

# Lecture 2: Search

Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

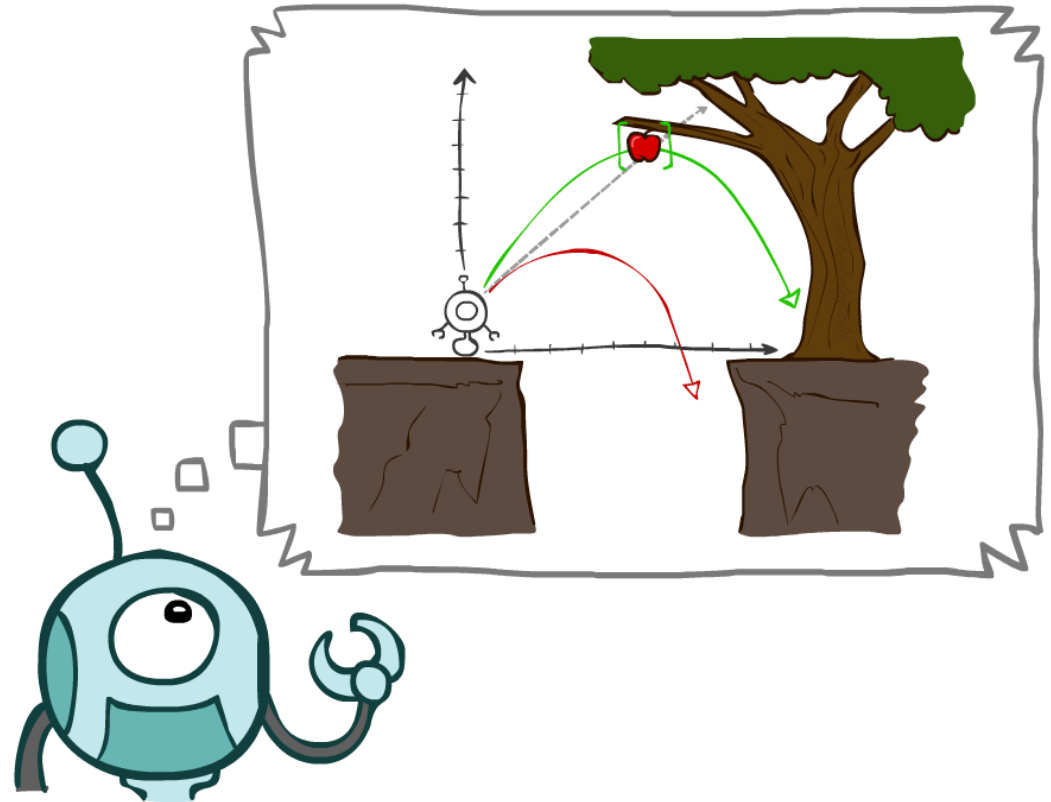
<https://shuaili8.github.io>

<https://shuaili8.github.io/Teaching/CS3317/index.html>

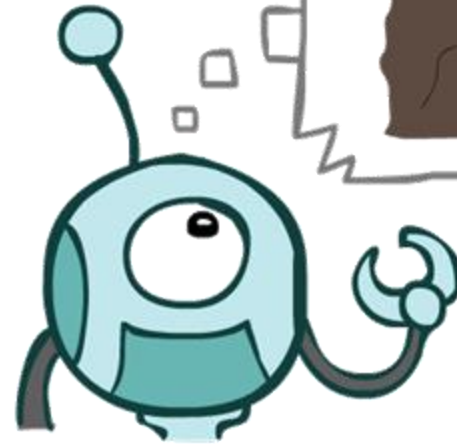
Part of slide credits: CMU AI & <http://ai.berkeley.edu>

# Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Informed Search
  - Heuristics
  - Greedy Search
  - A\* Search
  - Graph Search

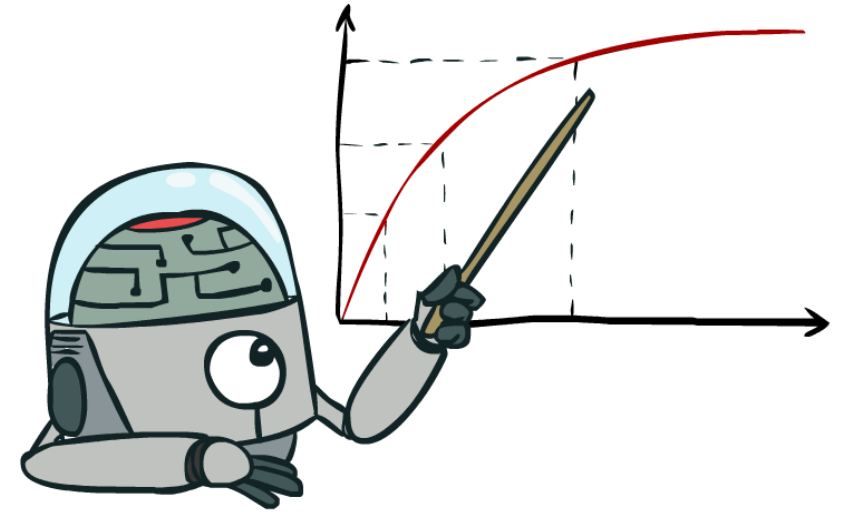


# Agents that Plan



# Rationality

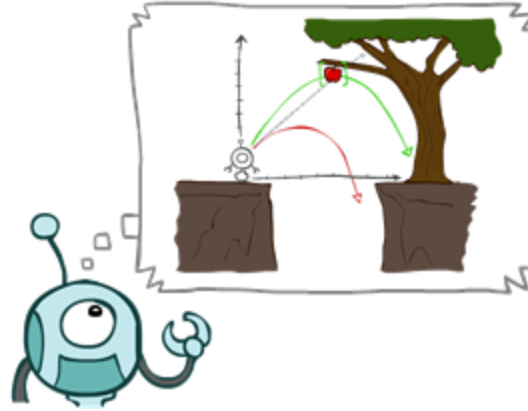
- What is rational depends on:
  - Performance measure
  - Agent's prior knowledge of environment
  - Actions available to agent
  - Percept/sensor sequence to date
- Being rational means **maximizing your expected utility**



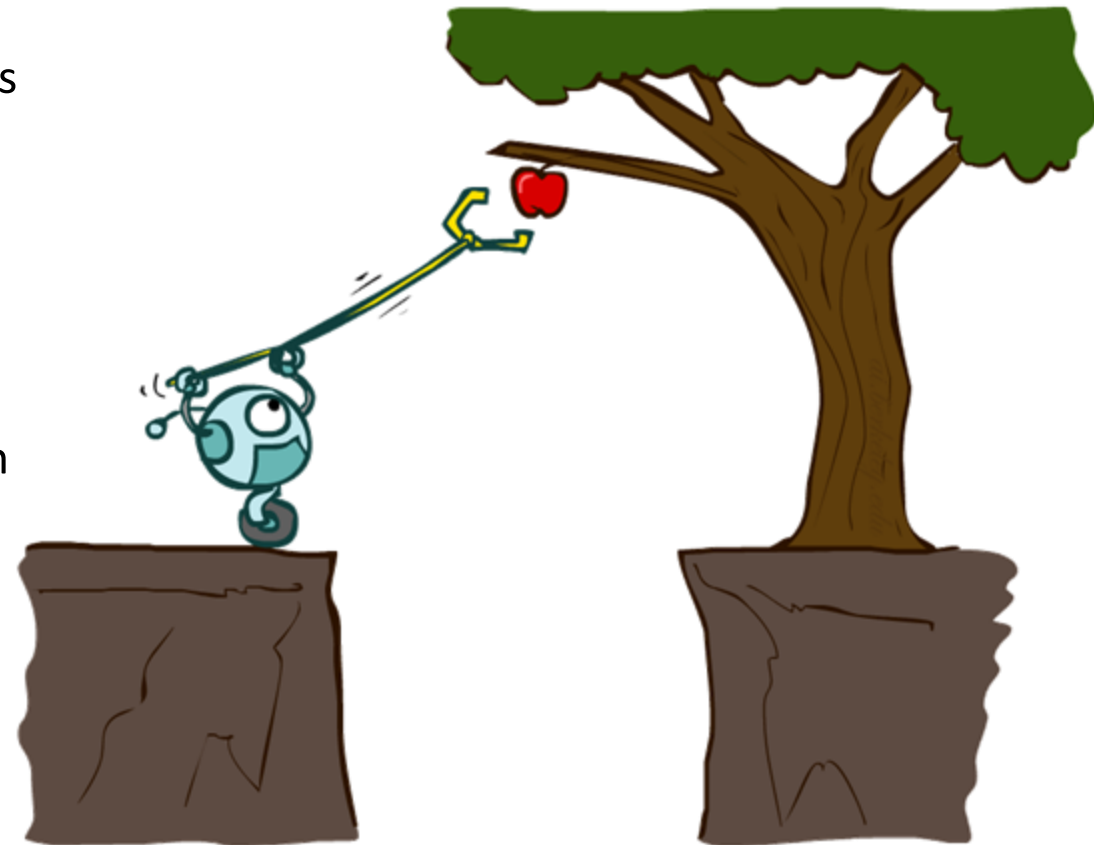
# Rational Agents

- Are rational agents *omniscient*? 无所不知的
  - No – they are limited by the available percepts
- Are rational agents *clairvoyant*? 透视的
  - No – they may lack knowledge of the environment dynamics
- Do rational agents *explore* and *learn*?
  - Yes – in unknown environments these are essential
- So rational agents are not necessarily successful, but they are *autonomous* (i.e., control their own behavior)

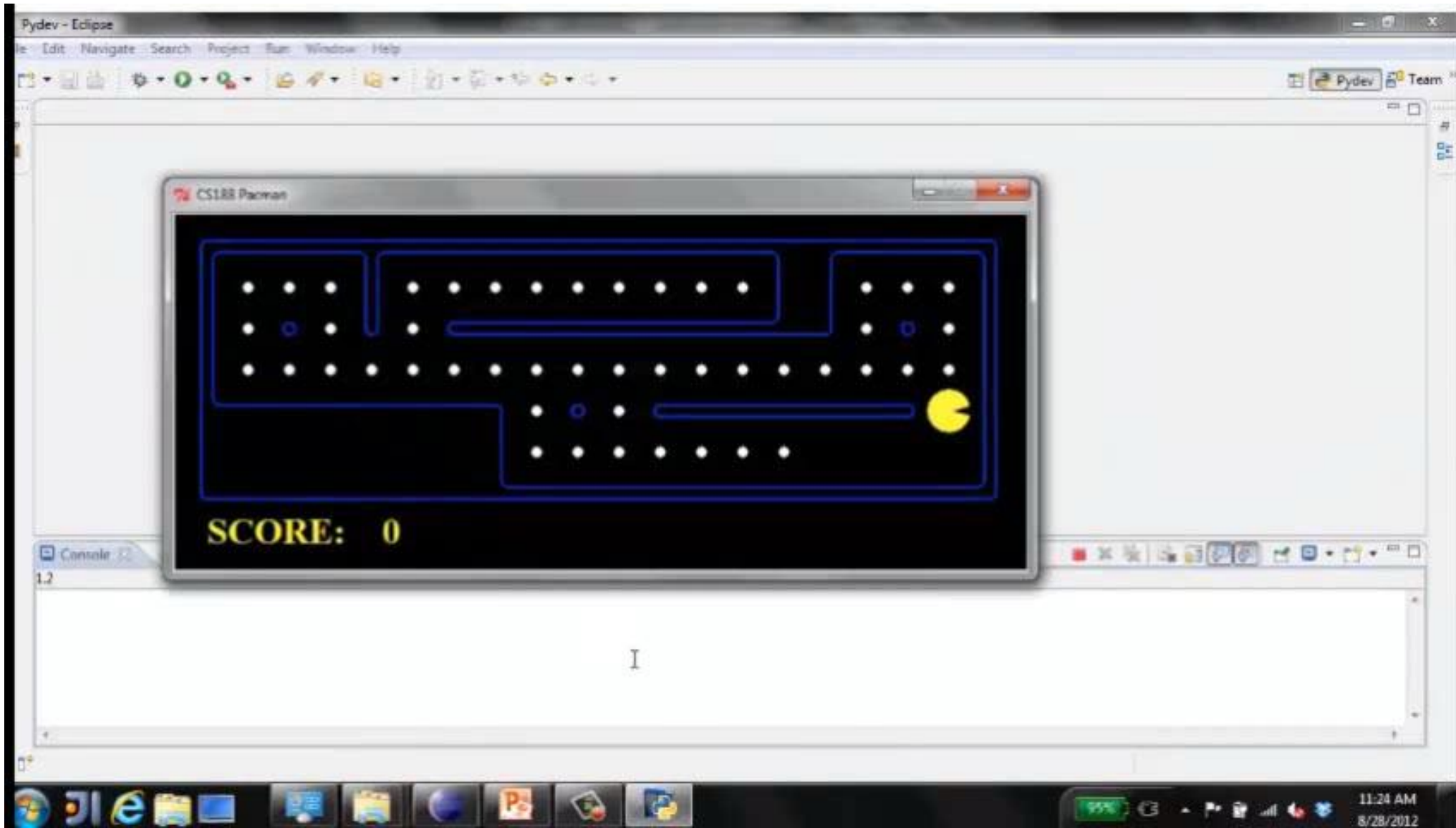
# Planning Agents



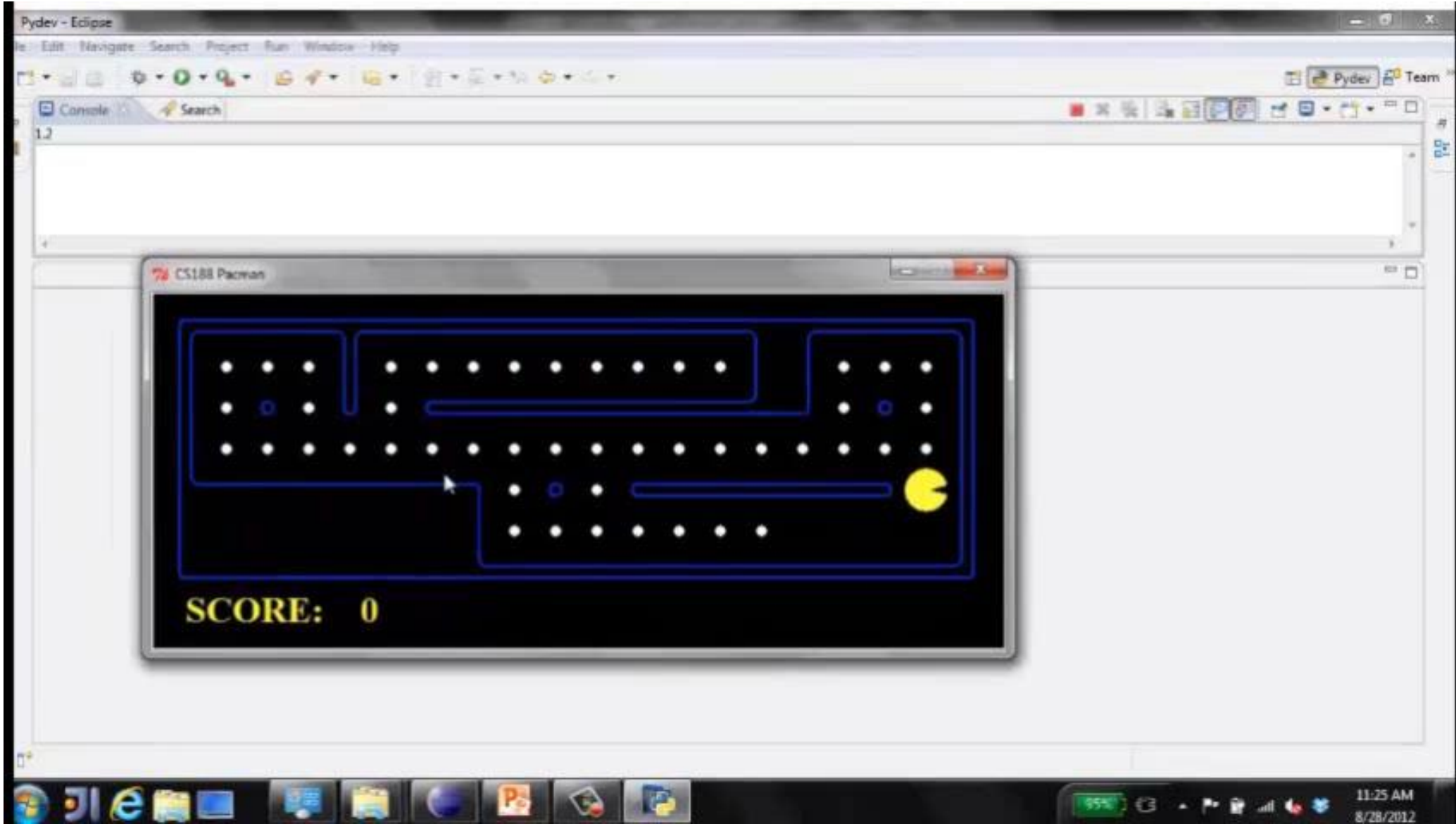
- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized or **predicted**) consequences of actions
  - Must have a **transition** model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - **Consider how the world WOULD BE**
- Spectrum of deliberativeness:
  - Generate complete, optimal plan offline, then execute
  - Generate a simple, greedy plan, start executing, replan when something goes wrong
- Optimal vs. complete planning
- Planning vs. replanning [Demo: re-planning (L2D3)]  
[Demo: mastermind (L2D4)]



# Video of Demo Replanning

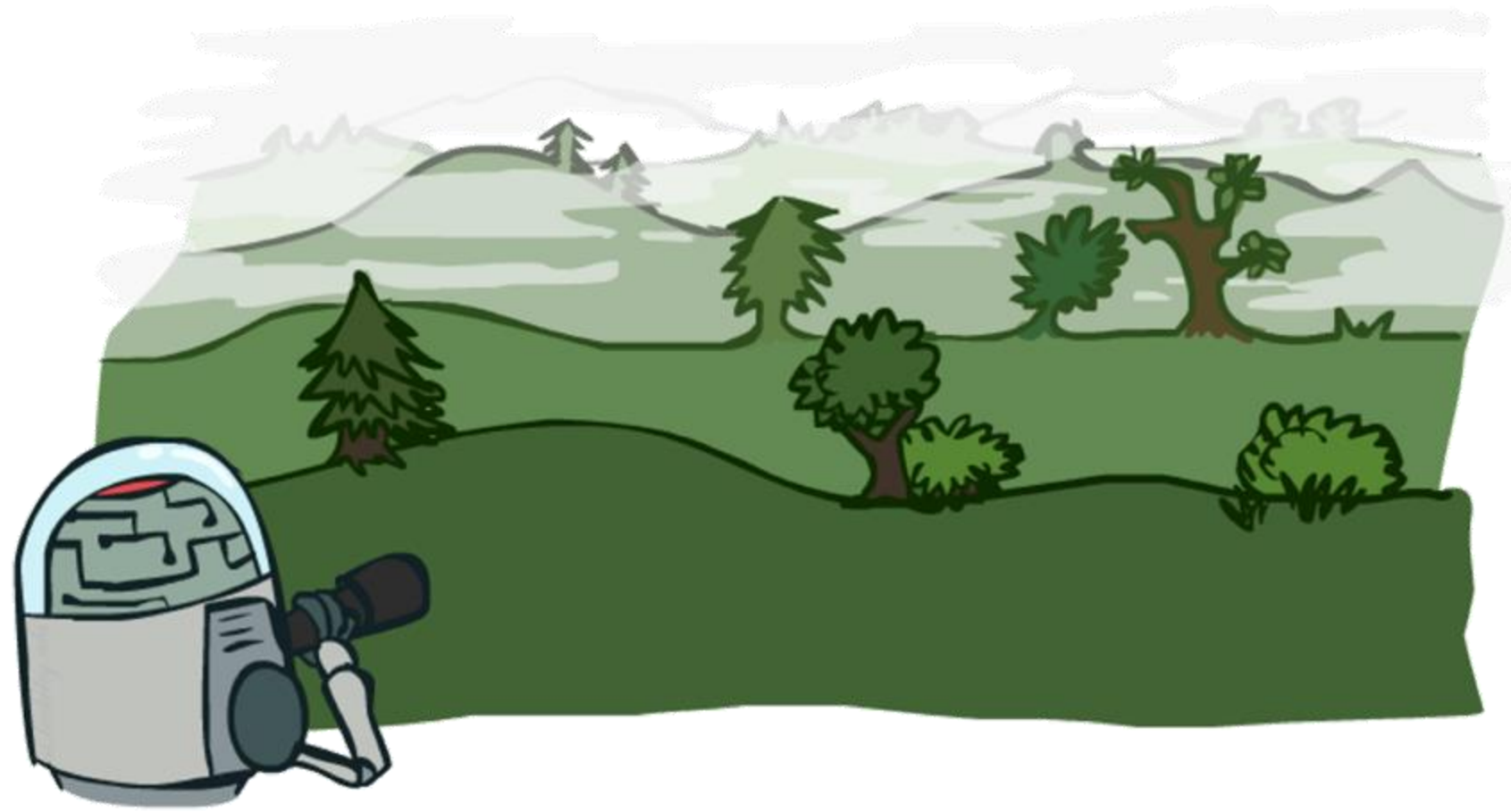


# Video of Demo Mastermind





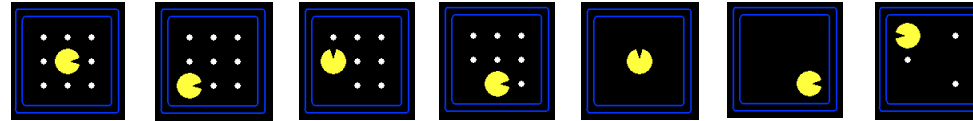
# Search Problems



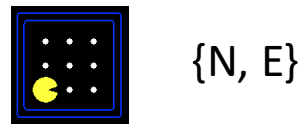
# Search Problems

- A **search problem** consists of:

- A state space

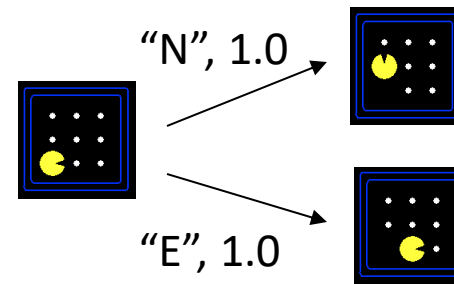


- For each state, a set **Actions(s)** of successors/actions



- A successor function

- A transition model  $T(s,a)$
- A step cost(reward) function  $c(s,a,s')$



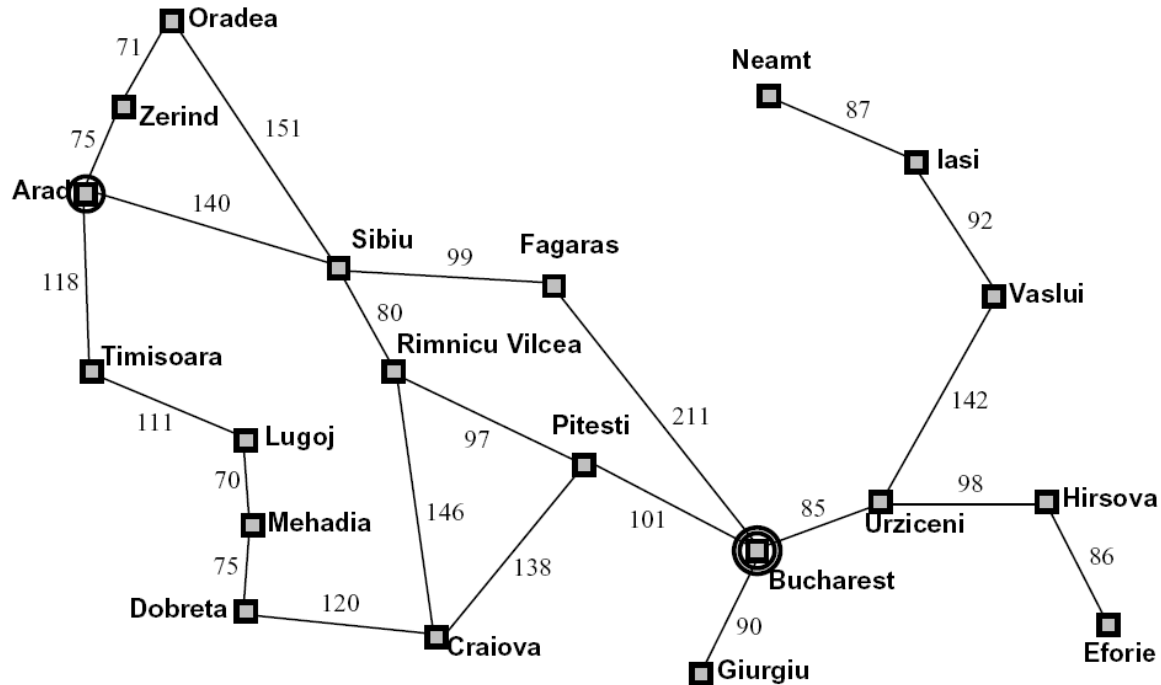
- A start state and a goal test

- A **solution** is a sequence of actions (a plan) which transforms the start state to a goal state

# Search Problems Are Models



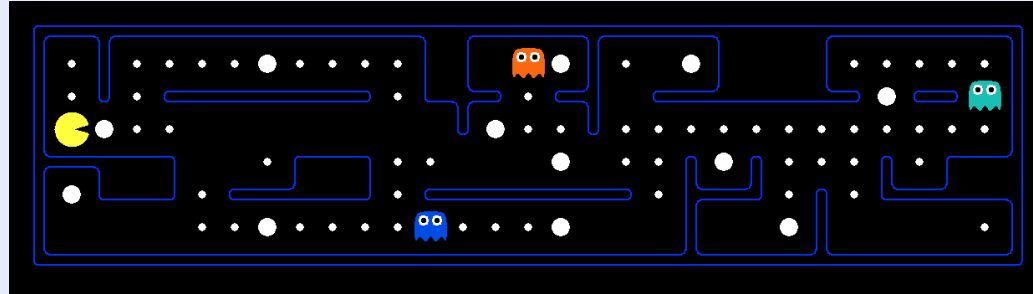
# Example: Traveling in Romania



- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with cost = distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

# What's in a State Space?

The **world state** includes every last detail of the environment

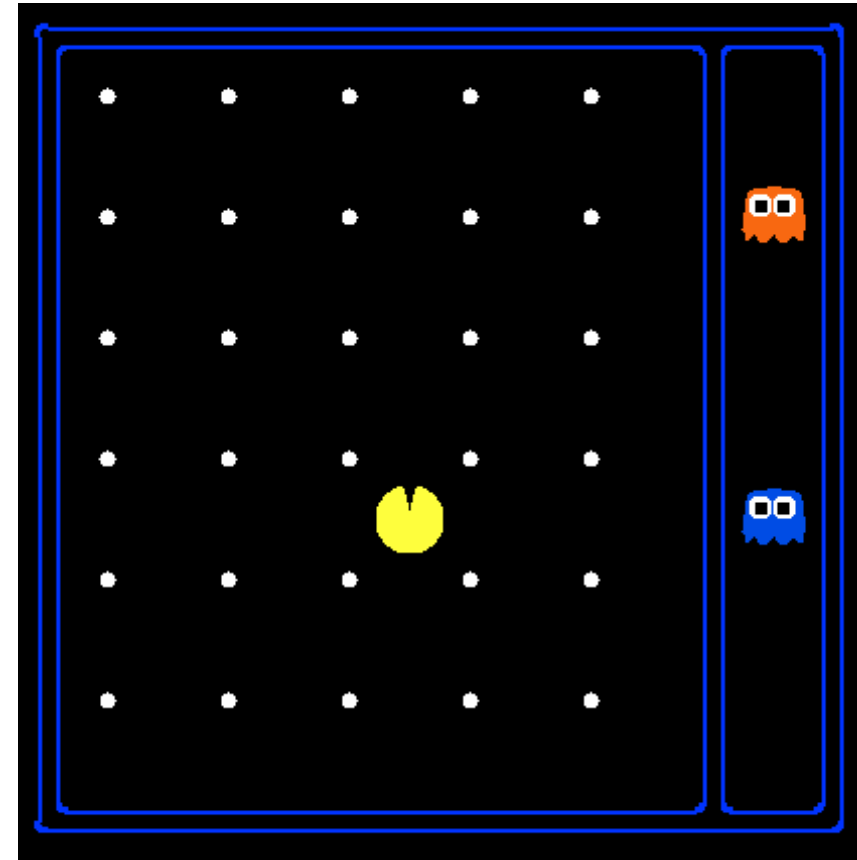


A **search state** keeps only the details needed for planning (abstraction)

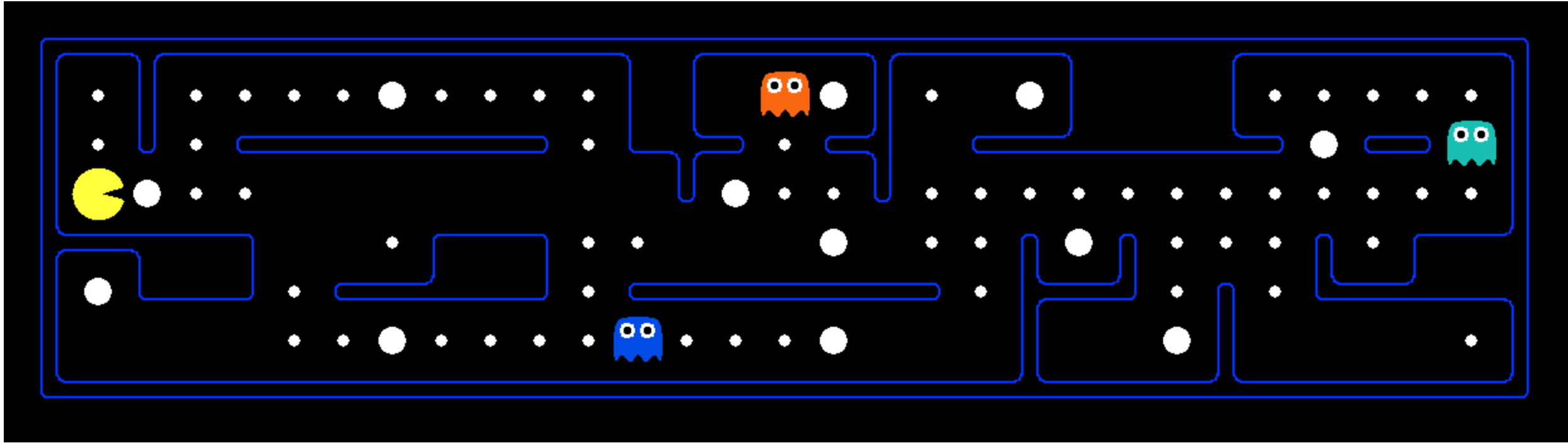
- **Problem: Pathing**
  - States:  $(x,y)$  location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is  $(x,y)=END$
- **Problem: Eat-All-Dots**
  - States:  $\{(x,y), \text{dot booleans}\}$
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

# State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW
- How many
  - World states?  
 $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing?  
120
  - States for eat-all-dots?  
 $120 \times (2^{30})$

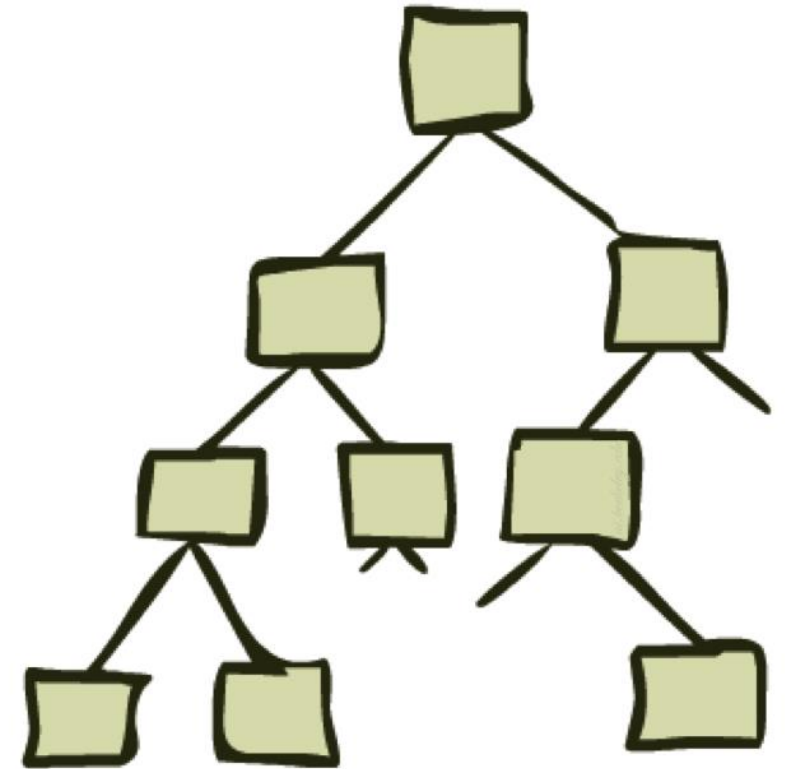


# Safe Passage



- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
  - (agent position, dot booleans, power pellet booleans, remaining scared time)

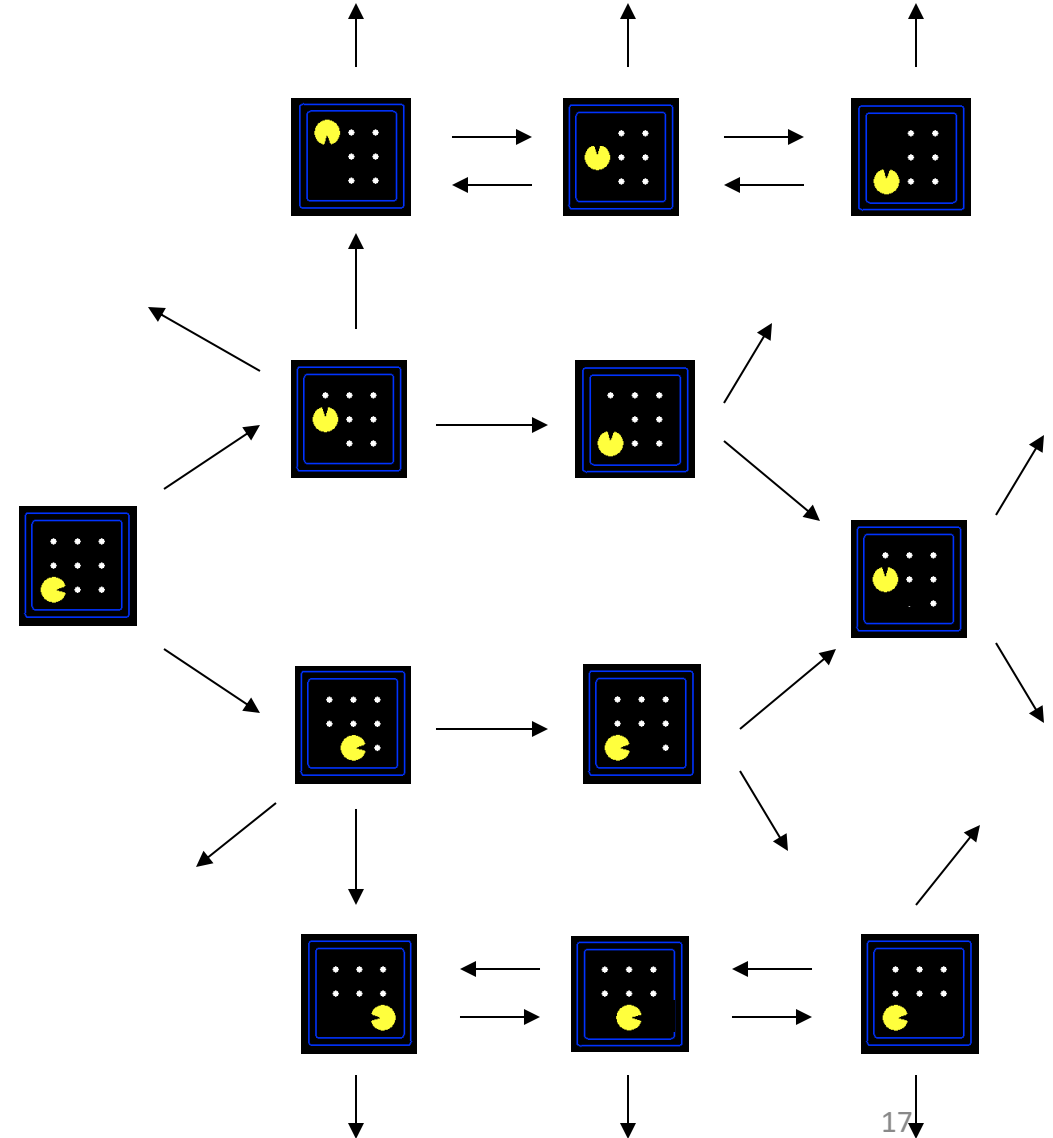
# State Space Graphs and Search Trees



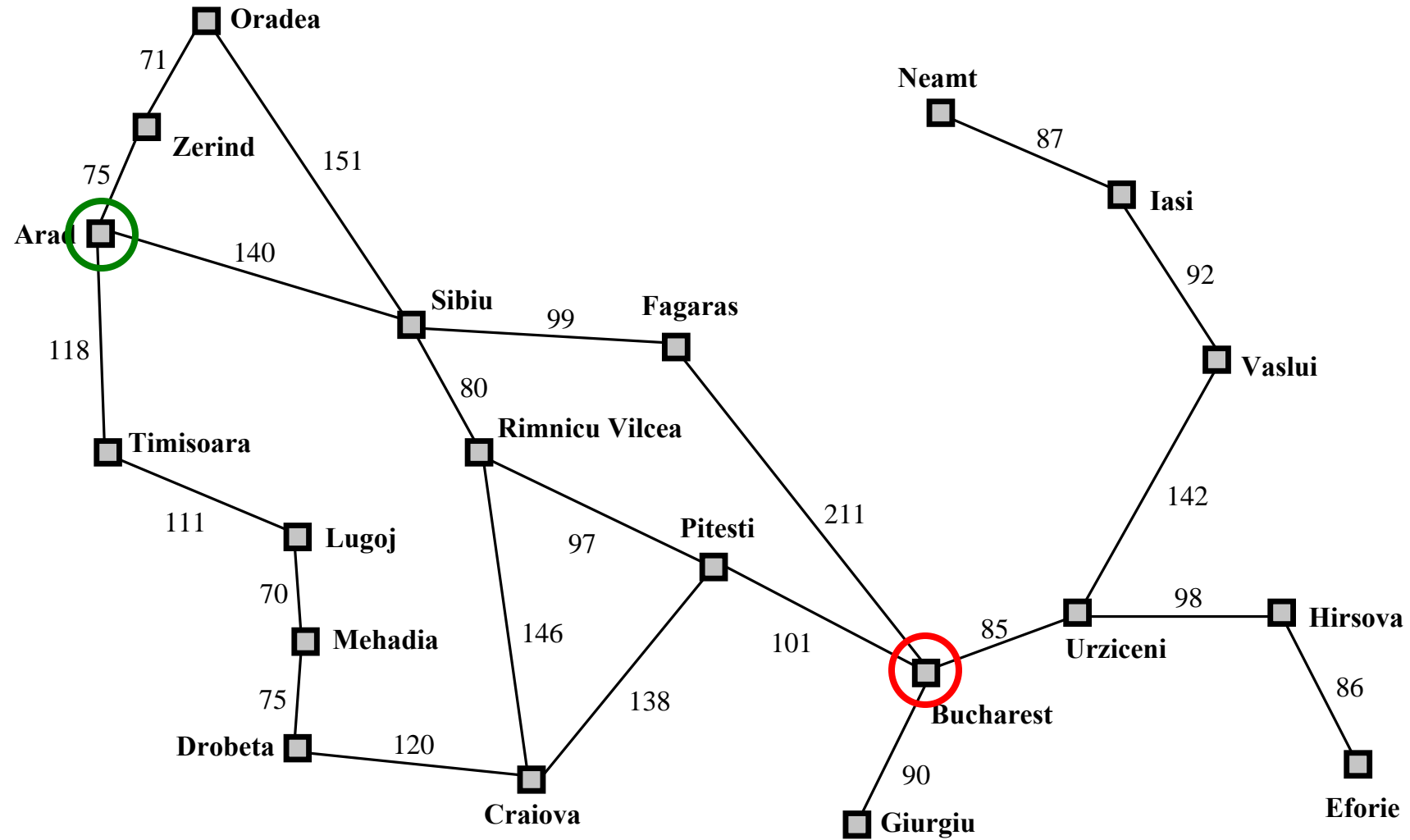


# State Space Graphs

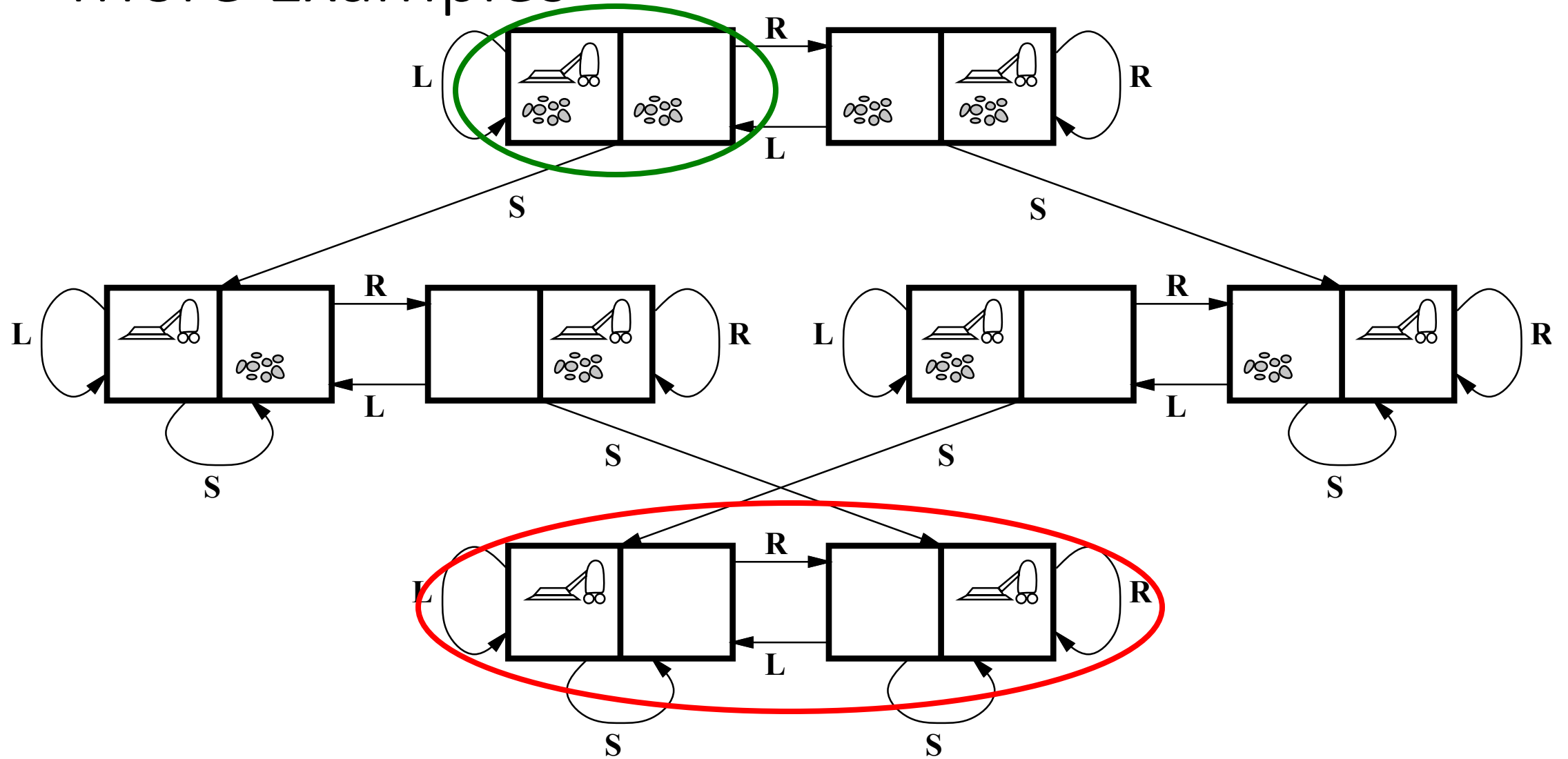
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



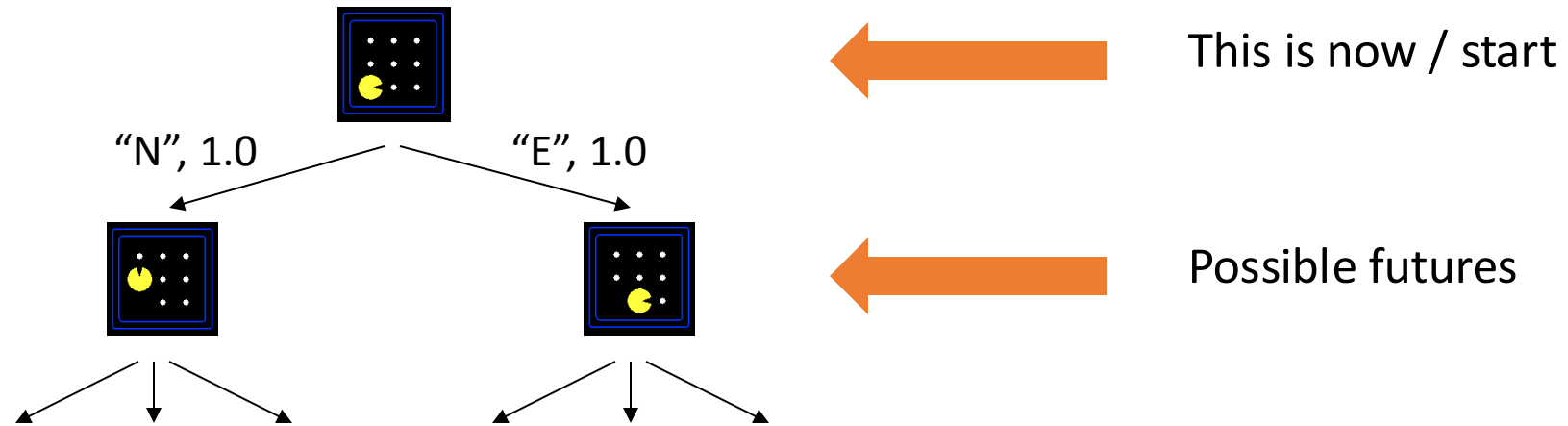
# More Examples



# More Examples



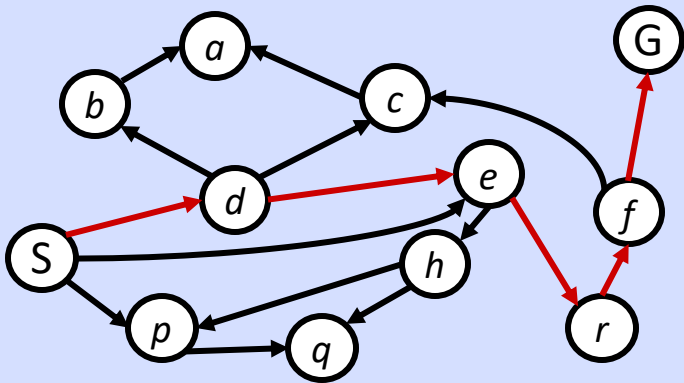
# Search Trees



- A search tree:
  - A “what if” tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to **PLANS** that achieve those states
  - **For most problems, we can never actually build the whole tree**

# State Space Graphs vs. Search Trees

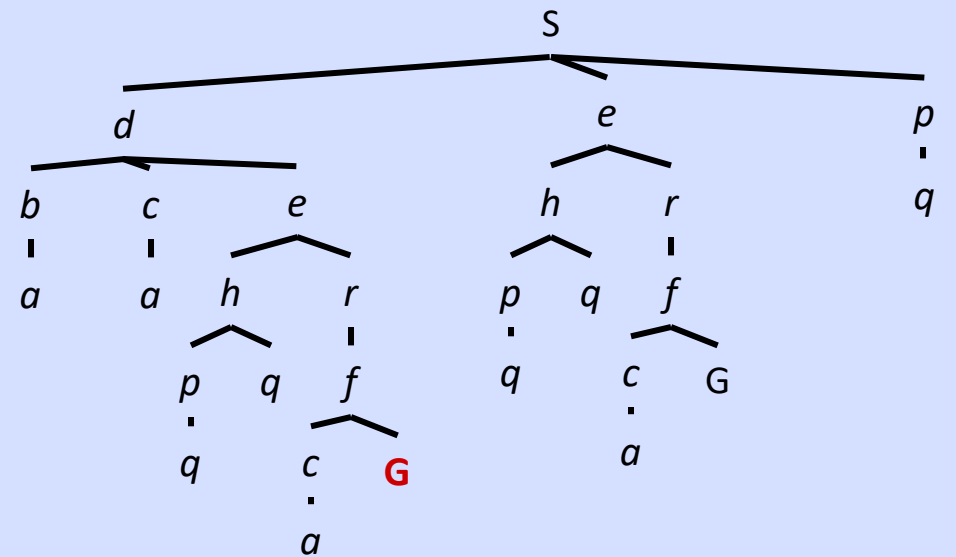
## State Space Graph



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

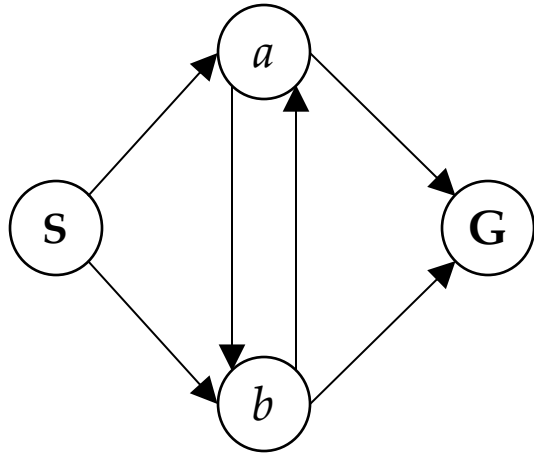
*We construct both  
on demand – and  
we construct as  
little as possible*

## Search Tree

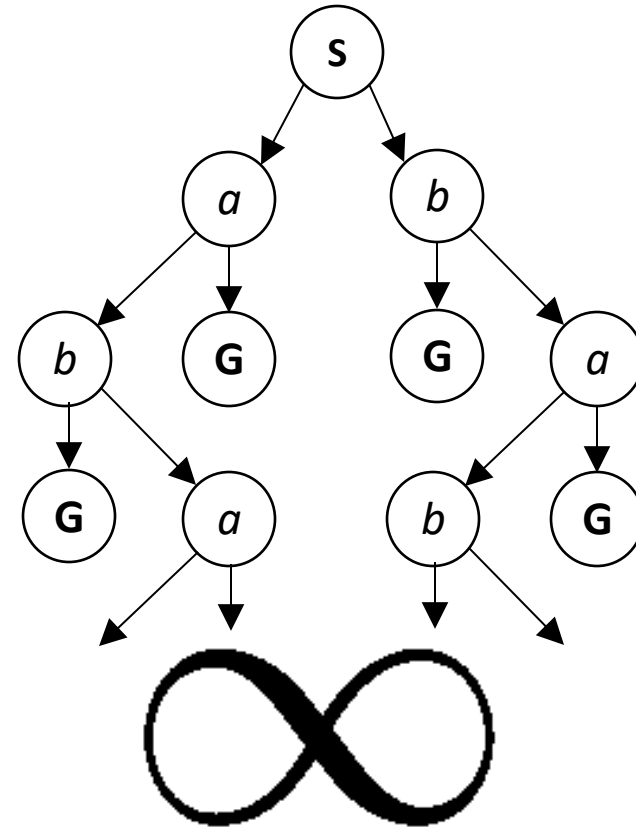


# State Space Graphs vs. Search Trees

Consider this 4-state graph:

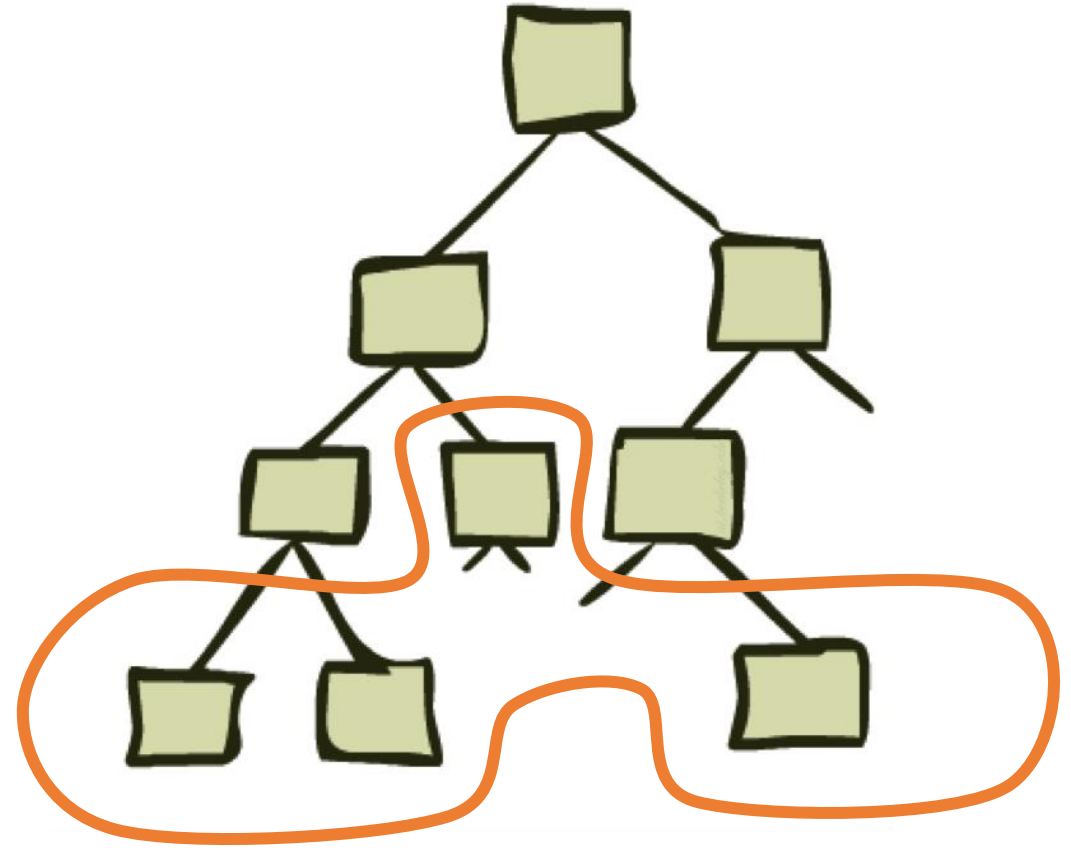


How big is its search tree (from S)?

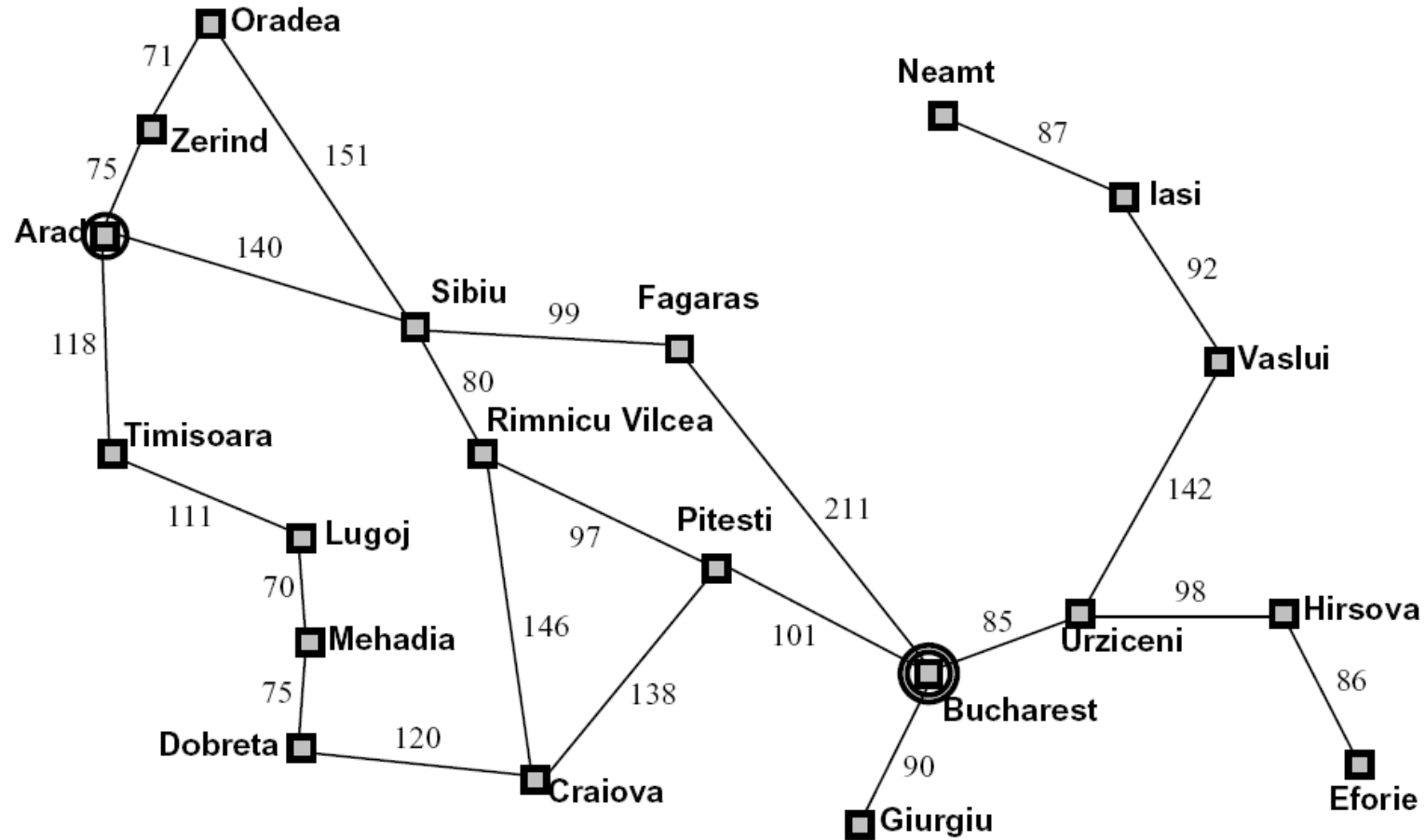


Important: Lots of repeated structure in the search tree!

# Tree Search

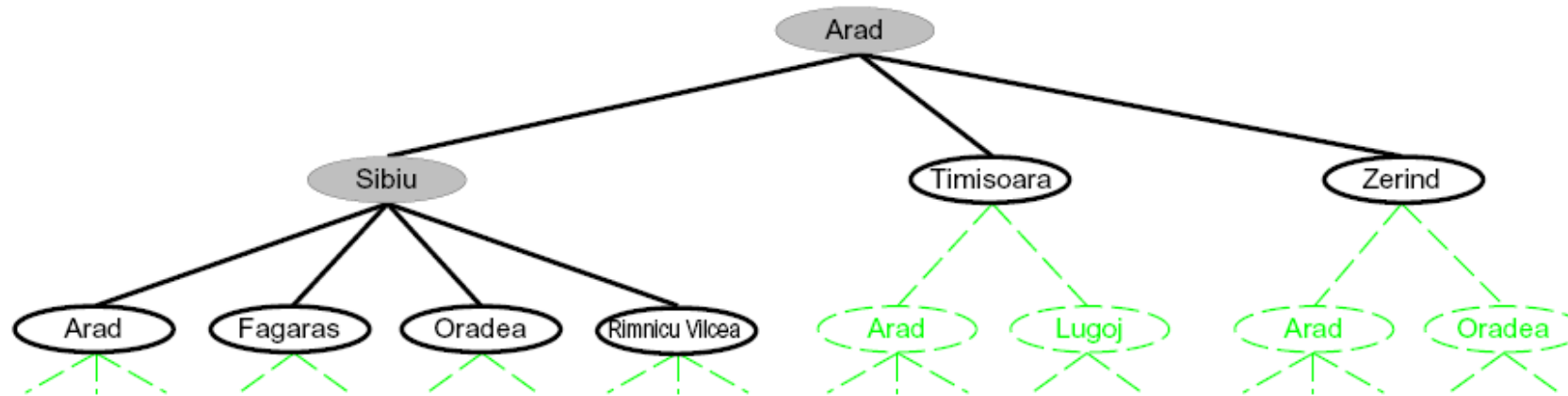


# Search Example: Romania





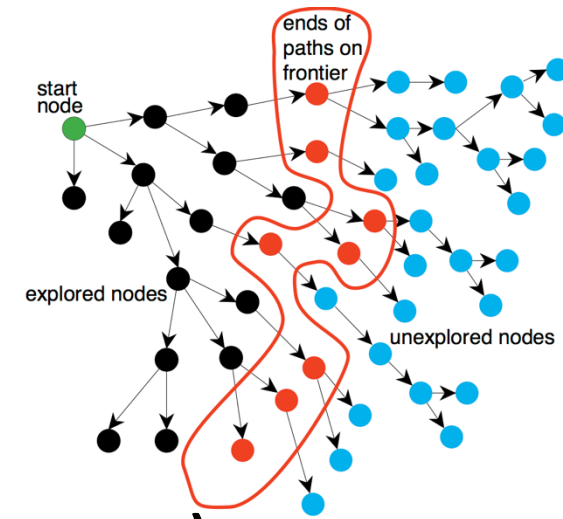
# Searching with a Search Tree



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

# General Tree Search

function `TREE_SEARCH(problem)` returns a solution, or failure



initialize the `frontier` as a specific work list (stack, queue, priority queue)

add initial state of `problem` to `frontier`

loop do

if the `frontier` is empty then

return failure

choose a `node` and remove it from the `frontier`

if the `node` contains a goal state then

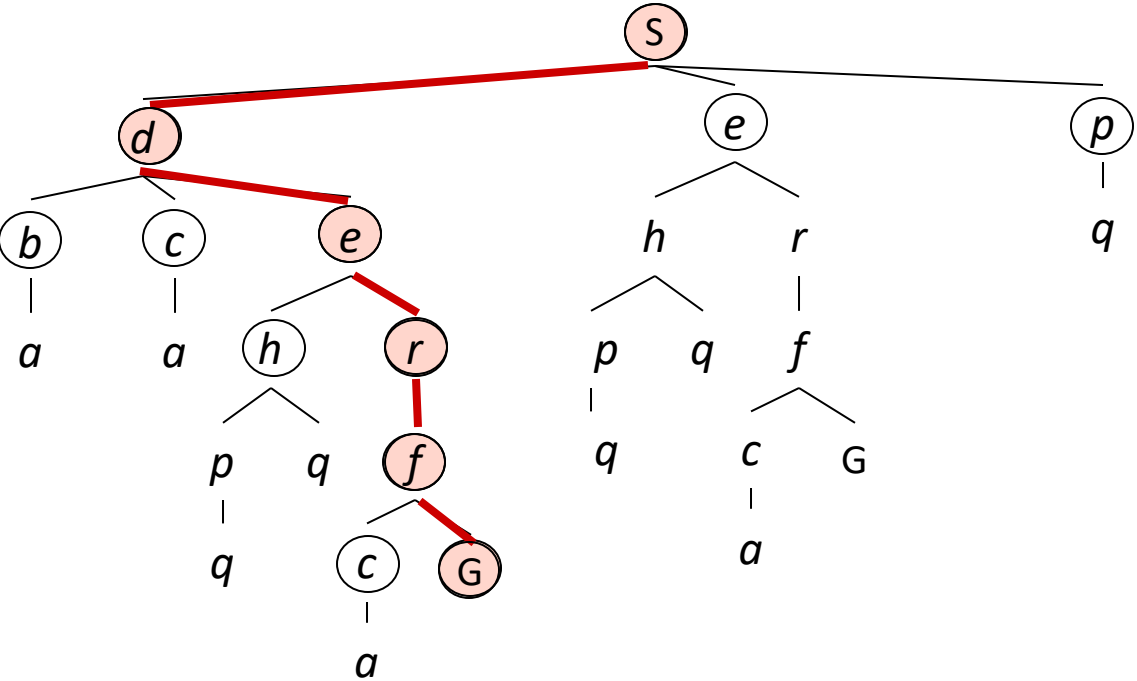
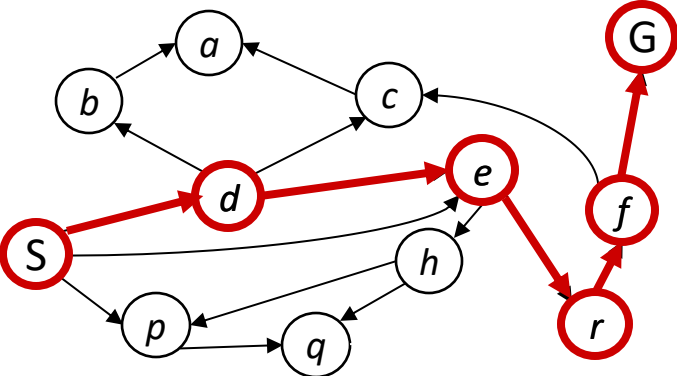
return the corresponding solution

for each resulting `child` from node

add `child` to the `frontier`

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question
  - which fringe nodes to explore?

# Example: Tree Search

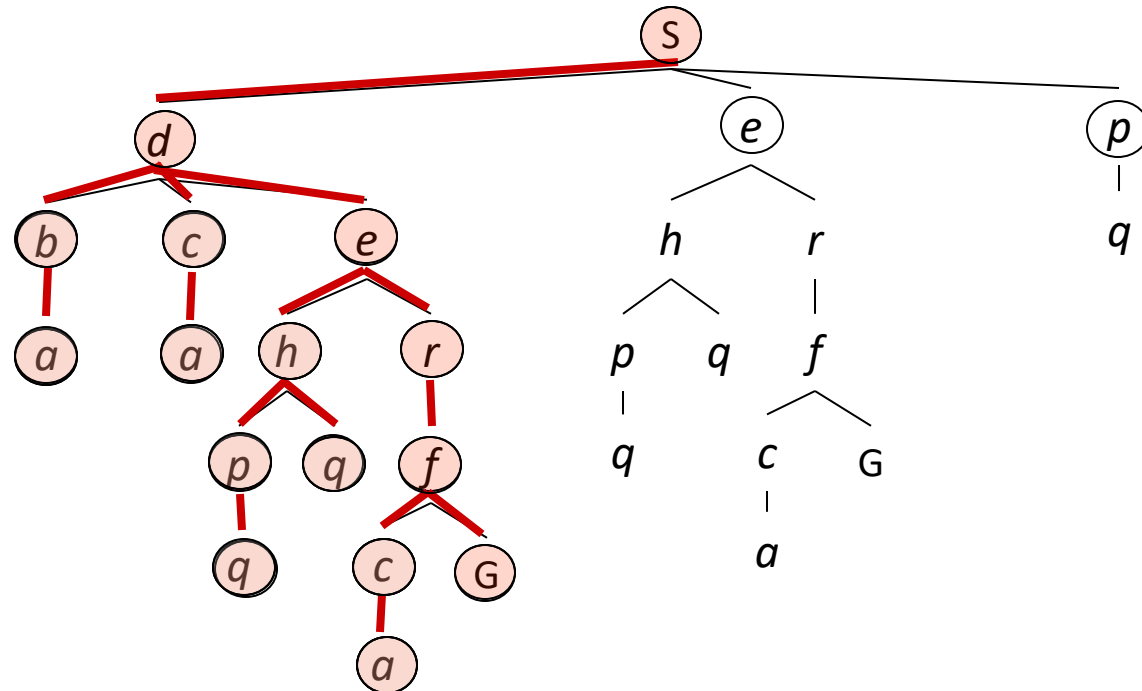
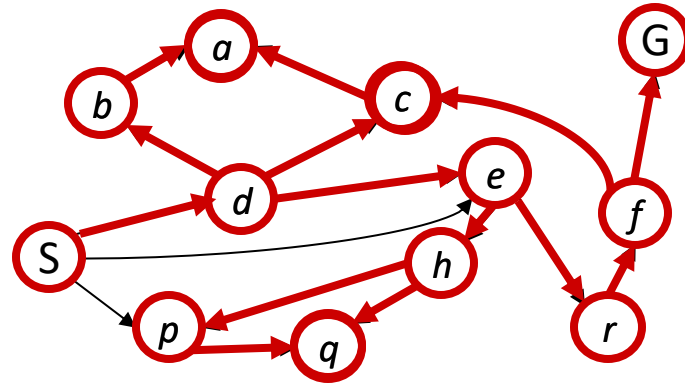


- ~~s~~
- ~~s → d~~
- s → e
- s → p
- s → d → b
- s → d → c
- ~~s → d → e~~
- s → d → e → h
- ~~s → d → e → r~~
- ~~s → d → e → r → f~~
- s → d → e → r → f → c
- ~~s → d → e → r → f → G~~

# Depth-First (Tree) Search

Strategy: expand a  
deepest node first

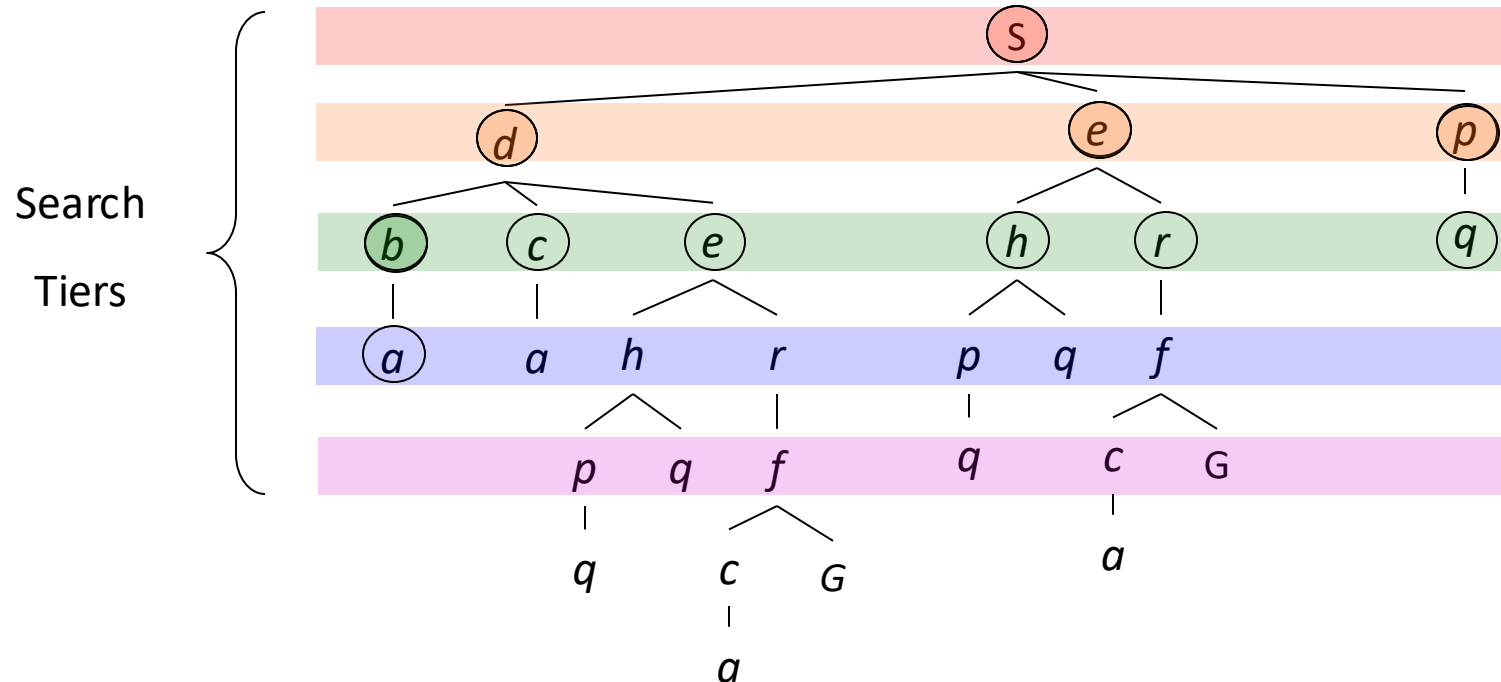
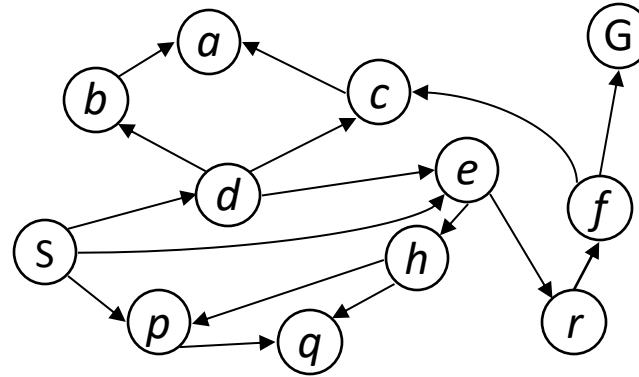
Implementation:  
Fringe is a *LIFO* stack



# Breadth-First (Tree) Search

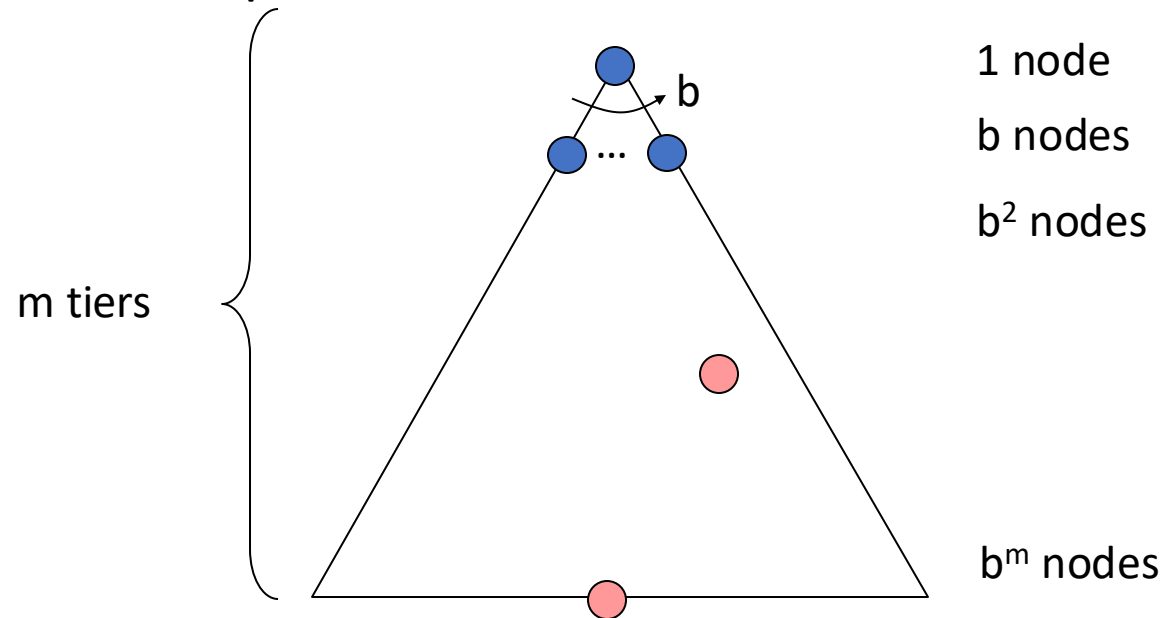
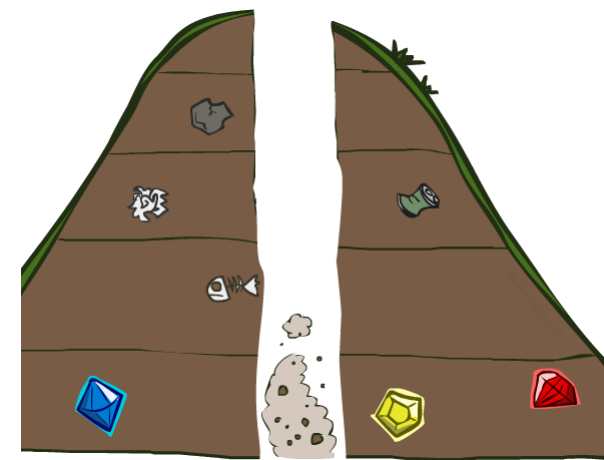
Strategy: expand a shallowest node first

Implementation: Fringe is a *FIFO* queue



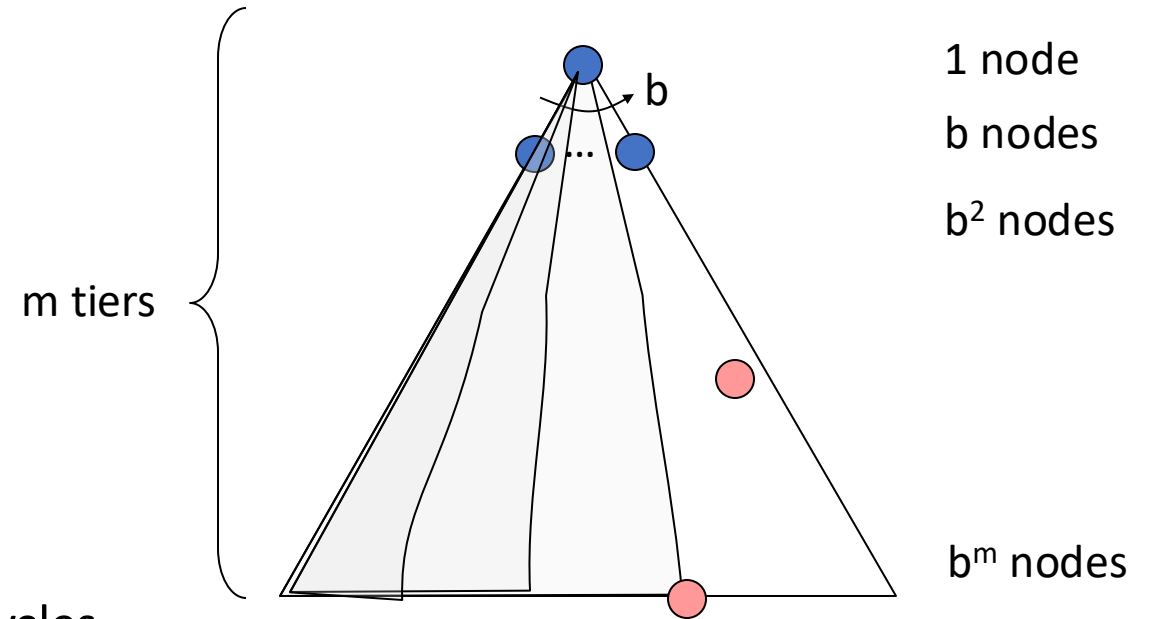
# Search Algorithm Properties

- 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?
  - $1 + b + b^2 + \dots + b^m = O(b^m)$



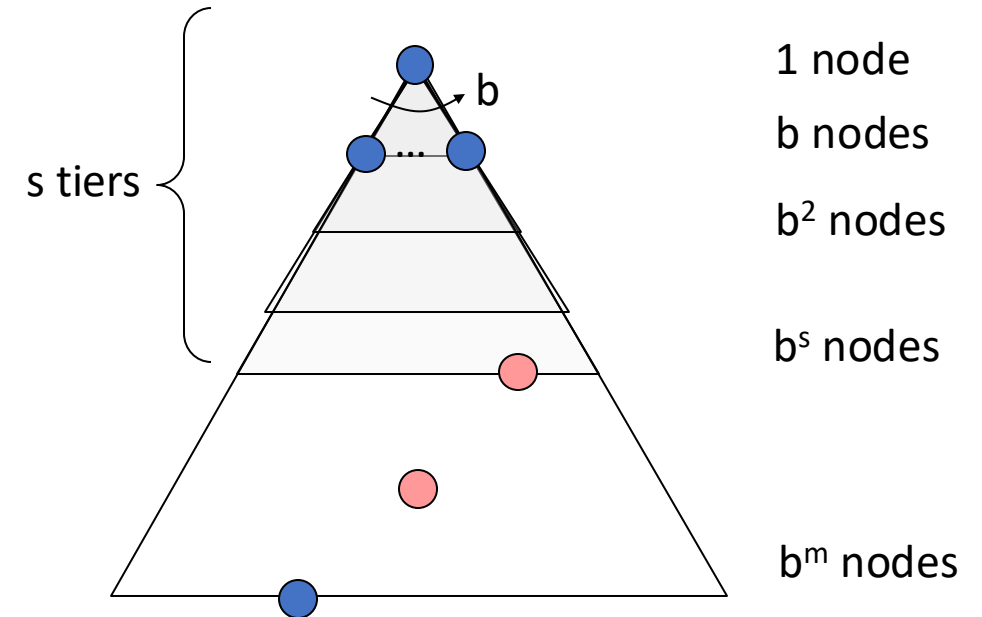
# Depth-First Search (DFS) Properties

- 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



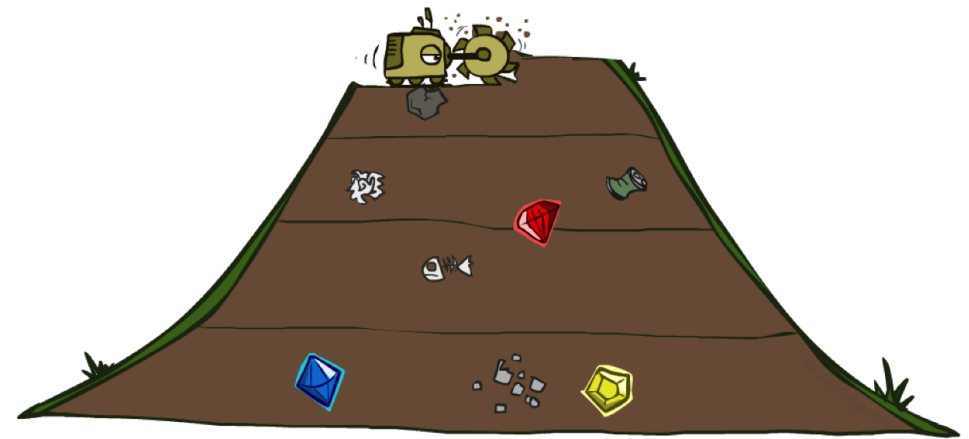
# Breadth-First Search (BFS) Properties

- 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
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



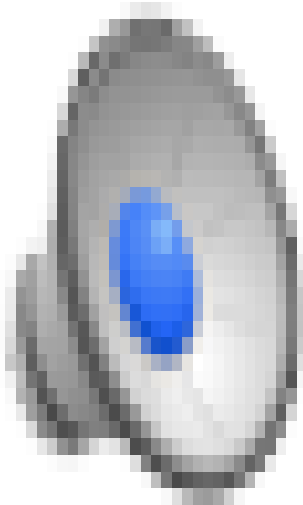
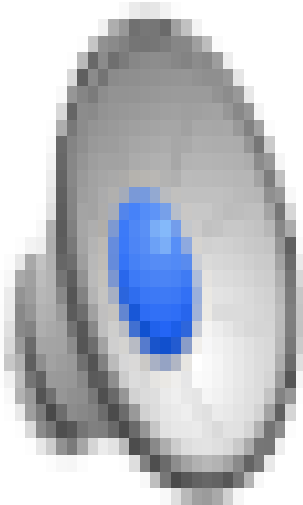


# DFS vs BFS



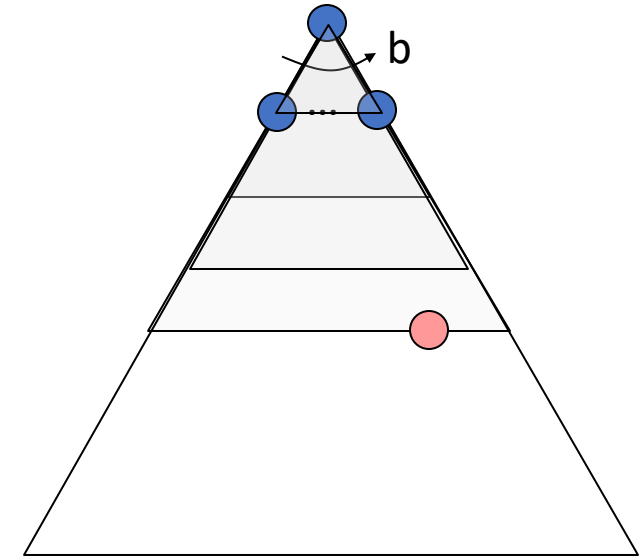
- When will BFS outperform DFS?
- When will DFS outperform BFS?

# Video of Demo Maze Water DFS/BFS



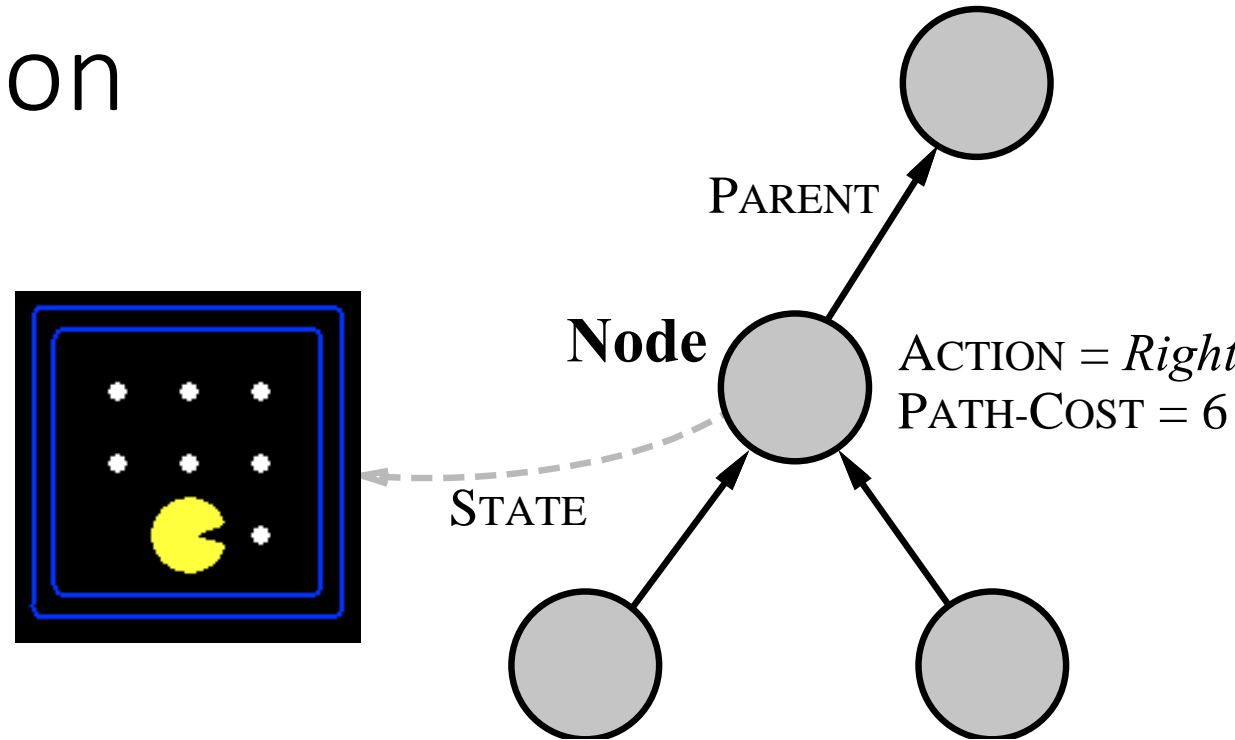
# Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution...
  - Run a DFS with depth limit 2. If no solution...
  - Run a DFS with depth limit 3. ....
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

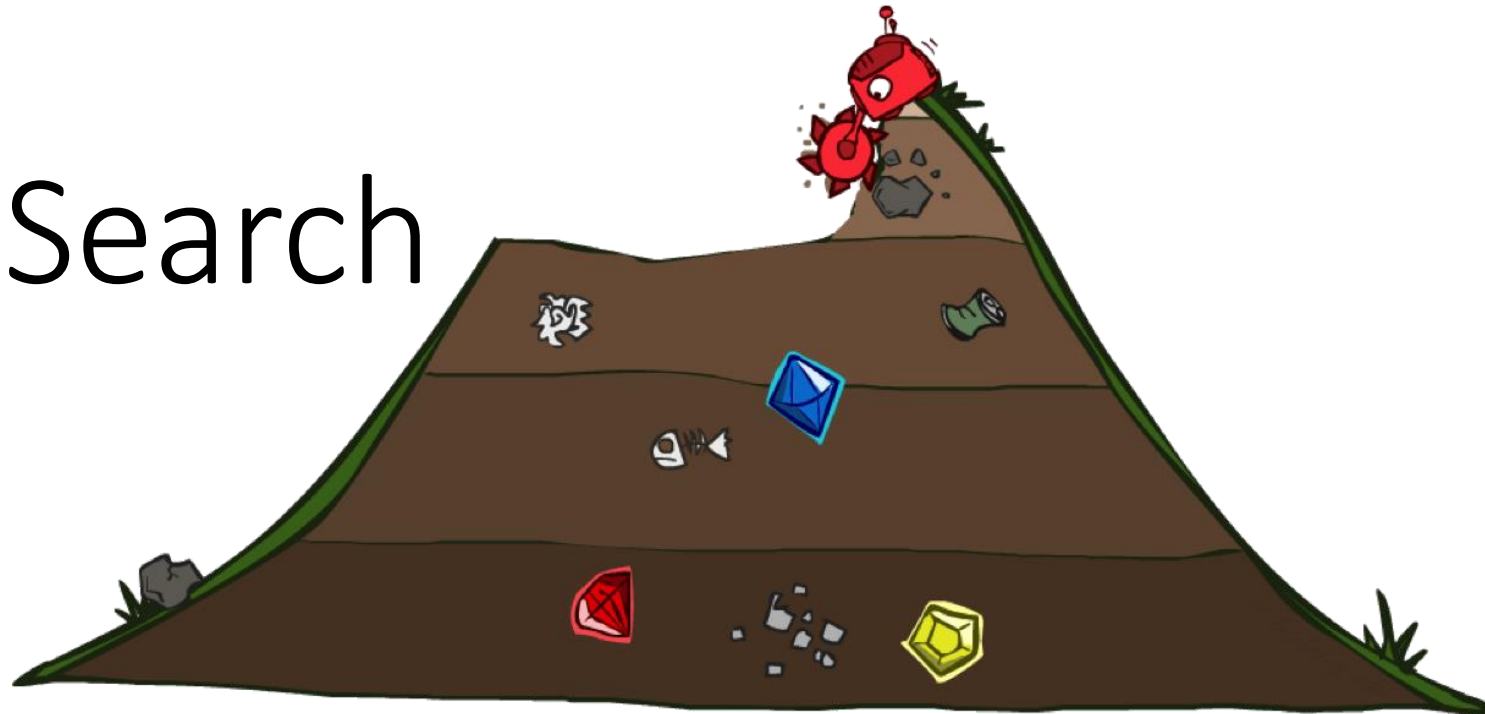


# A Note on Implementation

- Nodes have
  - `state`, `parent`, `action`, `path-cost`
- A child of `node` by action `a` has
  - `state` = `Transition(node.state, a)`
  - `parent` = `node`
  - `action` = `a`
  - `path-cost` = `node.path_cost` + `step_cost(node.state, a, self.state)`
- Extract solution by tracing back parent pointers, collecting actions

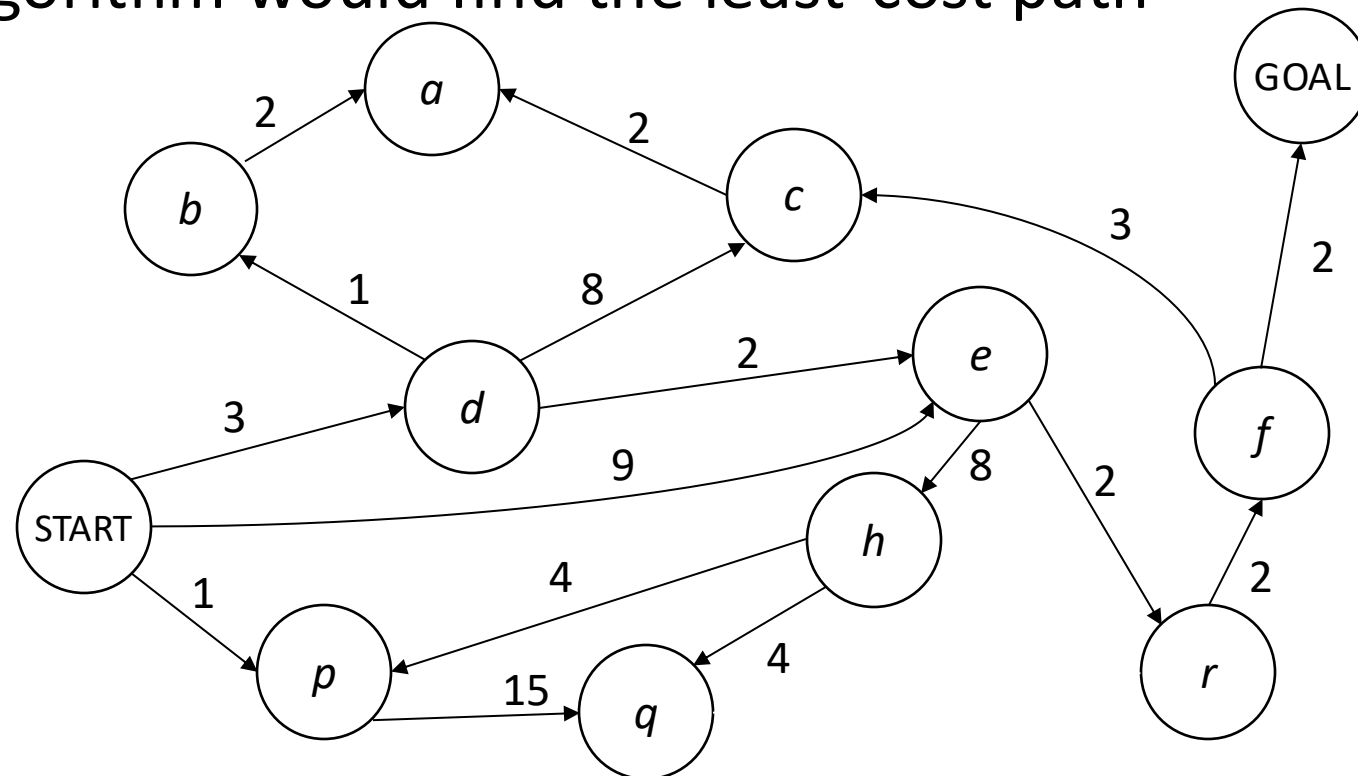


# Uniform Cost Search



# Finding a Least-Cost Path

- BFS finds the shortest path in terms of number of actions, but not the least-cost path
- A similar algorithm would find the least-cost path

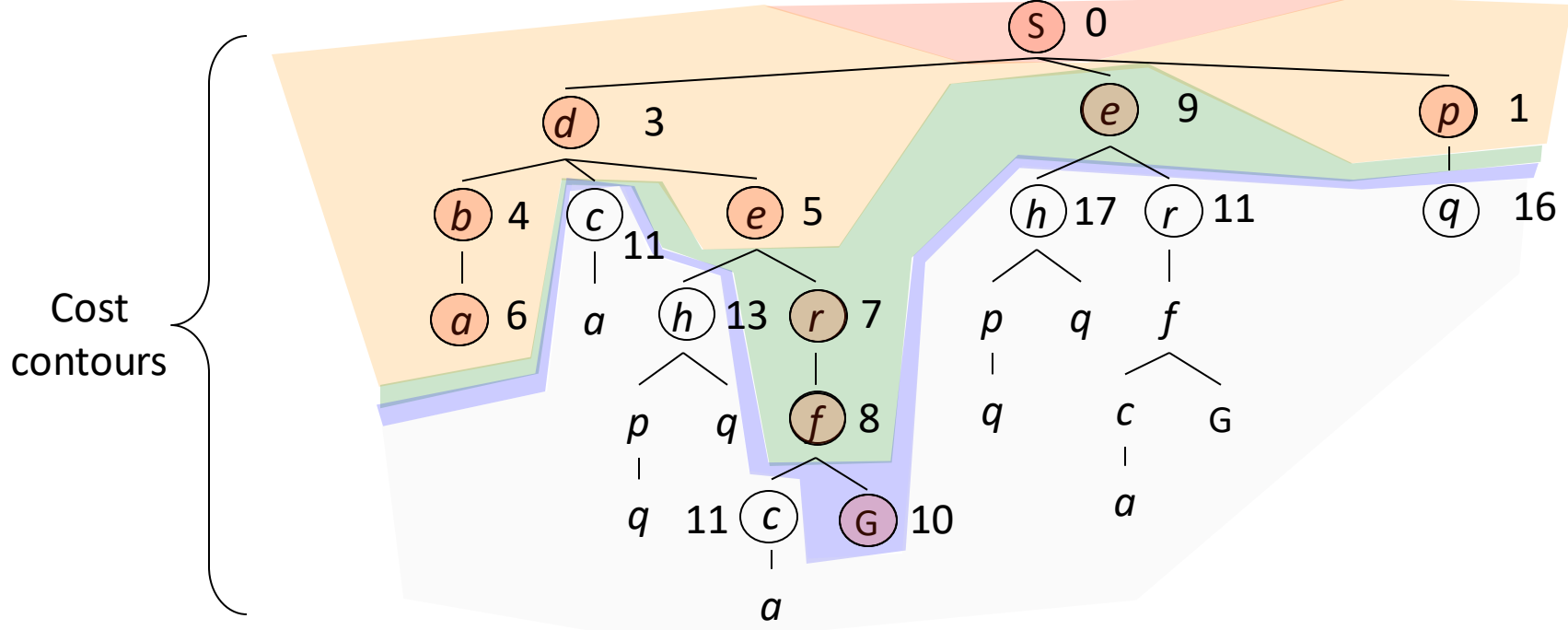
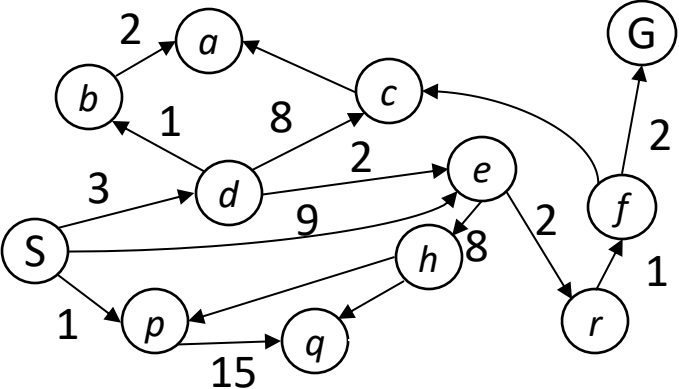


How?

# Uniform Cost Search

Strategy: expand a cheapest node first:

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



# Uniform Cost Search 2

**function** UNIFORM-COST-SEARCH(**problem**) **returns** a solution, or failure

initialize the **frontier** as a **priority queue** using **node's path\_cost** as the **priority**

add initial state of **problem** to **frontier** with **path\_cost = 0**

**loop do**

**if** the **frontier** is empty **then**

**return** failure

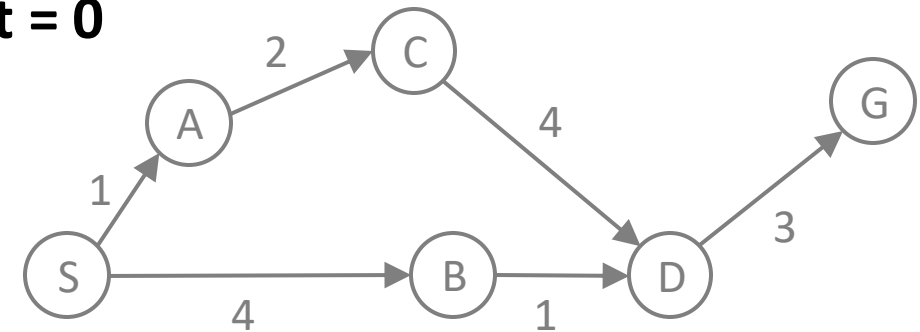
choose a **node** (with minimal **path\_cost**) and remove it from the **frontier**

**if** the **node** contains a goal state **then**

**return** the corresponding solution

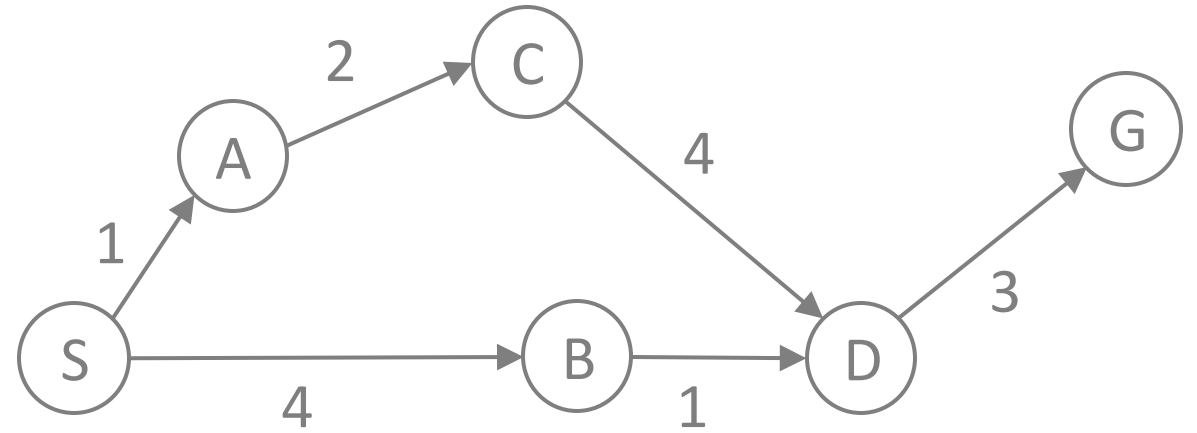
for each resulting **child** from node

add **child** to the **frontier** with **path\_cost = path\_cost(node) + cost(node, child)**

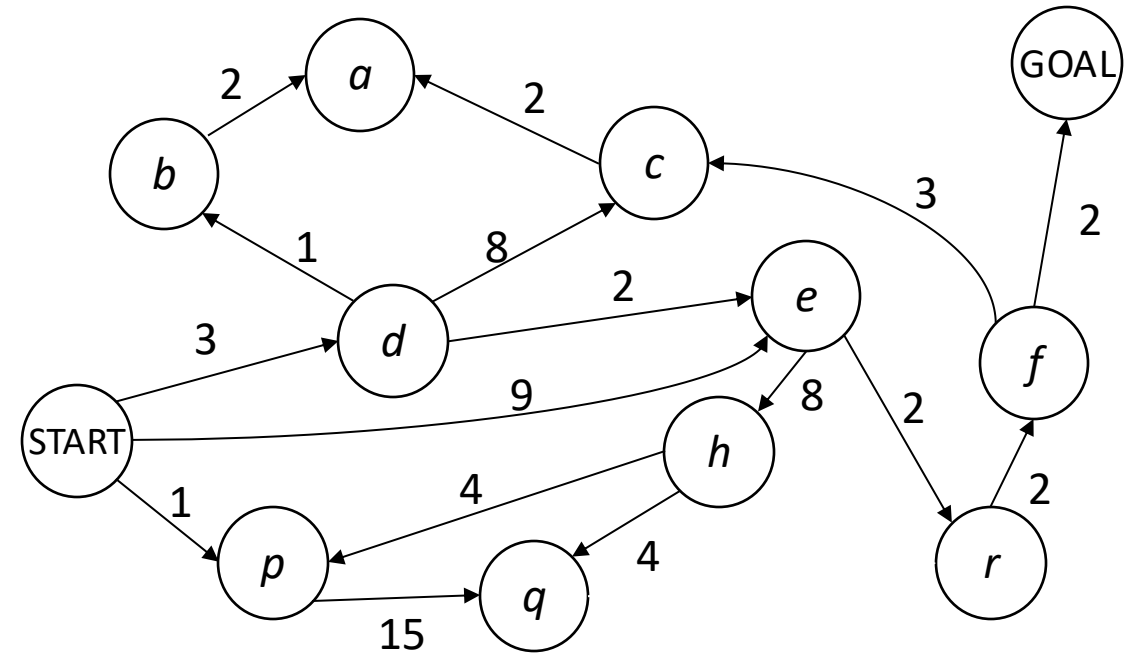




# Walk-through UCS

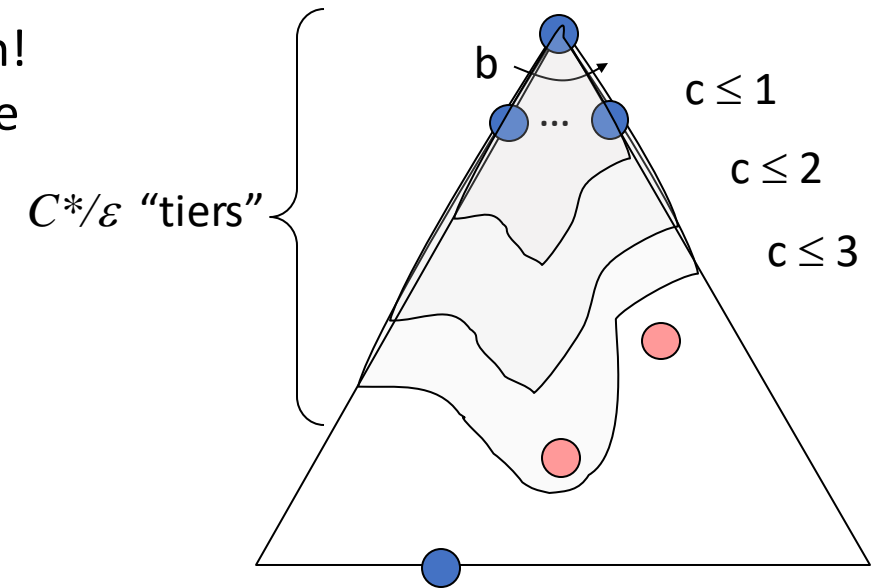


# Walk-through UCS



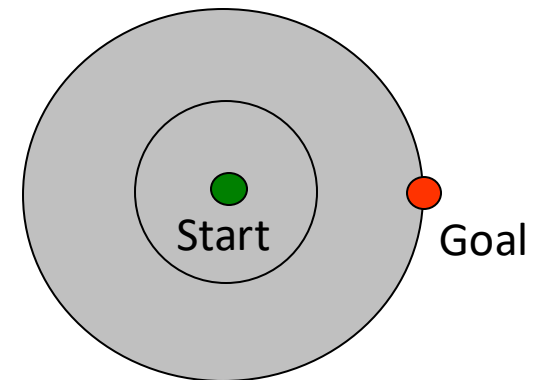
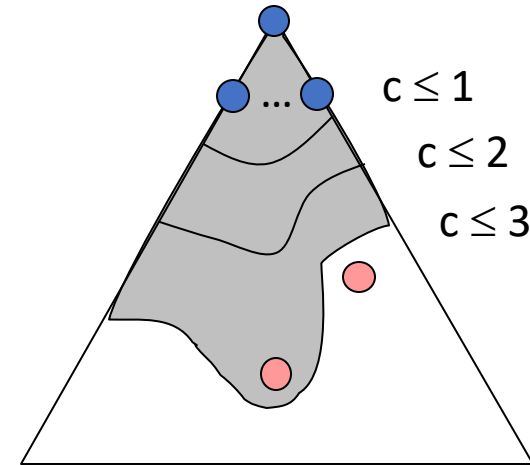
# Uniform Cost Search (UCS) Properties

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



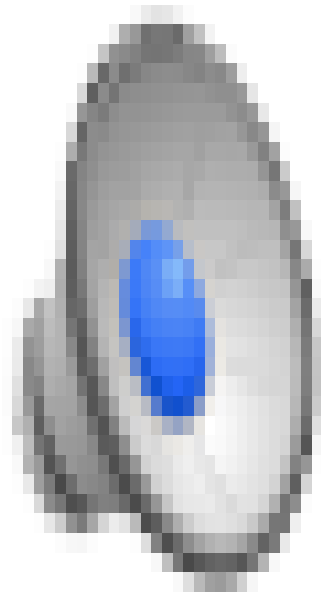
# Uniform Cost Issues

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

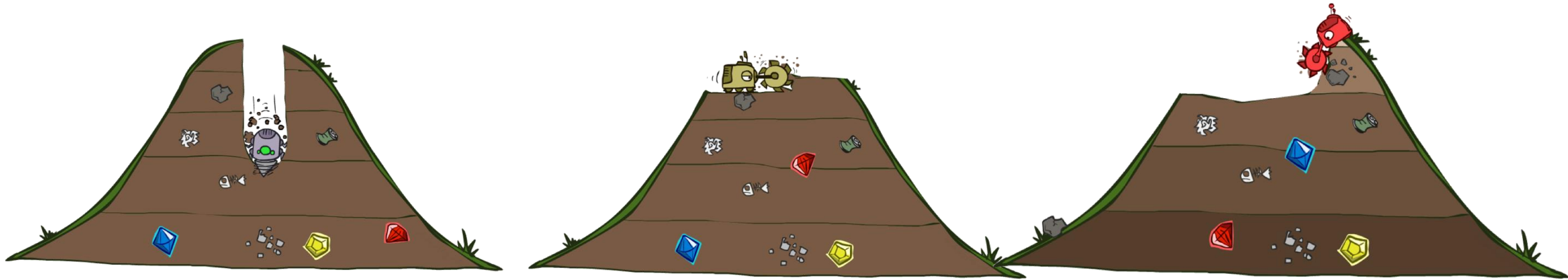


[Demo: empty grid UCS (L2D5)]  
[Demo: maze with deep/shallow  
water DFS/BFS/UCS (L2D7)]<sup>44</sup>

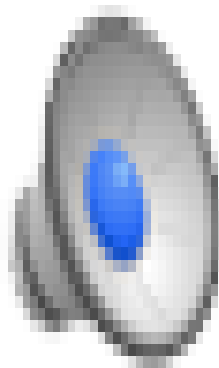
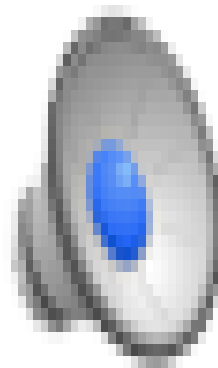
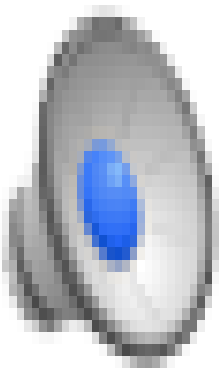
# Video of Demo Empty UCS (same cost)



# DFS, BFS, or UCS?

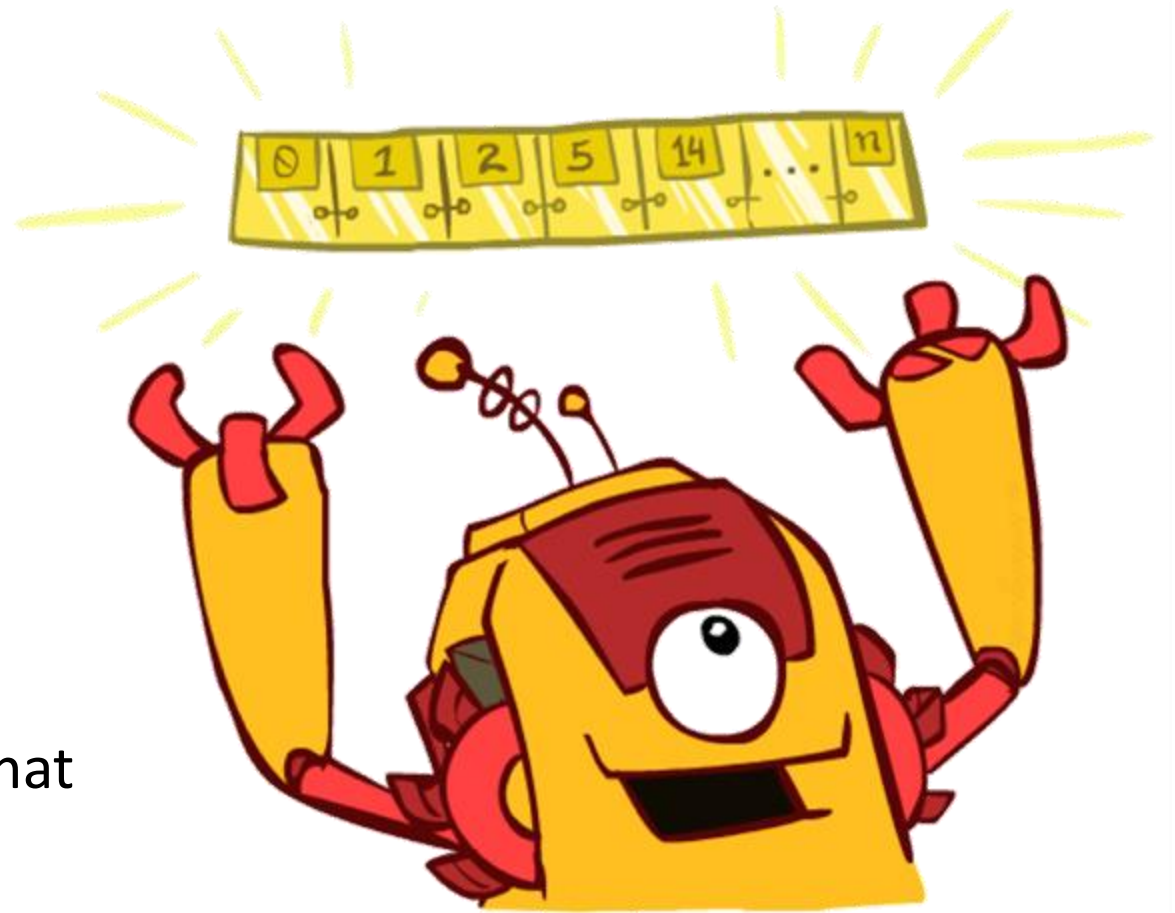


# Video of Demo Maze with Deep/Shallow Water



# The One Queue

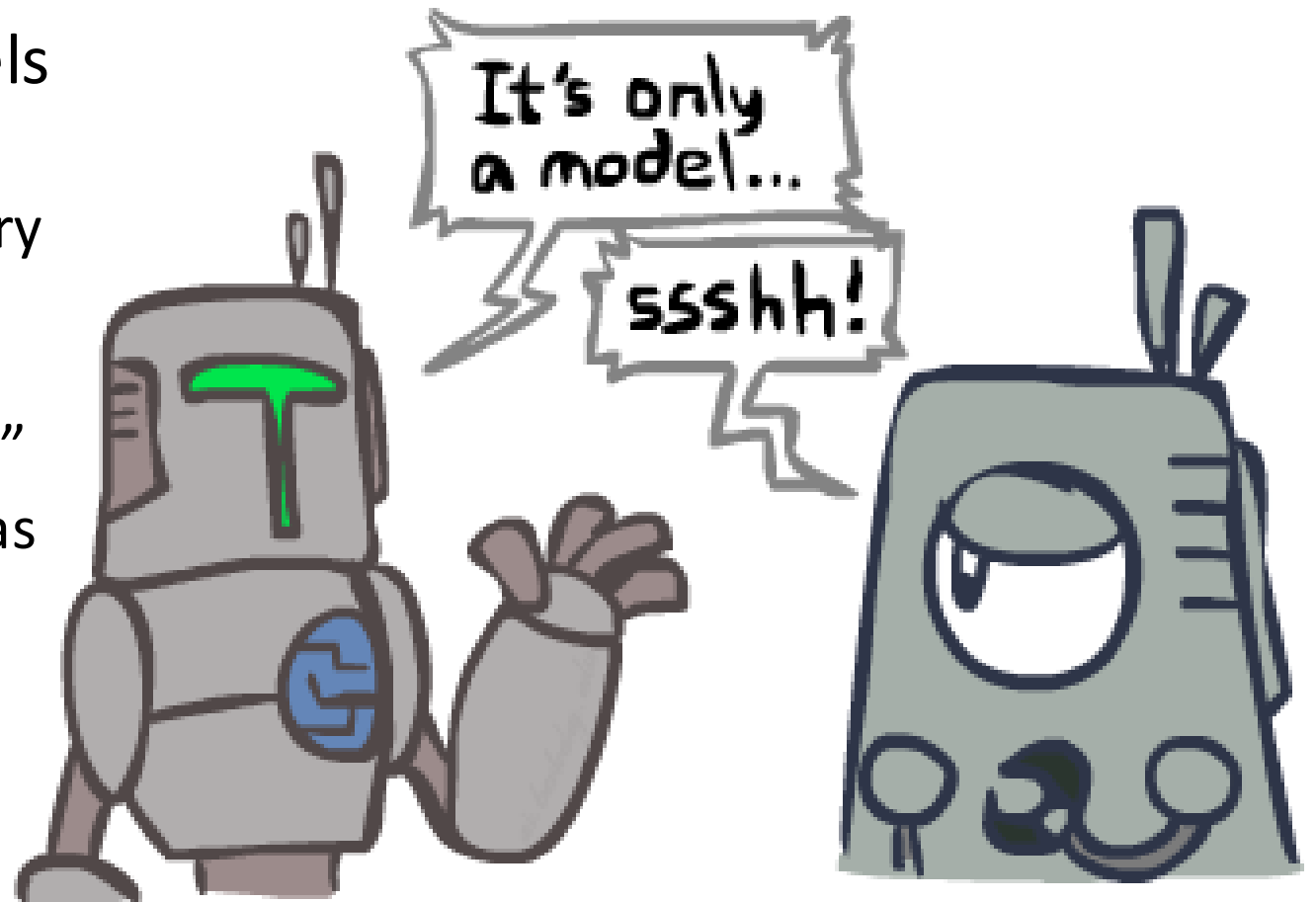
- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the  $\log(n)$  overhead from an actual priority queue, by using stacks and queues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



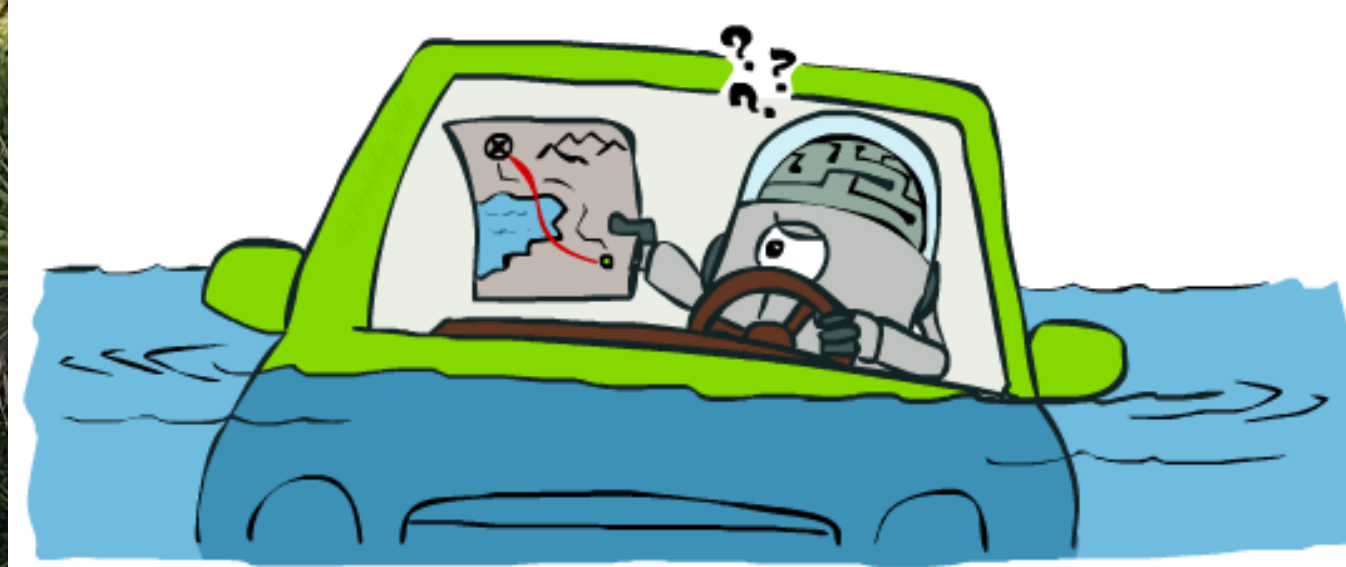
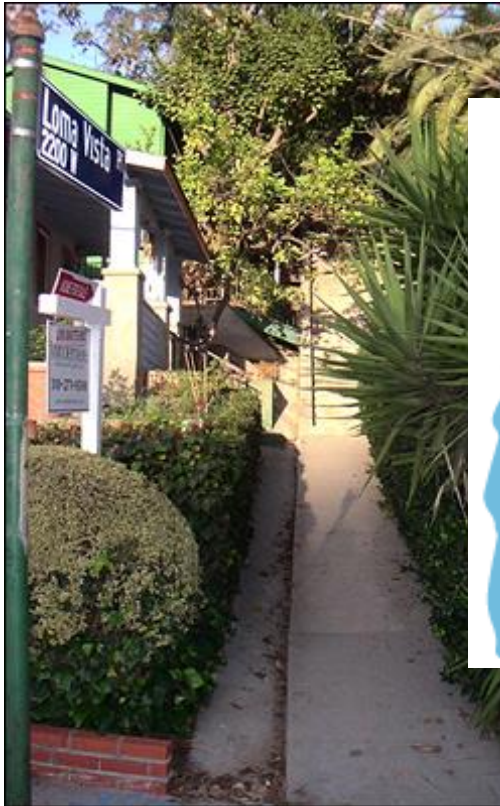


# Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all “in simulation”
  - Your search is only as good as your models...



# Search Gone Wrong?



Microsoft® MapPoint®

NORTH

ARCTIC OCEAN

ICELAND

Start

End

Stockholm

Helsinki

Riga

Vilnius

Berlin

Wrocław

Vienna

Zagreb

Bucharest

SWEDEN

FINLAND

RUSSIA

Poland

Ukraine

Hungary

Romania

Belarus

Czech Rep.

Germany

Italy

km 500 1000

mi 200 400 600

Legend

Zoom on map click

Start: Haugesund, Rogaland, Norway

End: Trondheim, Sør-Trøndelag, Norway

Total Distance: 2713.2 Kilometers

Estimated Total Time: 47 hours, 31 minutes

nrk.no/alltidmoro

# Informed Search

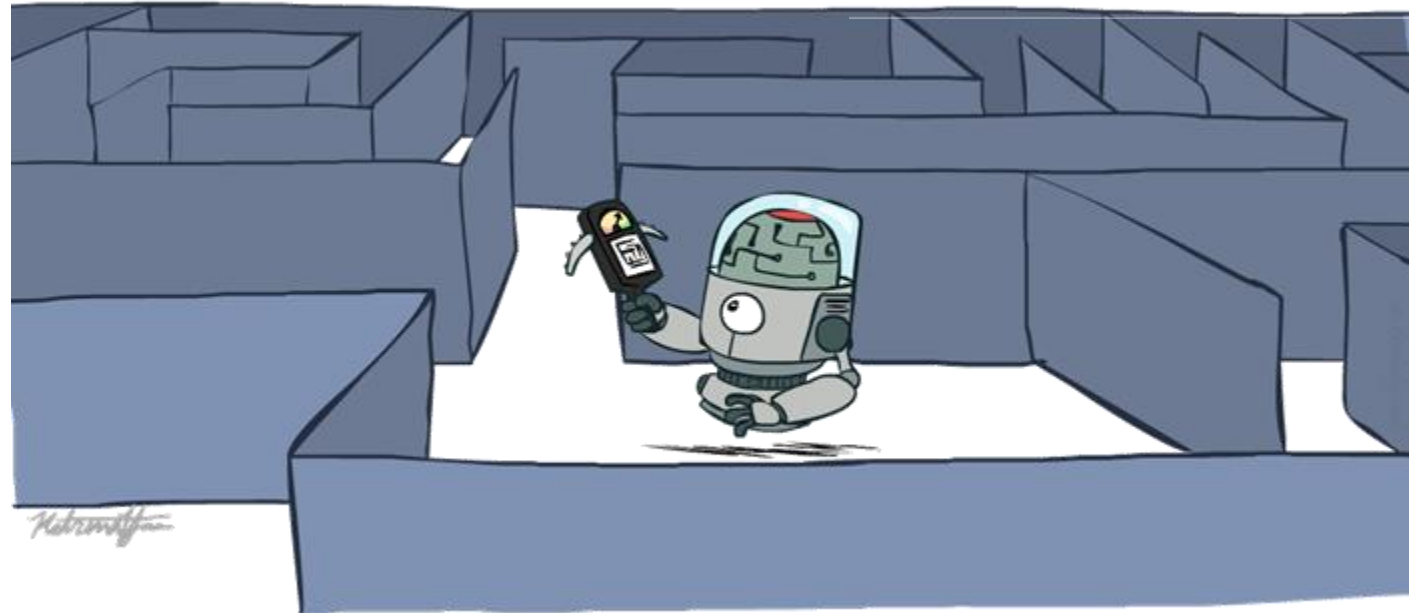
# Informed Search

- Uninformed Search
  - DFS
  - BFS
  - UCS



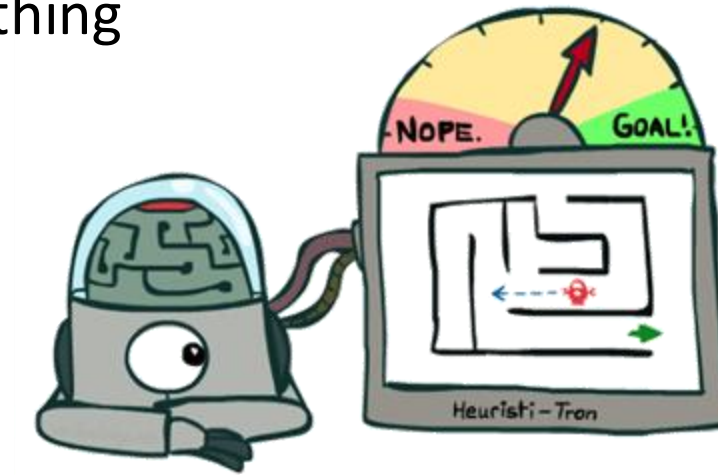
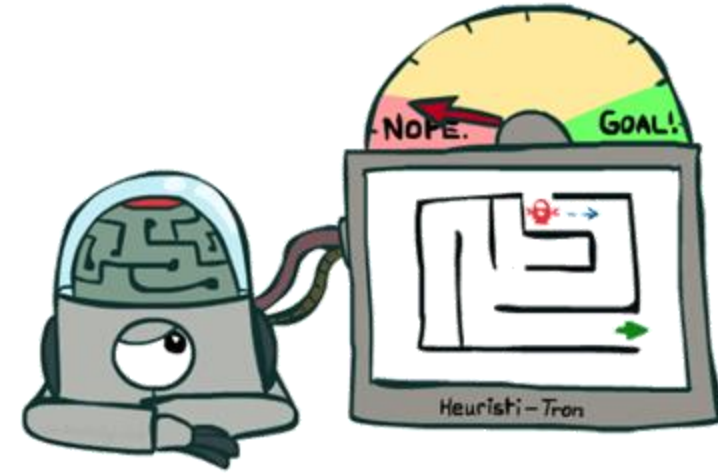
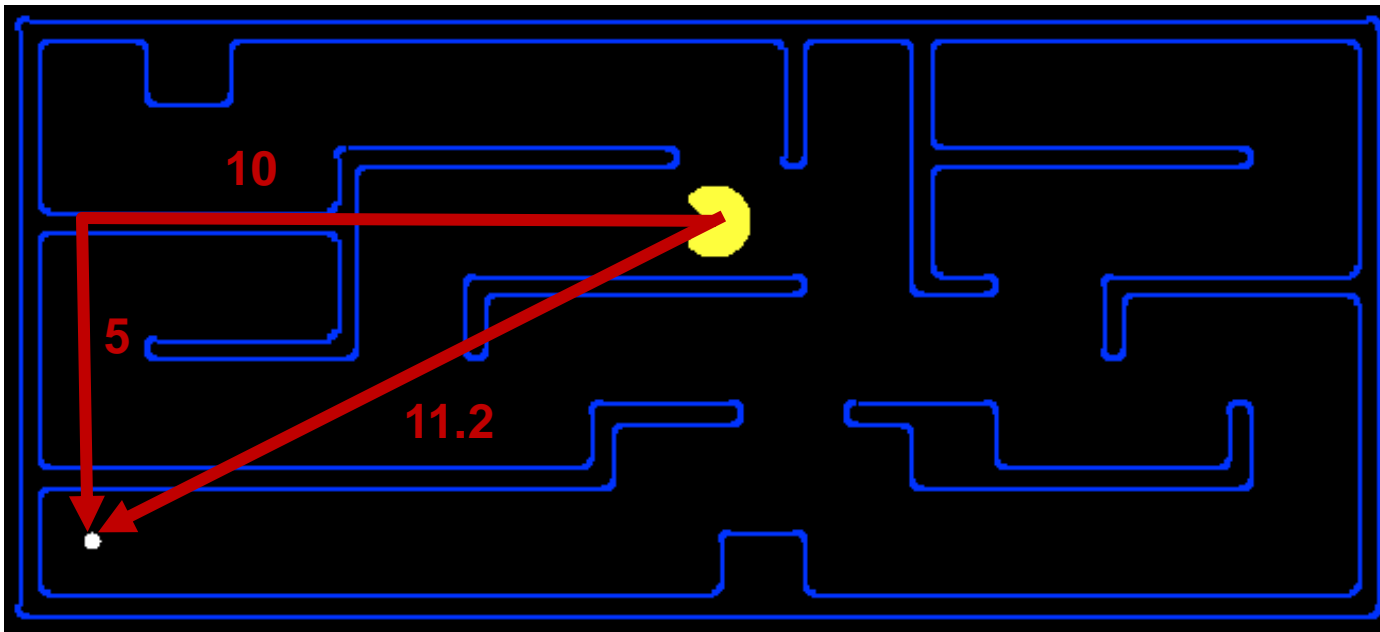
www.shutterstock.com · 149559782

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

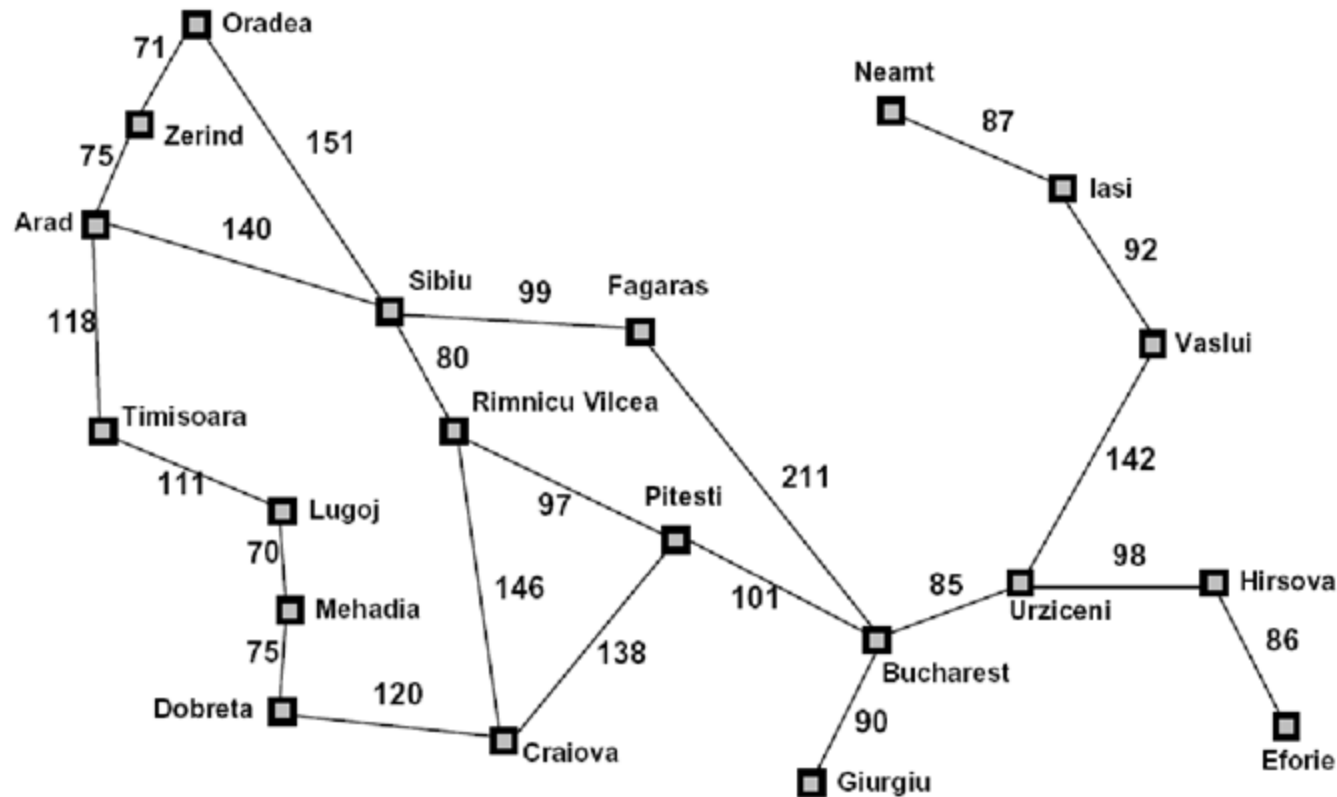


# Search Heuristics

- A heuristic is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - [Pathing?](#)
  - Examples: Manhattan distance, Euclidean distance for pathing



# Example: Heuristic Function (Euclidean distance to Bucharest)

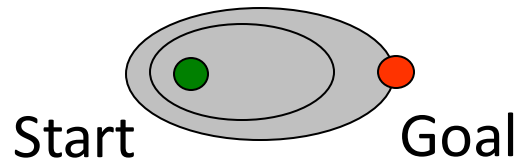


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

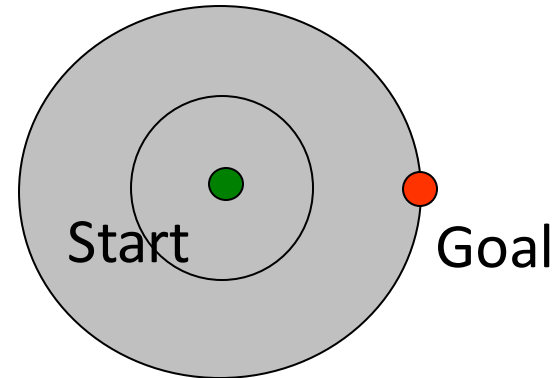
$h(\text{state}) \rightarrow \text{value}$

# Effect of heuristics

- Guide search *towards the goal* instead of *all over the place*



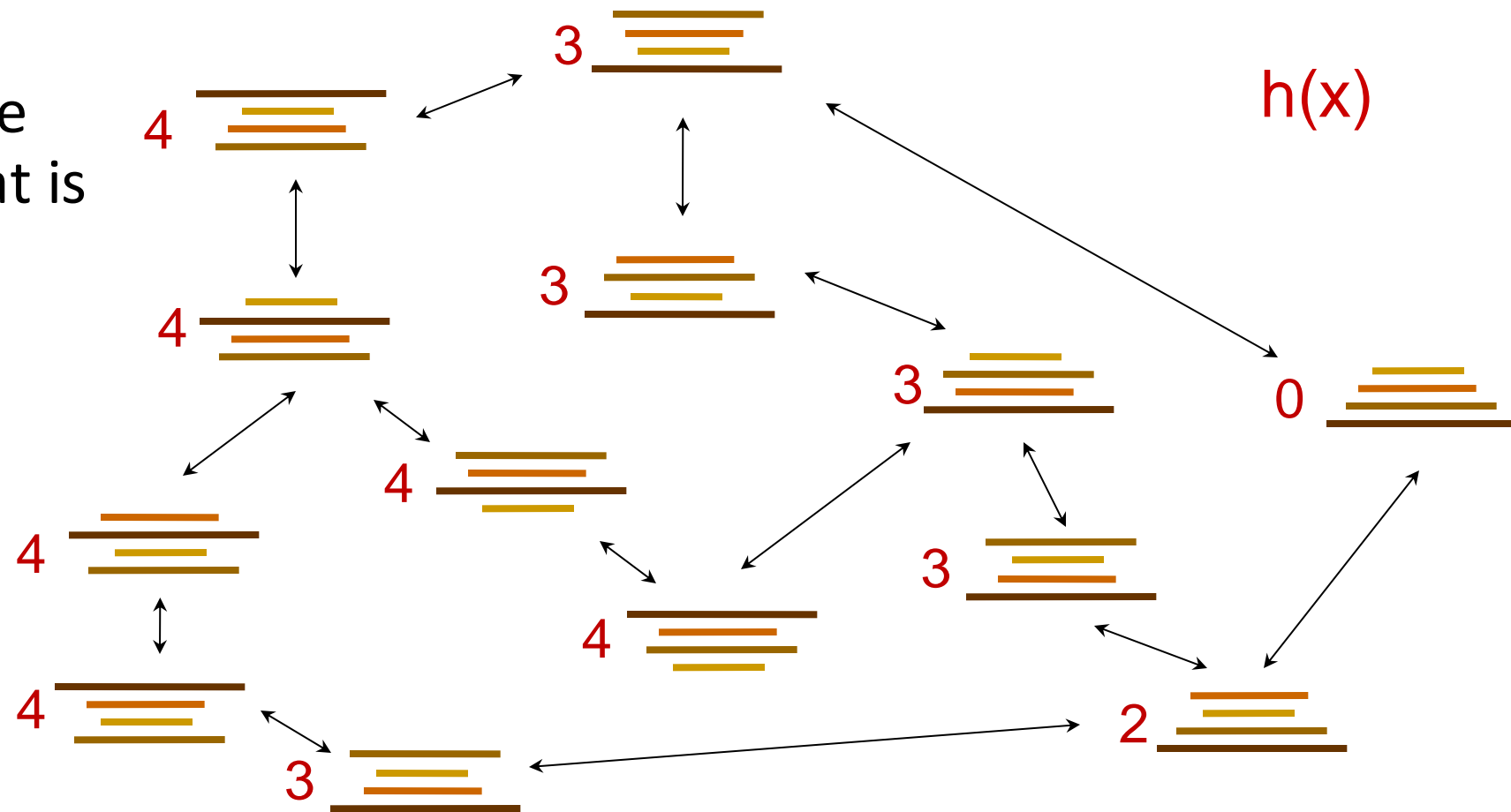
Informed



Uninformed

# Example: Heuristic Function 2

- Heuristic?
- E.g. the index of the largest pancake that is still out of place



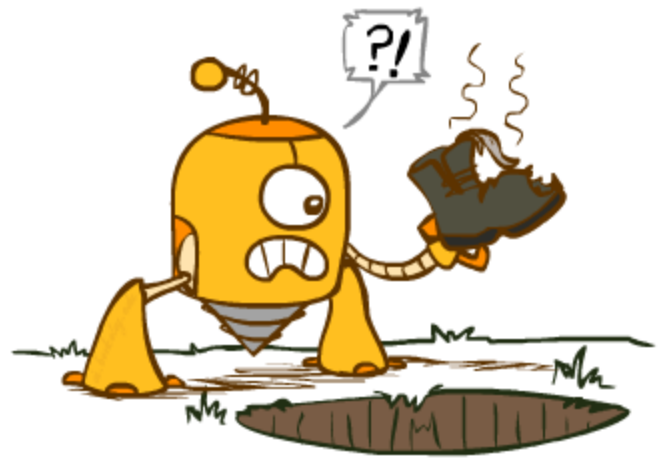
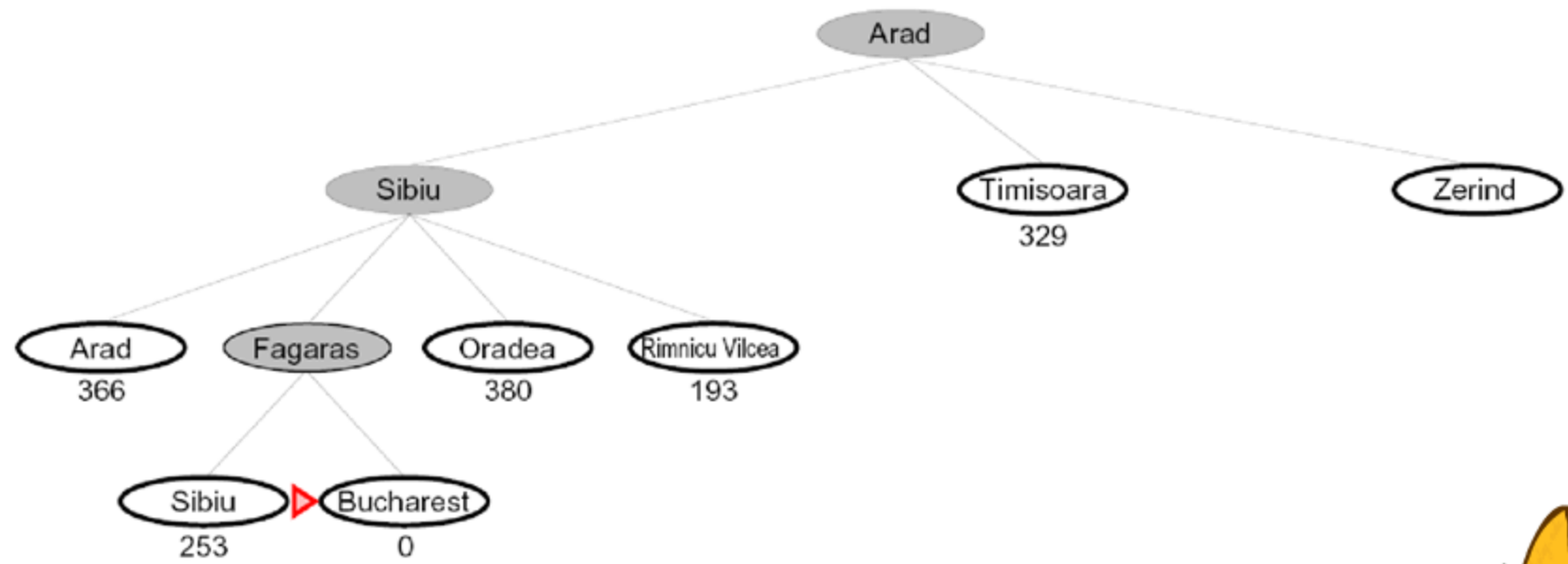
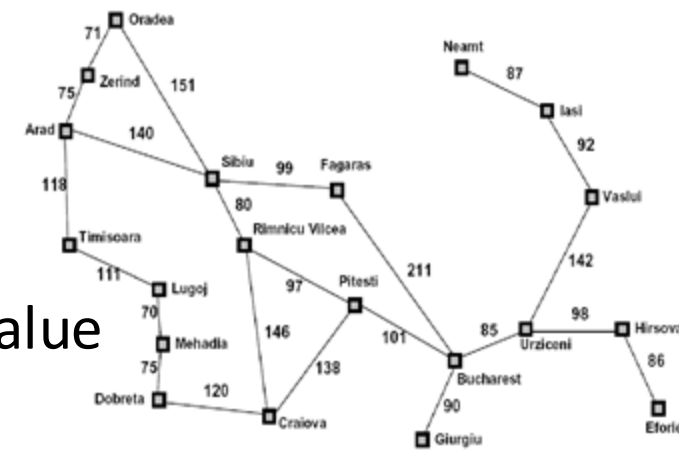


# Greedy Search



# Greedy Search

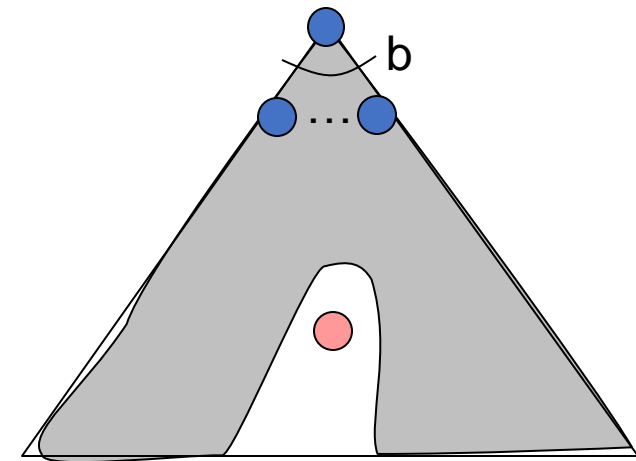
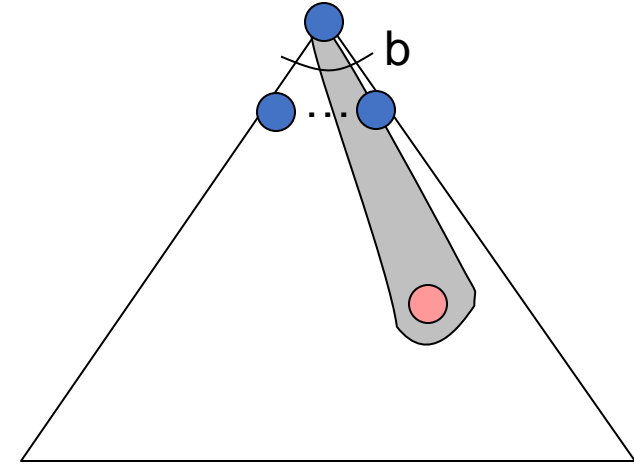
- Expand the node that seems closest to the goal, or least  $h(n)$  value



- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!
  - Why?
  - Heuristics might be wrong

# Greedy Search 2

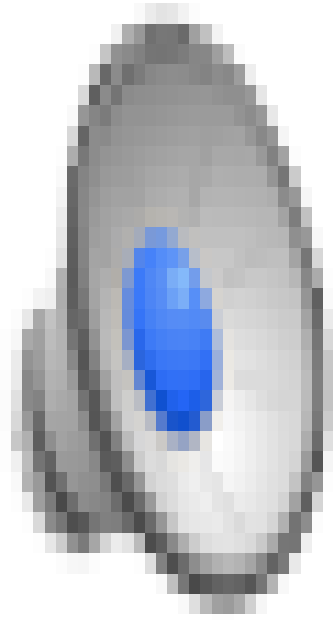
- Strategy: expand a node that **you think** is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state
- A common case:
  - Best-first takes you straight to the (wrong) goal
  - (It chooses a node even if it's at the end of a very long and winding road)
- Worst-case: like a badly-guided DFS
  - (It takes  $h$  literally even if it's completely wrong)



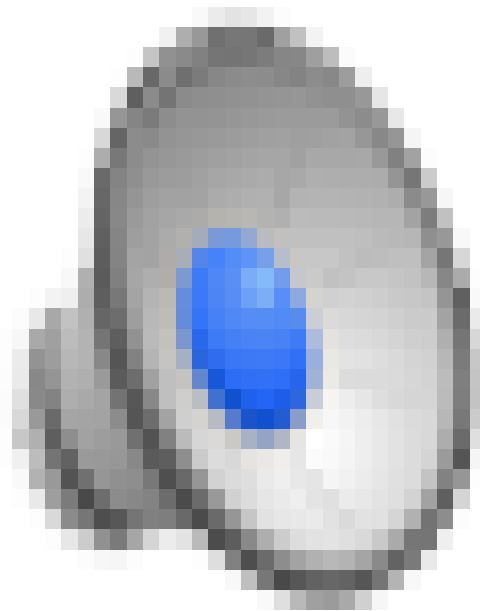
[Demo: contours greedy empty (L3D1)]

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

# Video of Demo Contours Greedy (Empty)



# Video of Demo Contours Greedy (Pacman Small Maze)



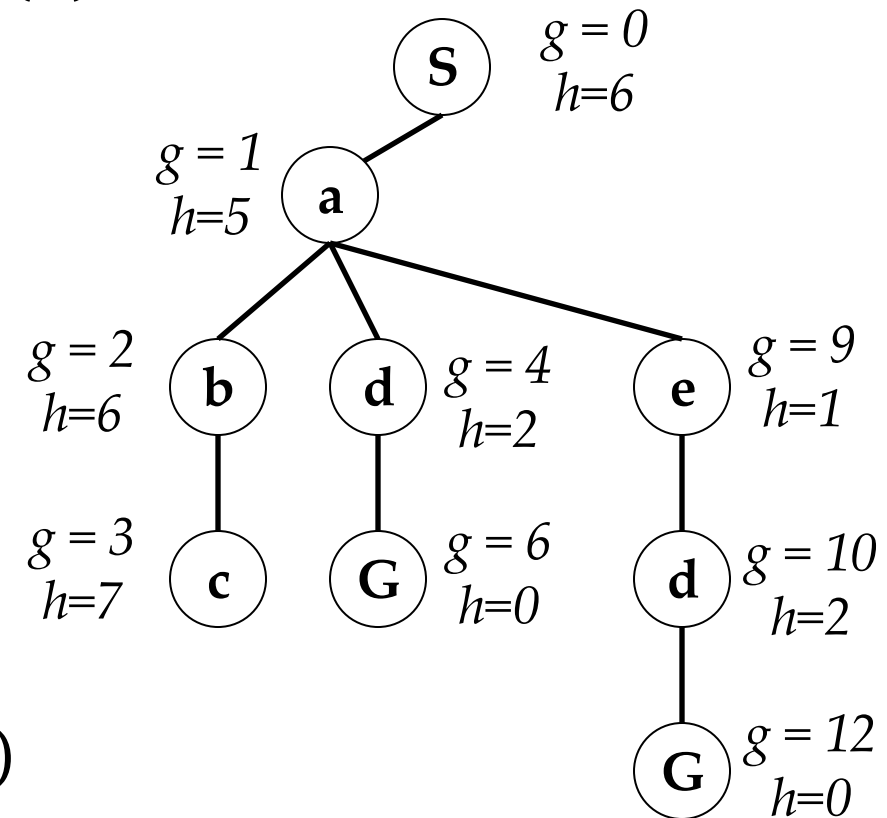
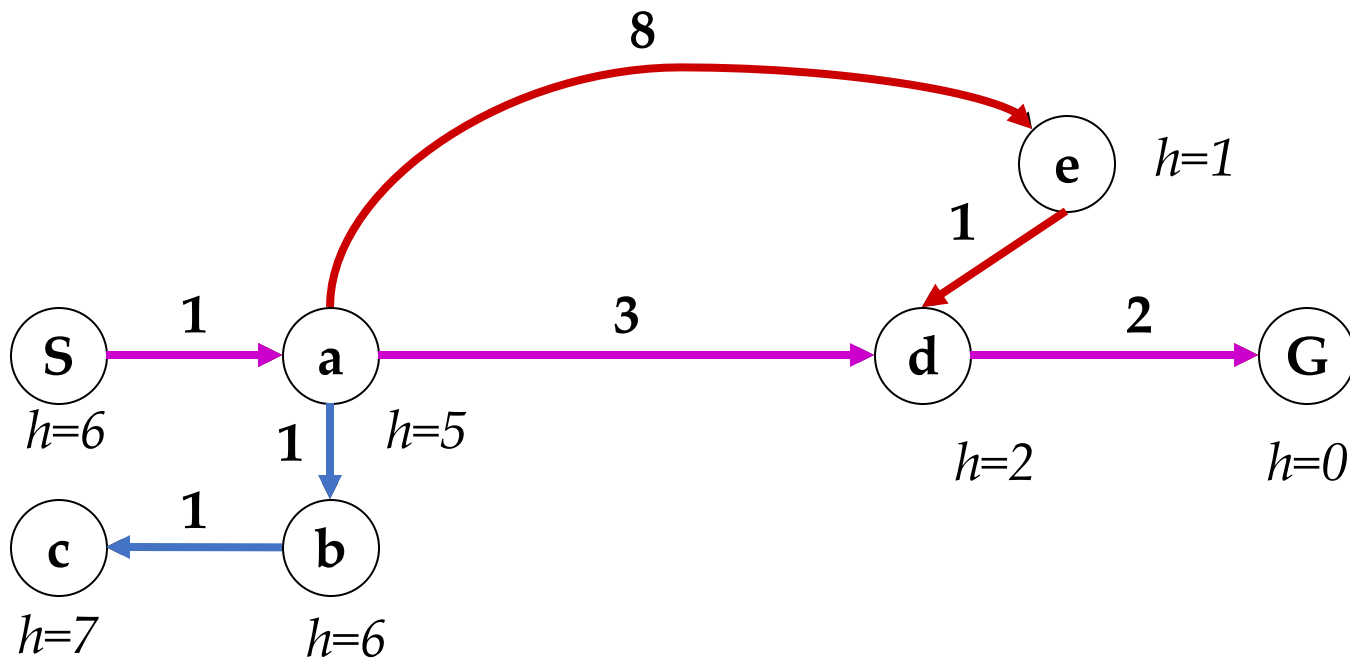
# A\* Search



# 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*  $h(n)$

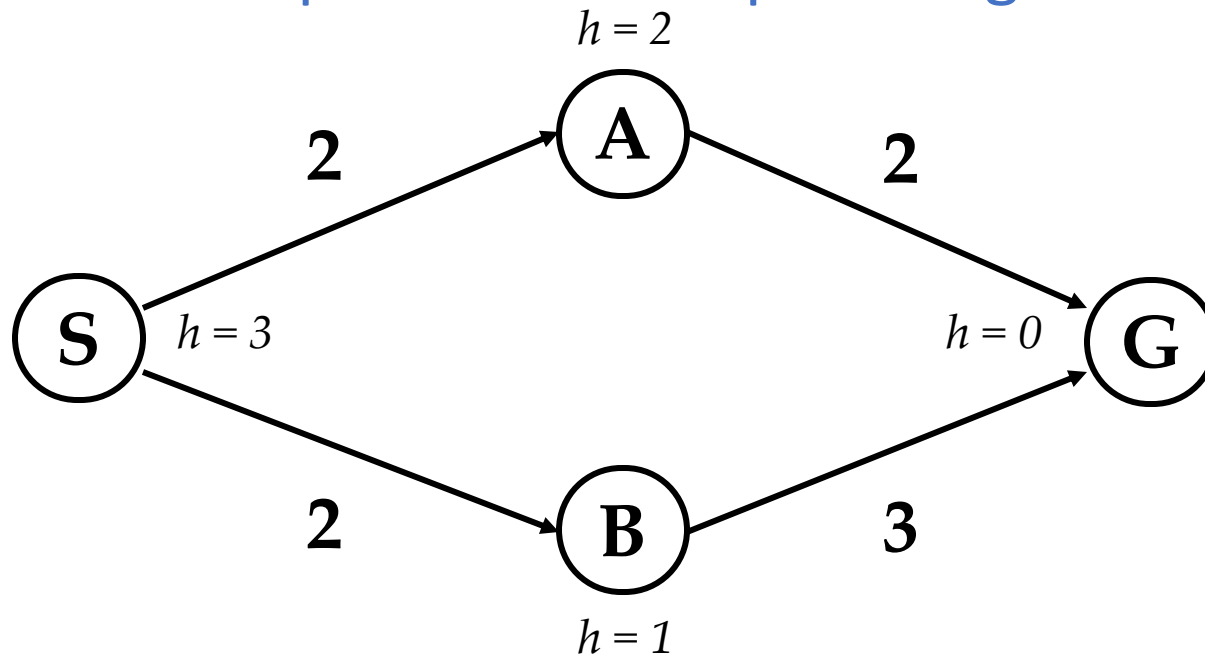


- **A\* Search** orders by the sum:  $f(n) = g(n) + h(n)$



# When should A\* terminate?

- Should we stop when we enqueue a goal?



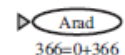
- No: only stop when we dequeue a goal

	g	h	+
<del>S</del>	<del>0</del>	<del>3</del>	<del>3</del>
<del>S-&gt;A</del>	<del>2</del>	<del>2</del>	<del>4</del>
<del>S-&gt;B</del>	<del>2</del>	<del>1</del>	<del>3</del>
S->B->G	5	0	5
S->A->G	4	0	4

# A\* Search

```
function A-STAR-SEARCH(problem) returns a solution, or failure
  initialize the frontier as a priority queue using  $f(n)=g(n)+h(n)$  as the priority
  add initial state of problem to frontier with priority  $f(S)=0+h(S)$ 
  loop do
    if the frontier is empty then
      return failure
    choose a node and remove it from the frontier
    if the node contains a goal state then
      return the corresponding solution
    for each resulting child from node
      add child to the frontier with  $f(n)=g(n)+h(n)$ 
```

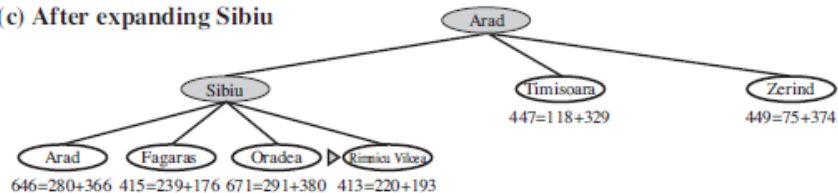
(a) The initial state



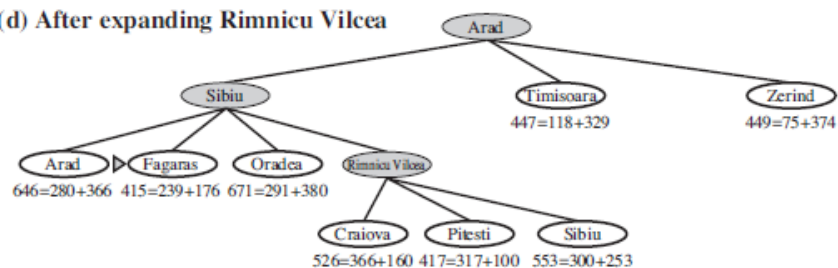
(b) After expanding Arad



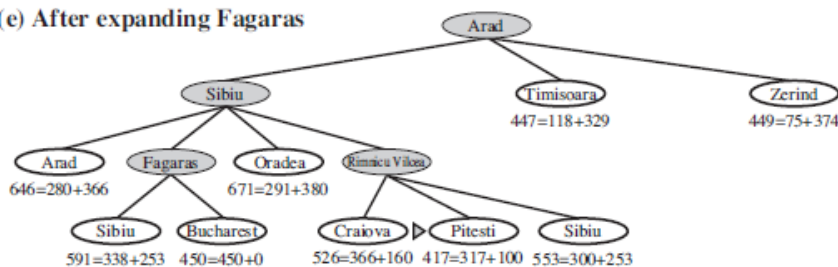
(c) After expanding Sibiu



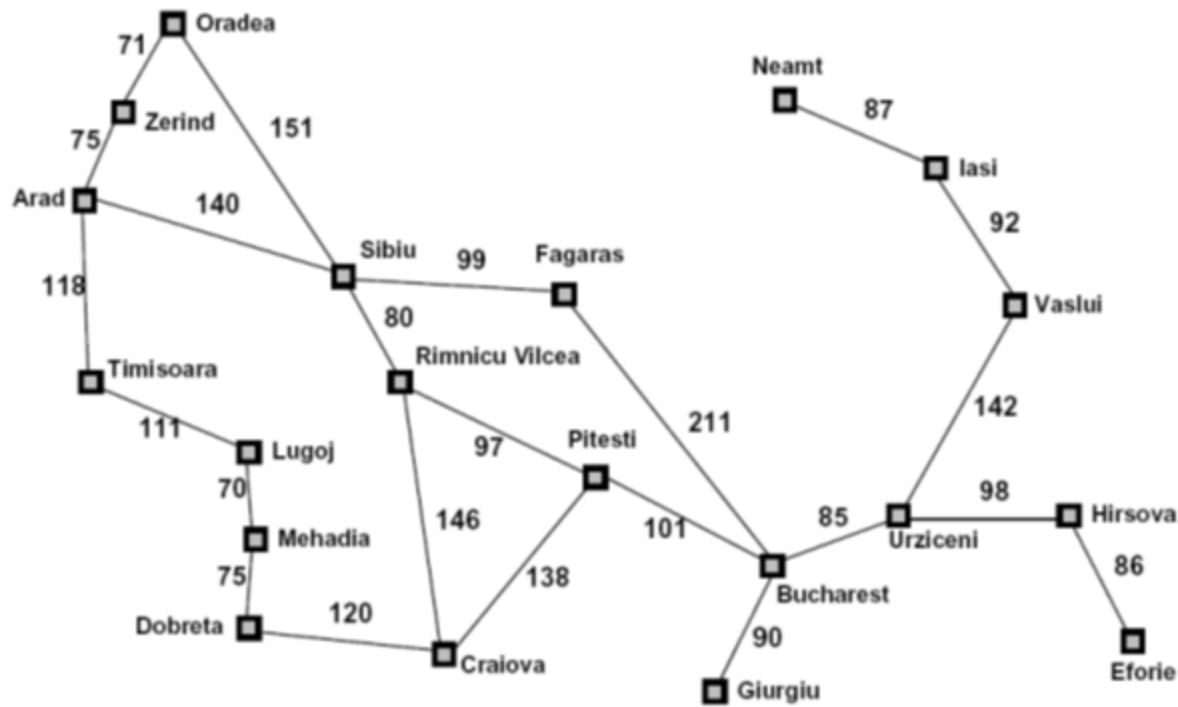
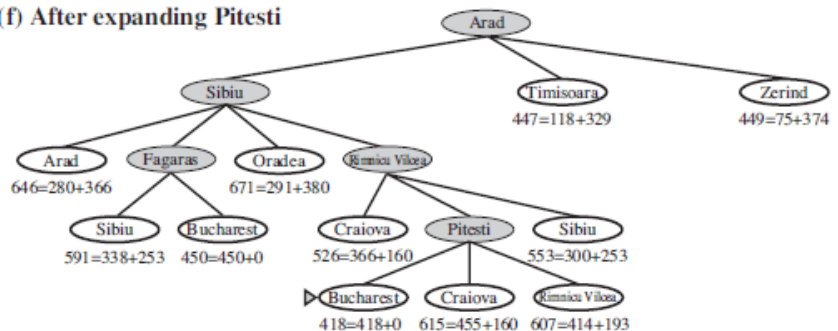
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras

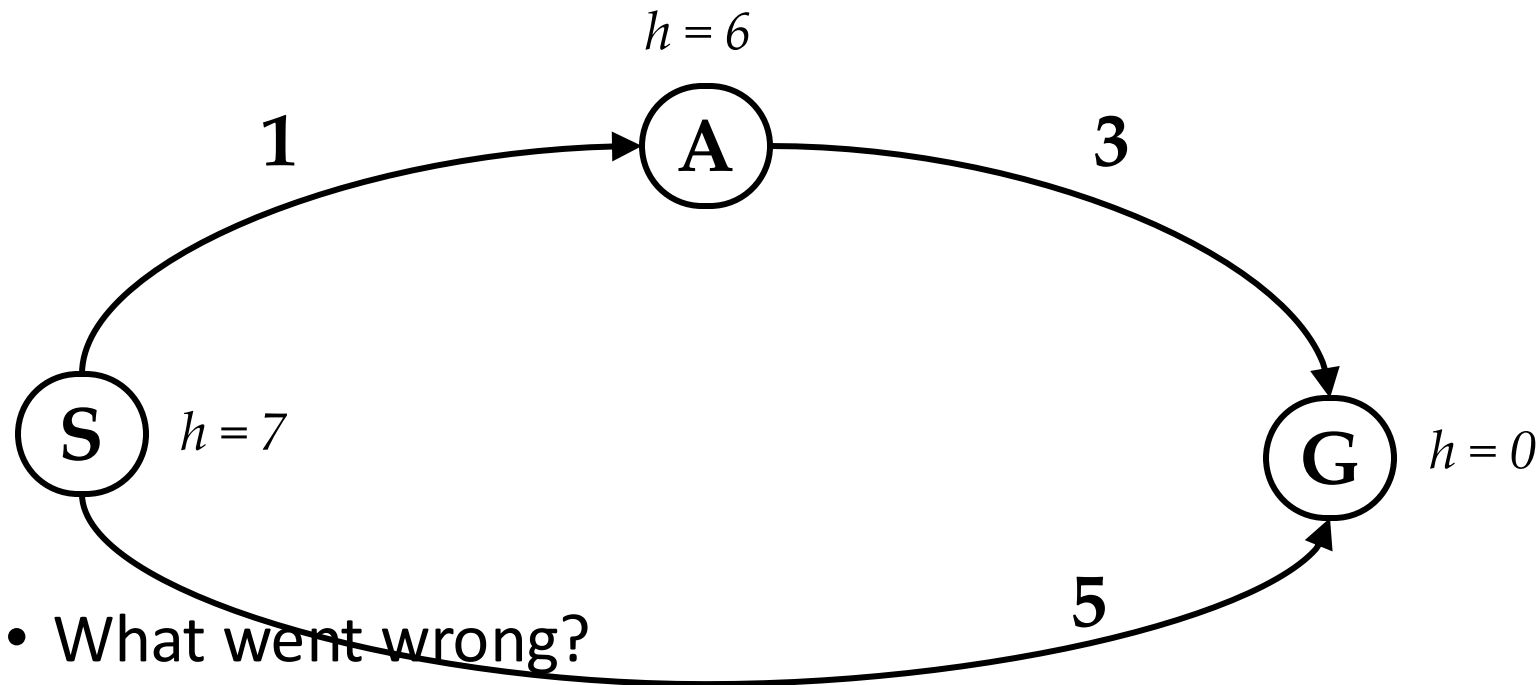


(f) After expanding Pitesti



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

# Is A\* Optimal?



	g	h	+
<del>S</del>	<del>0</del>	<del>7</del>	<del>7</del>
S->A	1	6	7
S->G	5	0	5

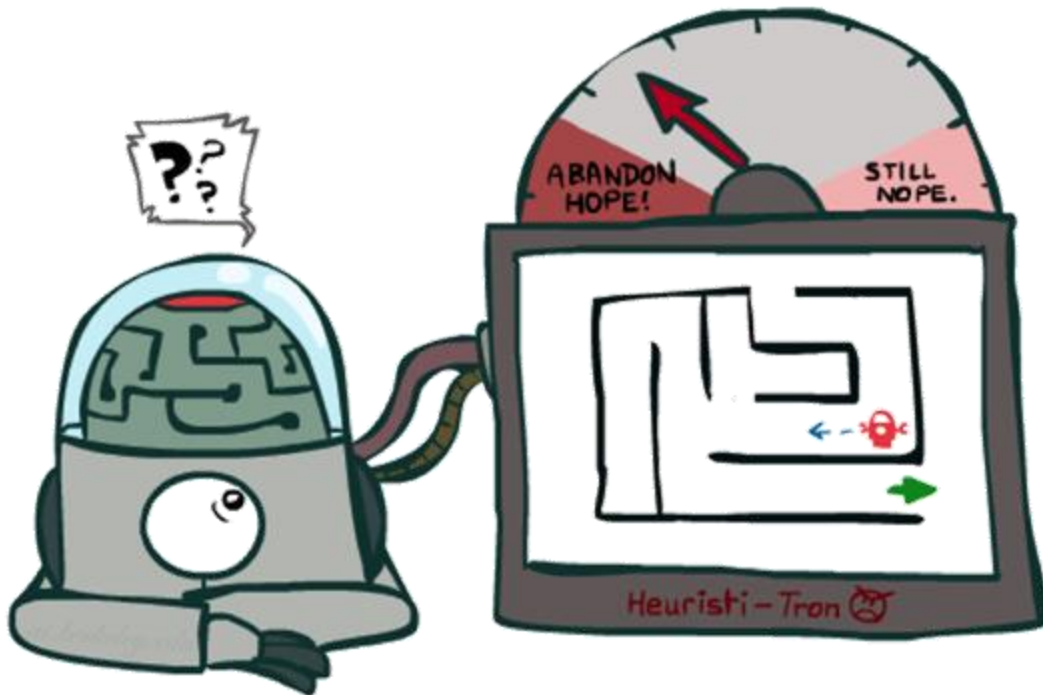
- What went wrong?
- $g(S \rightarrow \text{bad} \rightarrow G) > g(S \rightarrow \text{good} \rightarrow G) = g(S \rightarrow \text{good} \rightarrow A) + g(A \rightarrow \text{good} \rightarrow G) \geq g(S \rightarrow \text{good} \rightarrow A) + h(A)$
- **Actual** bad goal cost < **estimated** good goal cost
- We need estimates to be less than actual costs!

# The Price is Wrong...

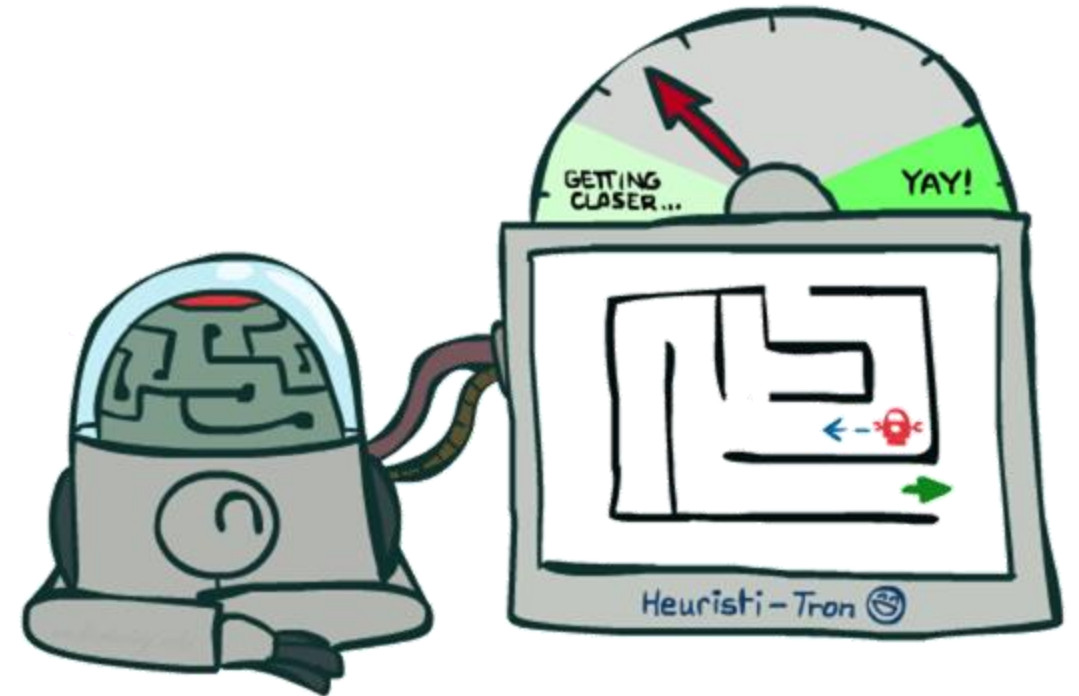
- Closest bid without going over...



# Admissible Heuristics: Ideas



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

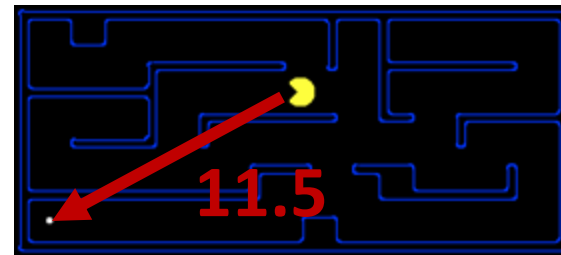
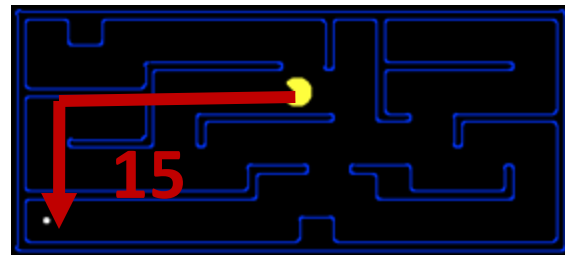


Admissible (optimistic) heuristics  
slow down bad plans but  
never outweigh true costs

# 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:

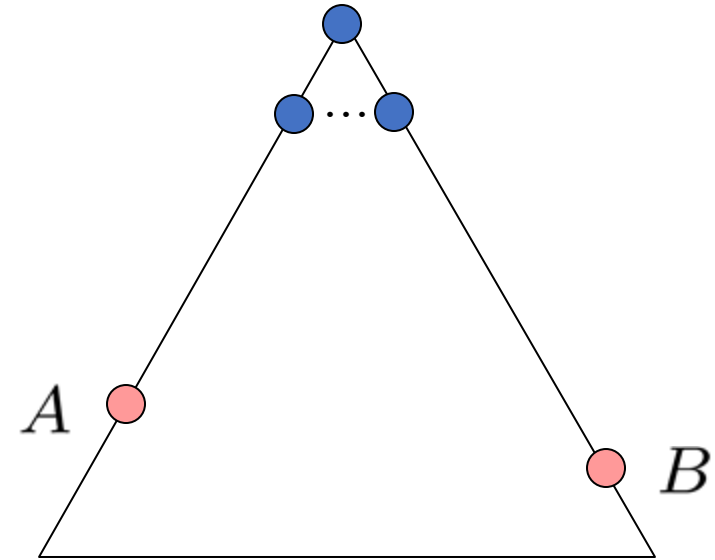


0.0

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

# Optimality of A\* Tree Search

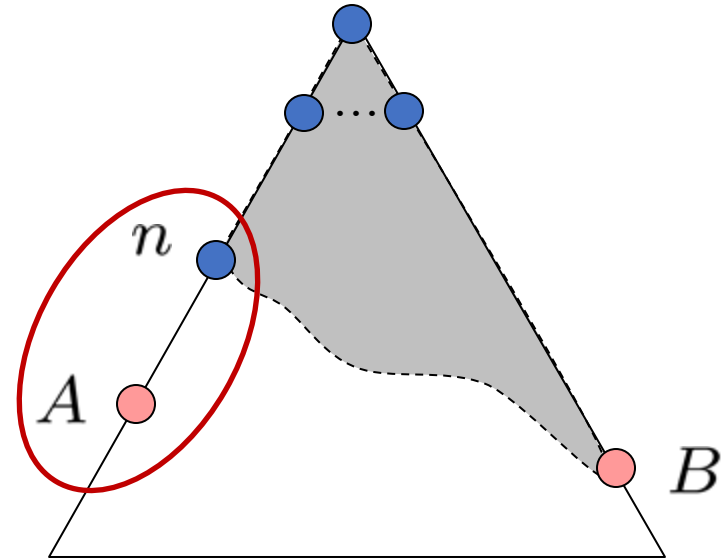
- Assume:
  - A is an optimal goal node
  - B is a suboptimal goal node
  - h is admissible
- Claim:
  - A will exit the fringe before B





# Optimality of A\* Tree Search: Blocking

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



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

$$f(n) \leq g(A)$$

$$g(A) = f(A)$$

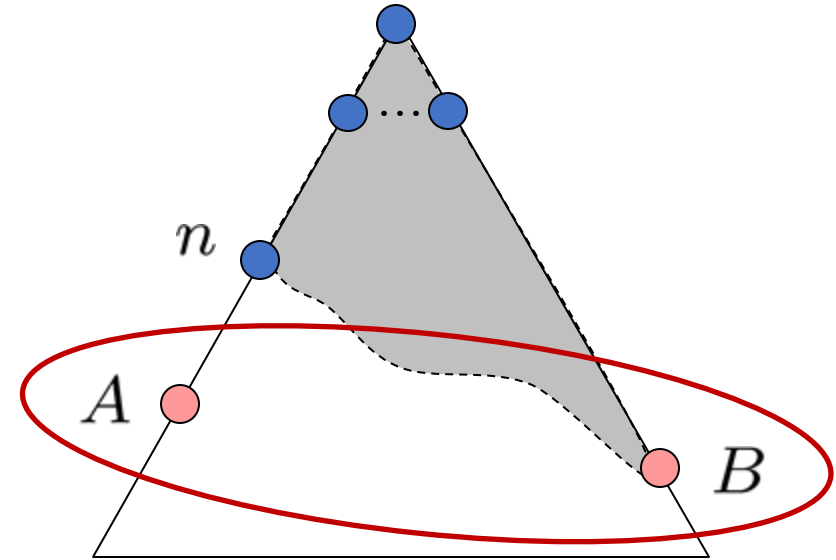
Definition of f-cost

Admissibility of  $h$

$h = 0$  at a goal

# Optimality of A\* Tree Search: Blocking 2

- 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  $f(B)$



$$g(A) < g(B)$$

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

B is suboptimal

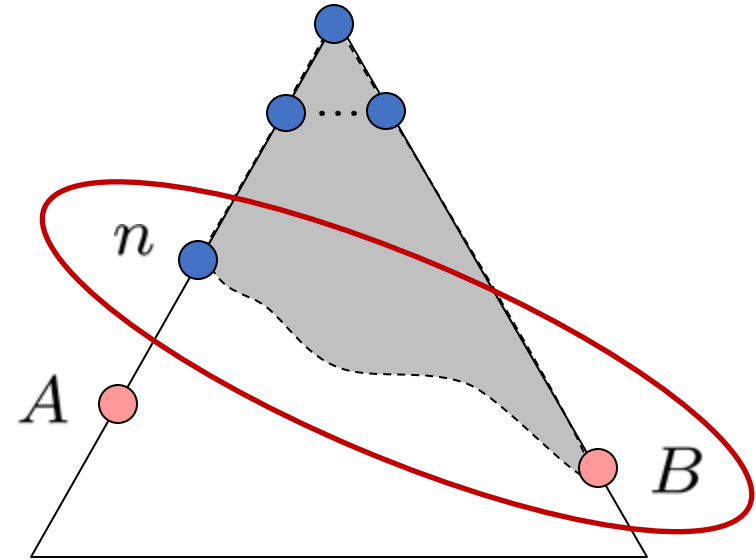
$h = 0$  at a goal

# Optimality of A\* Tree Search: Blocking 3

- 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

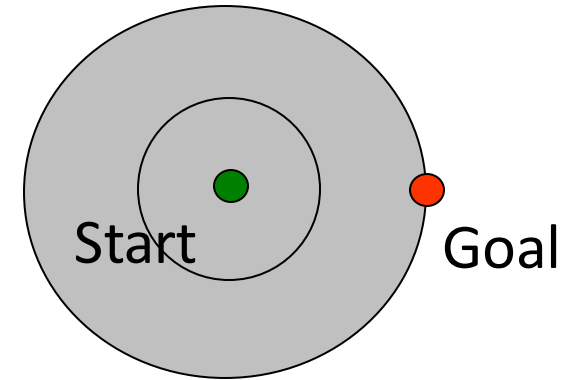
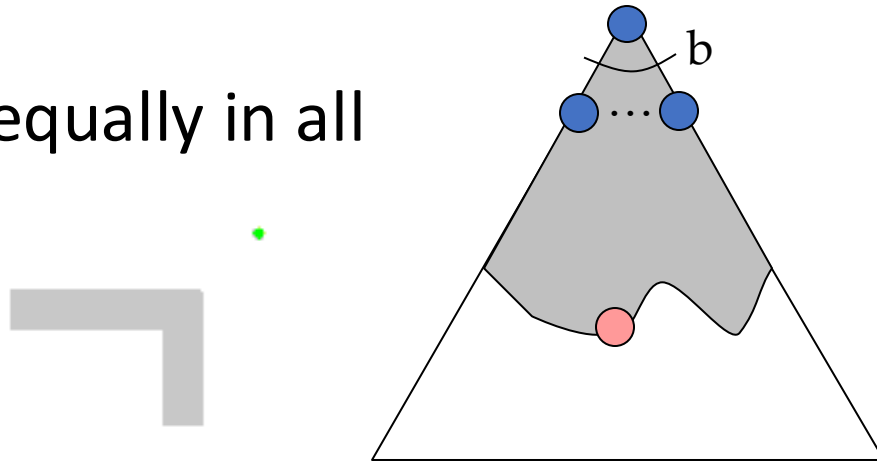


$$f(n) \leq f(A) < f(B)$$

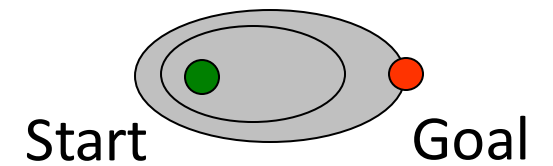
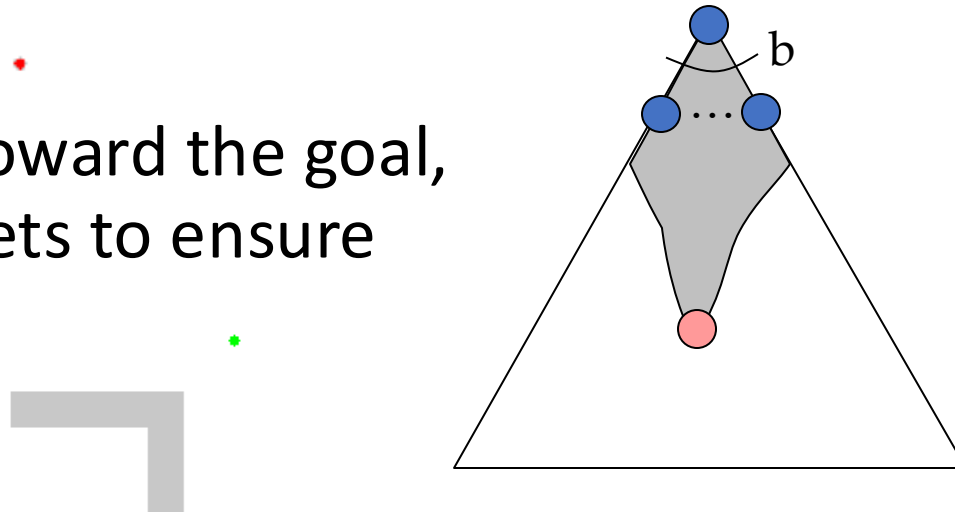


# UCS vs A\*

- Uniform-cost expands equally in all “directions”



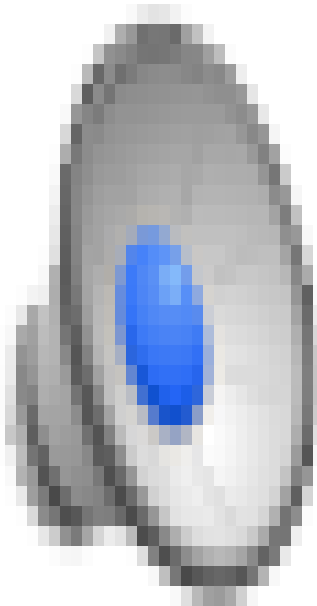
- A\* expands mainly toward the goal, but does hedge its bets to ensure optimality



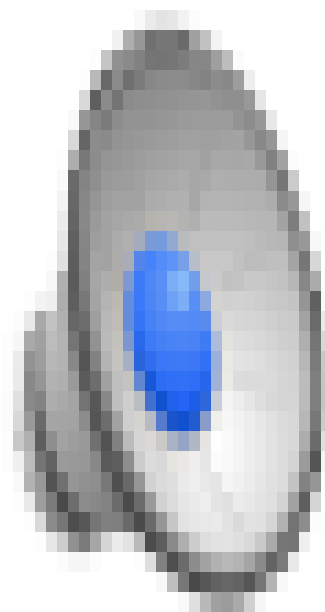
[Demo: contours UCS / greedy / A\* empty (L3D1)]

[Demo: contours A\* pacman small maze (L3D5)]

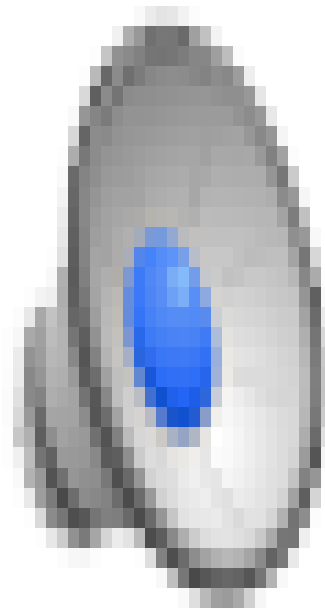
# Video of Demo Contours (Empty) -- UCS



# Video of Demo Contours (Empty) -- Greedy

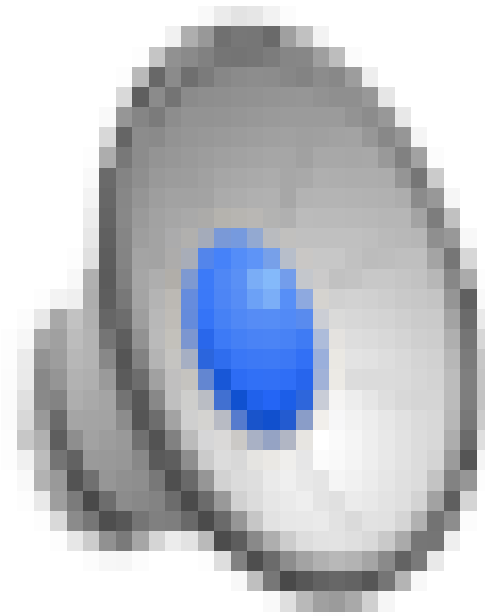


# Video of Demo Contours (Empty) – $A^*$



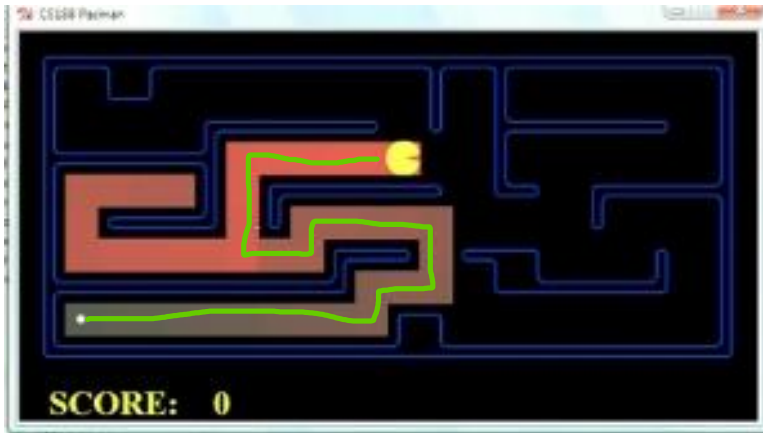
# Video of Demo Contours (Pacman Small Maze)

–  $A^*$

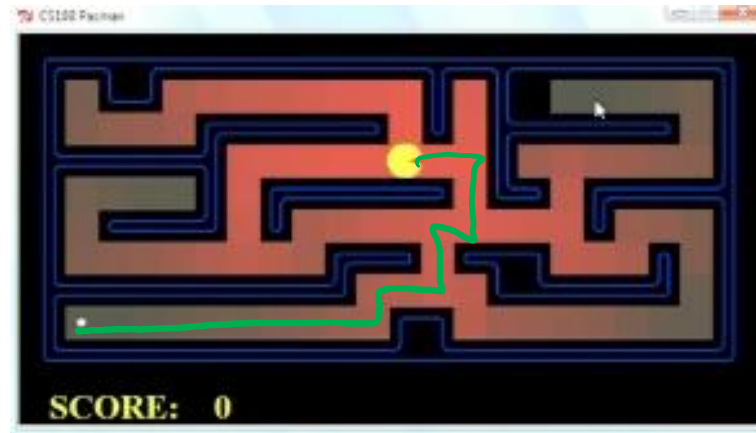




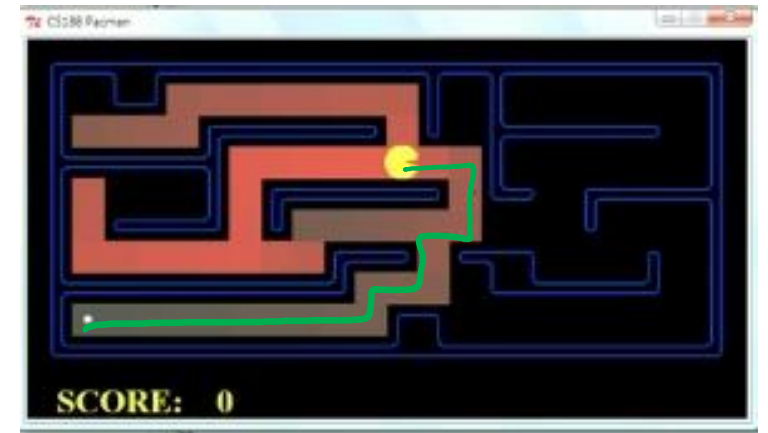
# Comparison



Greedy



Uniform Cost



A\*

# A\* Applications

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

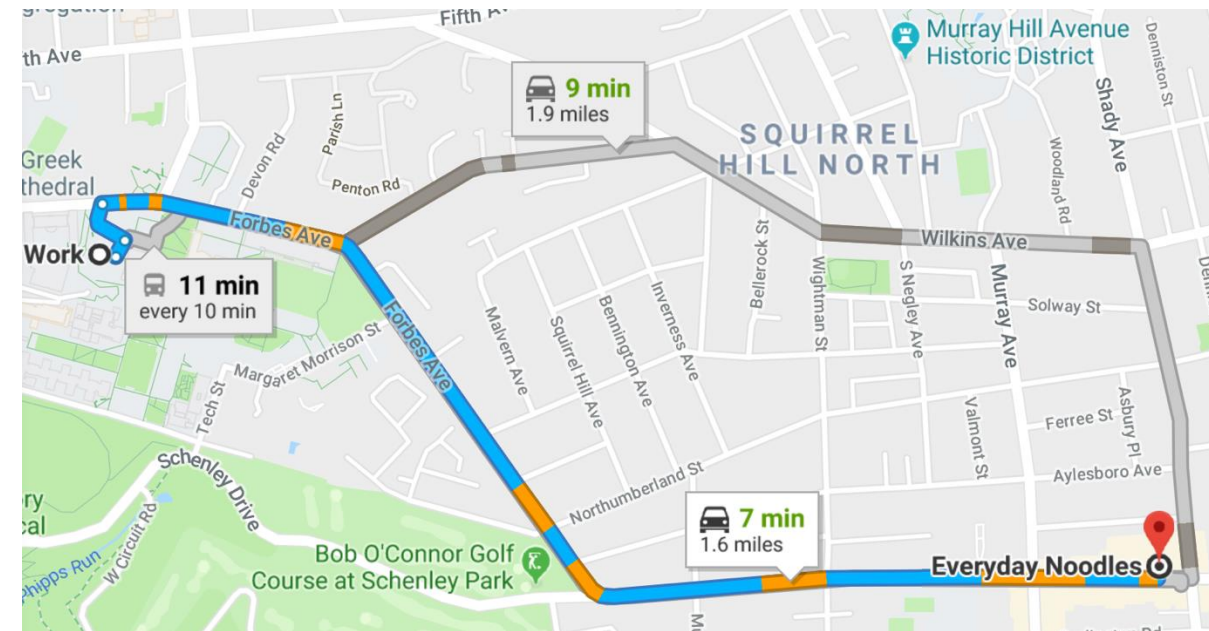
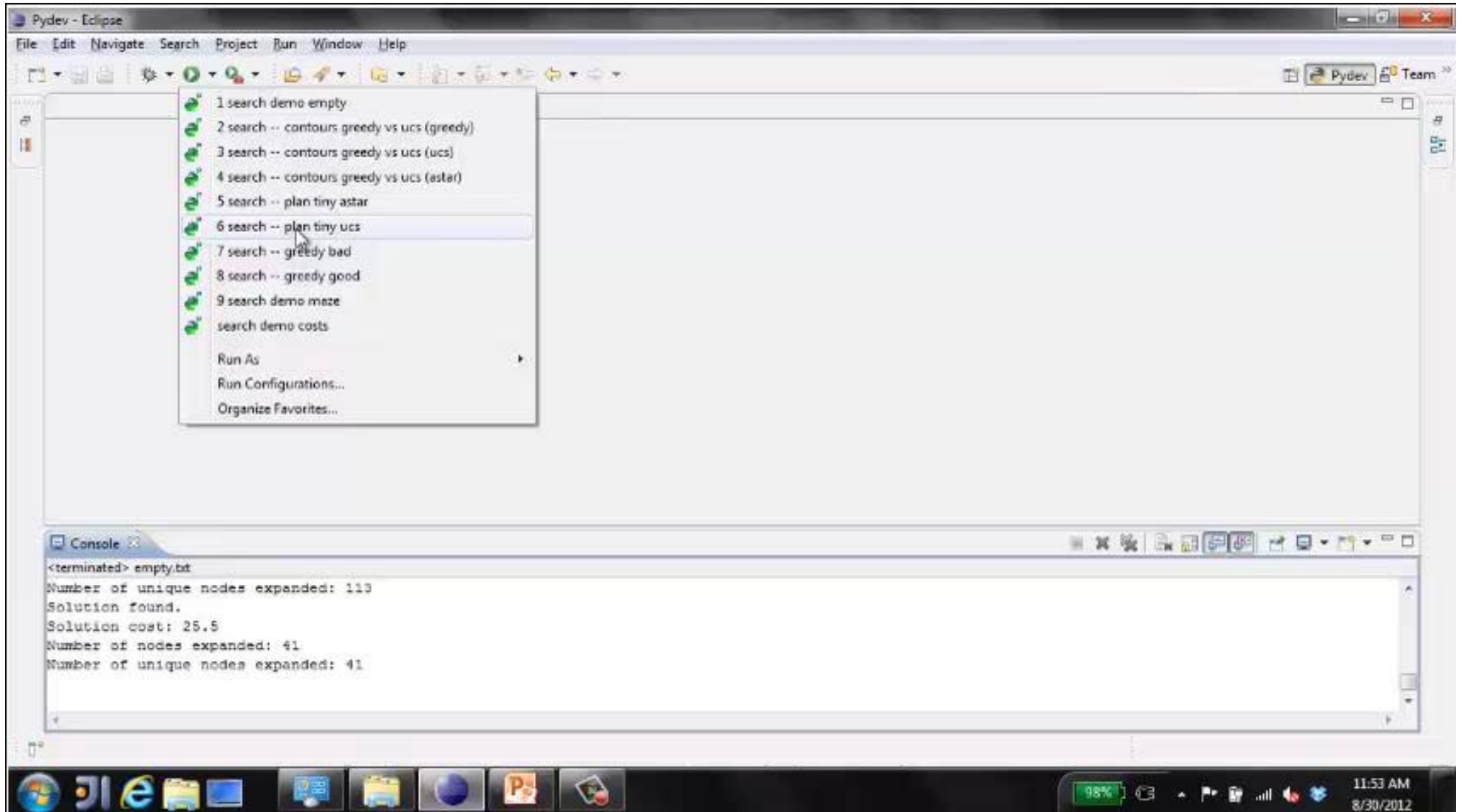


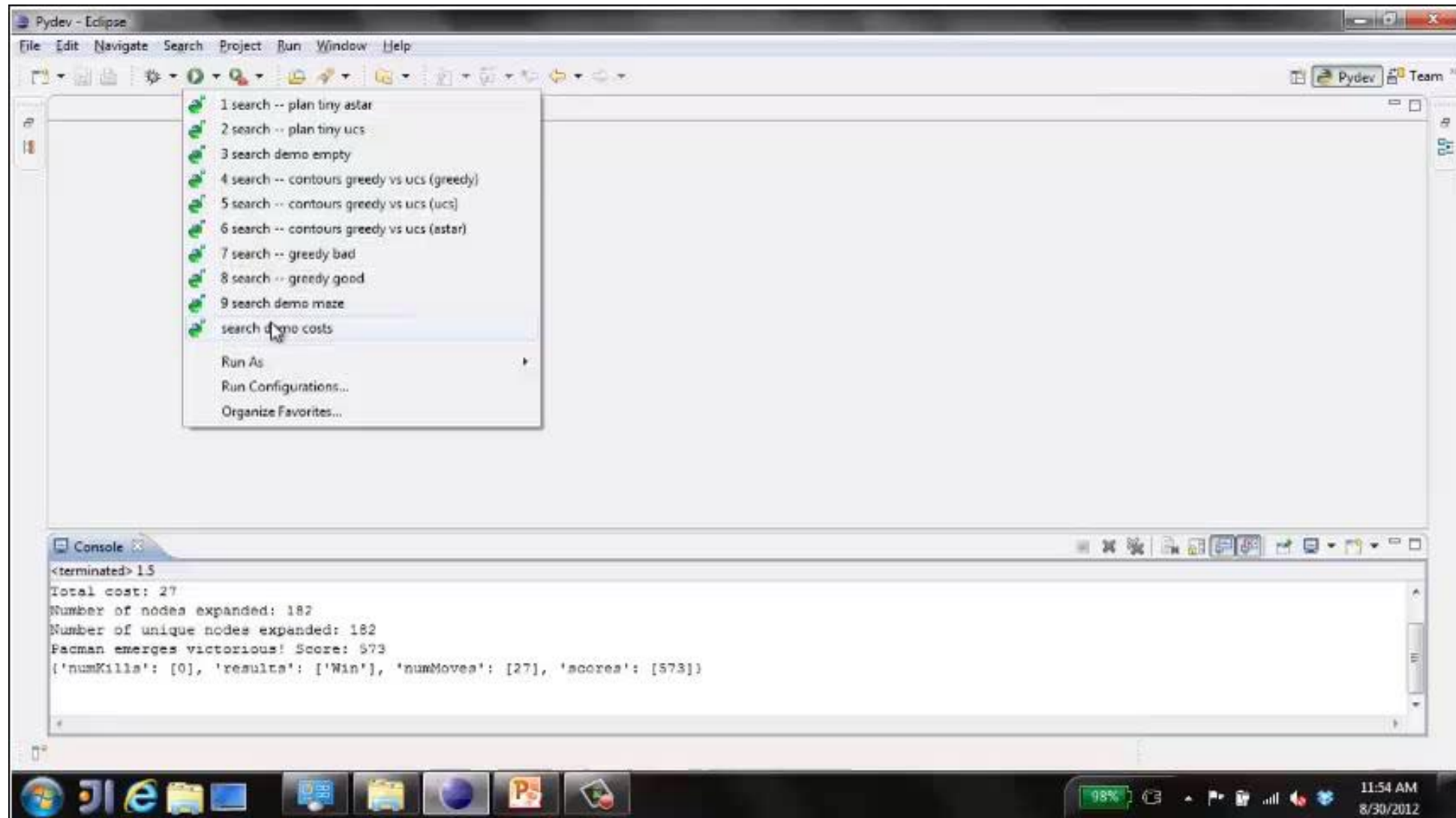
Image: maps.google.com

[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



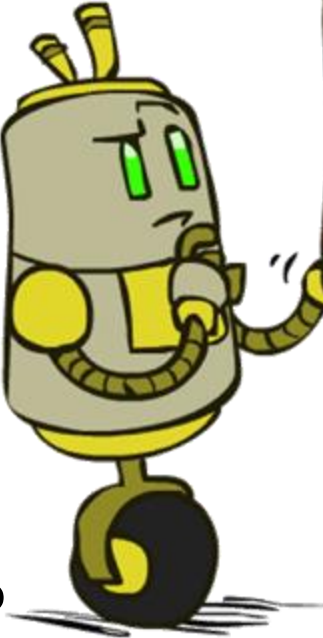




# Example: 8 Puzzle

7	2	4
5		6
8	3	1

Start State



3	7	1
2	4	5
	8	6

Actions

	1	2
3	4	5
6	7	8

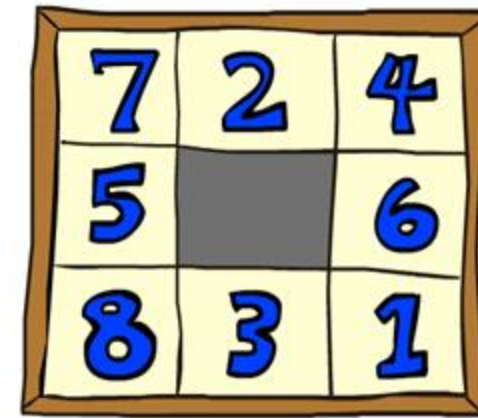
Goal State

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

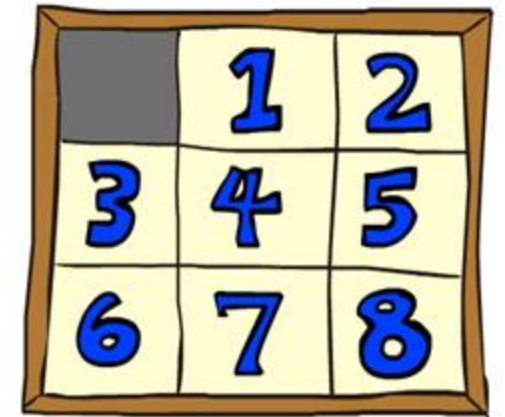
Admissible  
heuristics?

# Example: 8 Puzzle - 2

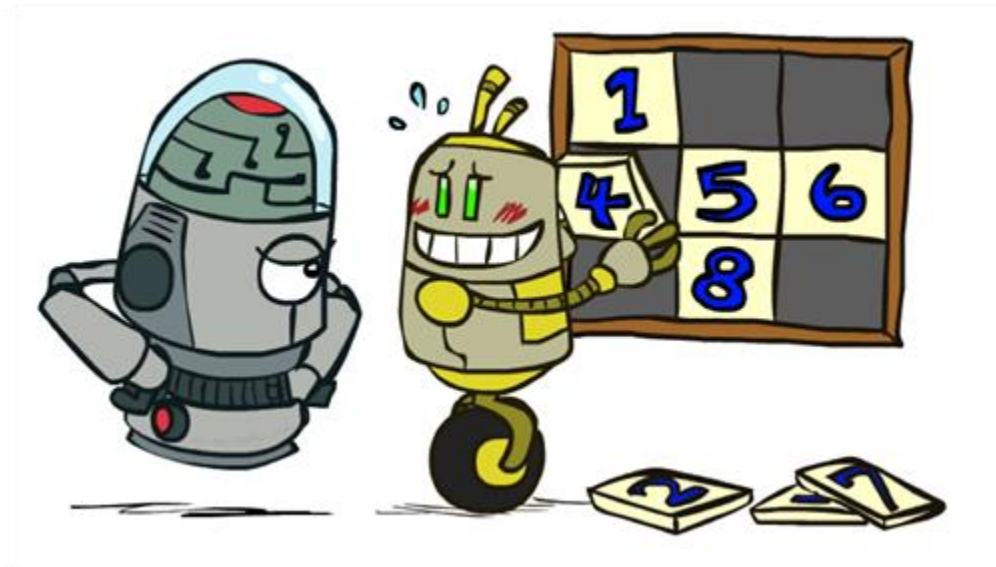
- Heuristic: Number of **tiles** misplaced
- Why is it admissible?
- $h(\text{start}) = 8$
- This is a relaxed-problem heuristic



Start State



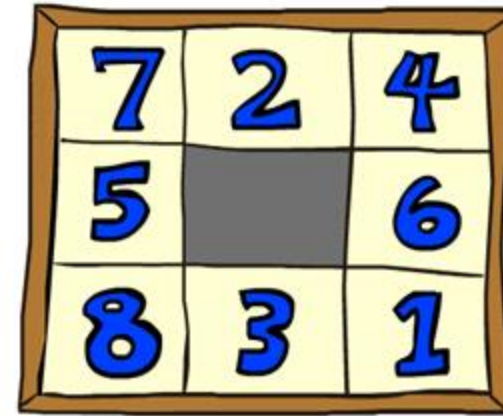
Goal State



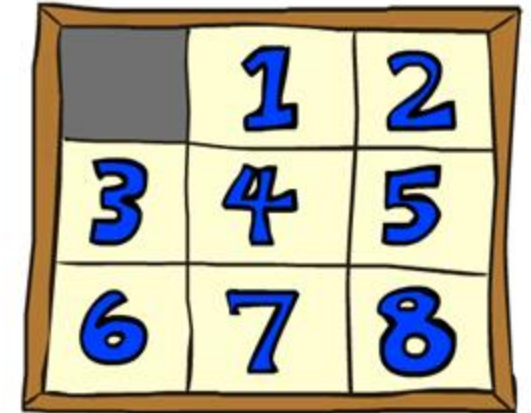
Average nodes expanded when the optimal path has...			
	...4 steps	...8 steps	...12 steps
UCS	112	6,300	$3.6 \times 10^6$
TILES	13	39	227

# Example: 8 Puzzle - 3

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?



Start State



Goal State

- Total Manhattan distance

- Why is it admissible?

- $h(\text{start}) = 3 + 1 + 2 + \dots = 18$

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



# Example: 8 Puzzle - 4

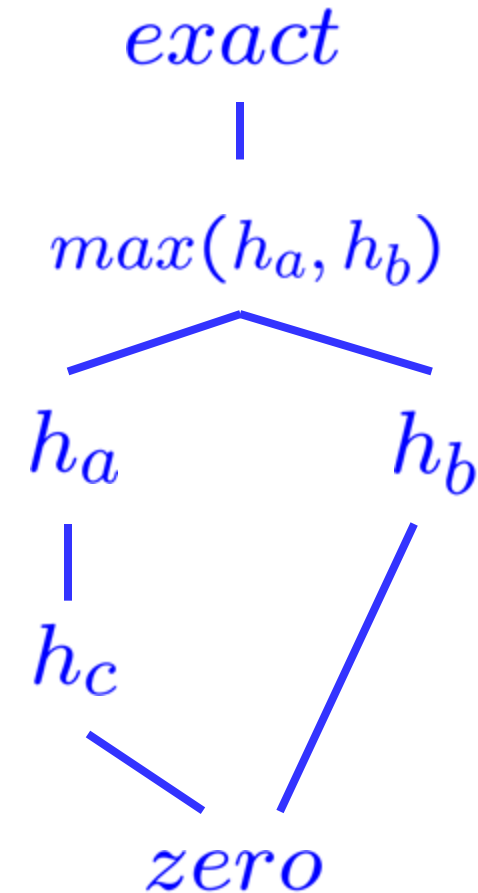
- How about using the actual cost as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

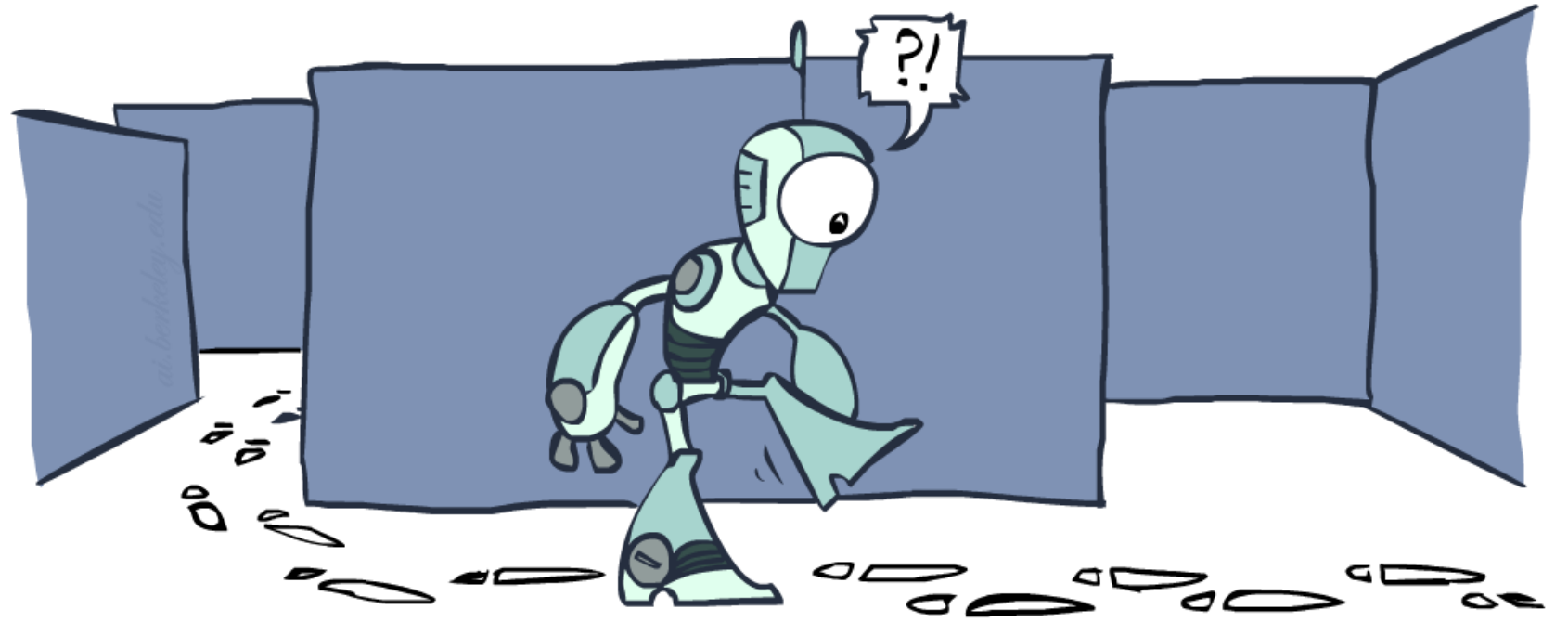


- With  $A^*$ : a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

# Combining Heuristics, Dominance

- Dominance:  $h_a \geq h_c$  if
$$\forall n : h_a(n) \geq h_c(n)$$
  - Roughly speaking, larger is better as long as both are admissible
- 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, but usually too expensive

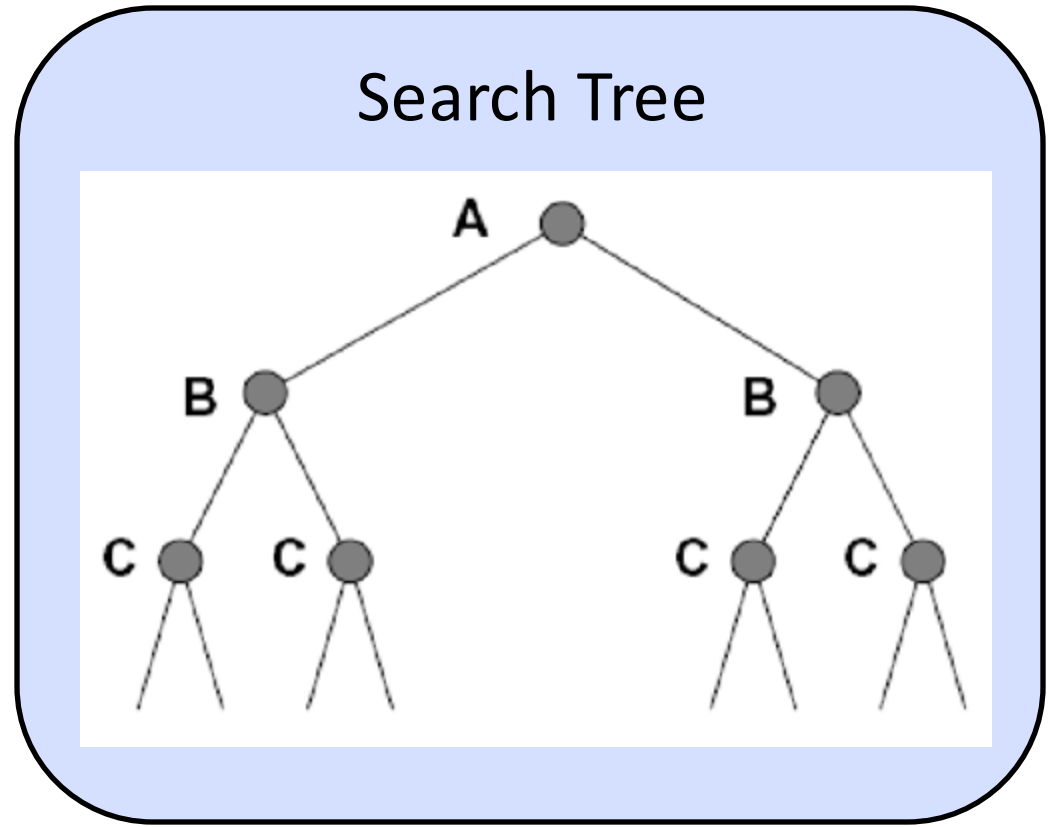
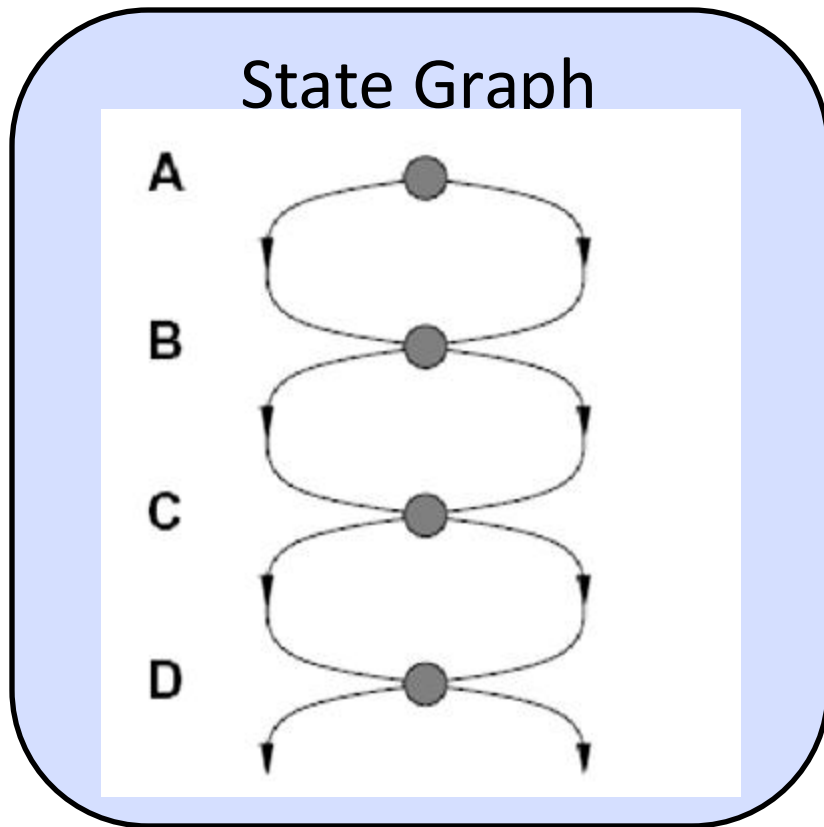




# Graph Search

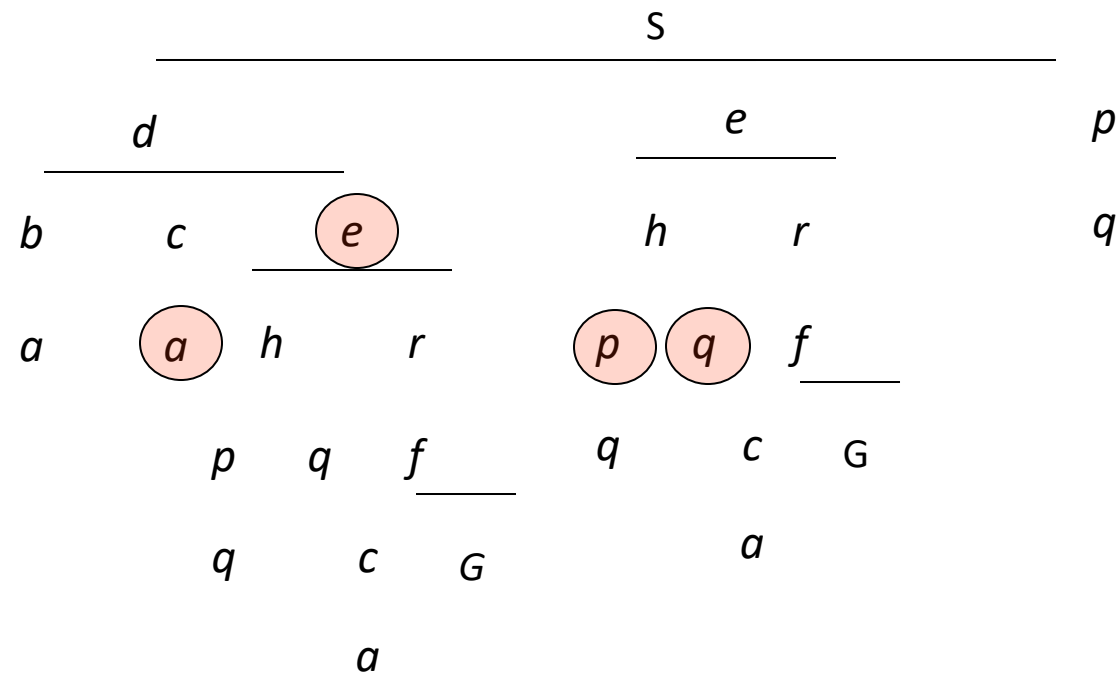
# Tree Search: Extra Work!

- 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 2

- Idea: never **expand** a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”, “explored 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/explored set
- Important: **store the closed/explored set as a set**, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

**function** GRAPH\_SEARCH(**problem**) **returns** a solution, or failure

**initialize the explored set to be empty**

initialize the **frontier** as a specific work list (stack, queue, priority queue)

add initial state of **problem** to **frontier**

**loop do**

**if** the **frontier** is empty **then**

**return** failure

choose a **node** and remove it from the **frontier**

**if** the **node** contains a goal state **then**

**return** the corresponding solution

**add the node state to the explored set**

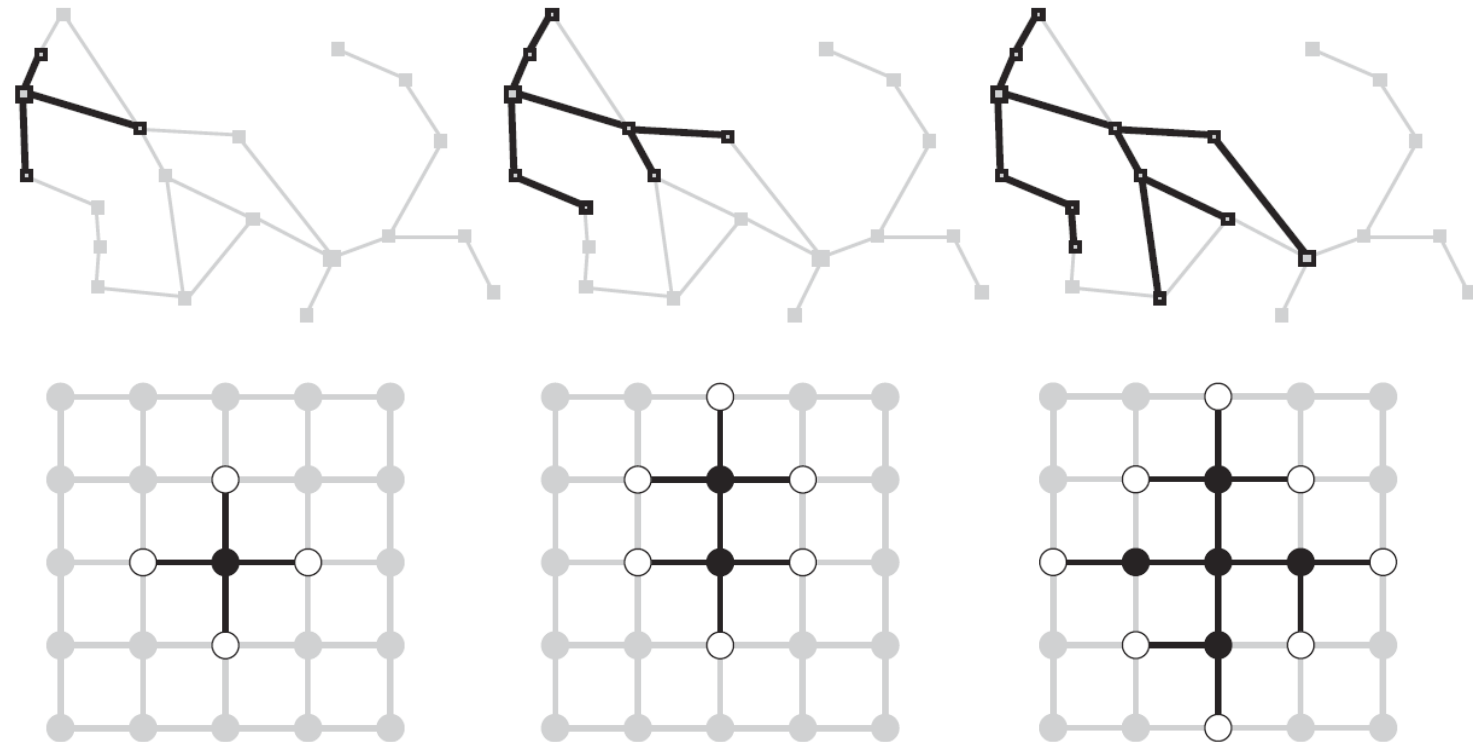
for each resulting **child** from node

**if** the **child** state is not already in the **frontier** or **explored set** **then**

add **child** to the **frontier**

# Graph Search 3

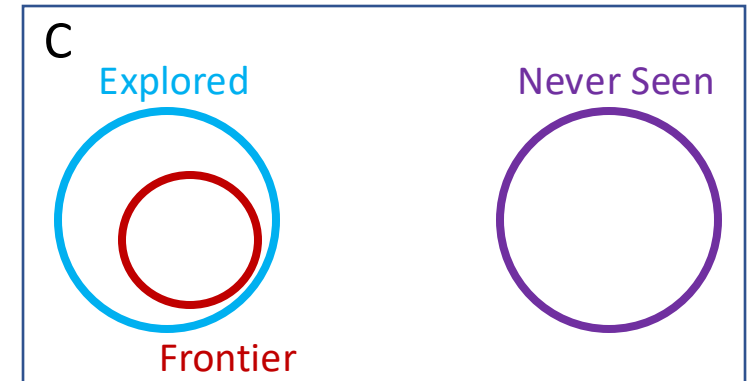
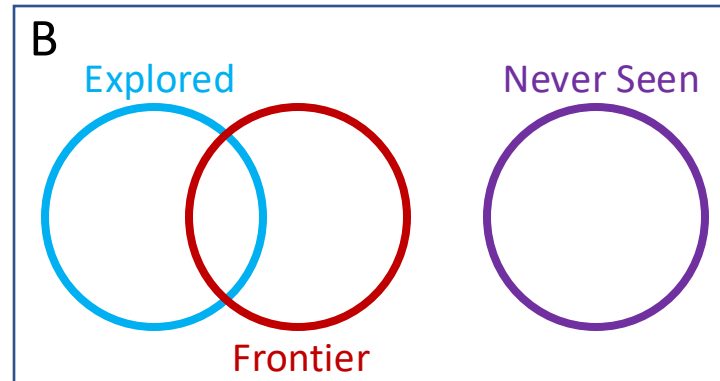
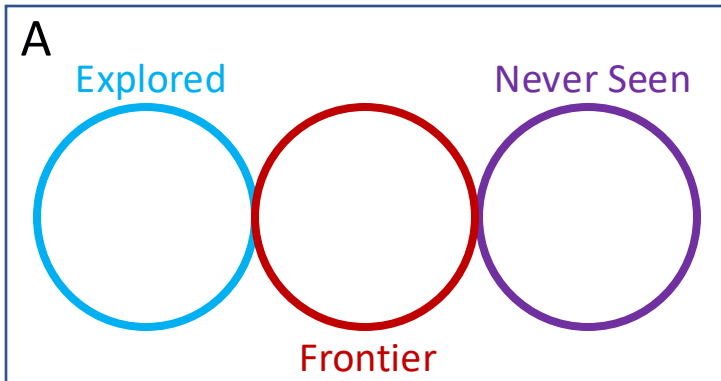
- This graph search algorithm overlays a tree on a graph
- The **frontier** states separate the **explored** states from **never seen** states





# Quiz

- What is the relationship between these sets of states after each loop iteration in `GRAPH_SEARCH`?
- (Loop invariants!!!)



**function** UNIFORM-COST-GRAPH-SEARCH(**problem**) **returns** a solution, or failure

initialize the **explored set** to be empty

initialize the **frontier** as a **priority queue** using **node's path\_cost** as the **priority**

add initial state of **problem** to **frontier** with **path\_cost = 0**

**loop do**

**if** the **frontier** is empty **then**

**return** failure

    choose a **node** and remove it from the **frontier**

**if** the **node** contains a goal state **then**

**return** the corresponding solution

    add the **node** state to the **explored set**

    for each resulting **child** from node

**if** the **child** state is not already in the **frontier** or **explored set** **then**

            add **child** to the **frontier**

**else if** the **child** is already in the **frontier** with higher **path\_cost** **then**

            replace that **frontier** node with **child**

function A-STAR-GRAPH-SEARCH(problem) returns a solution, or failure

initialize the explored set to be empty

initialize the frontier as a priority queue using  $f(n) = g(n) + h(n)$  as the priority

add initial state of problem to frontier with priority  $f(S) = 0 + h(S)$

loop do

if the frontier is empty then

return failure

choose a node and remove it from the frontier

if the node contains a goal state then

return the corresponding solution

add the node state to the explored set

for each resulting child from node

if the child state is not already in the frontier or explored set then

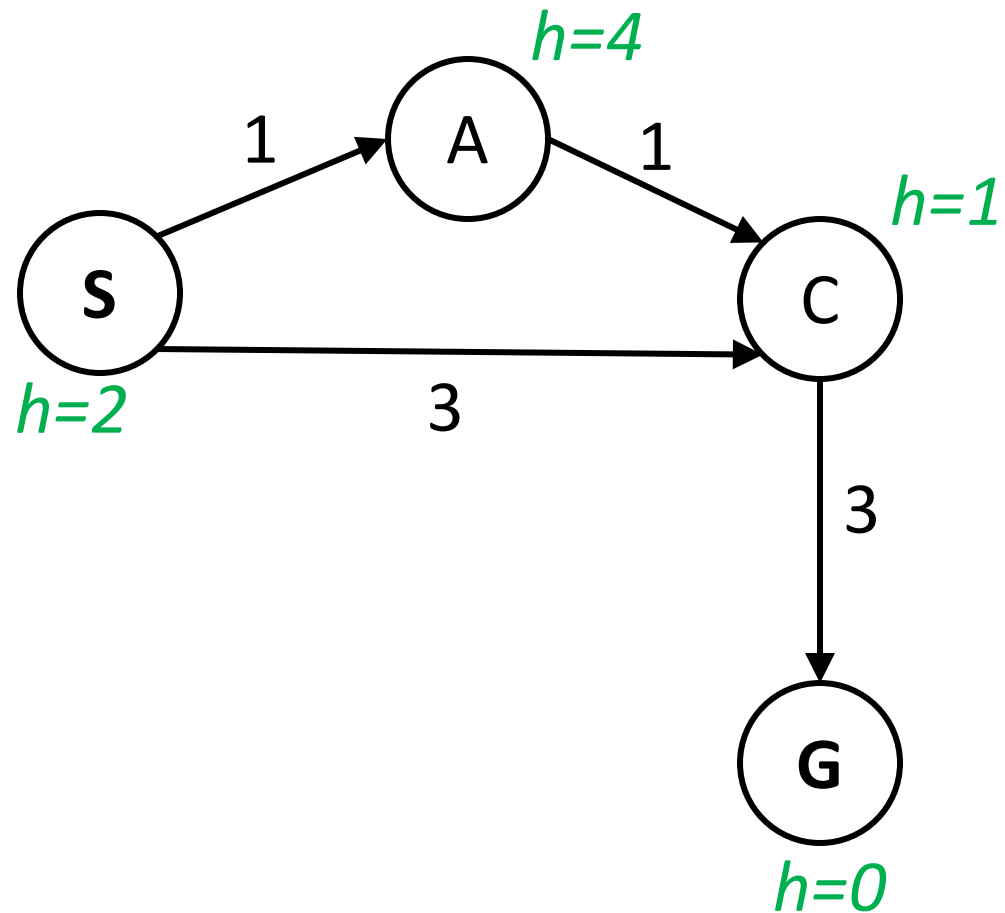
add child to the frontier

else if the child is already in the frontier with higher  $f(n)$  then

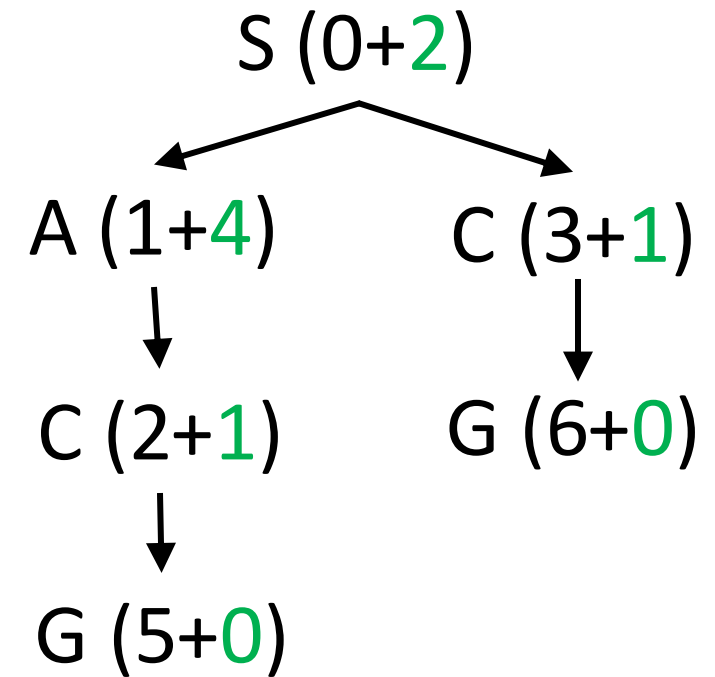
replace that frontier node with child

# A\* Tree Search

State space graph

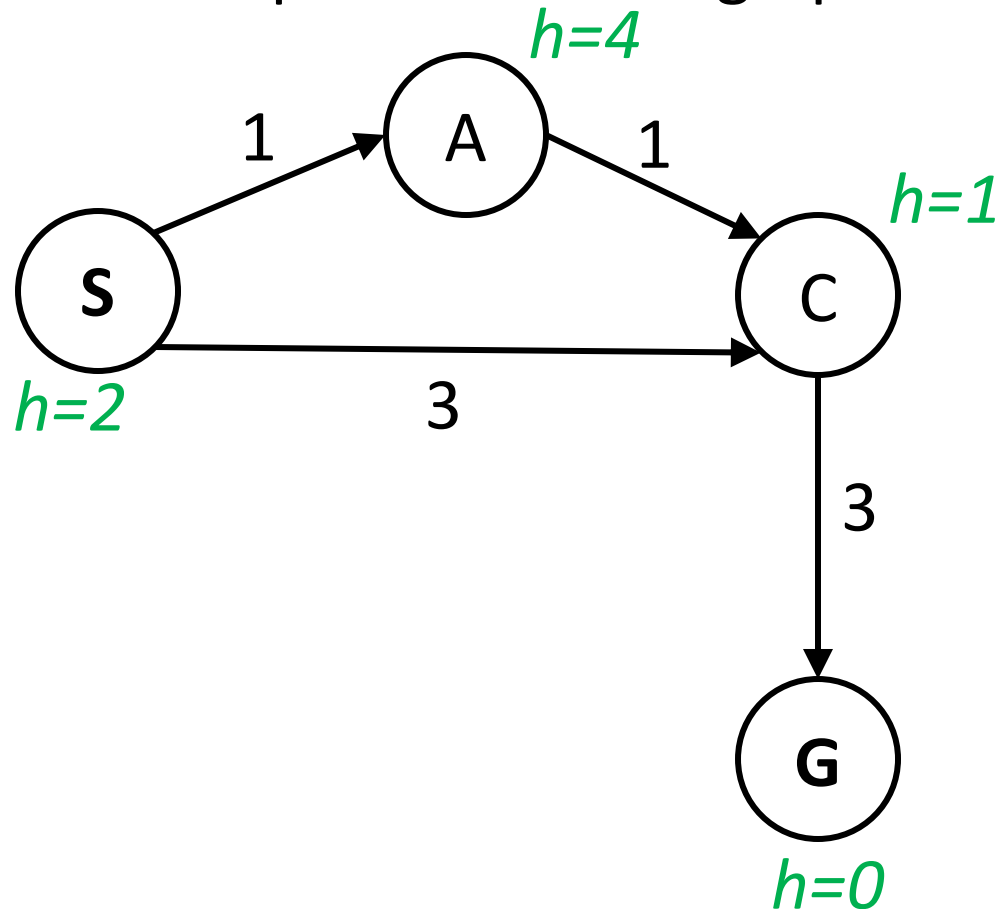


Search tree



# Quiz: A\* Graph Search

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



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

Frontier

~~S-0+2~~  
~~S->A 1+4~~  
~~S->C 3+1~~  
**S->C->G 6+0**

Explored set:

S C A

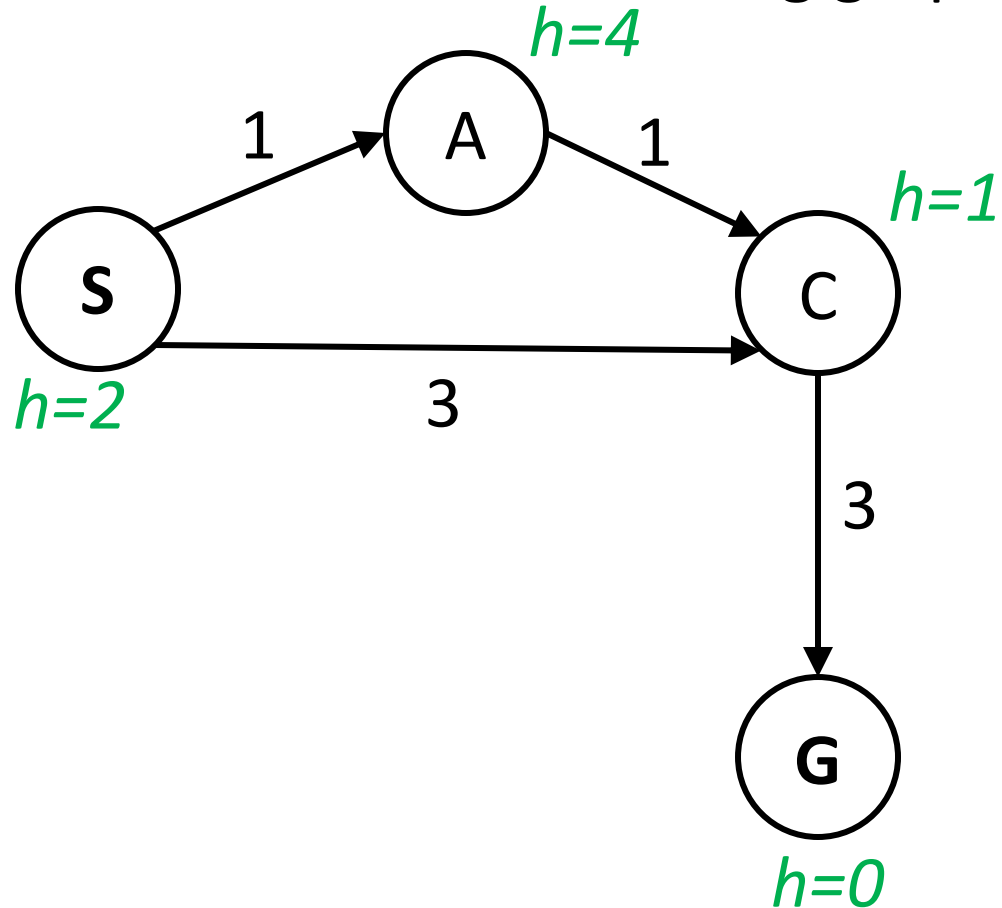
$g(S \rightarrow \text{good} \rightarrow A) + h(A)$

$\leq g(S \rightarrow \text{good} \rightarrow A) + g(A \rightarrow \text{good} \rightarrow C) + h(C)$

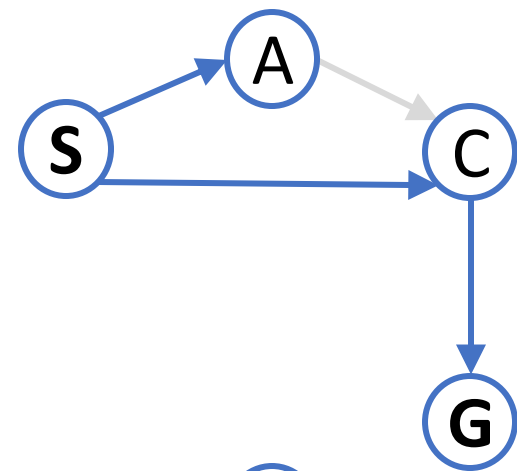
$< g(S \rightarrow \text{bad} \rightarrow C) + h(C)$

# Quiz: A\* Graph Search 2

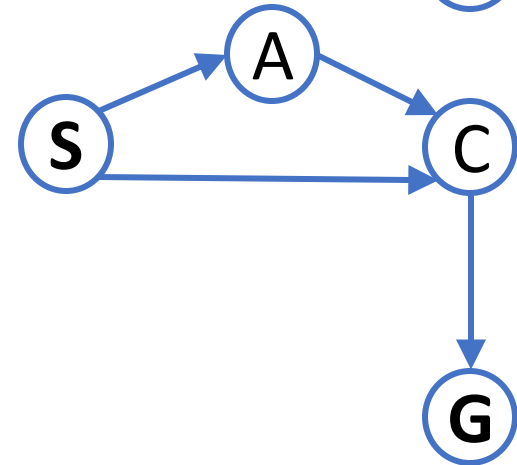
- What does the resulting graph tree look like?



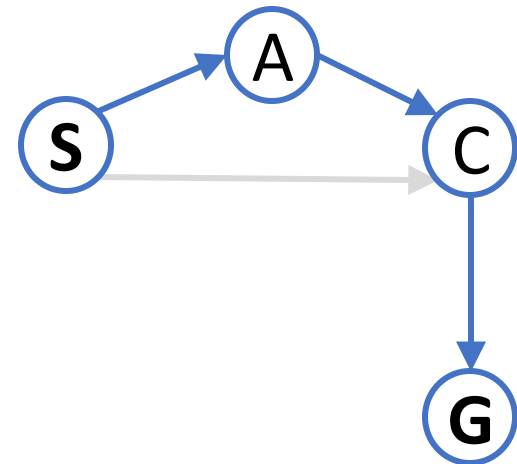
A)



B)

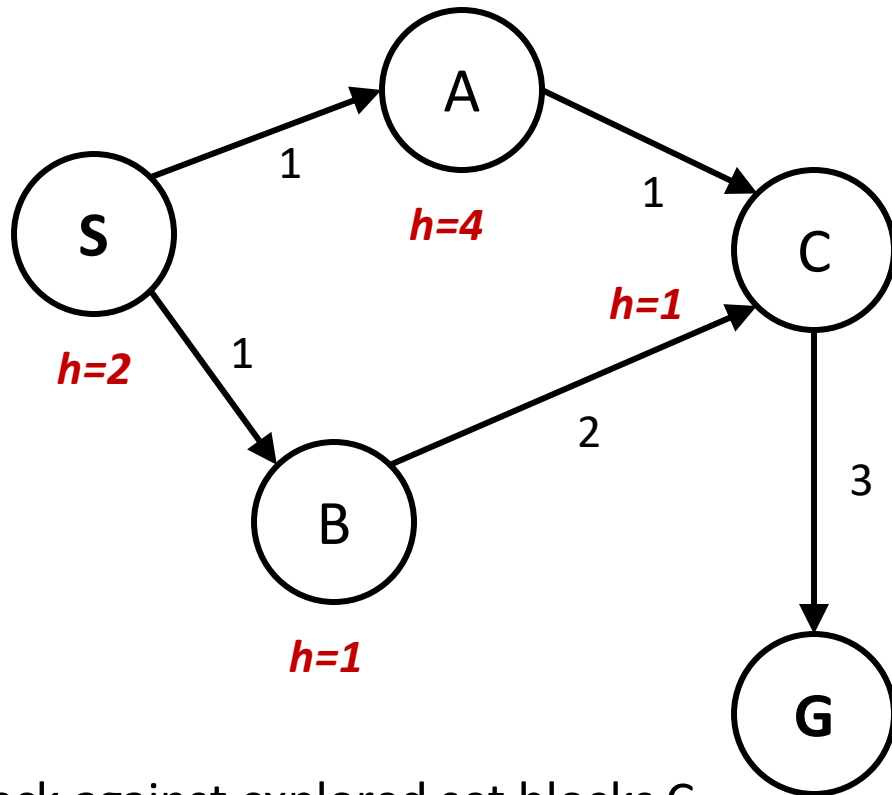


C & D)

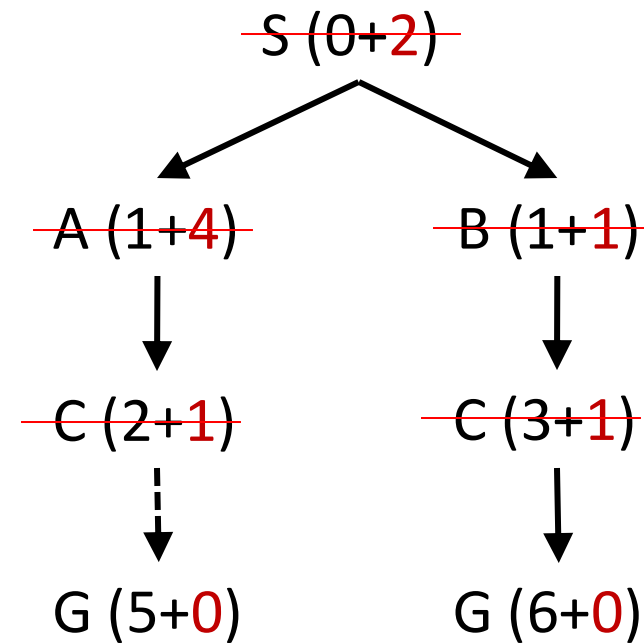


# A\* Graph Search Gone Wrong?

State space graph



Search tree

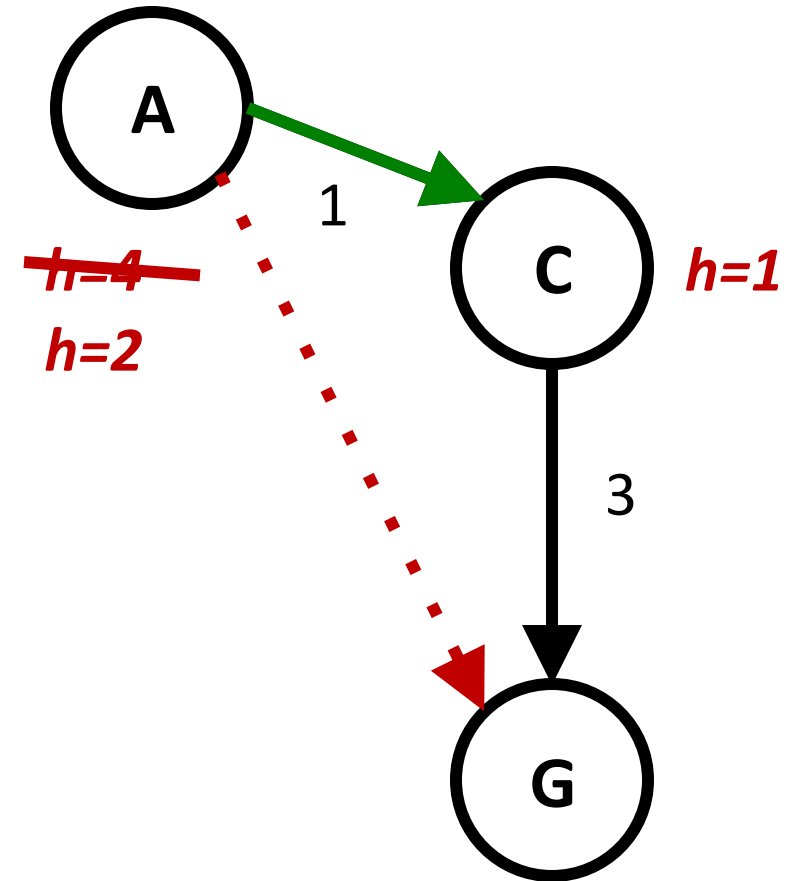


- Simple check against explored set blocks C
- Fancy check allows new C if cheaper than old  $h=0$  but requires recalculating C's descendants

Explored Set: S B C A

# Consistency of Heuristics

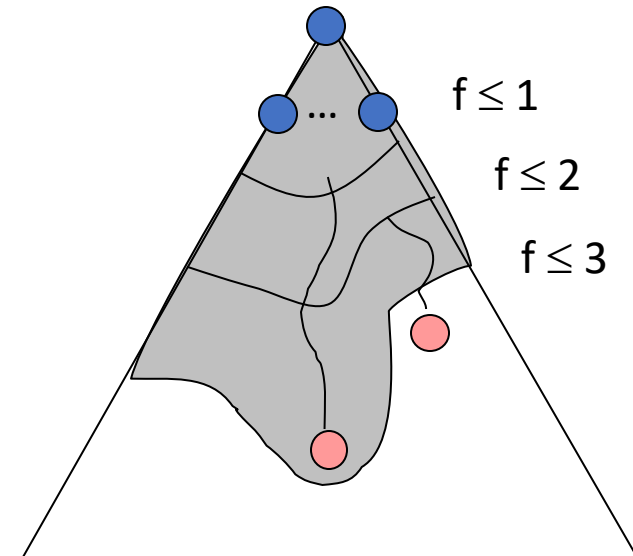
- Main idea: estimated heuristic costs  $\leq$  actual costs
  - Admissibility: heuristic cost  $\leq$  actual cost to goal
    - $h(A) \leq$  actual cost from A to G
  - Consistency: heuristic “arc” cost  $\leq$  actual cost for each arc
    - $h(A) - h(C) \leq \text{cost}(A \text{ to } C)$
    - triangle inequality:  $h(A) \leq c(A-C) + h(C)$
- Consequences of consistency:
  - The f value along a path never decreases (never overestimate future)
    - $h(A) \leq \text{cost}(A \text{ to } C) + h(C)$
  - A\* graph search is optimal





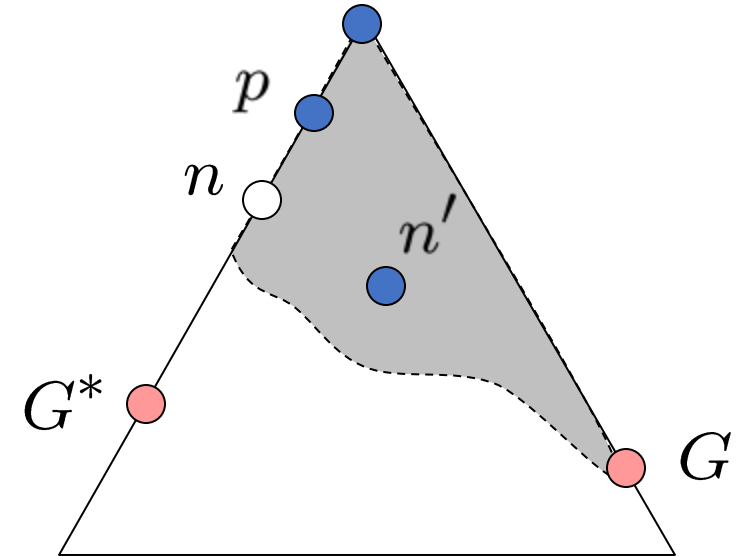
# Optimality of A\* Graph Search

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



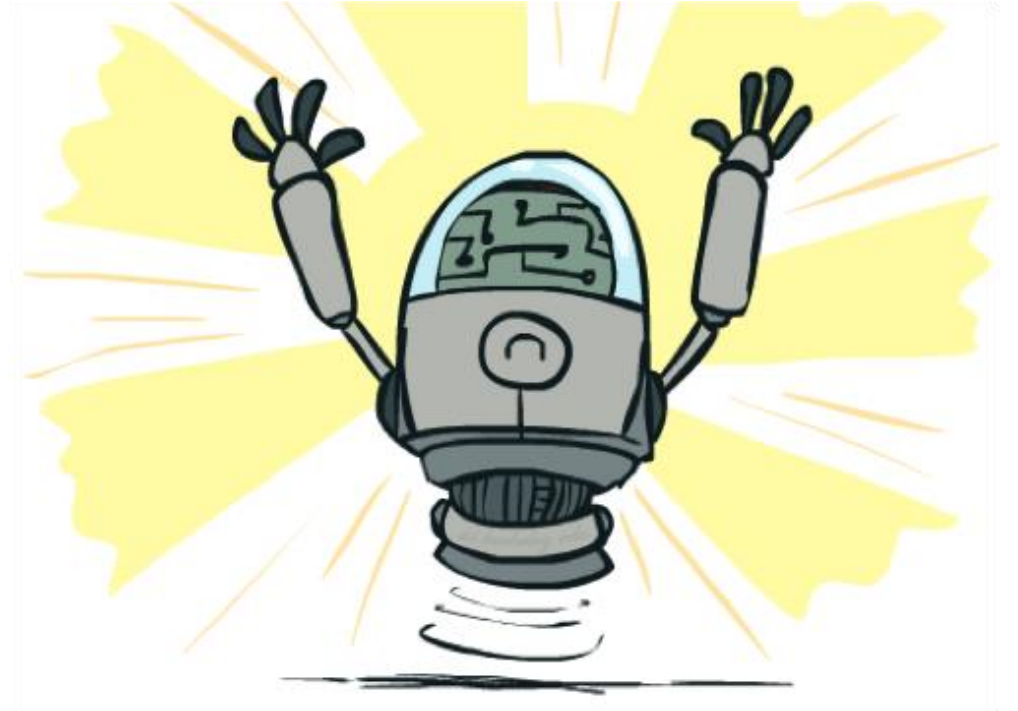
# Optimality of A\* Graph Search: Proof

- For any  $n$  on path to  $G^*$ , let  $n'$  be a worse node for **the same state**
- Let  $p$  be the ancestor of  $n$  that was on the queue when  $n'$  was added in the queue
- Claim:  $p$  will be expanded before  $n'$ 
  - $f(p) \leq f(n)$  because of **consistency**
  - $f(n) < f(n')$  because  $n'$  is suboptimal
  - $p$  would have been expanded before  $n'$
- Thus  $n$  will be expanded before  $n'$
- All ancestors of  $G^*$  are not blocked



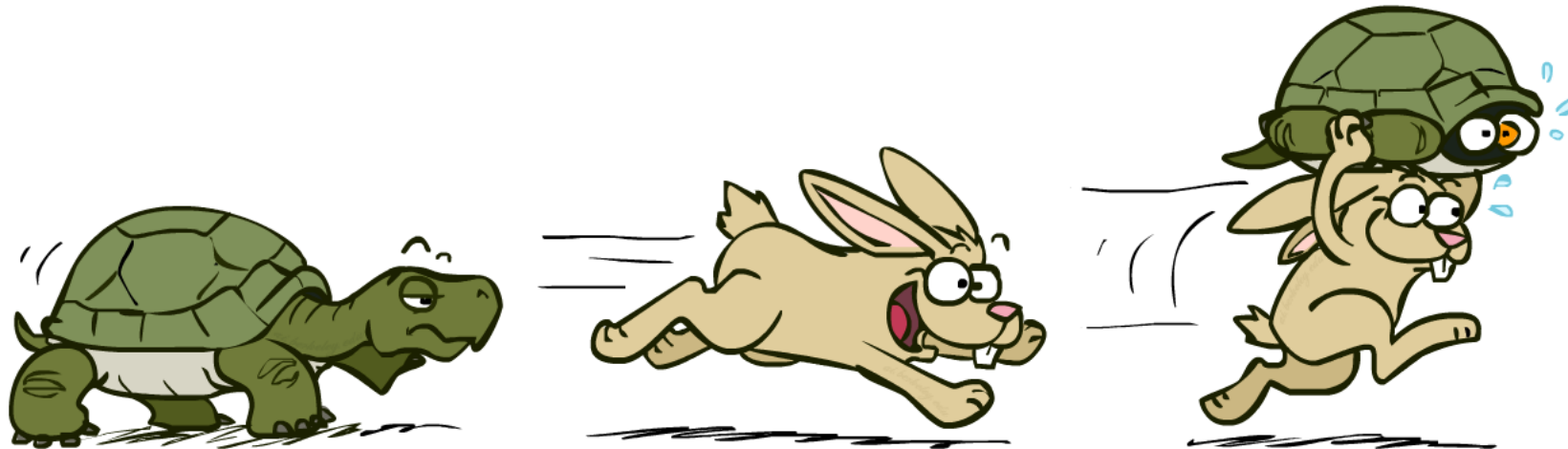
# Optimality of A\* Search

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



# Summary of A\*

- 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



# Summary

- Rational agents
- Search problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Informed Search Methods
  - Heuristics
  - Greedy Search
  - A\* Search
  - Graph Search

**Shuai Li**

<https://shuaili8.github.io>

# Questions?