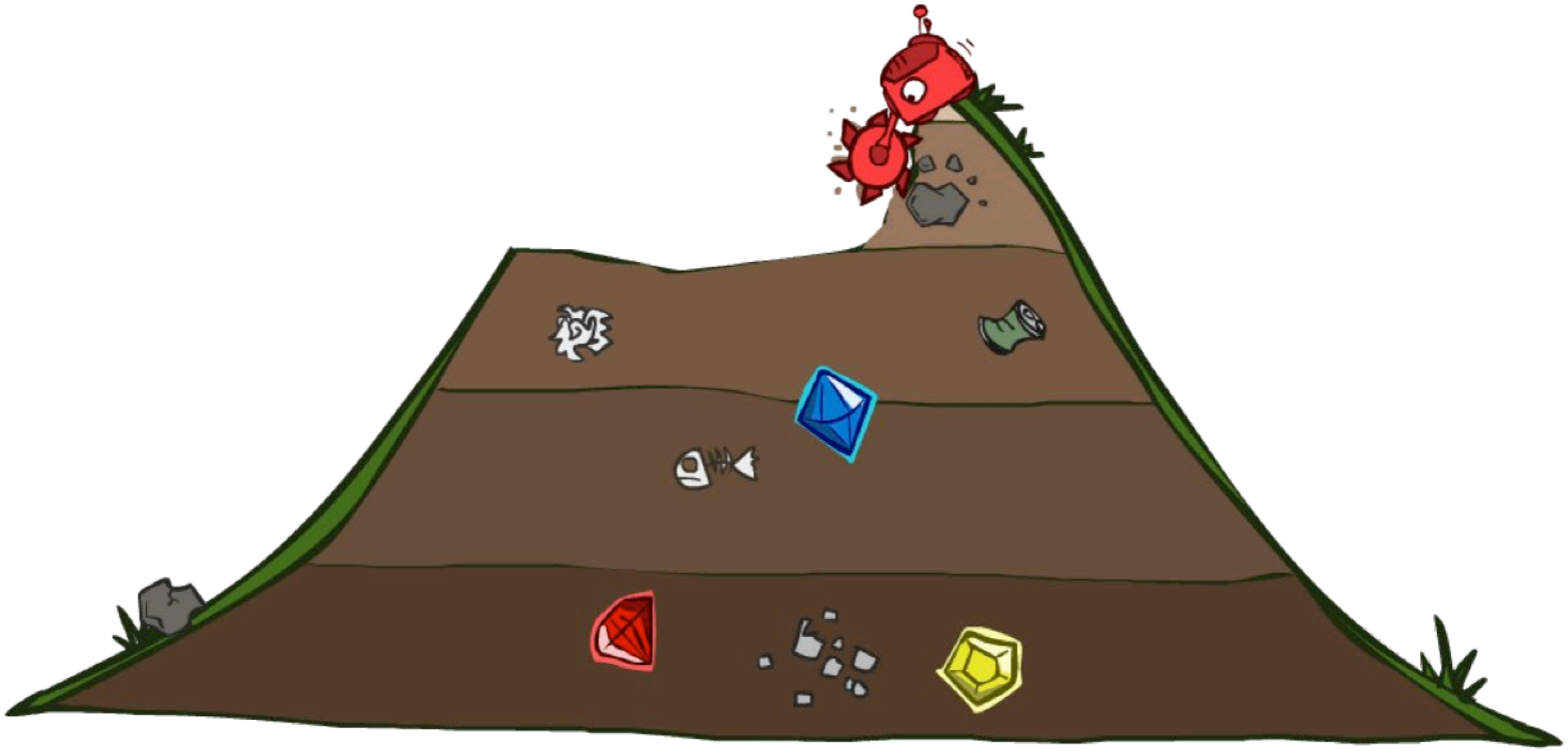
Video of Demo Maze Water DFS/BFS (part 2)

Cost-Sensitive Search



BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path. We will now cover
a similar algorithm which does find the least-cost path.

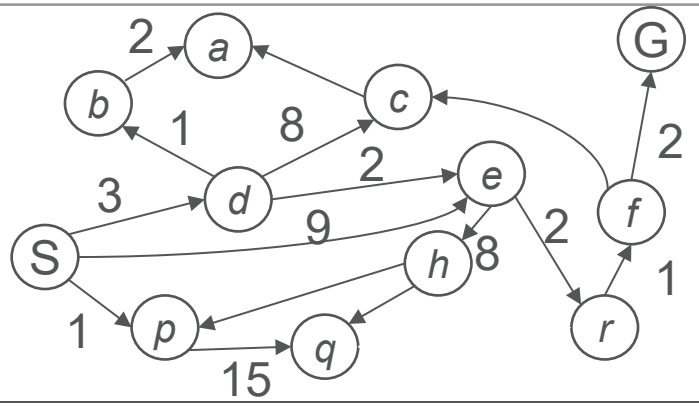
Uniform Cost Search



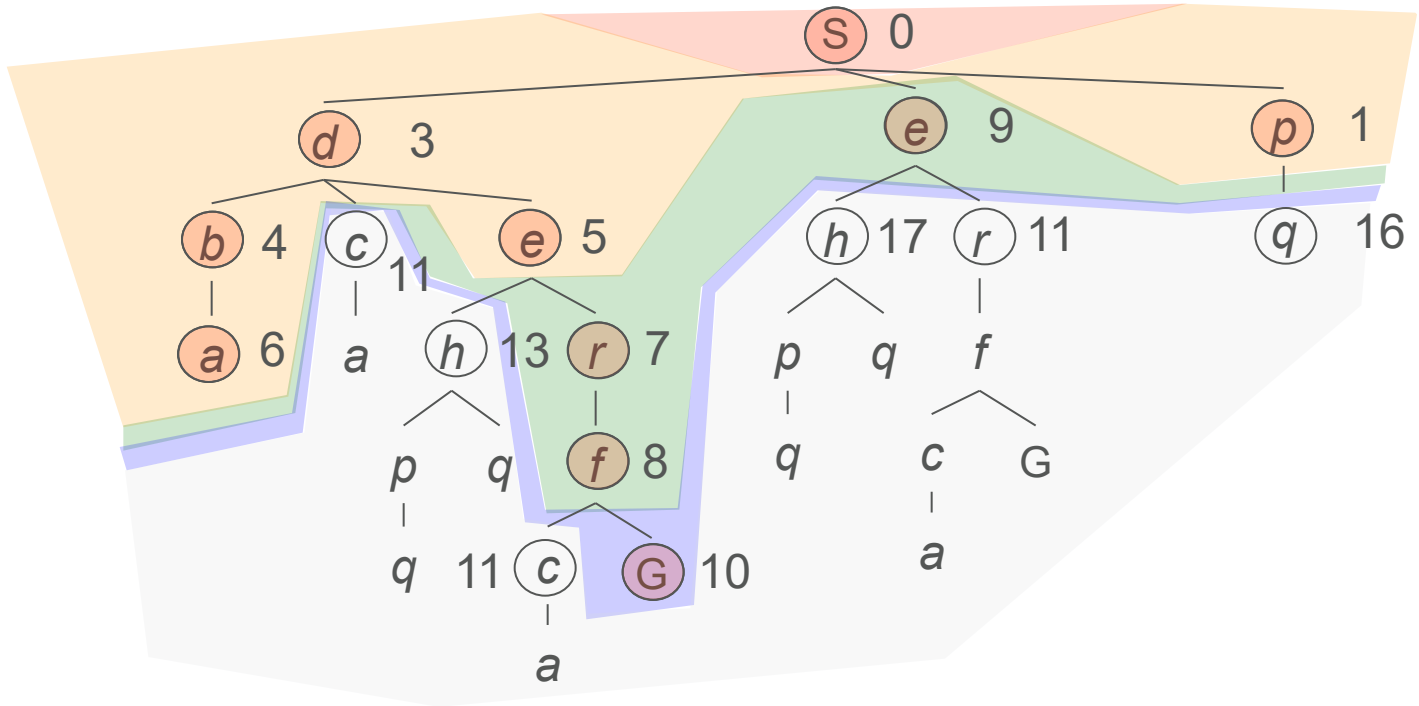
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

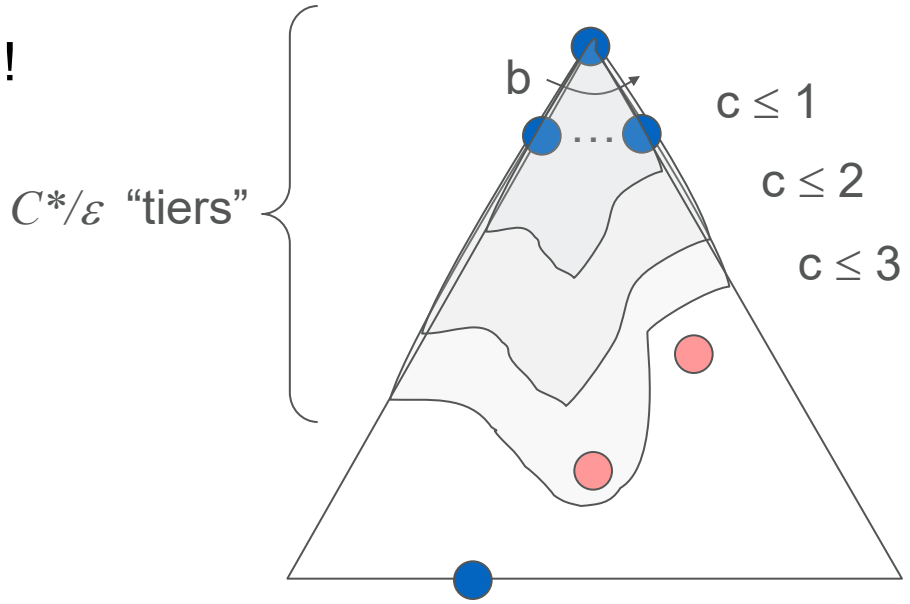


Cost contours



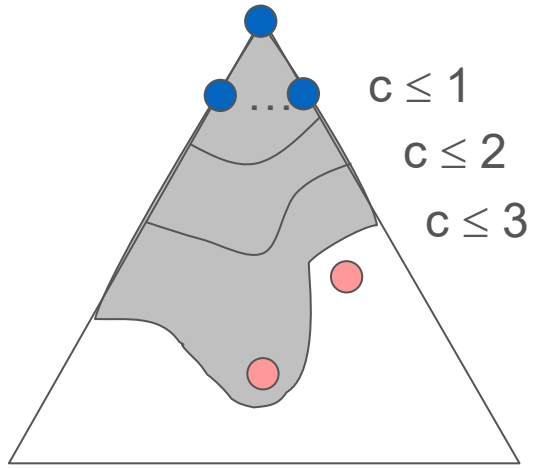
Uniform Cost Search (UCS) Properties

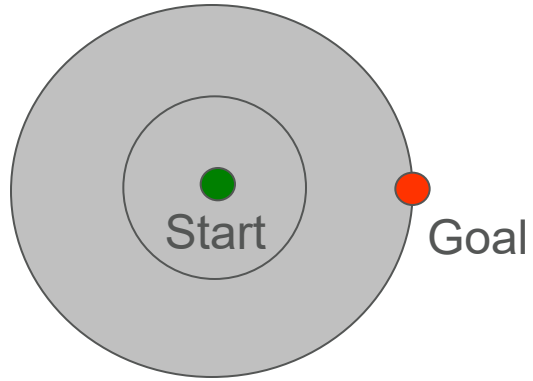
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If the solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?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



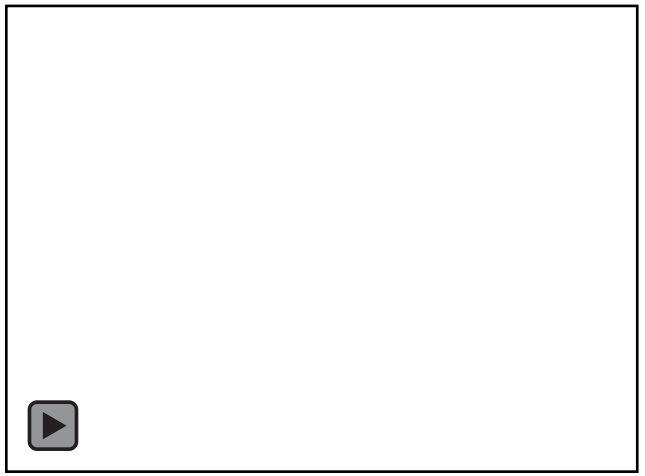
Uniform Cost Issues

- Remember: UCS explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location

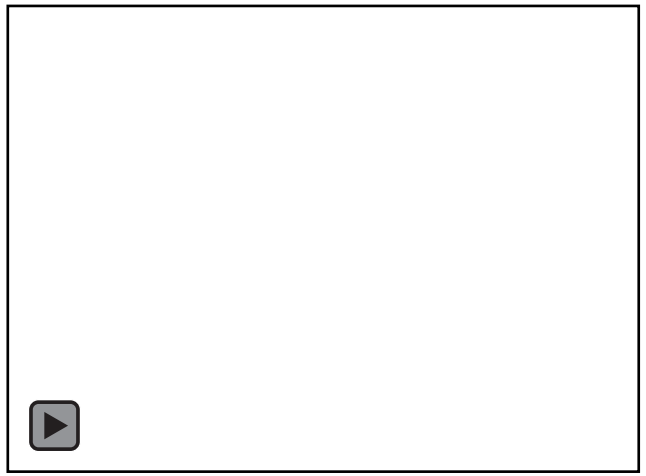




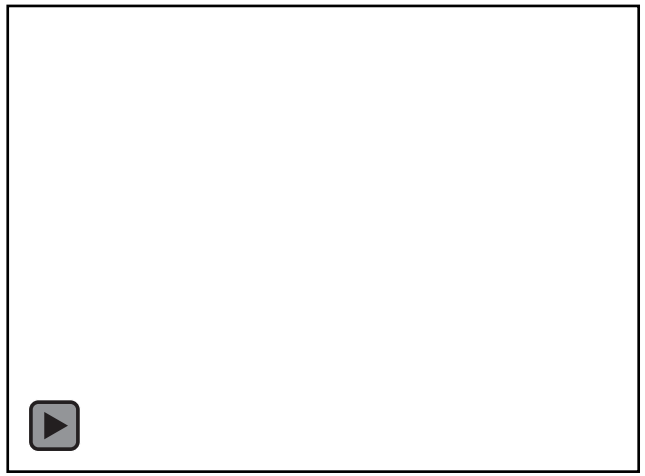
Video of Demo Maze with Deep/Shallow Water DFS, BFS, UCS



DFS

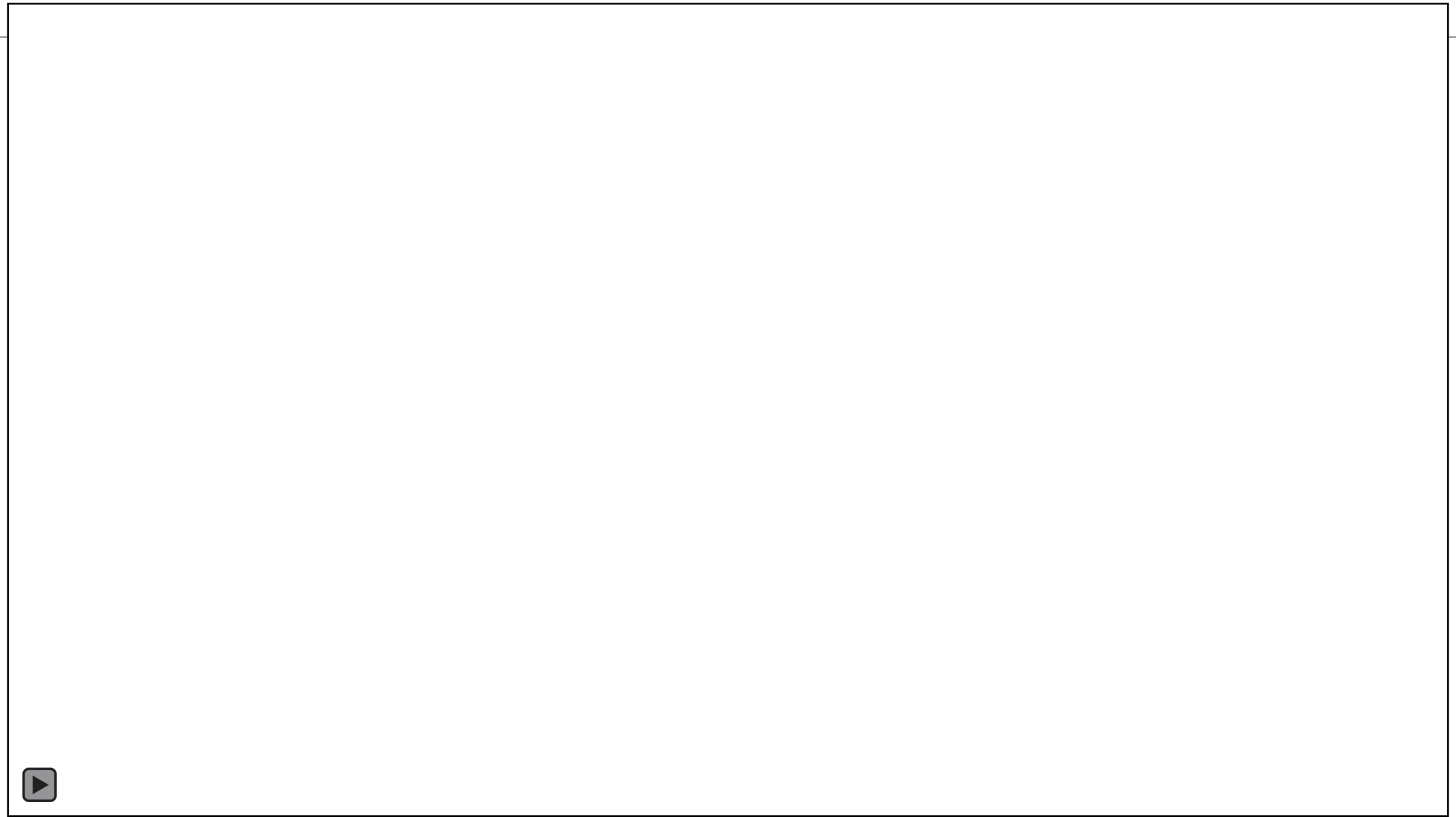


BFS

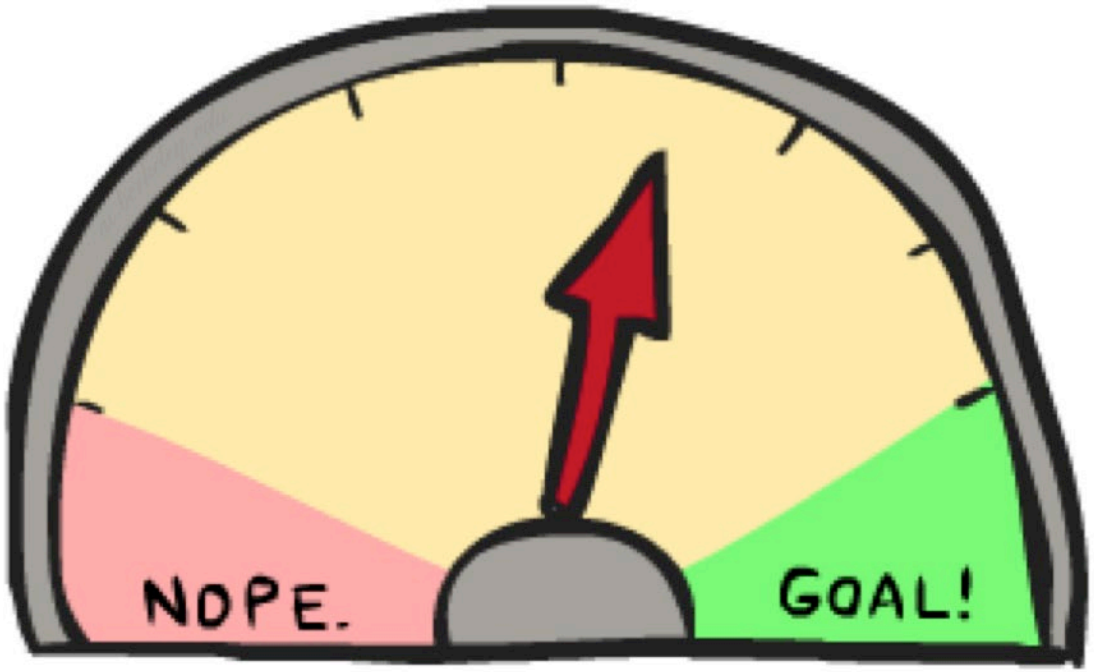


UCS

Video of Demo Contours UCS Pacman Small Maze

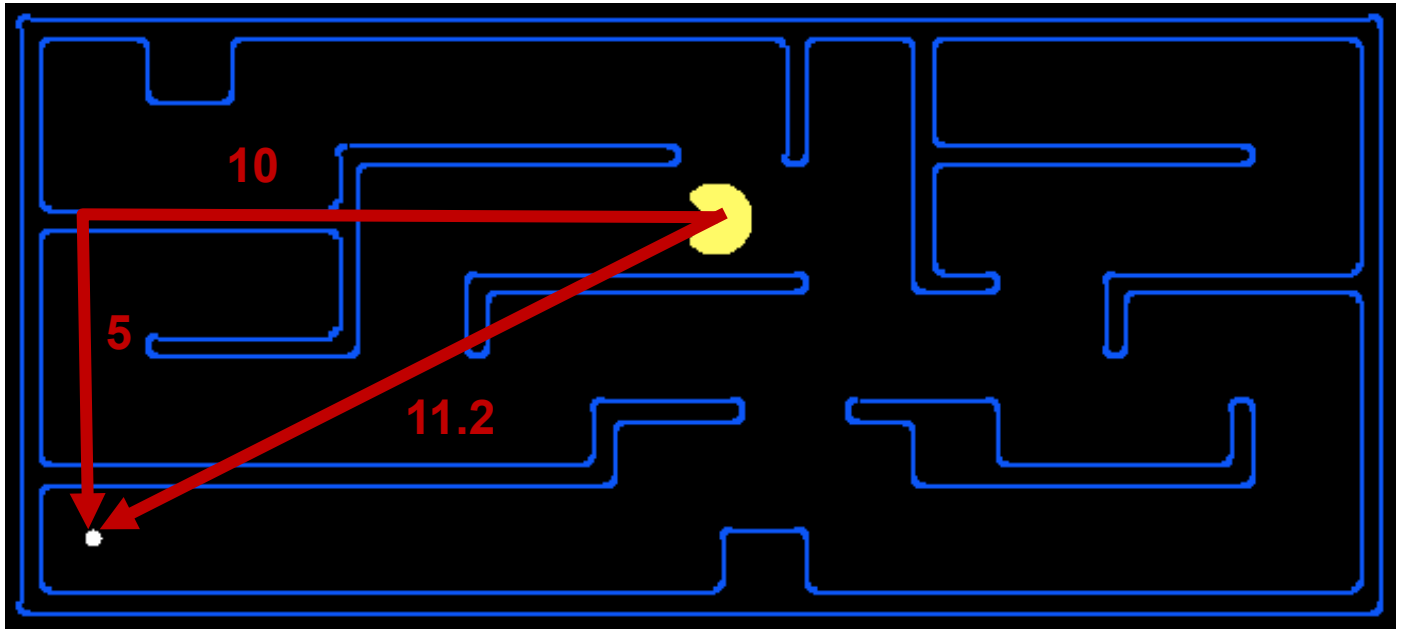


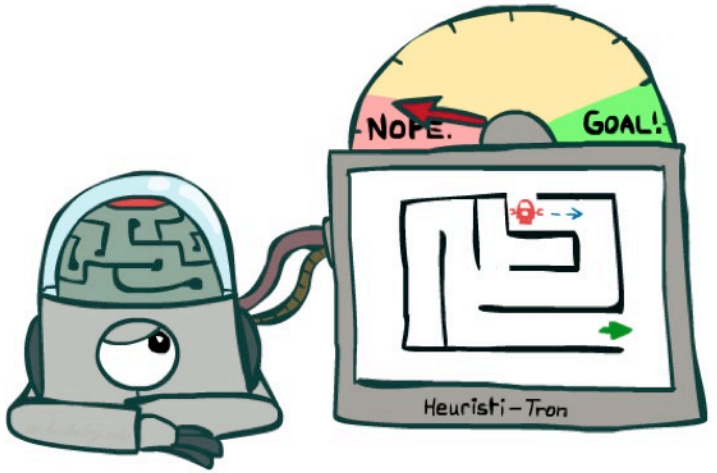
Informed Search

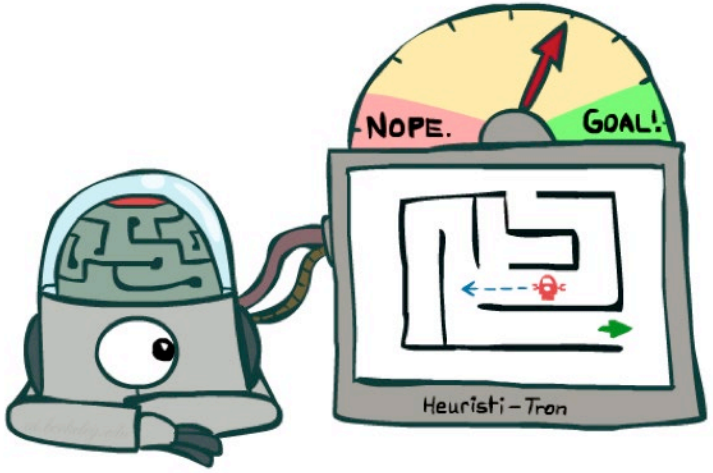


Search Heuristics

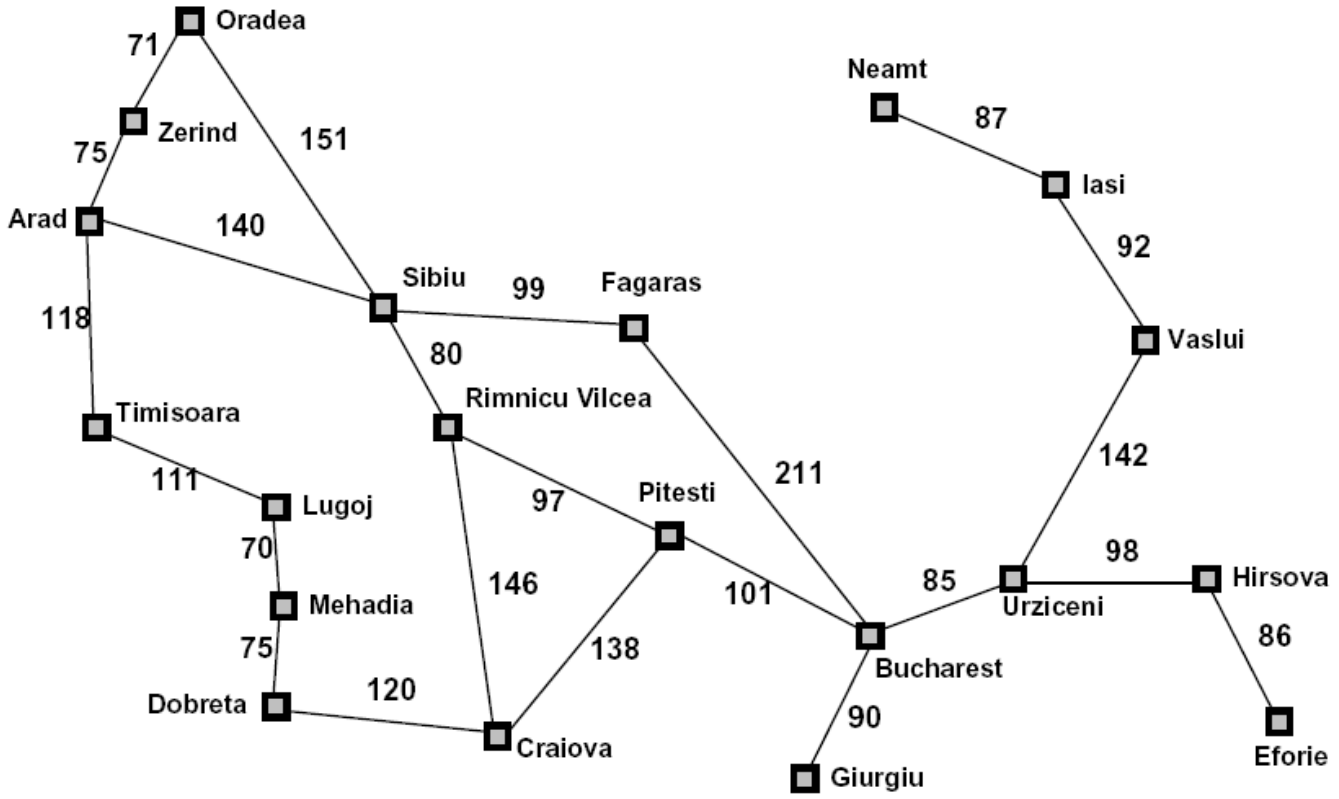
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing







Example: Heuristic Function

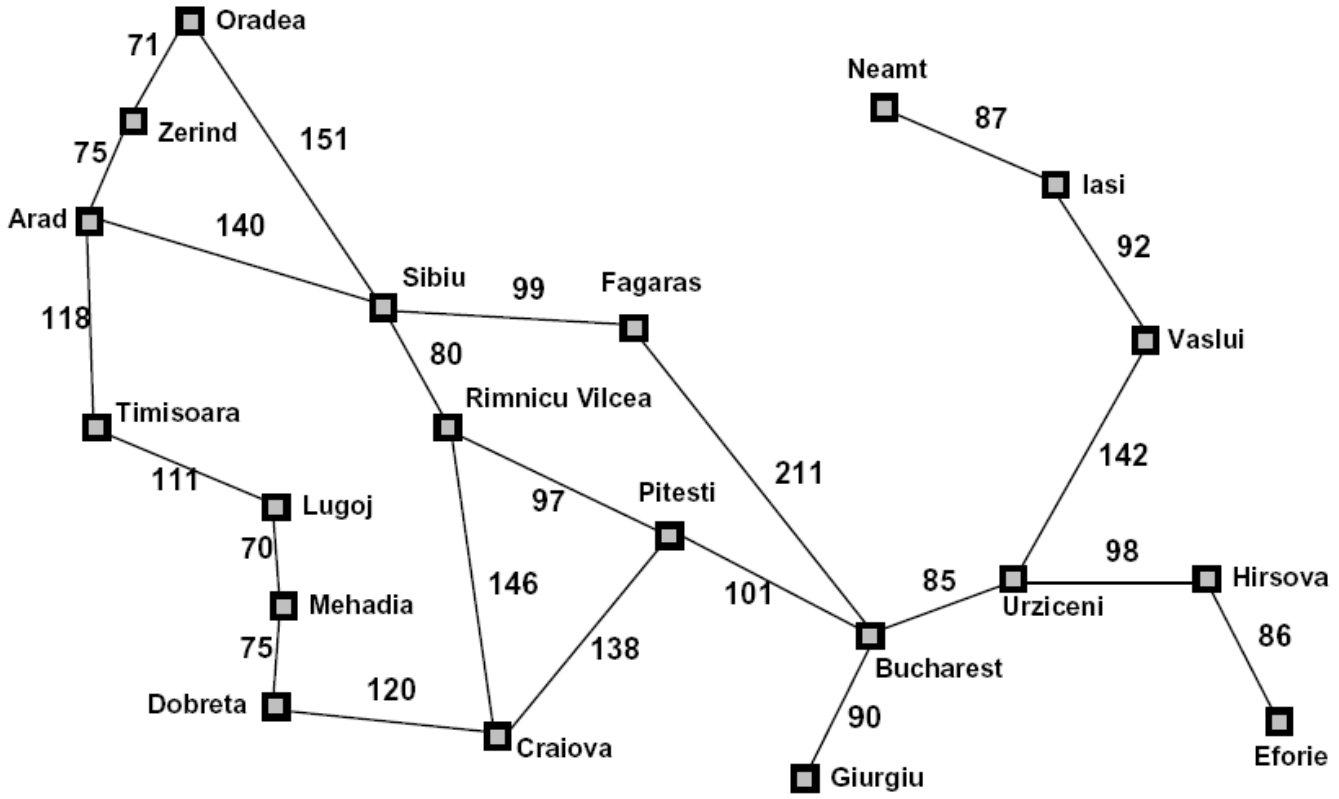


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Greedy Search

Example: Heuristic Function

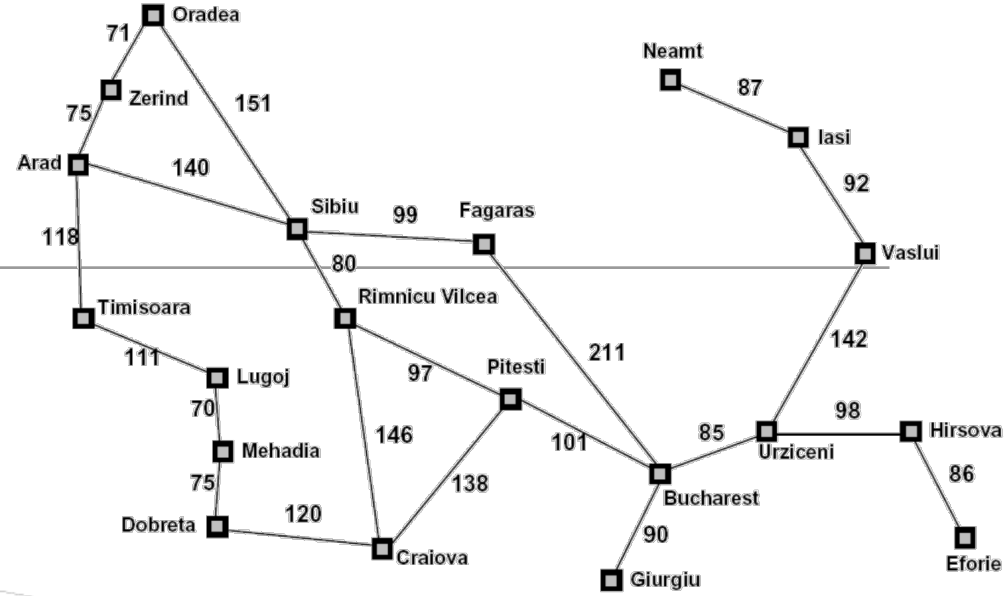


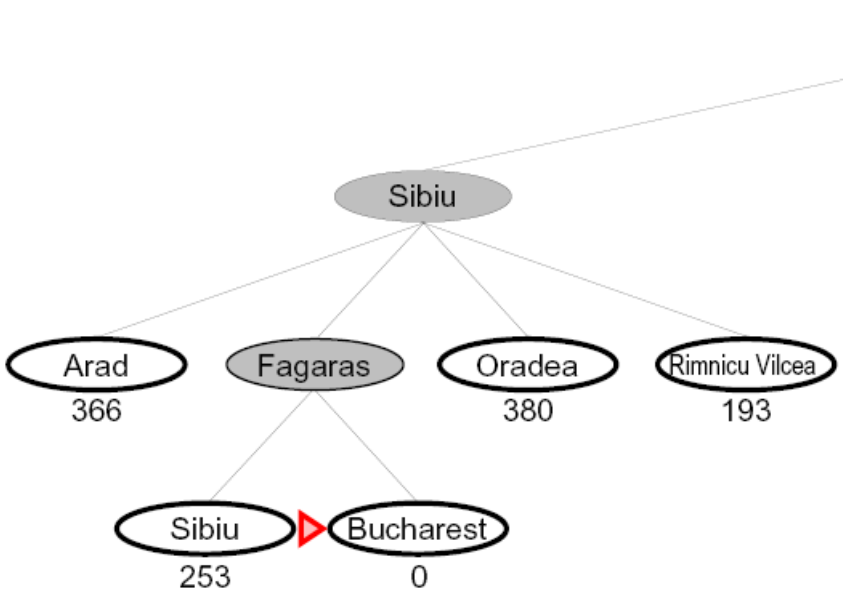
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

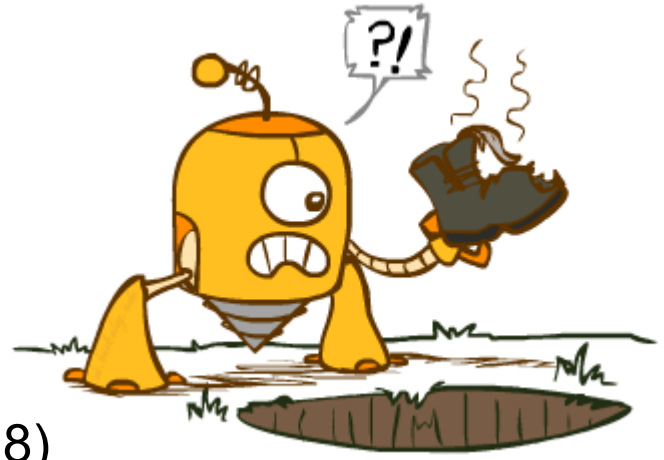
h(x)

Greedy Search

- Expand the node that seems closest…





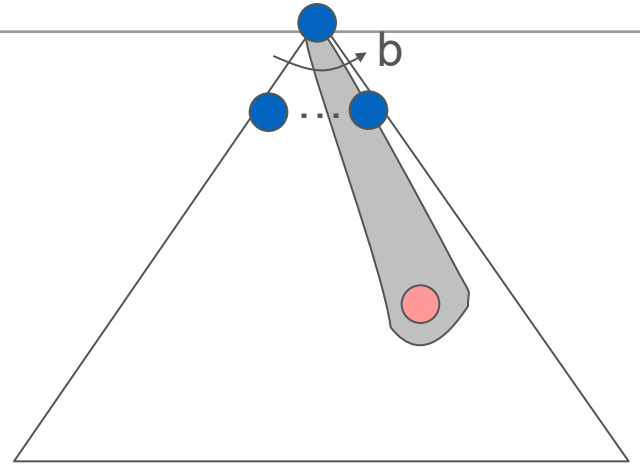


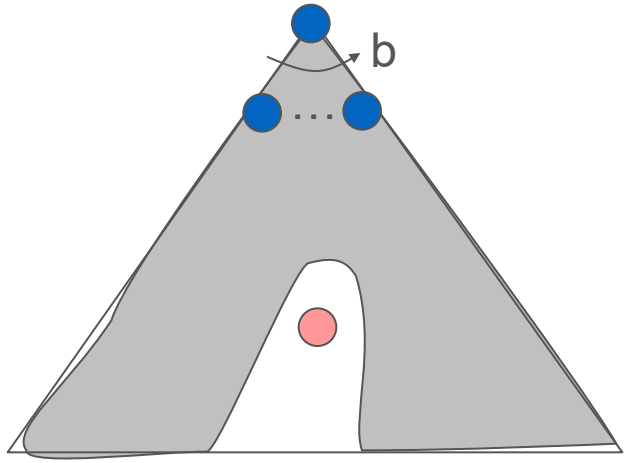
Found Path: Arad->Sibiu->**Faragas**->Bucharest (Dist: 450)

Optimal Path: Arad->Sibiu->**Rimnicu Vilcea->Pitesti-**>Bucharest (Dist:418)

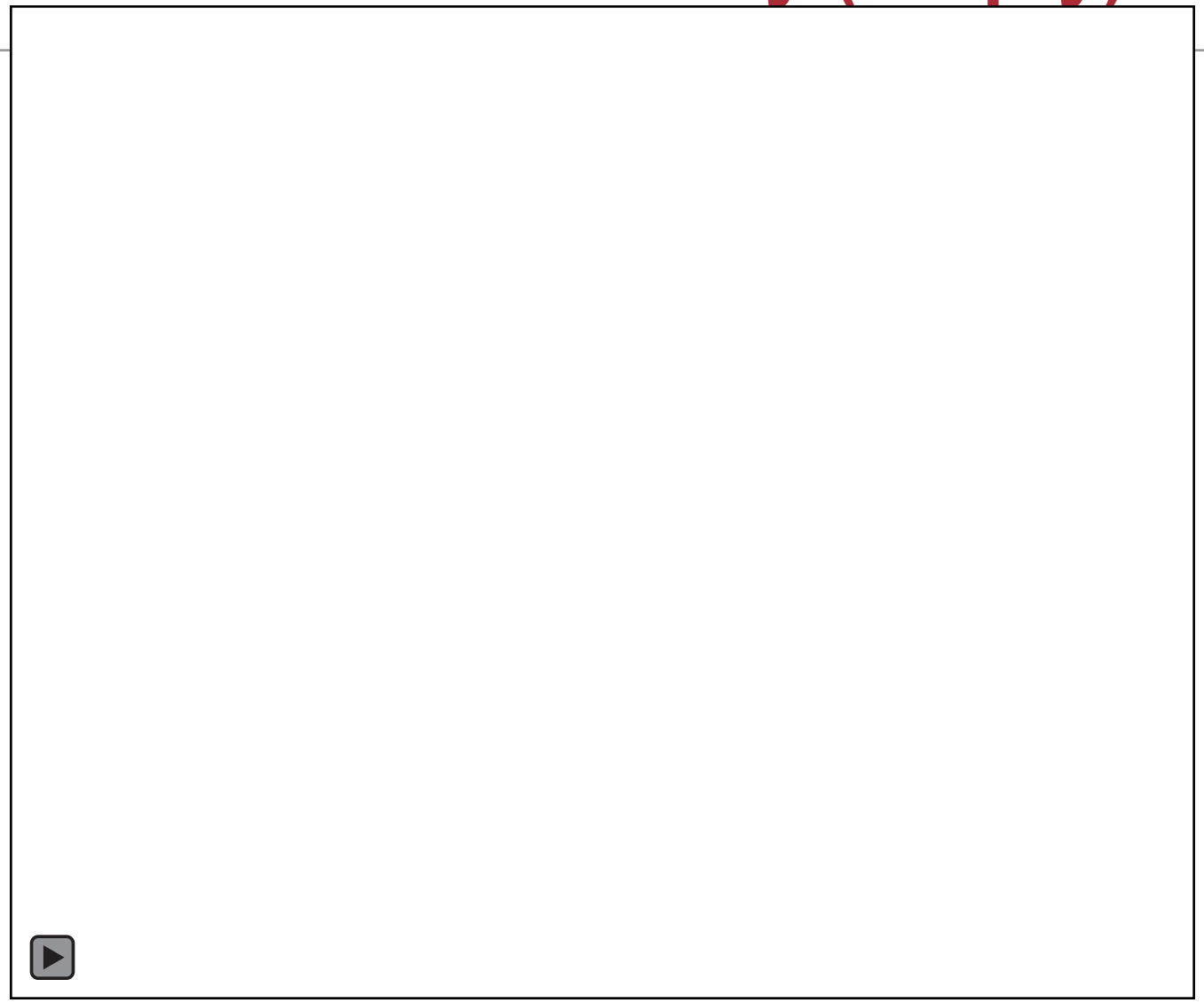
Greedy Search

- 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)



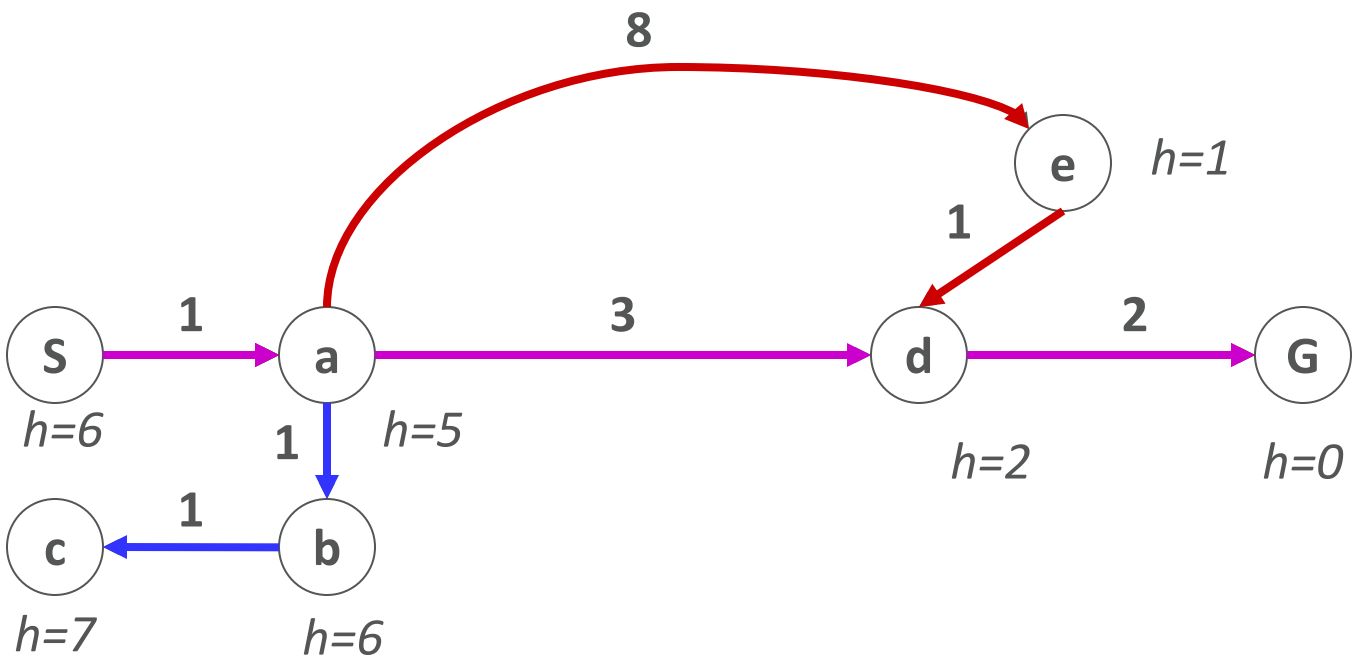
Video of Demo Contours Greedy (Pacman Small Maze)

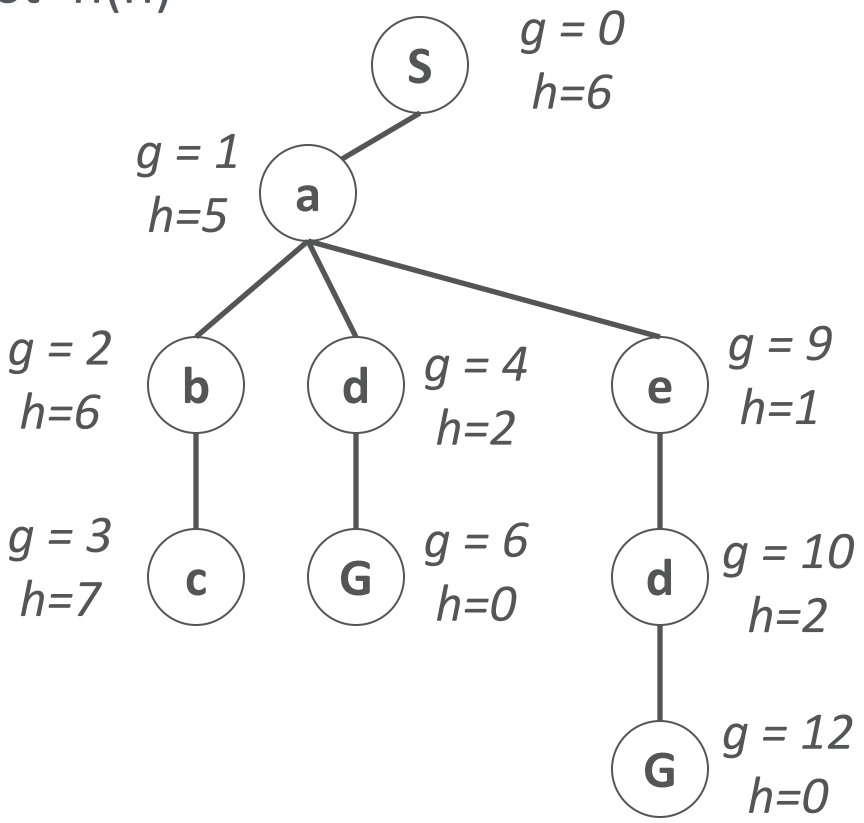
A* Search

A* Search

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)

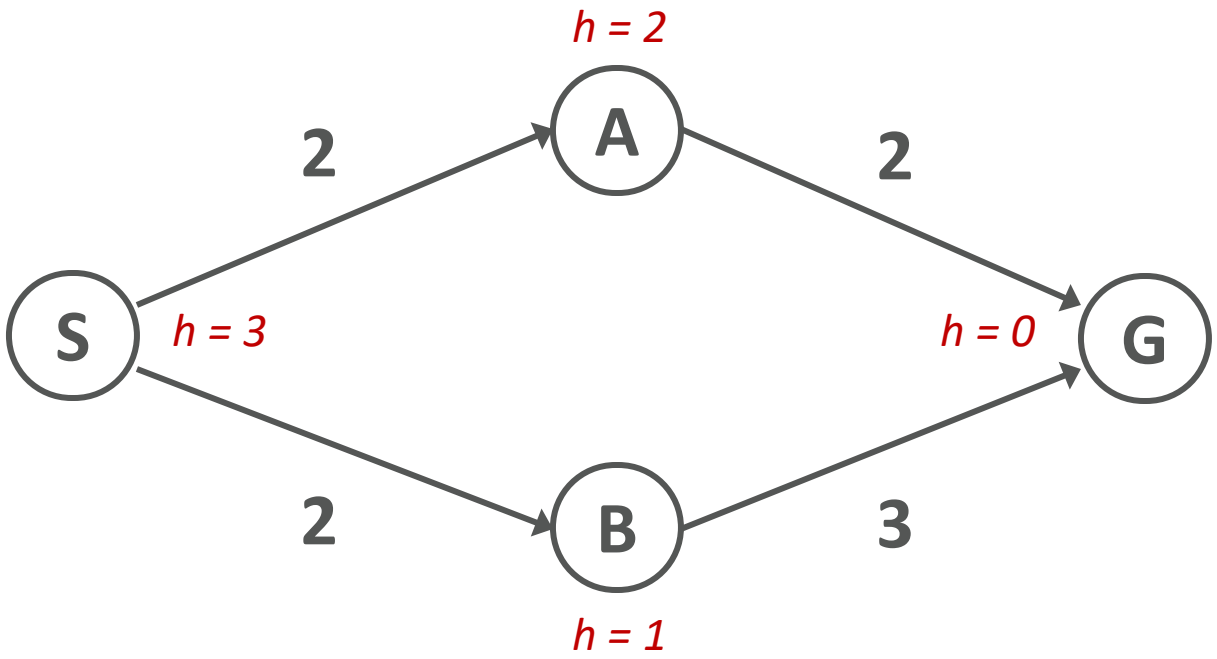




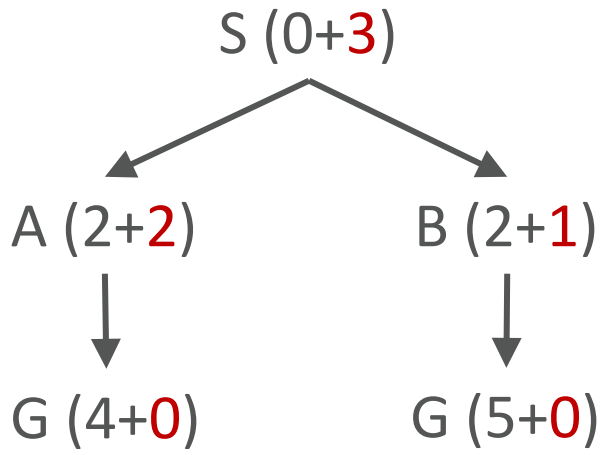
- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

When should A* terminate?



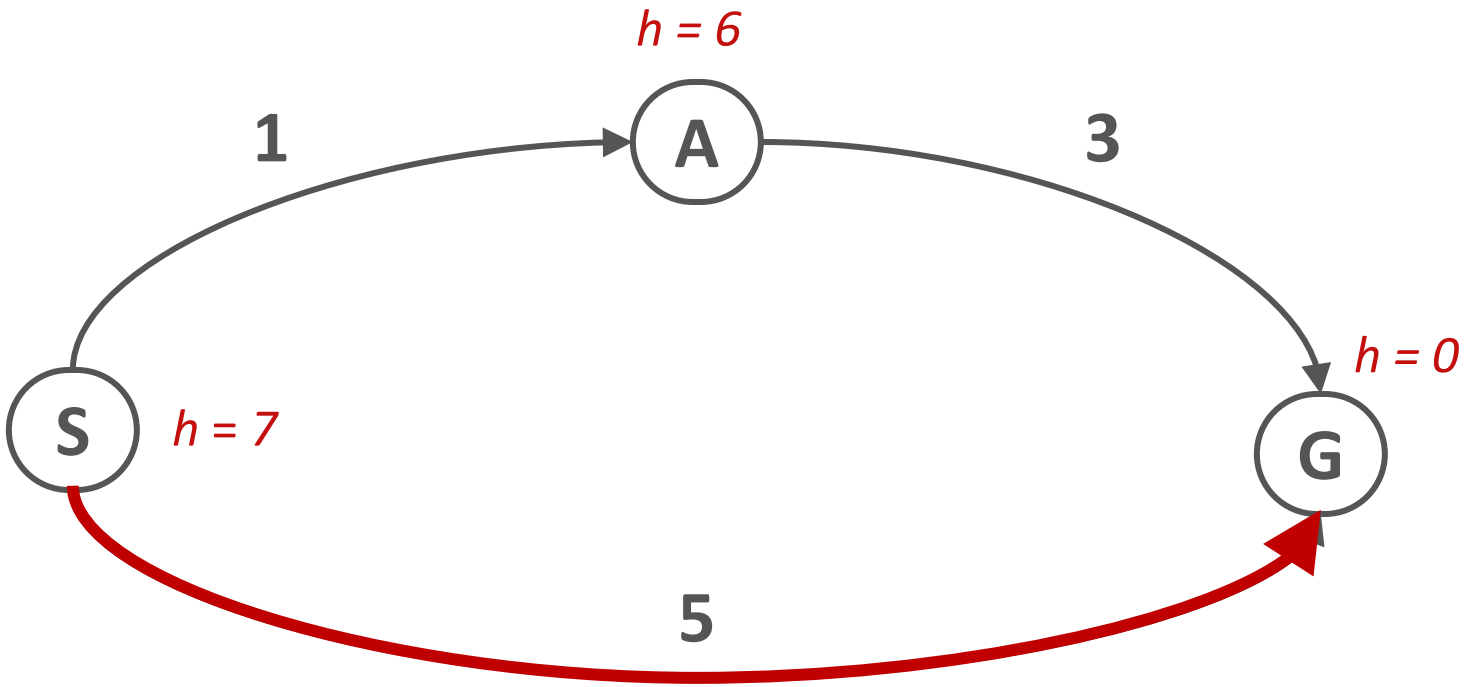
Search tree



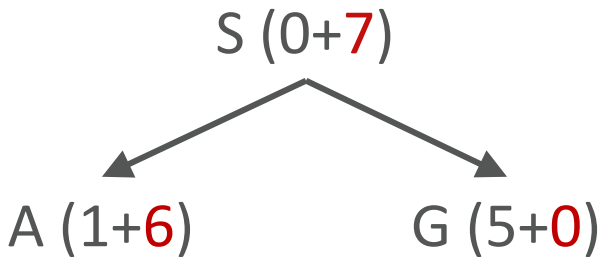
Shall we stop when we add a goal to the fringe?

No: we only stop when we deque the goal

Is A* Optimal?

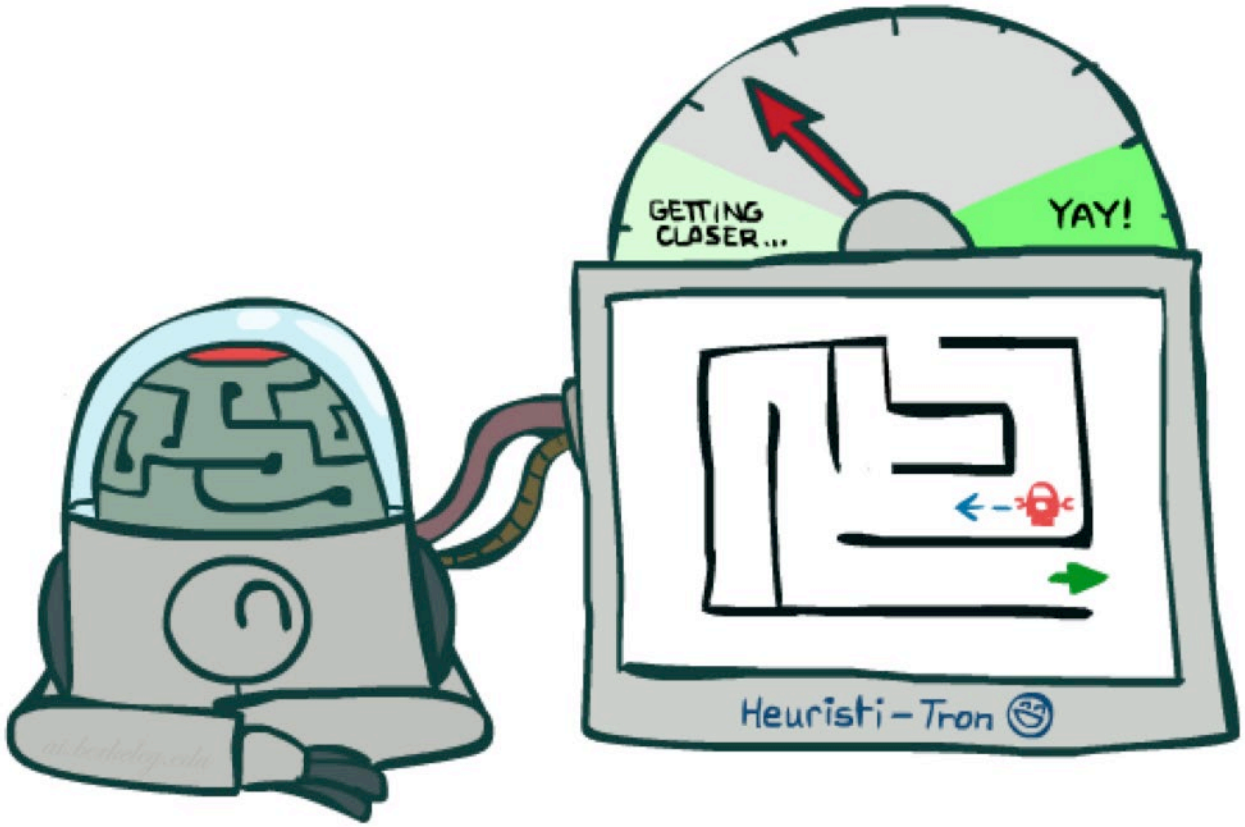


Search tree

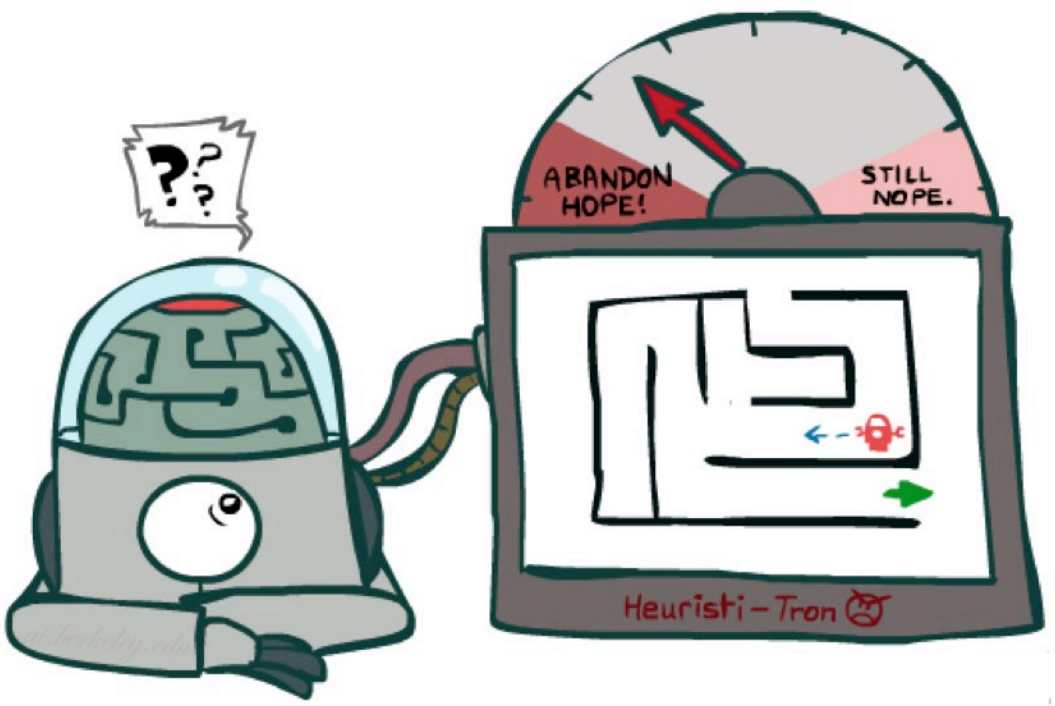


- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

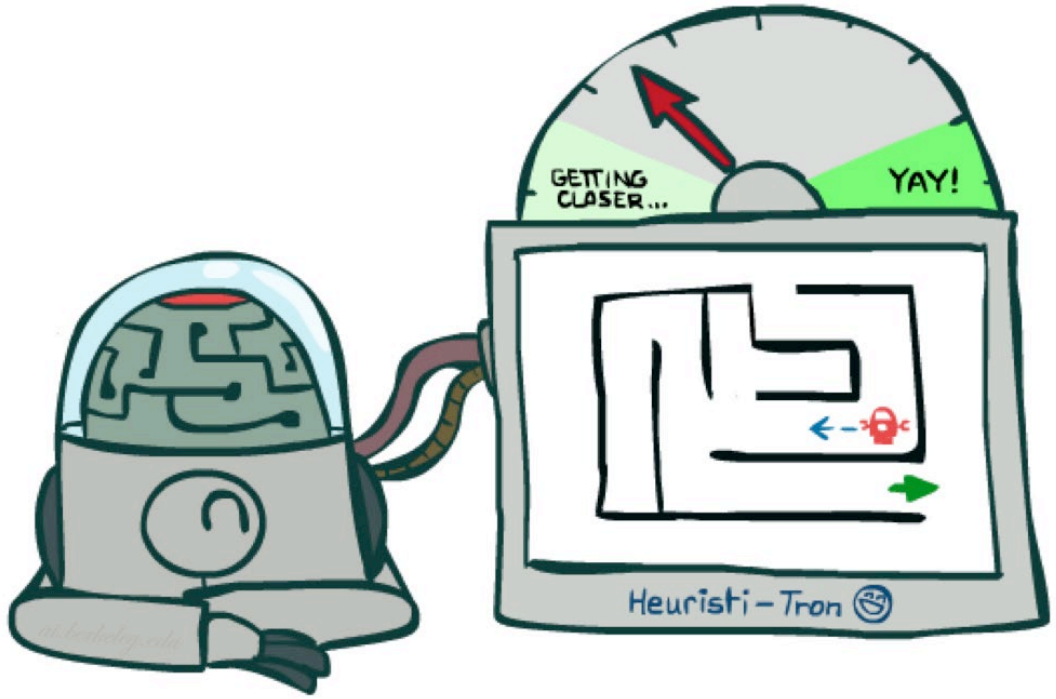
Admissible Heuristics



Idea: Admissibility



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



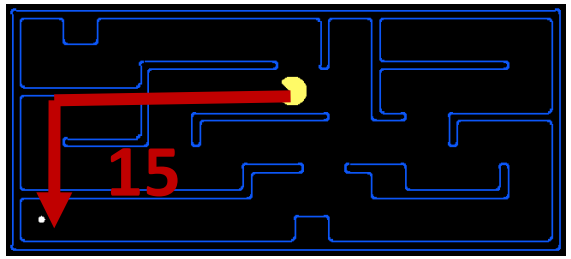
Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

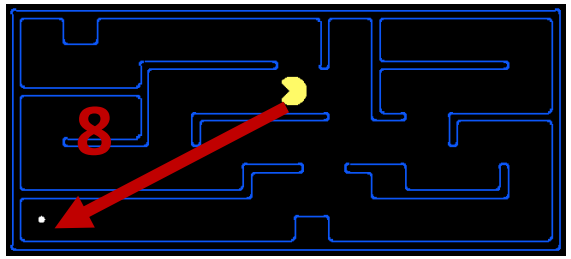
Admissible Heuristics

- 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:

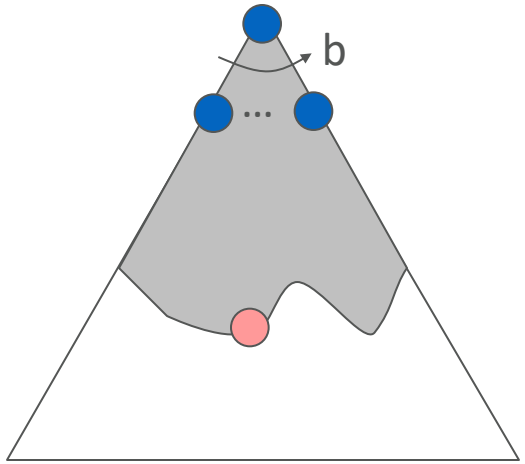




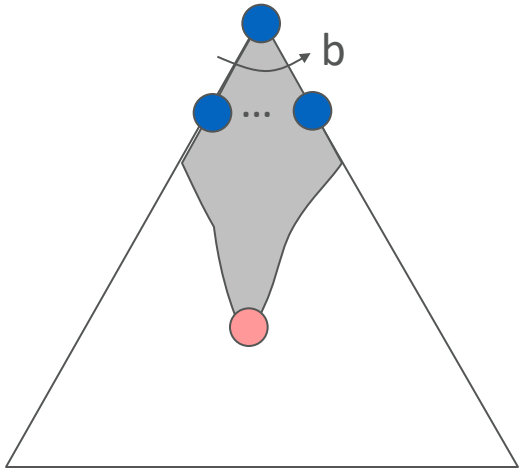
- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Properties of A*

Uniform-Cost

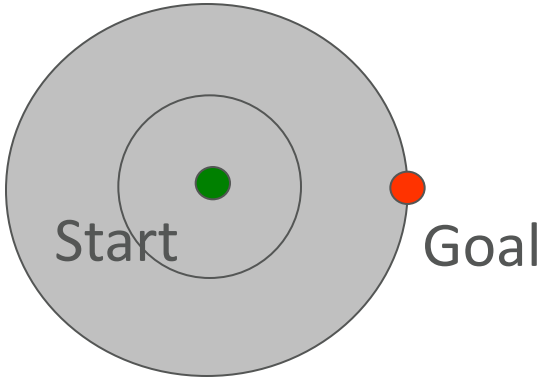


A*

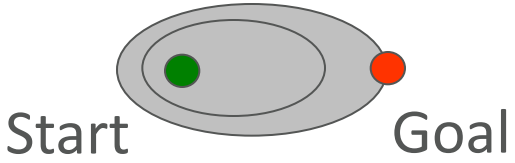


UCS vs A* Contours

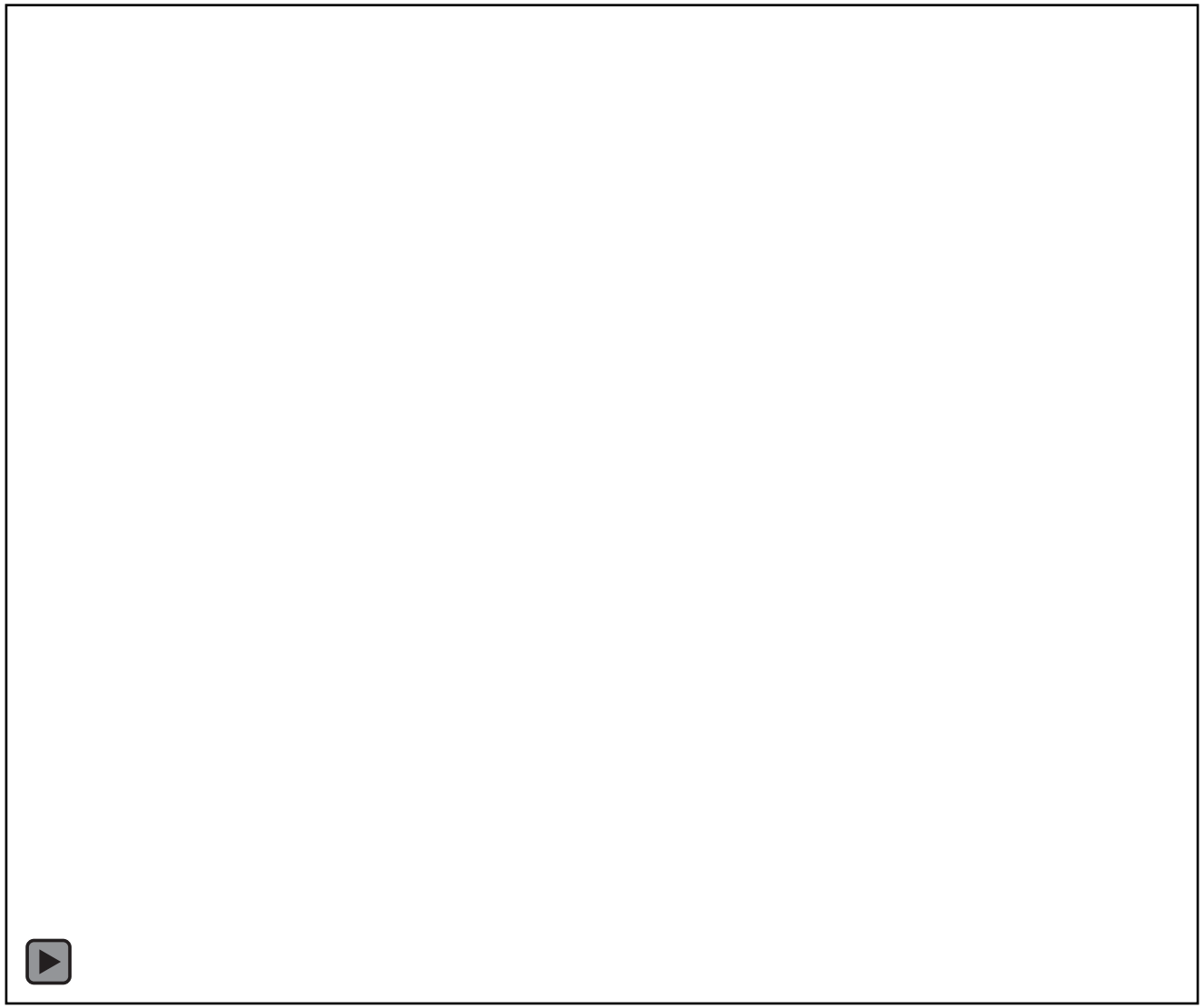
- Uniform-cost expands equally in all “directions”



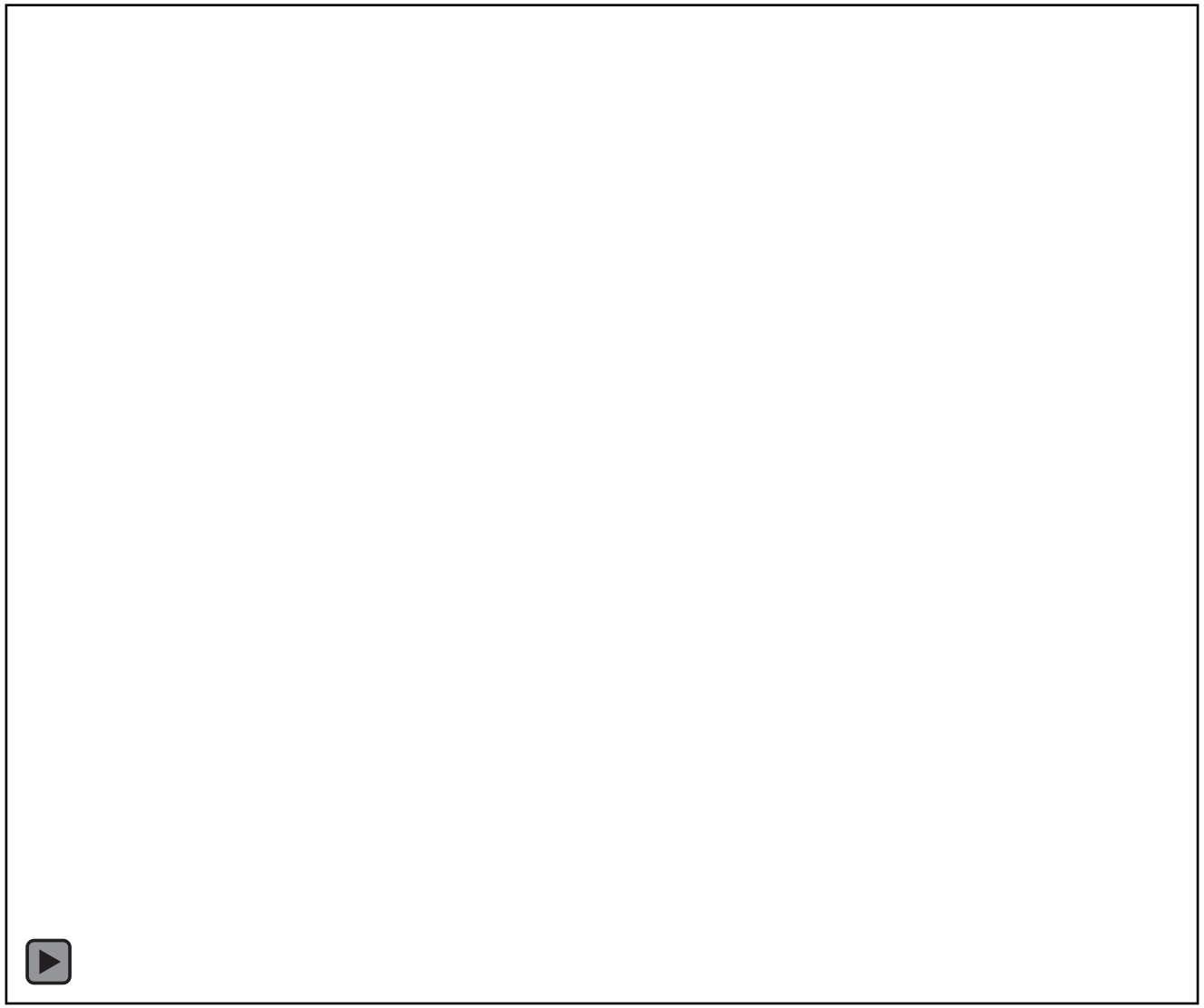
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



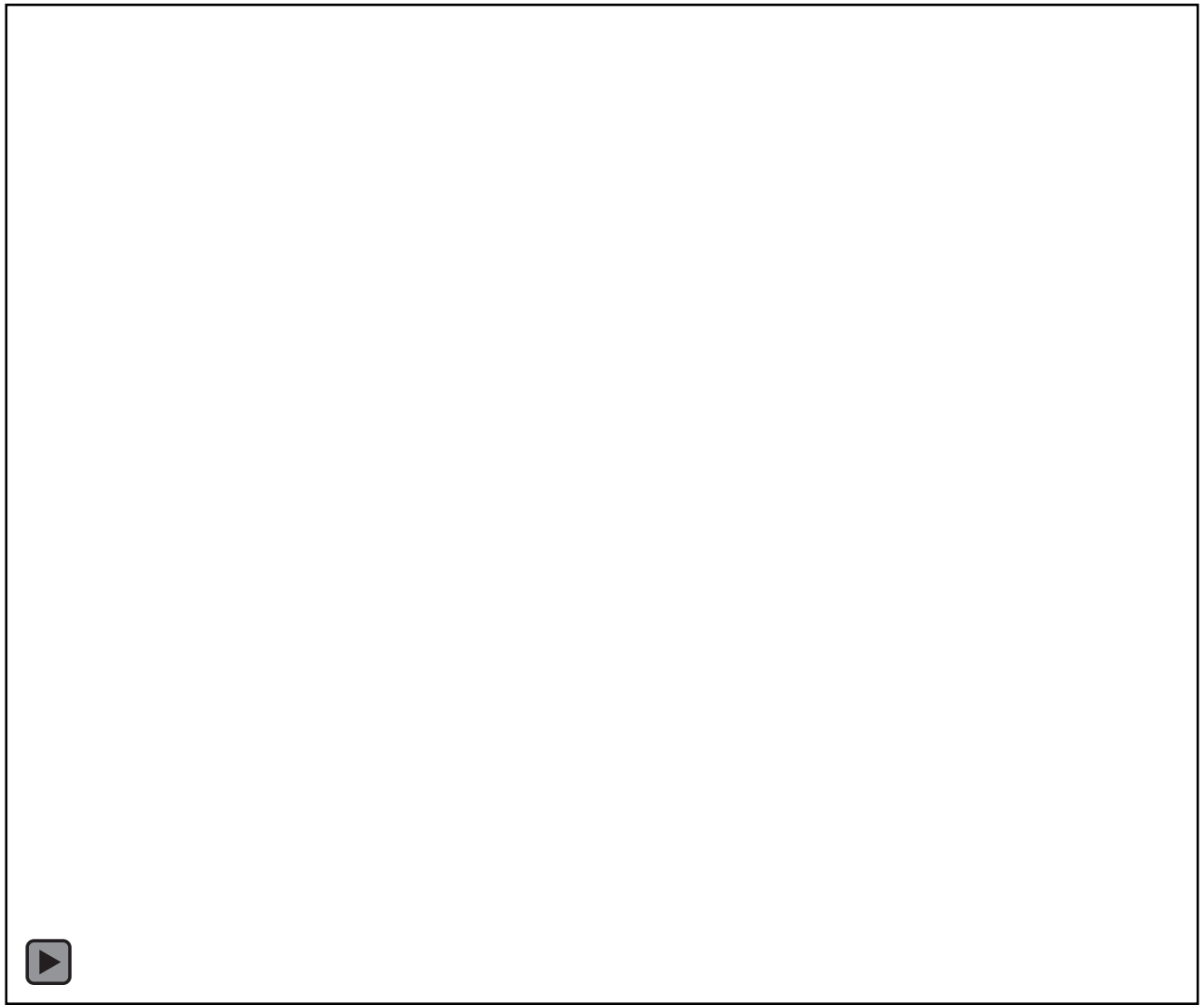
Video of Demo Contours (Empty) -- UCS



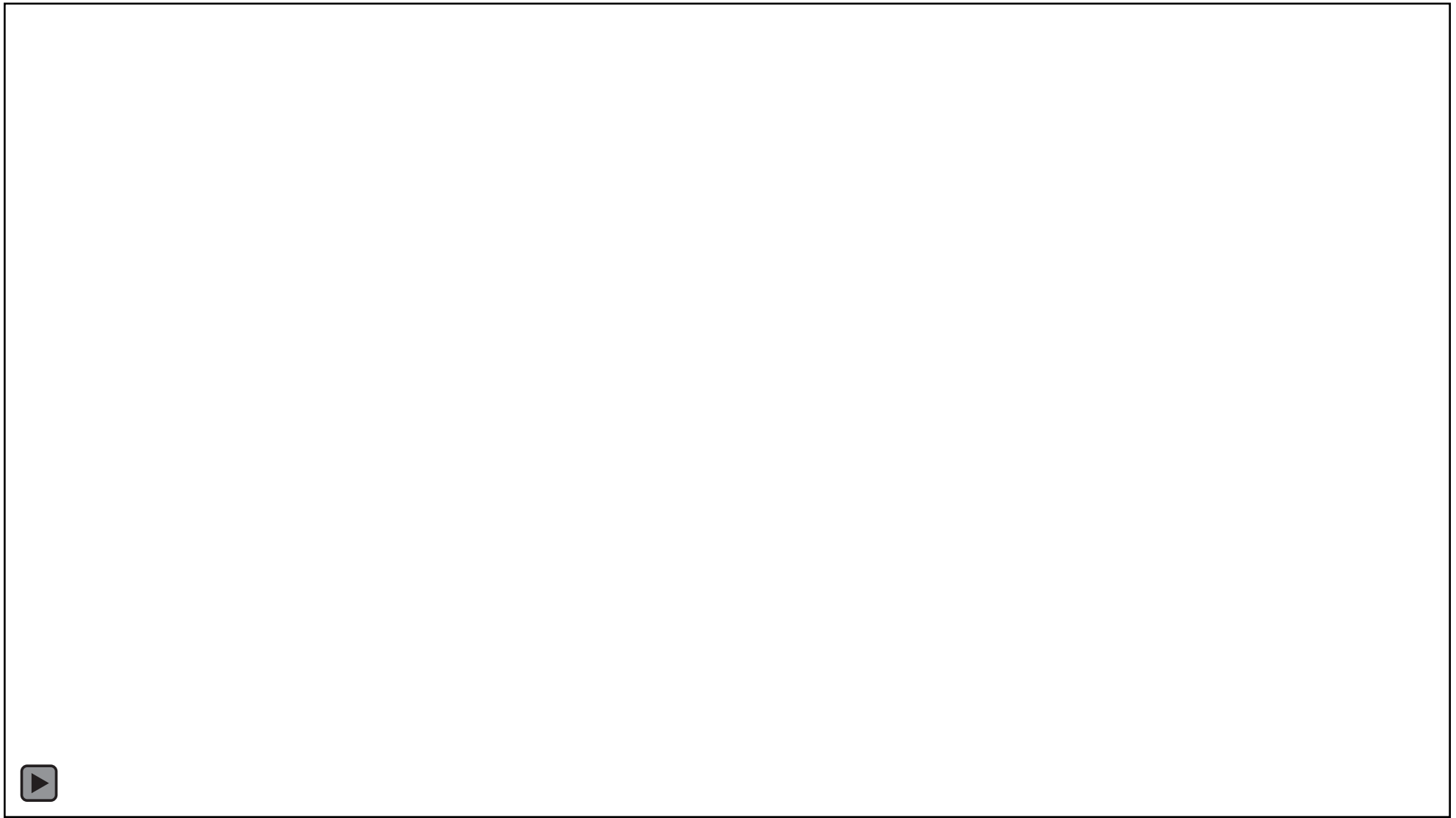
Video of Demo Contours (Empty) -- Greedy



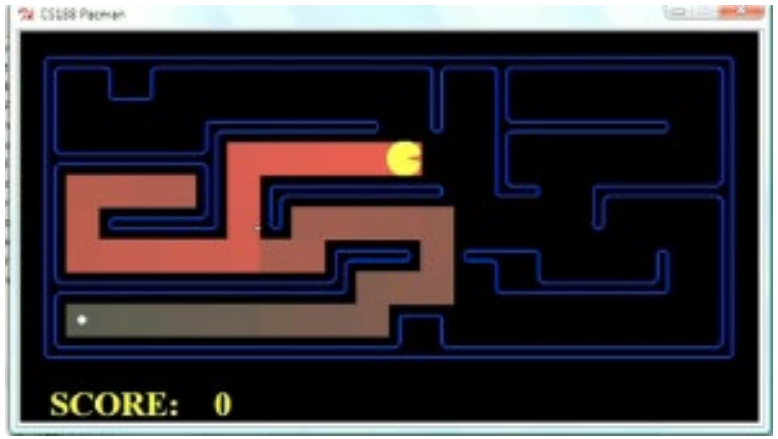
Video of Demo Contours (Empty) – A*



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

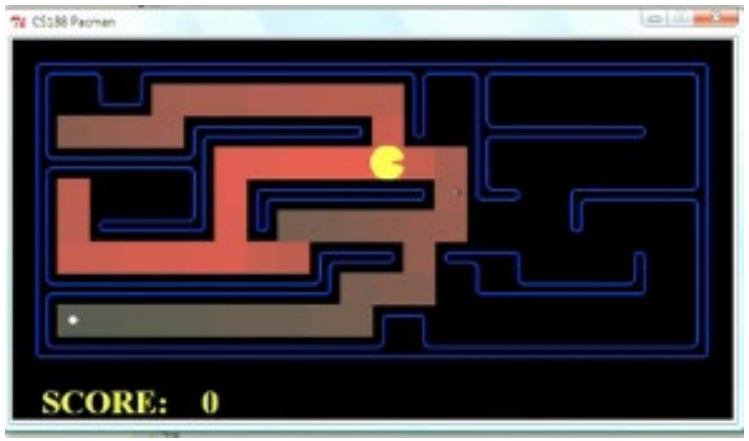


Comparison



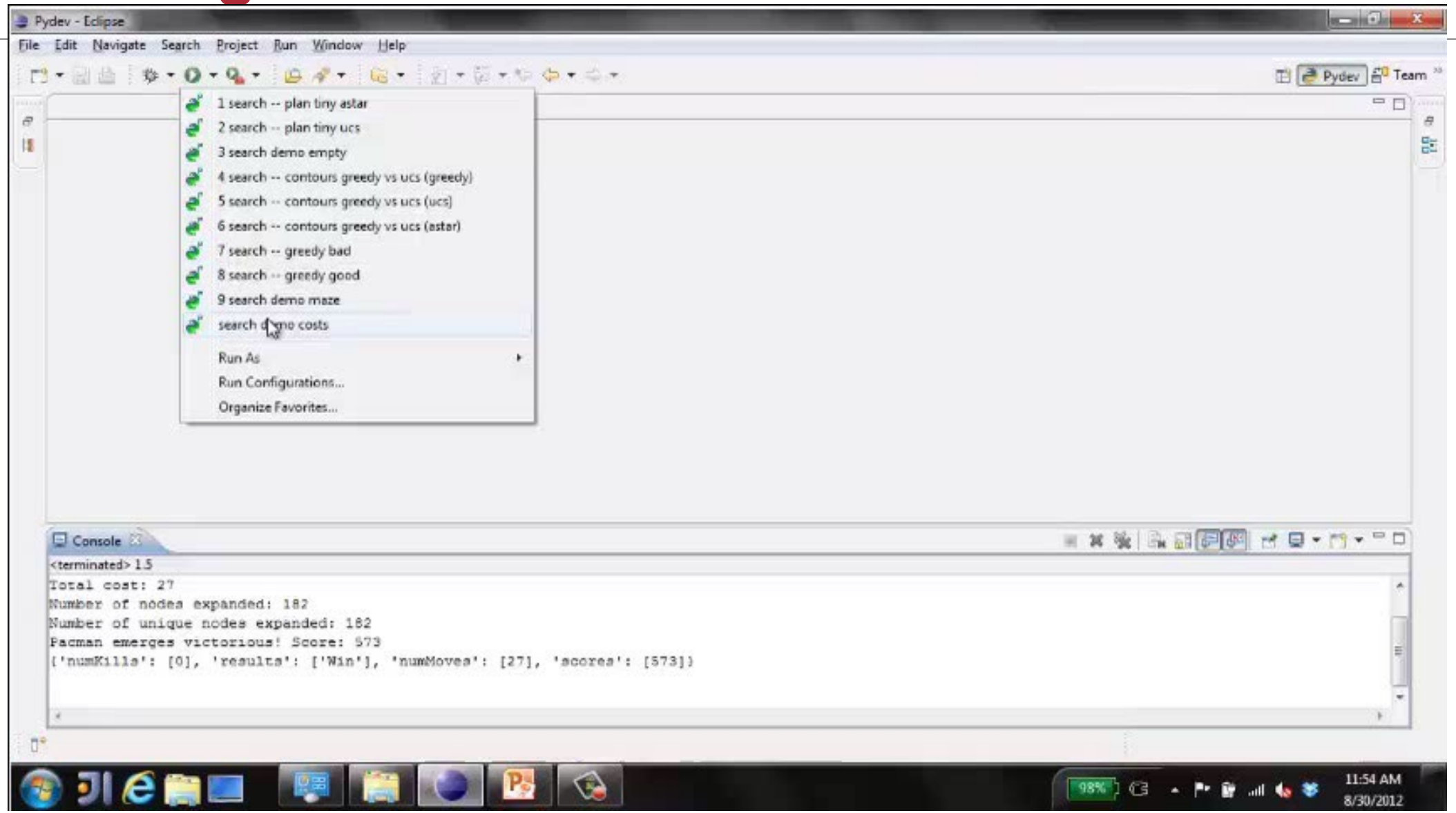
Greedy

Uniform Cost



A*

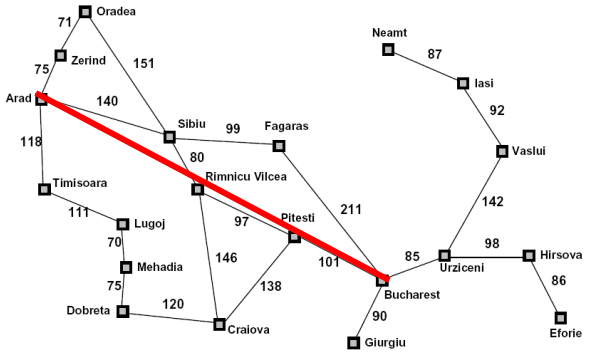
Video of Demo Empty Water Shallow/Deep – Guess Algorithm

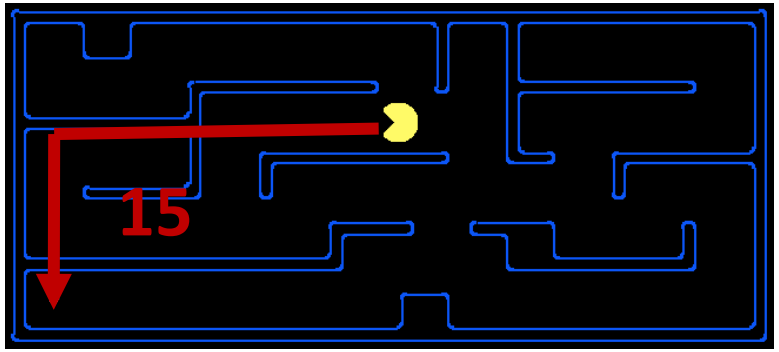


Creating Admissible Heuristics

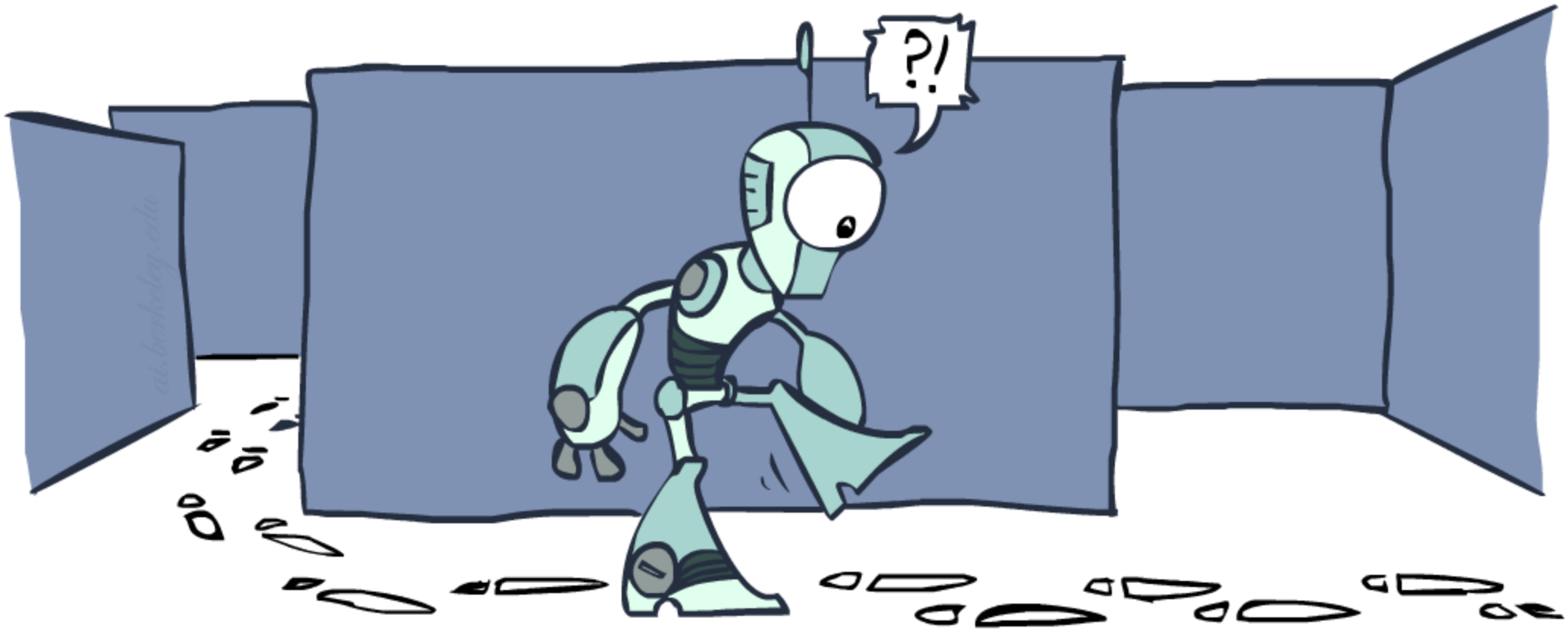
- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available

366



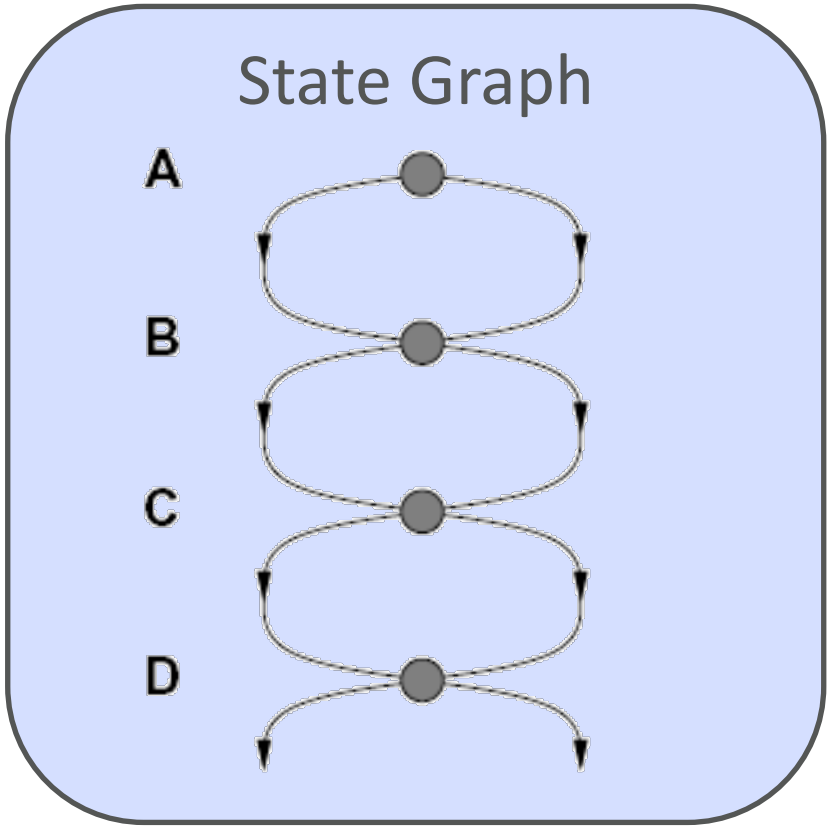


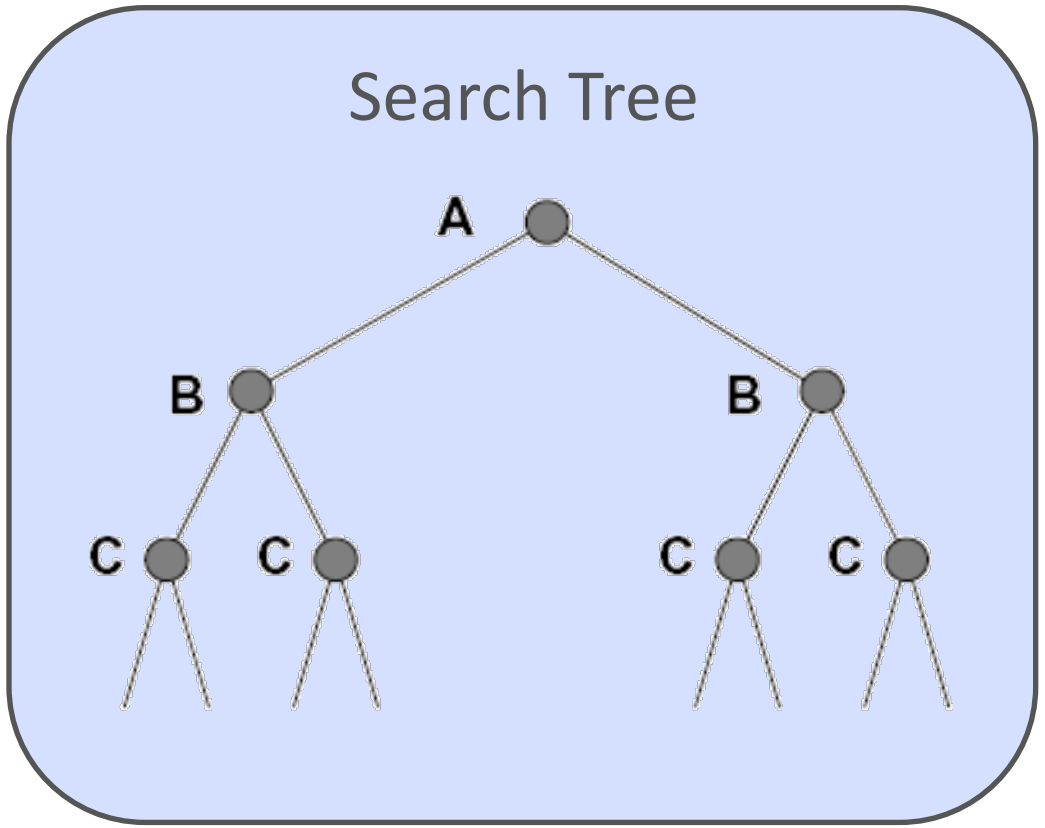
Graph Search



Tree Search: Extra Work!

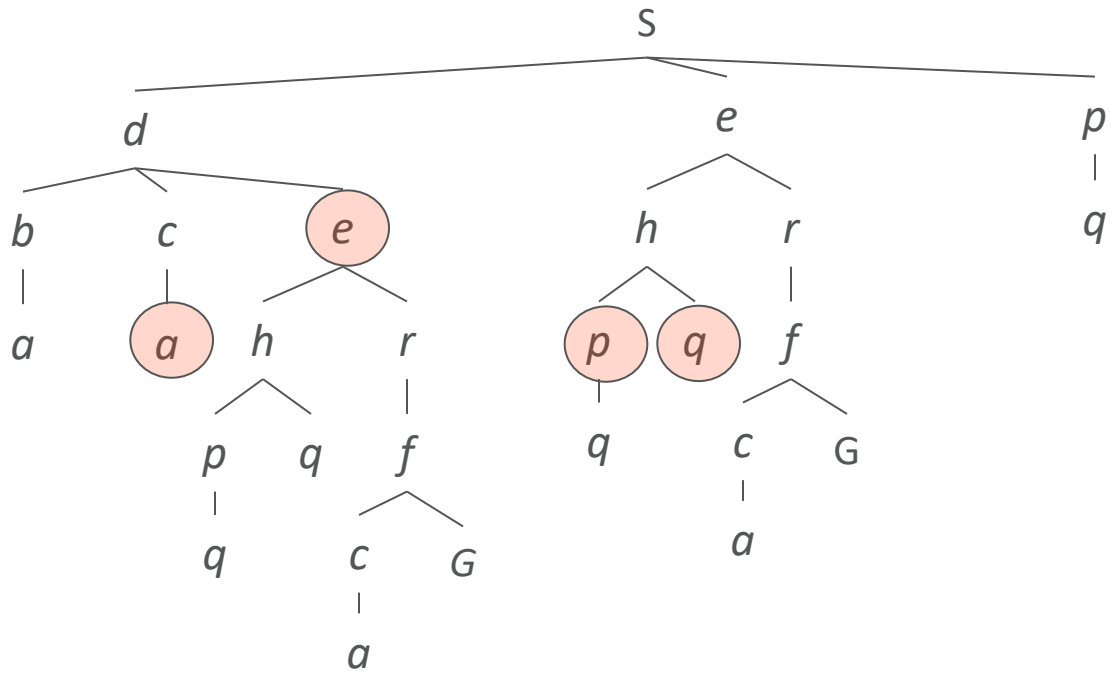
- Failure to detect repeated states can cause exponentially more work.





Graph Search

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

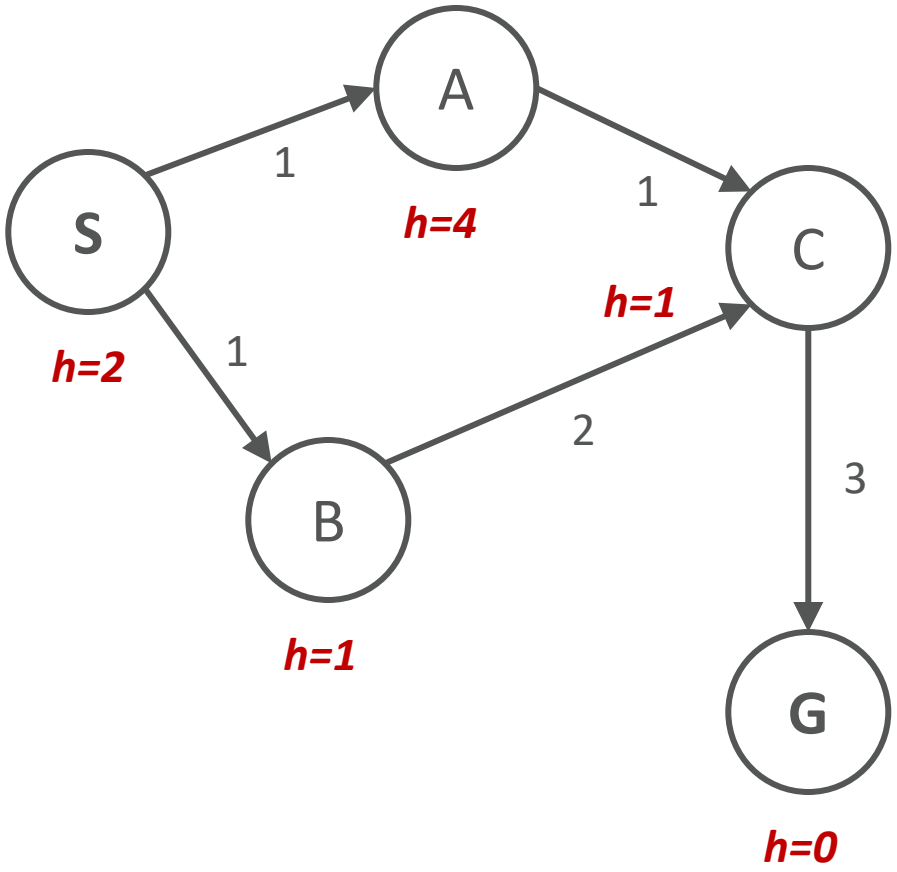


Graph Search

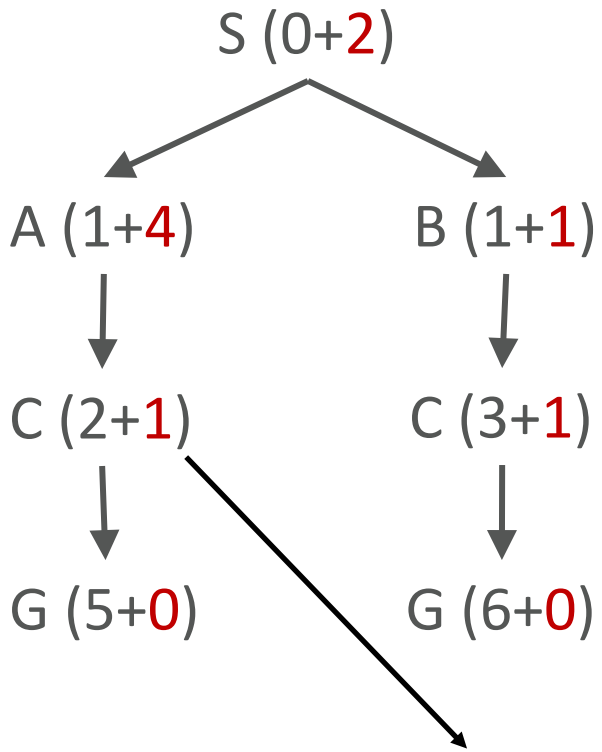
- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: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

A* Graph Search Gone Wrong?

State space graph

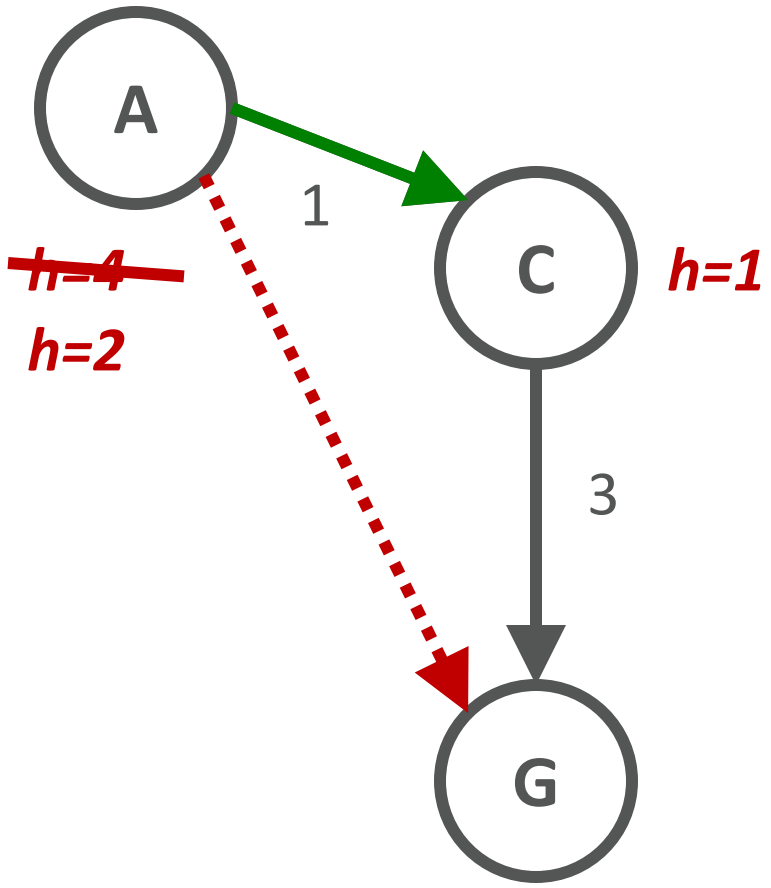


Search tree



A* would go stop here since we have already visited node C

Consistency of Heuristics



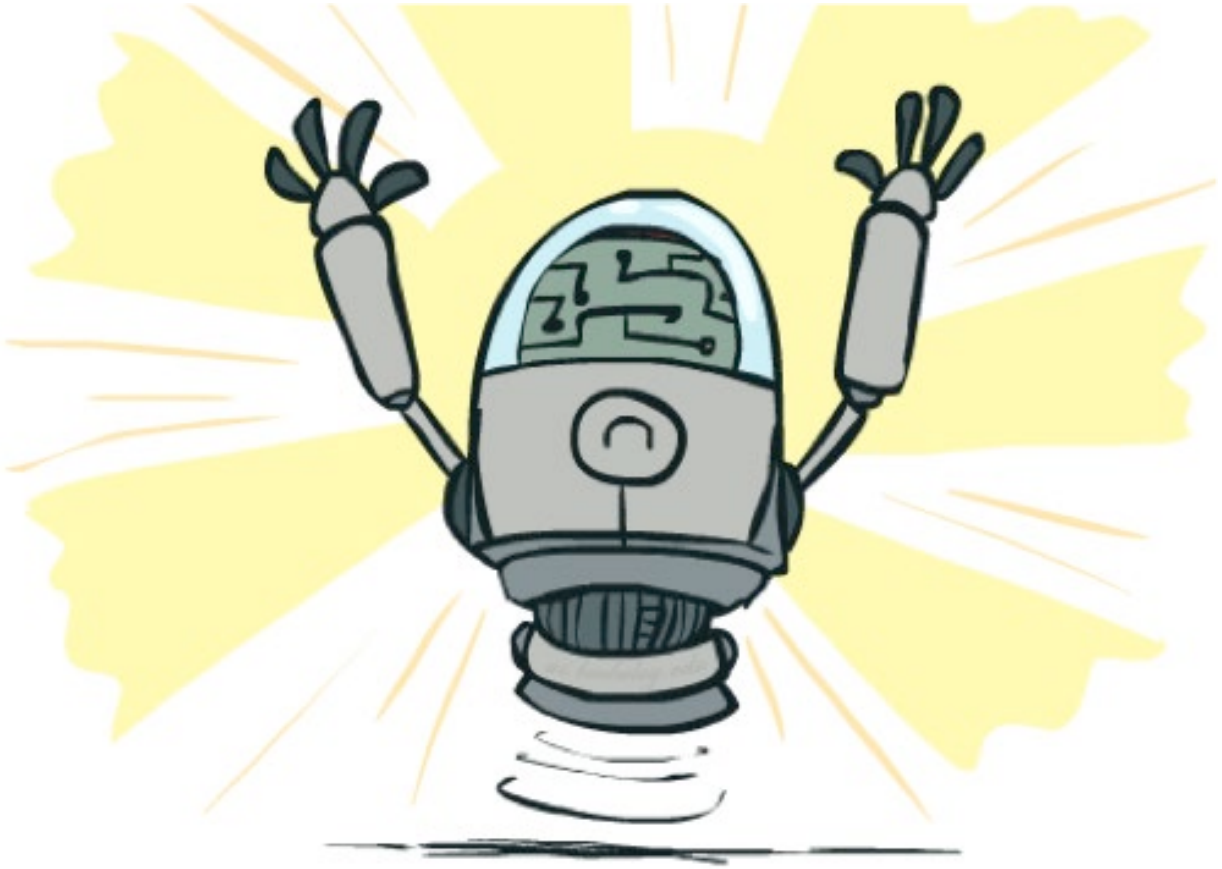
- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    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



Optimality of A* Graph Search with consistent heuristic

To reach node D from a sub-optimal path (B->D):

The c(R->B->D)+h(D) < c(R->C) + h(C) [We will prove this false]

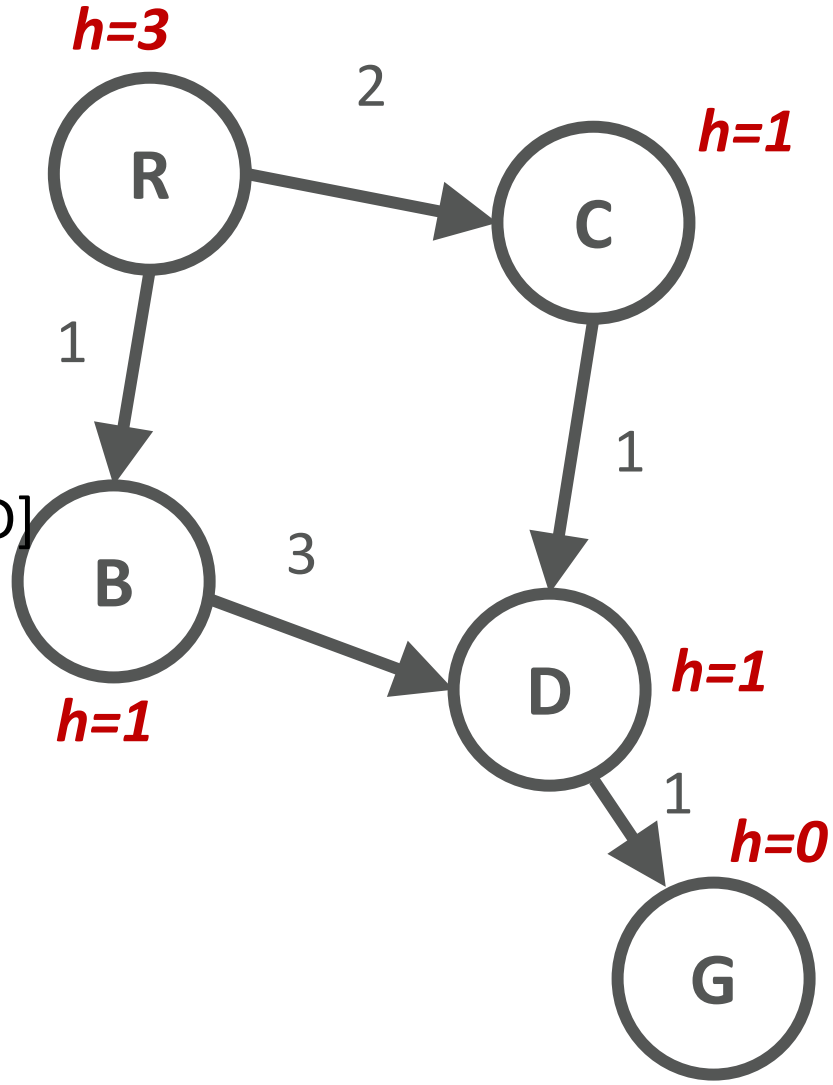
We know:

1. c(R->B->D) > c(R->C->D)= c(R->C)+c(C->D) [Optimal path C->D]

2. c(C, D)=> h(C) – h(D) [Consistent heuristic]

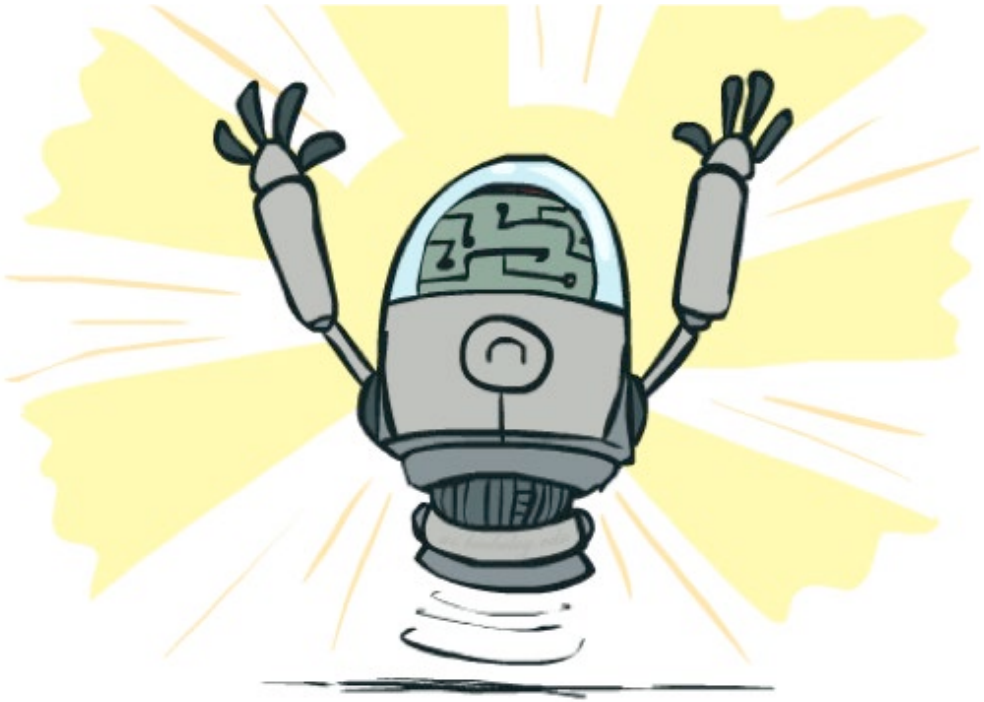
(1),(2) --> cost(R->B->D) >= cost(R->C)+ h(C) – h(D)

--> cost(R->B->D) +h(D) > cost(R->C)+ h(C)



Optimality

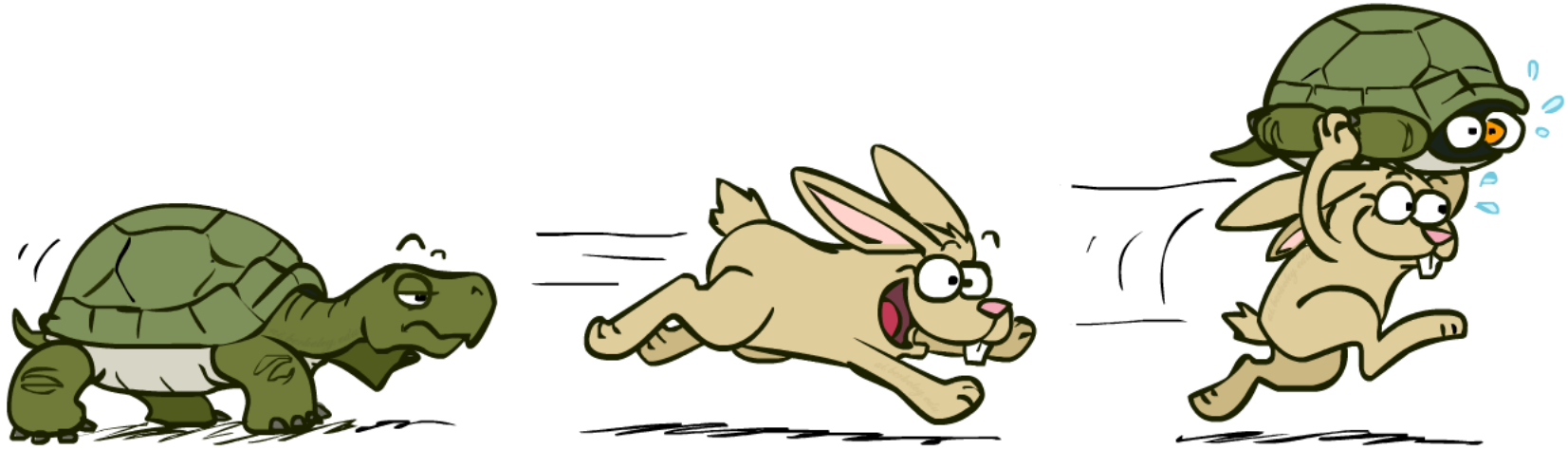
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially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

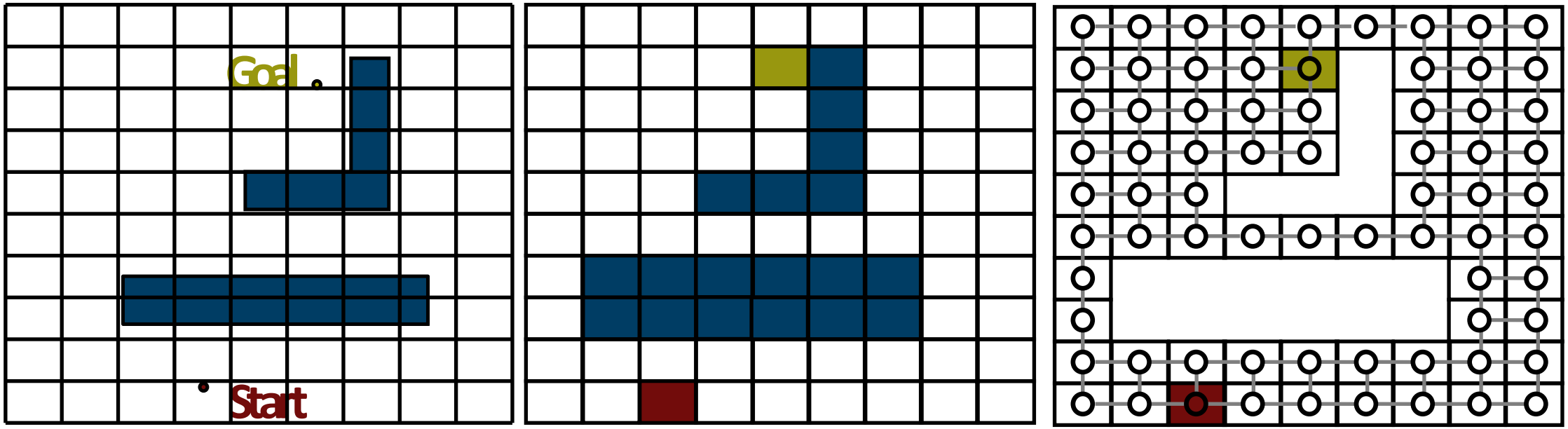
```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for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

A* using a grid discretization

A* using a grid discretization



The good :

- Optimal
- Complete
- Efficient

The bad :

- Not actually the optimal path
- Rough Approximation of C-space
- Poor Scalability, Curse of Dimensionality

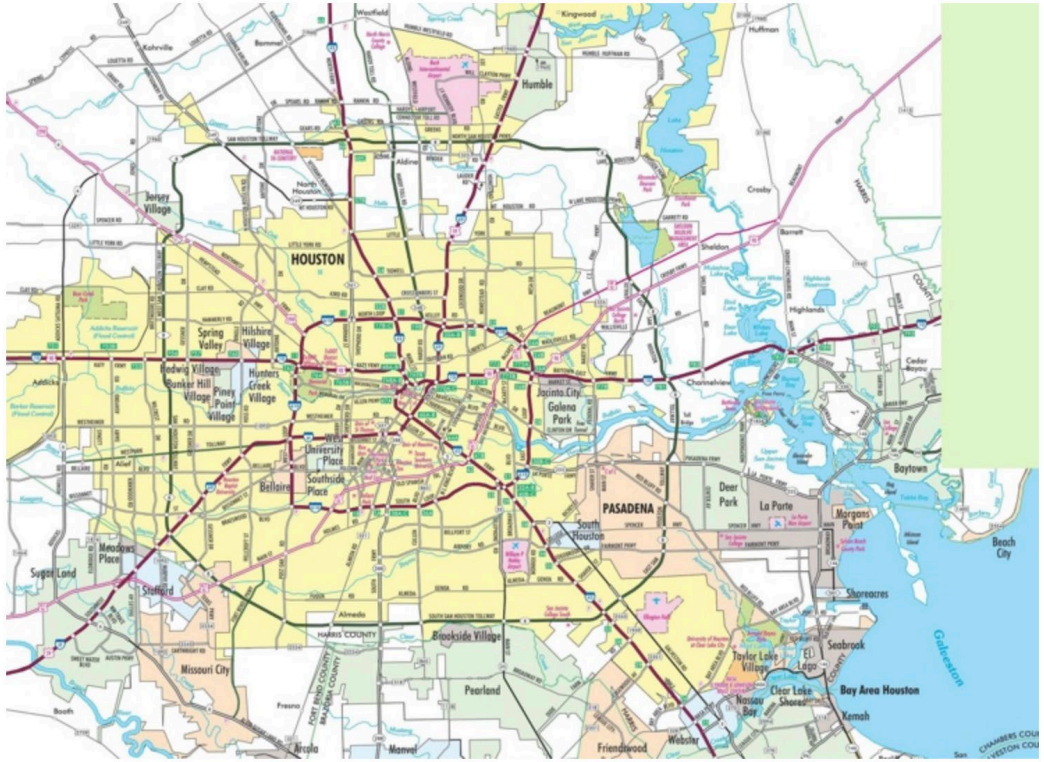
Can we Fix this?

Better Graph representation of Qfree?

Graph representations of Qfree (Roadmaps):

- Visibility Graph
  - Generalized Visibility Graph
  - Reduced Visibility Graph
- Voronoi Diagrams
- Silhouette Method

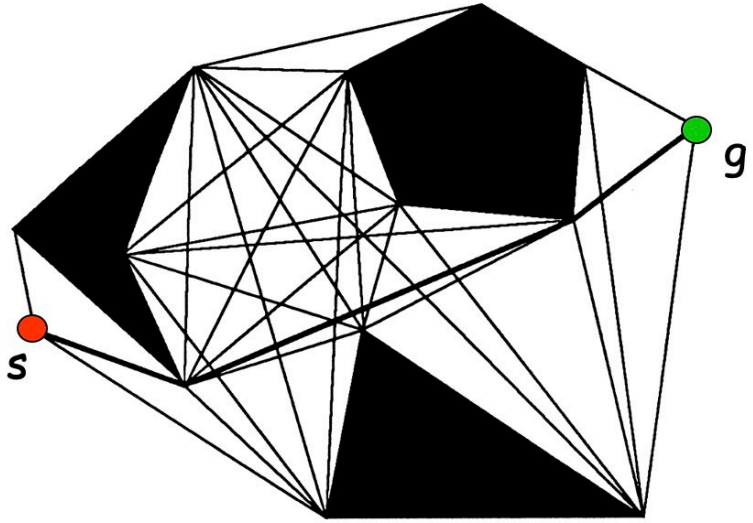
Click to see large



A roadmap is a representation of Qfree as a graph or a network of paths.

Visibility Graph methods

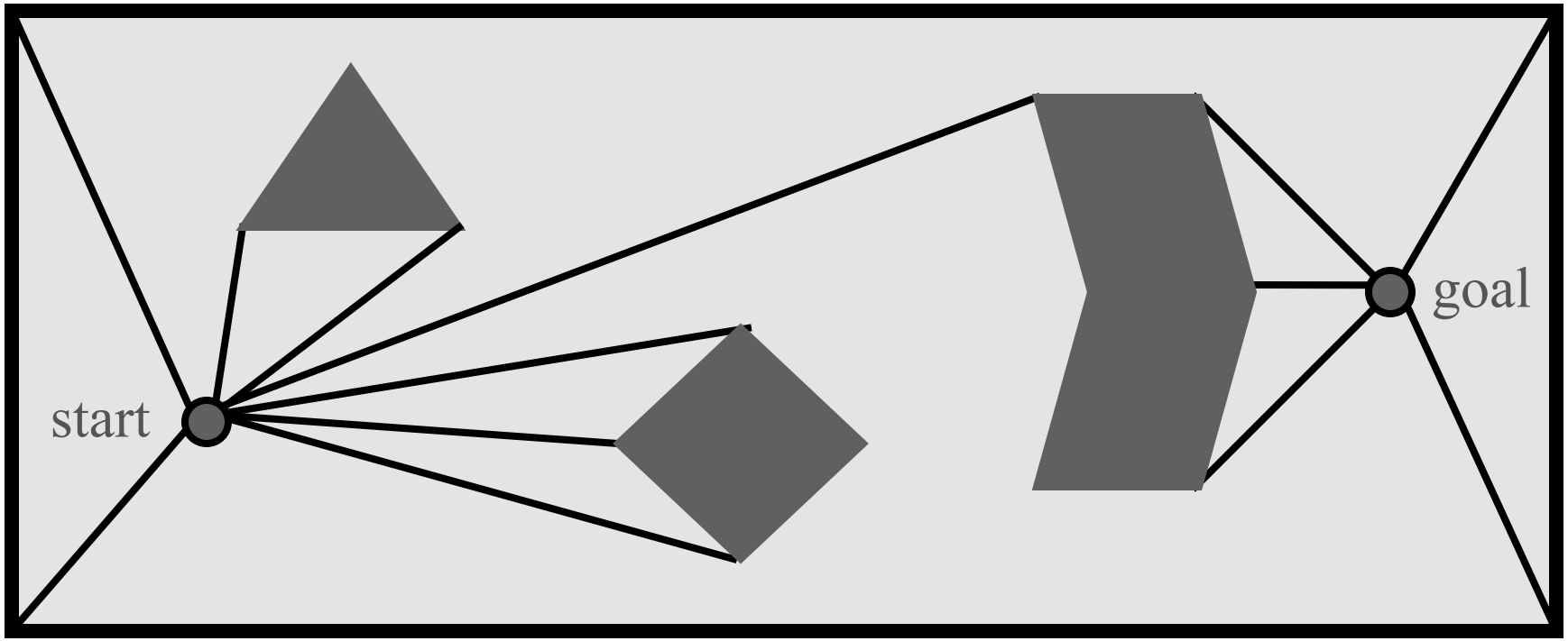
- Defined for polygonal obstacles
- Nodes correspond to vertices of obstacles
- Nodes are connected if
  - they are already connected by an edge on an obstacle
  - the line segment joining them is in free space
- Not only is there a path on this roadmap, but it is the *shortest* path
- If we include the start and goal nodes, they are automatically connected
- Algorithms for constructing them can be efficient
  - O(n^3) brute force



s and g are part of the roadmap if known, but roadmaps can be constructed without s, g

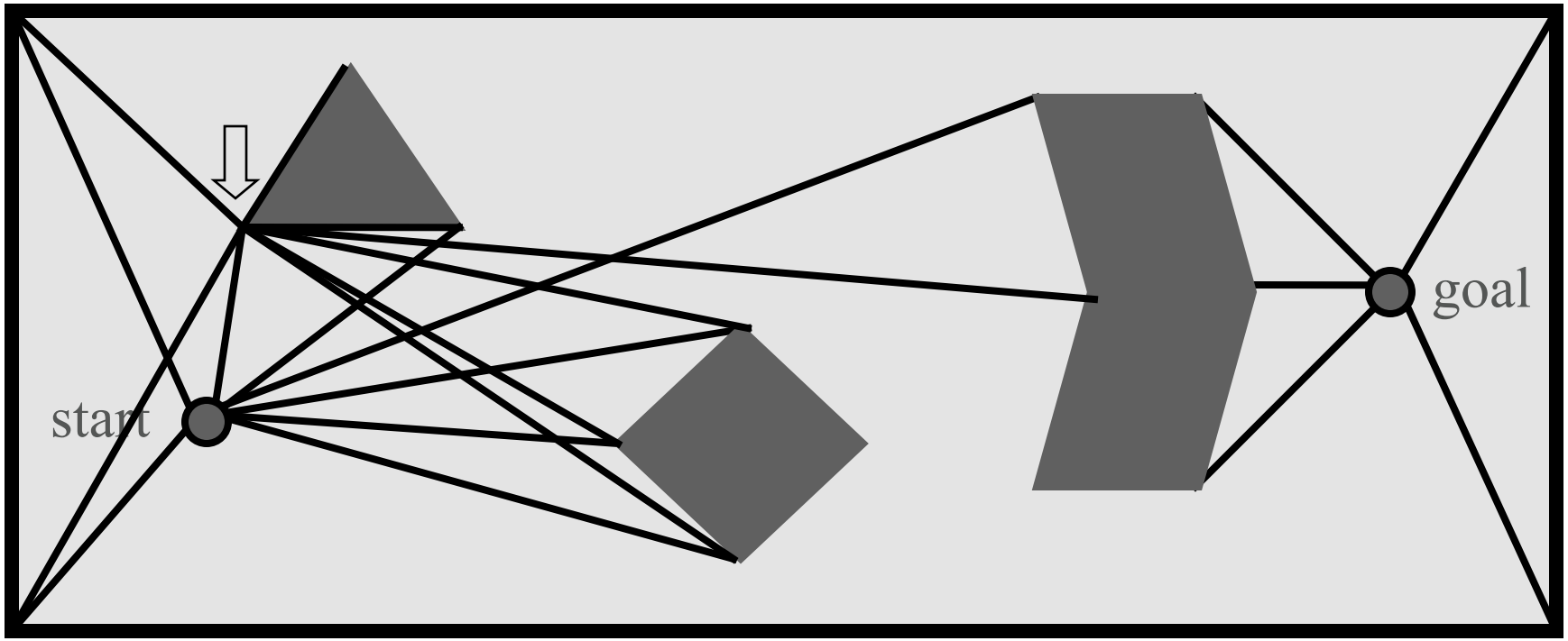
The Visibility Graph in Action (Part 1)

- First, draw lines of sight from the start and goal to all “visible” vertices and corners of the world.



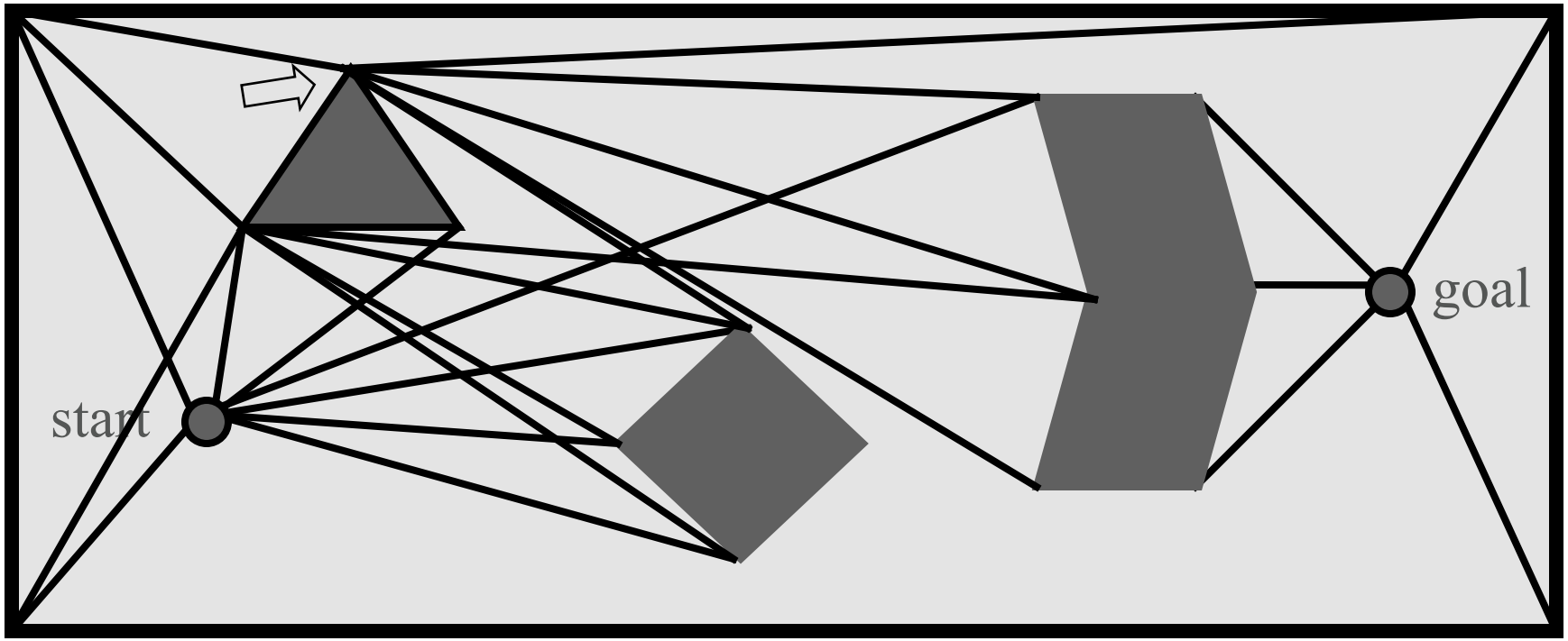
The Visibility Graph in Action (Part 2)

- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.



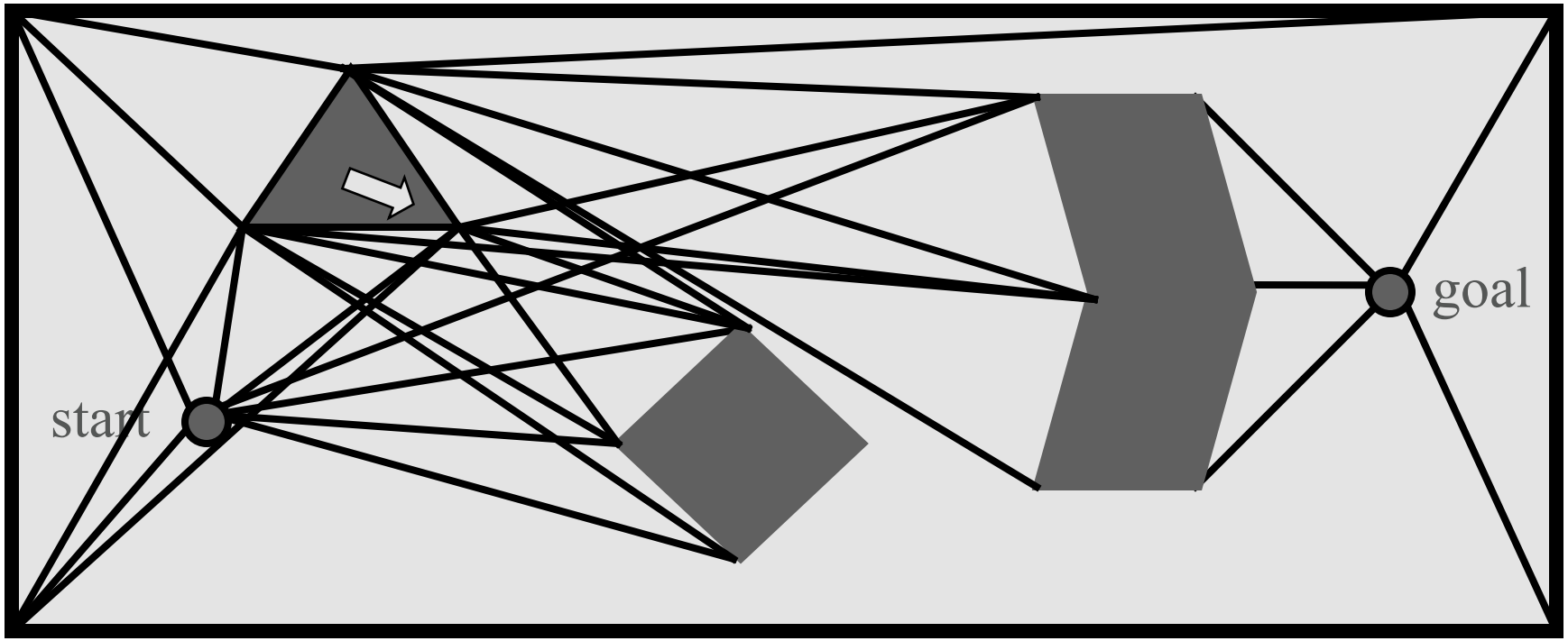
The Visibility Graph in Action (Part 3)

- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.



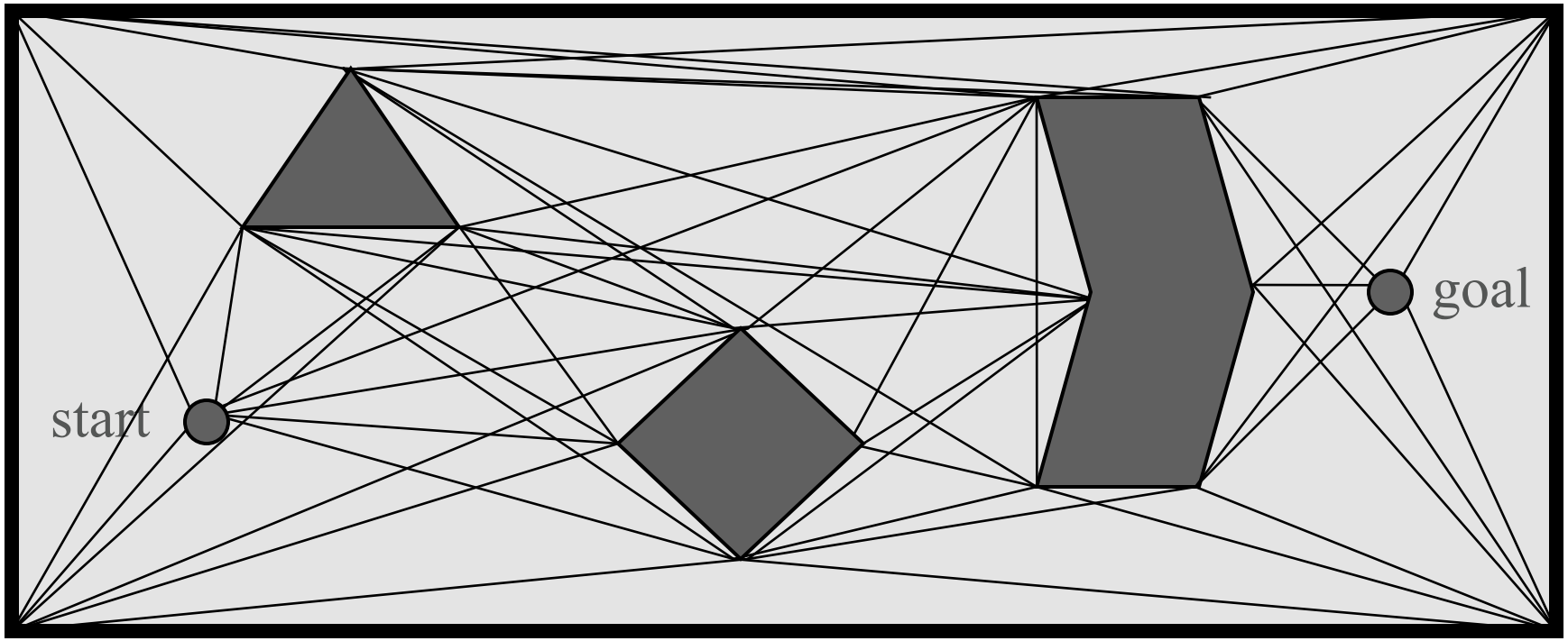
The Visibility Graph in Action (Part 4)

- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.



The Visibility Graph (Done)

- Repeat until you're done.



Complexity of Visibility Graph

Simple algorithm: $O(n^3)$ time

Rotational sweep: $O(n^2 \log n)$

Optimal algorithm: $O(n^2)$

Space: $O(n^2)$

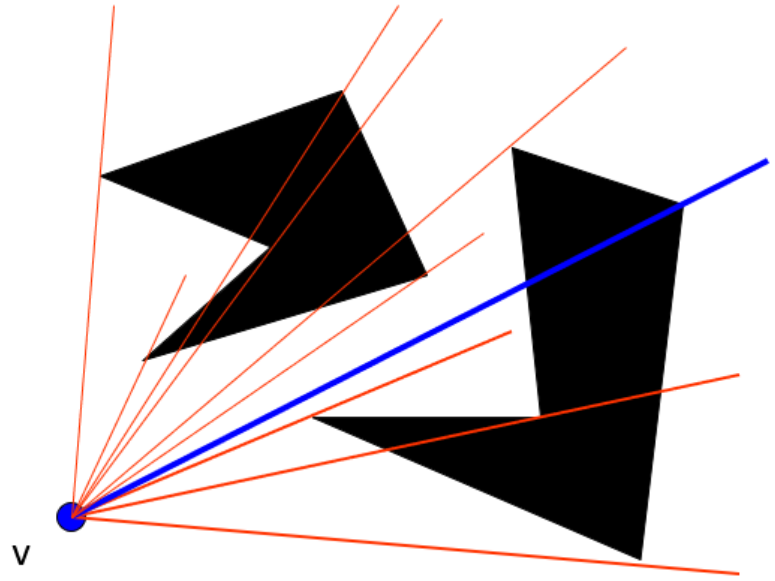
Rotational Sweep Idea

Simple algorithm: $O(n^3)$ time

Rotational sweep: $O(n^2 \log n)$

Optimal algorithm: $O(n^2)$

Space: $O(n^2)$



Key to the approach: incrementally maintain the set of edges that intersect the sweep line, sorted in order of increasing distance from v

Reduced Visibility Graph

Simple algorithm: $O(n^3)$ time

Rotational sweep: $O(n^2 \log n)$

Optimal algorithm: $O(n^2)$

Space: $O(n^2)$

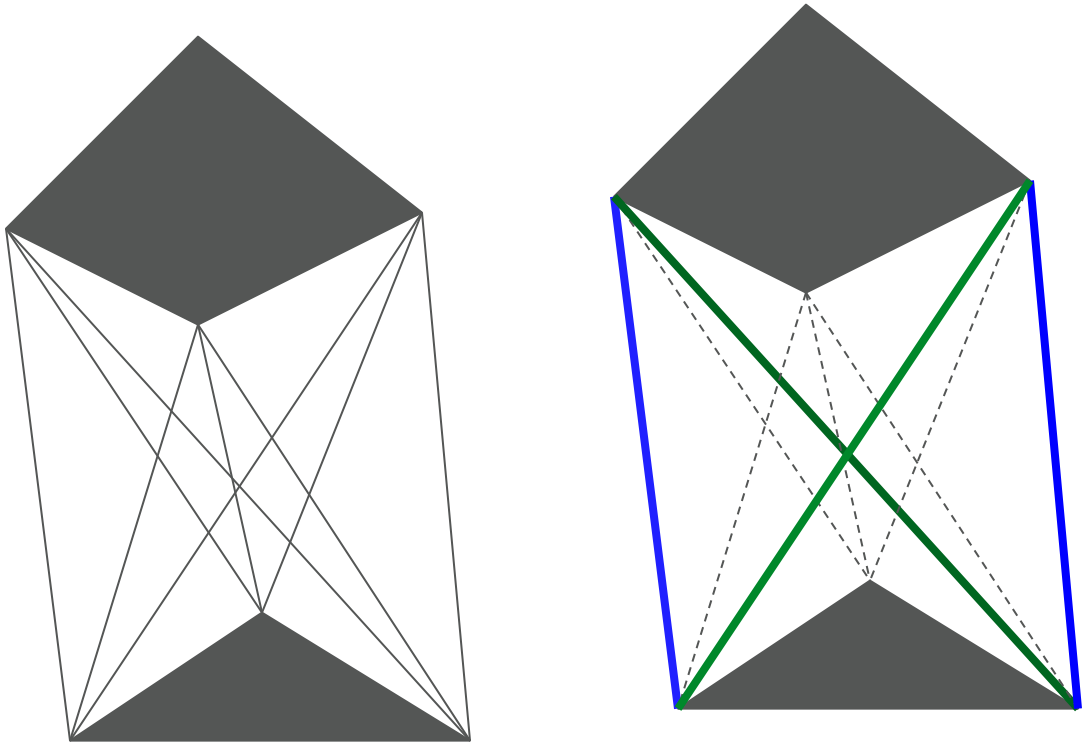
Reduced Visibility Graph

- The visibility graph has too many edges

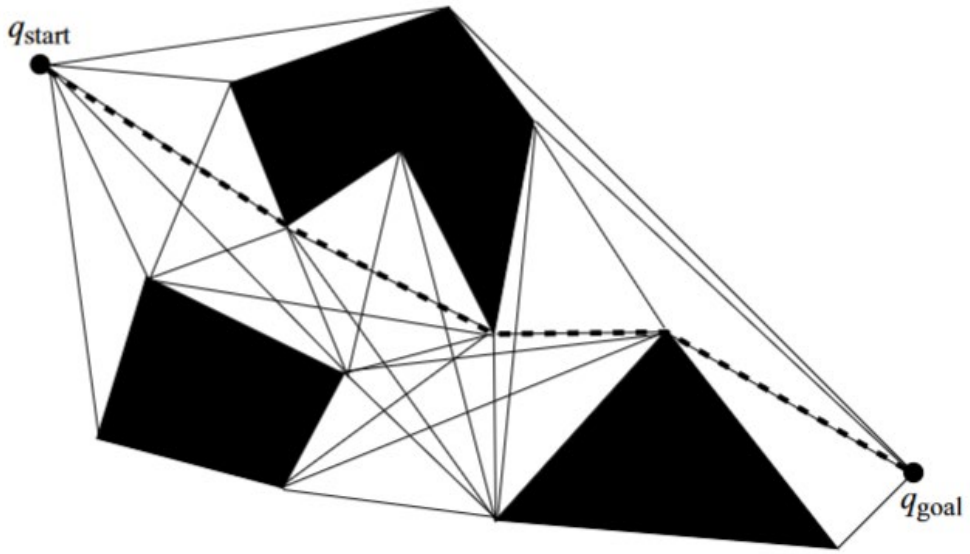
Keep only:

1.Tangent segments (whole obstacle from same side)

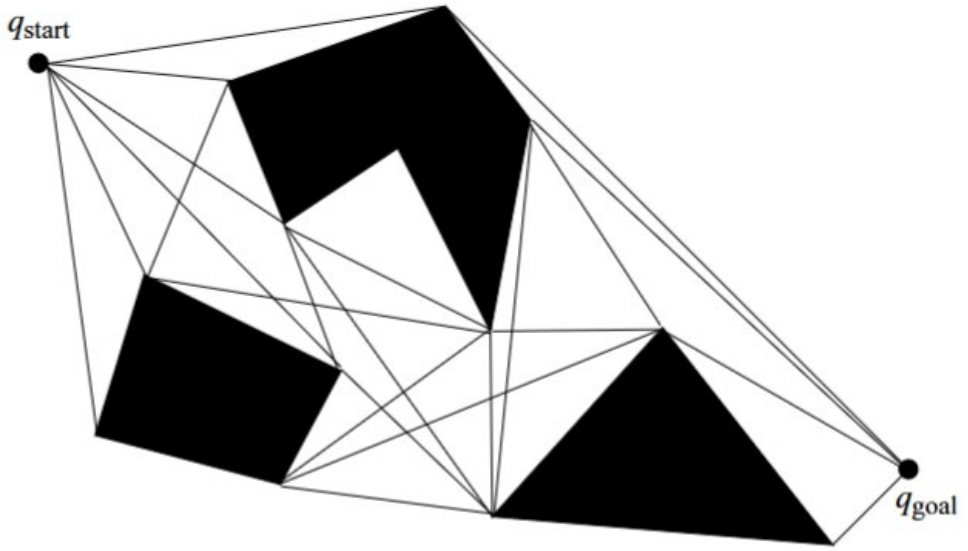
2. Supporting segments (whole obstacle from opposite sides)



Reduced Visibility Graph

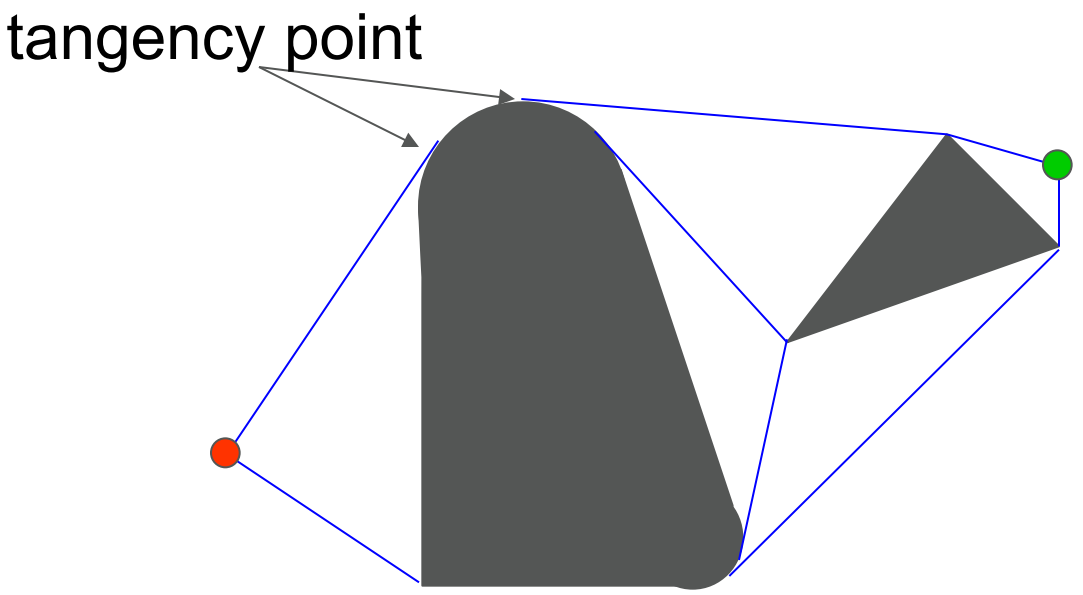


Visibility graph



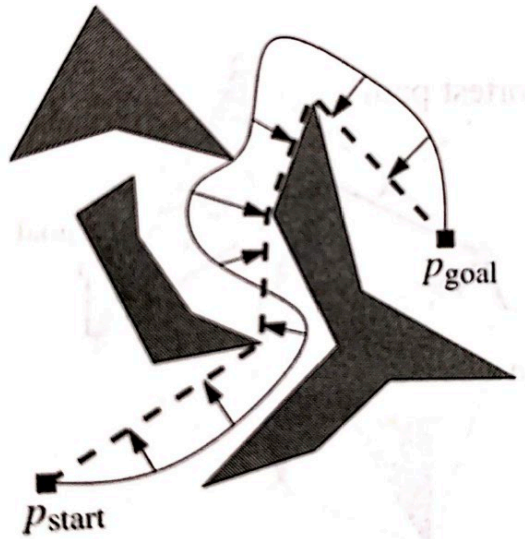
Reduced visibility graph

Generalized (Reduced) Visibility Graph



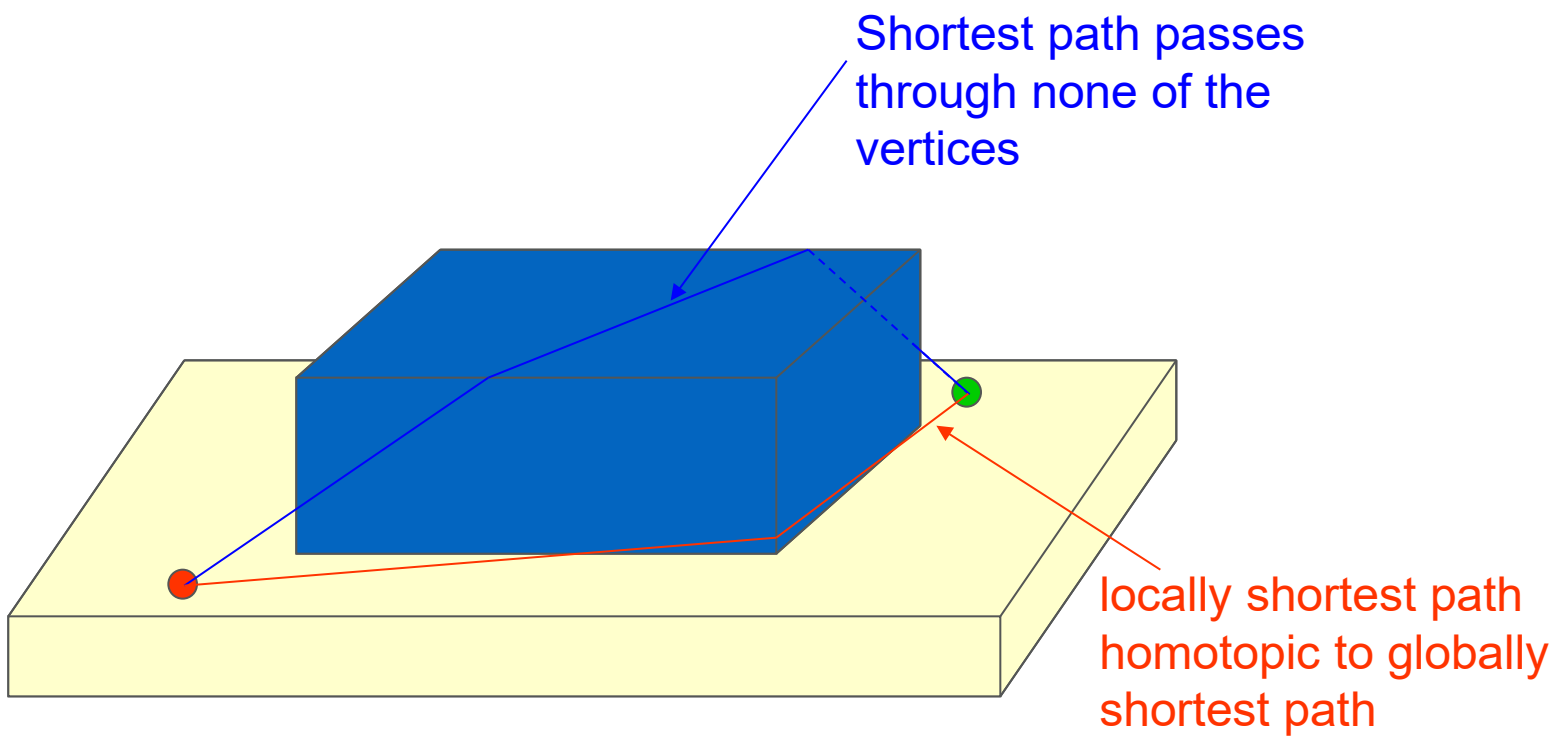
Visibility Graphs and Shortest Paths

Theorem: Any shortest path between pstart and pgoal among a set S of disjoint polygonal obstacles is a polygonal path whose inner vertices are vertices of S.



Interestingly, this all only works in $\Re^2$

Visibility Graphs for 3D space?

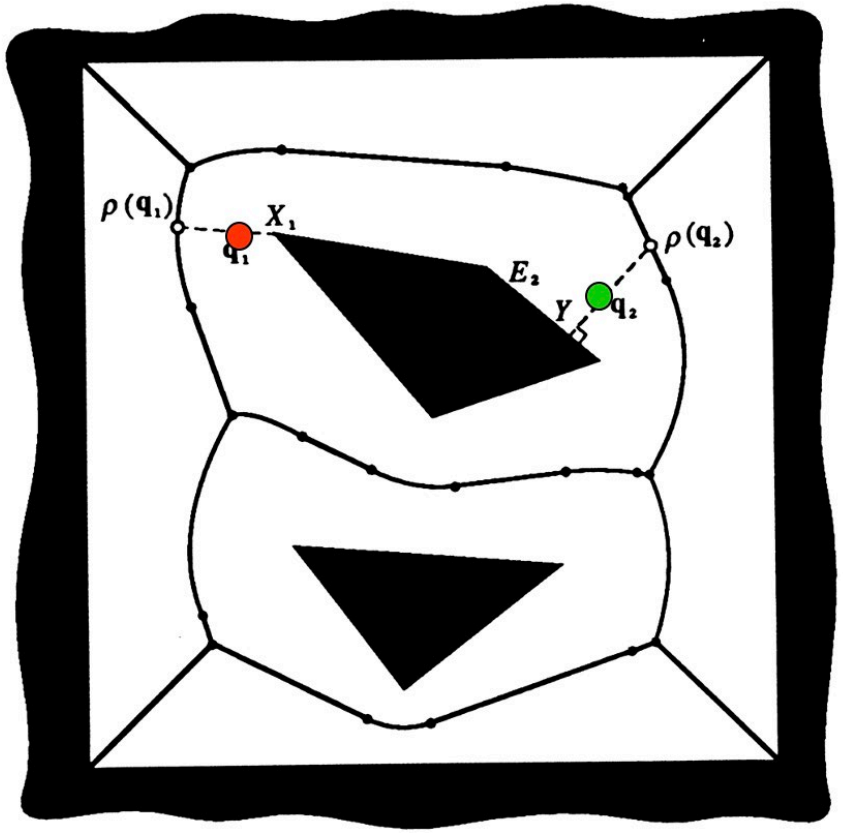


Computing the shortest collision-free path in a polyhedral space is NP-hard

Voronoi Diagram Optimize Clearance

Introduced by Computational Geometry researchers. Generate paths that maximizes clearance.

- O(n log n) time
- O(n) space



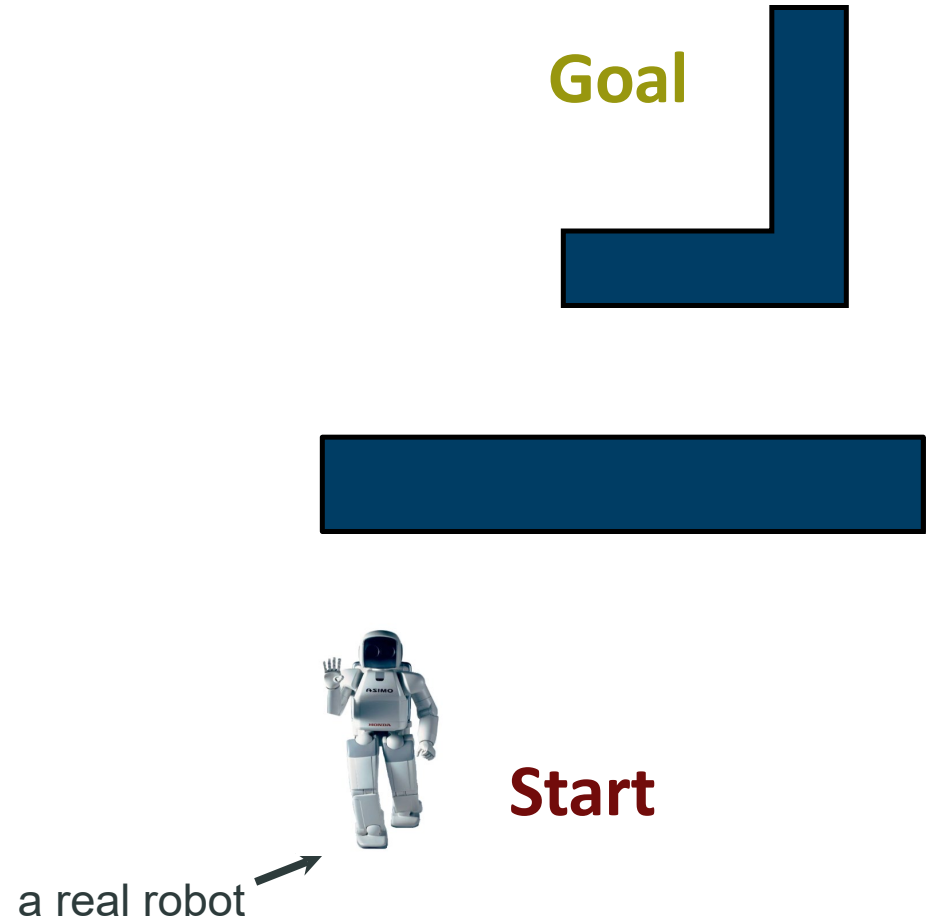
Silhouette Method

First complete general method that applies to spaces of any dimension and is singly exponential in # of dimensions [Canny, 87]

- No practical algorithm exists though

Planning Methods for Point Robots in 2D

1. Bug Algorithms
2. Potential Fields
3. Search + Grids
4. Search + Roadmaps

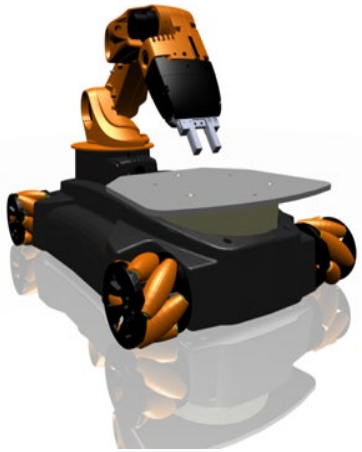


Robots

Honda Asimo

NASA Robonaut 2

Willow Garage PR2



KUKA youBot

DFKI AILA

Rethink Robotics Baxter

Intel HERB

Care-O-Bot

How?