

# Warm-up as you walk in

Write the pseudo code for breadth first search and depth first search

- Iterative version, not recursive

```
class TreeNode
    TreeNode[] children()
    boolean isGoal()
```

```
BFS(TreeNode start)...
```

```
DFS(TreeNode start)...
```

# Announcements

If you are not on Piazza, Gradescope, and Canvas

- E-mail me: [pvirtue@cmu.edu](mailto:pvirtue@cmu.edu)

No class next Mon 1/21, MLK Holiday

Recitation starting this Fri 3pm, GHC 4401 (recommended)

- Bring laptop if you can (not required)
- Start P0 before recitation to make sure Python 3.6 is working for you!

Reminder to be respectful of quiet areas in campus buildings

# Announcements

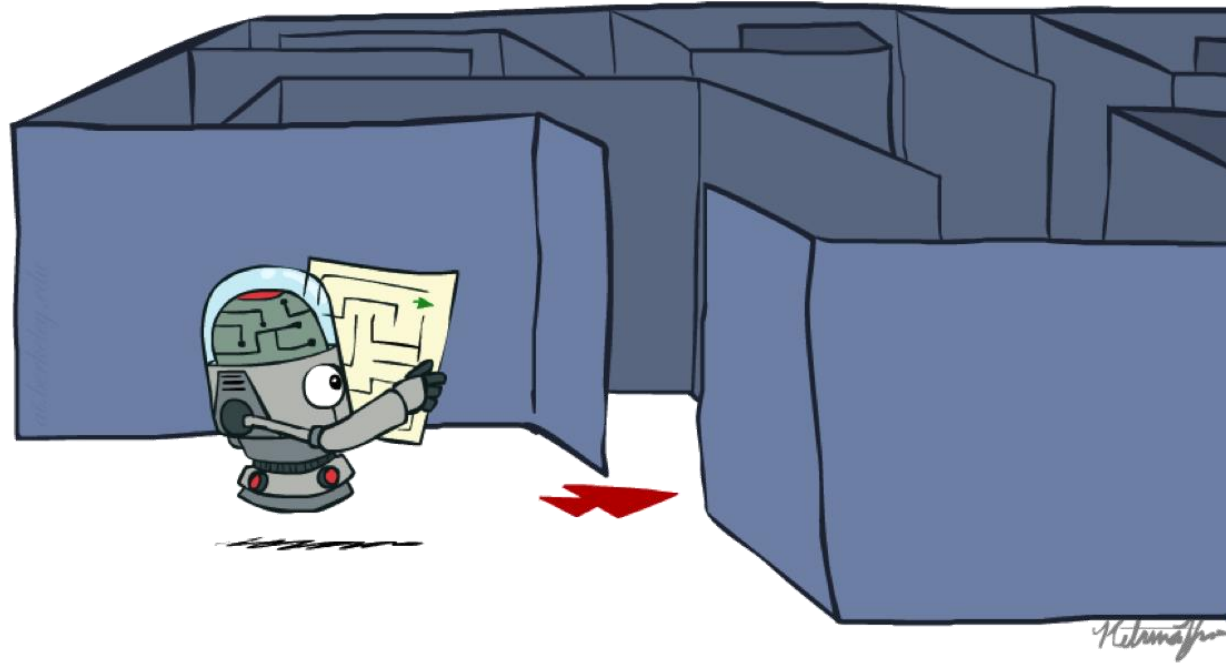
## Assignments:

- HW1 (online)
  - Released at 4:30 pm today
  - Due Tue 1/22, 10 pm
- P0: Python & Autograder Tutorial
  - Required, but worth zero points
  - Due Thu 1/24, 10 pm
  - No pairs, submit individually

Remaining programming assignments may be done in pairs

# AI: Representation and Problem Solving

## Agents and Search



Instructors: Pat Virtue & Stephanie Rosenthal

Slide credits: CMU AI, <http://ai.berkeley.edu>

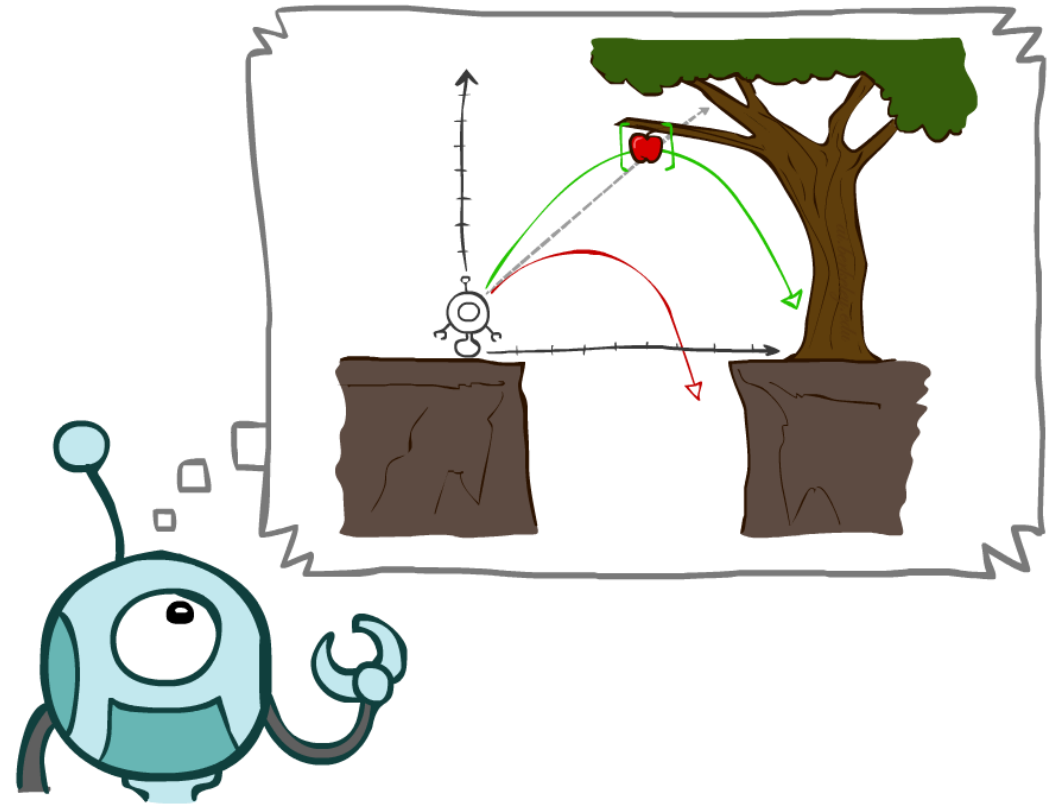
# Today

## Agents and Environment

## Search Problems

## Uninformed Search Methods

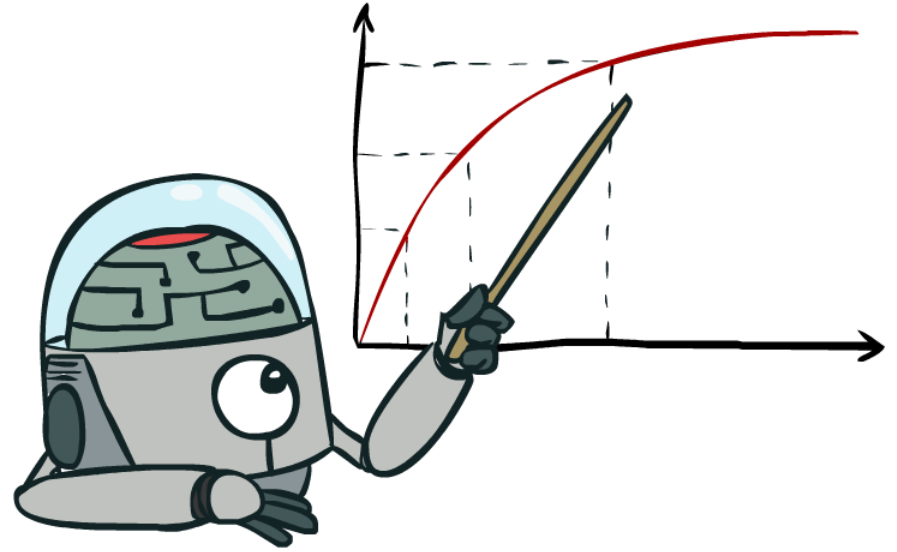
- Depth-First Search
- Breadth-First Search
- Uniform-Cost Search



# Rationality, contd.

## What is rational depends on:

- Performance measure
- Agent's prior knowledge of environment
- Actions available to agent
- Percept sequence to date



Being rational means **maximizing your expected utility**

# Rational Agents

Are rational agents *omniscient*?

- No – they are limited by the available percepts

Are rational agents *clairvoyant*?

- 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., transcend initial program)

# Task Environment - PEAS

## Performance measure

- -1 per step; +10 food; +500 win; -500 die; +200 hit scared ghost

## Environment

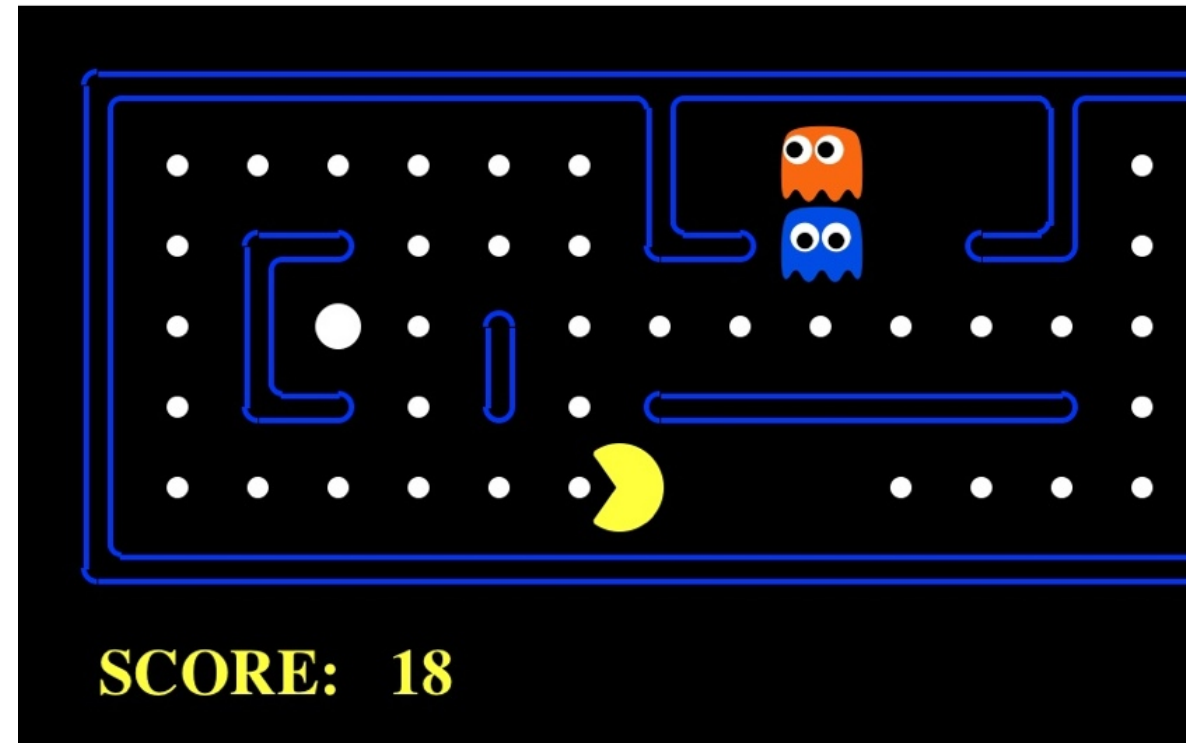
- Pacman dynamics (incl ghost behavior)

## Actuators

- North, South, East, West, (Stop)

## Sensors

- Entire state is visible





# PEAS: Automated Taxi

## Performance measure

- Income, happy customer, vehicle costs, fines, insurance premiums

## Environment

- US streets, other drivers, customers

## Actuators

- Steering, brake, gas, display/speaker

## Sensors

- Camera, radar, accelerometer, engine sensors, microphone



# Environment Types

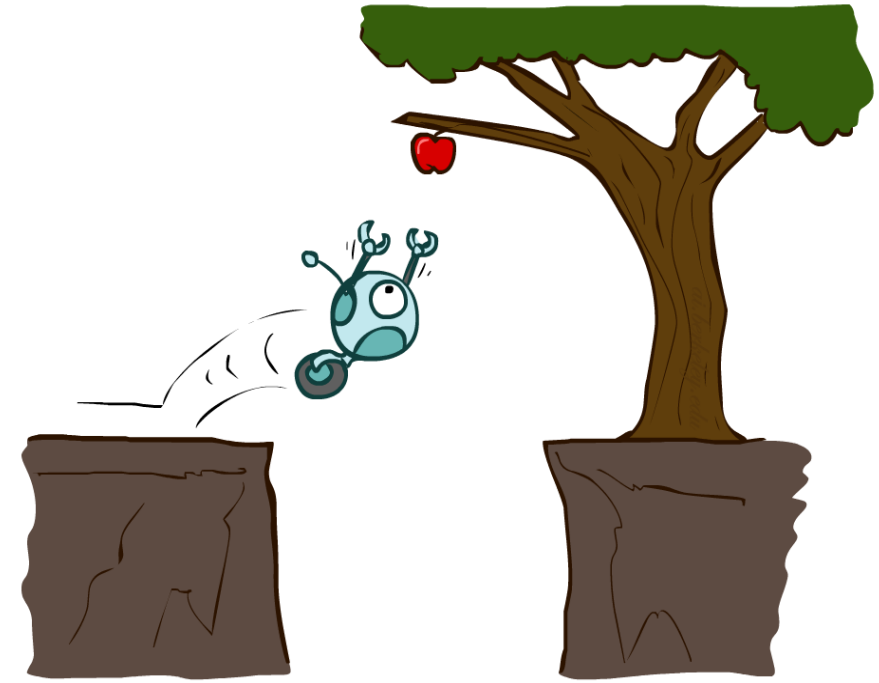
|                               | Pacman | Taxi |
|-------------------------------|--------|------|
| Fully or partially observable |        |      |
| Single agent or multi-agent   |        |      |
| Deterministic or stochastic   |        |      |
| Static or dynamic             |        |      |
| Discrete or continuous        |        |      |

# Reflex Agents

## Reflex agents:

- Choose action based on current percept (and maybe memory)
- May have memory or a model of the world's current state
- Do not consider the future consequences of their actions
- Consider how the world IS

Can a reflex agent be rational?



[Demo: reflex optimal (L2D1)]

[Demo: reflex optimal (L2D2)]

# Demo Reflex Agent

[Demo: reflex optimal (L2D1)]

[Demo: reflex optimal (L2D2)]

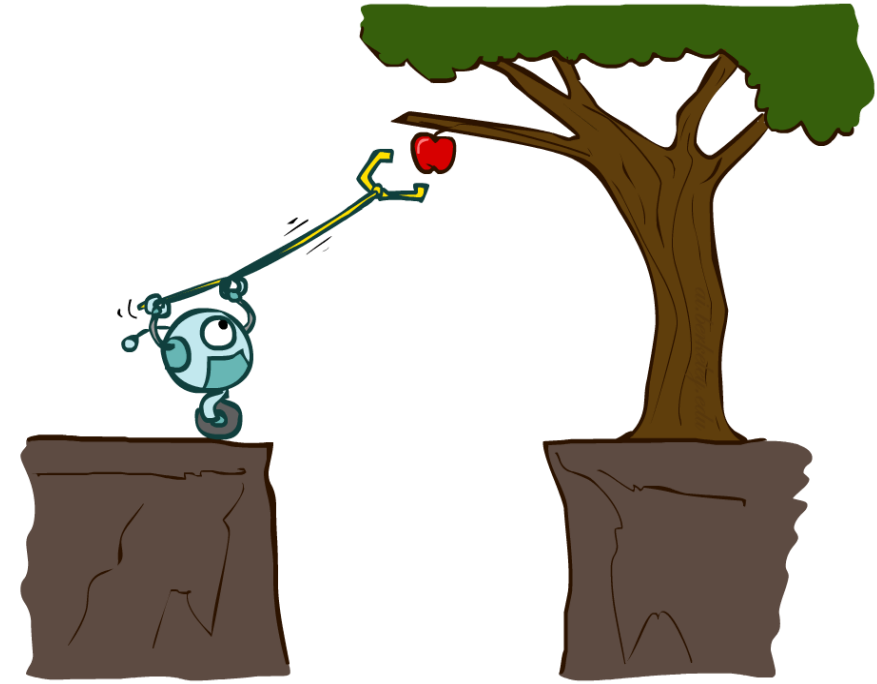
# Agents that Plan Ahead

## Planning agents:

- Decisions based on *predicted consequences* of actions
- Must have a *transition model*: how the world evolves in response to actions
- Must formulate a goal
- 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



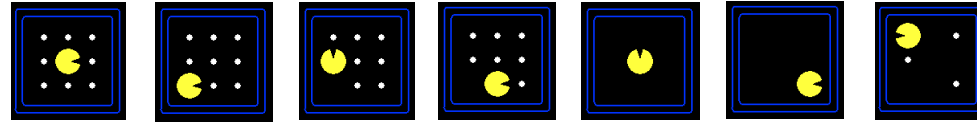
# Search Problems



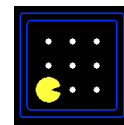
# Search Problems

A **search problem** consists of:

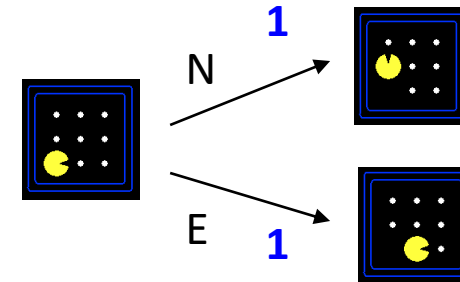
- A state space



- For each state, a set  
Actions(s) of allowable actions



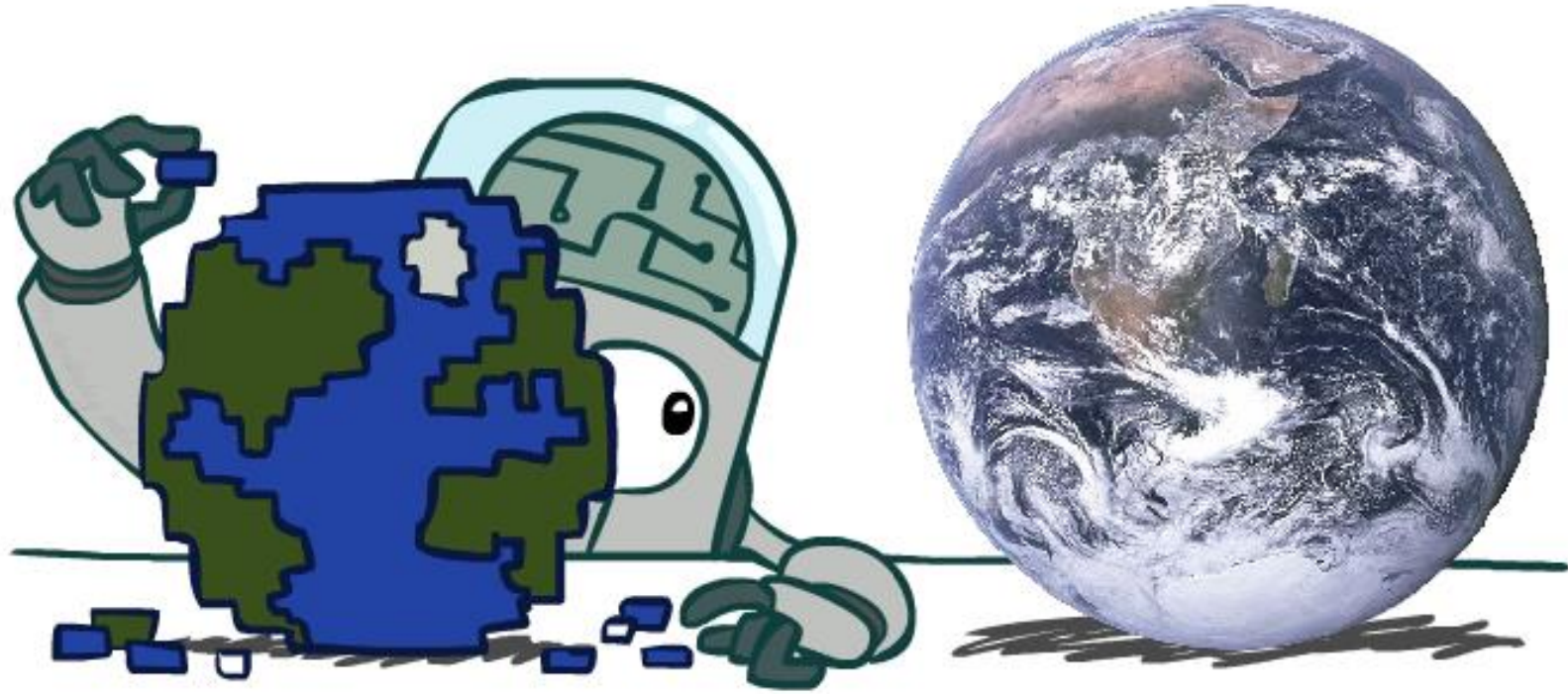
{N, E}



- A transition model  $\text{Result}(s,a)$
- A step cost function  $c(s,a,s')$
- A start state and a goal test

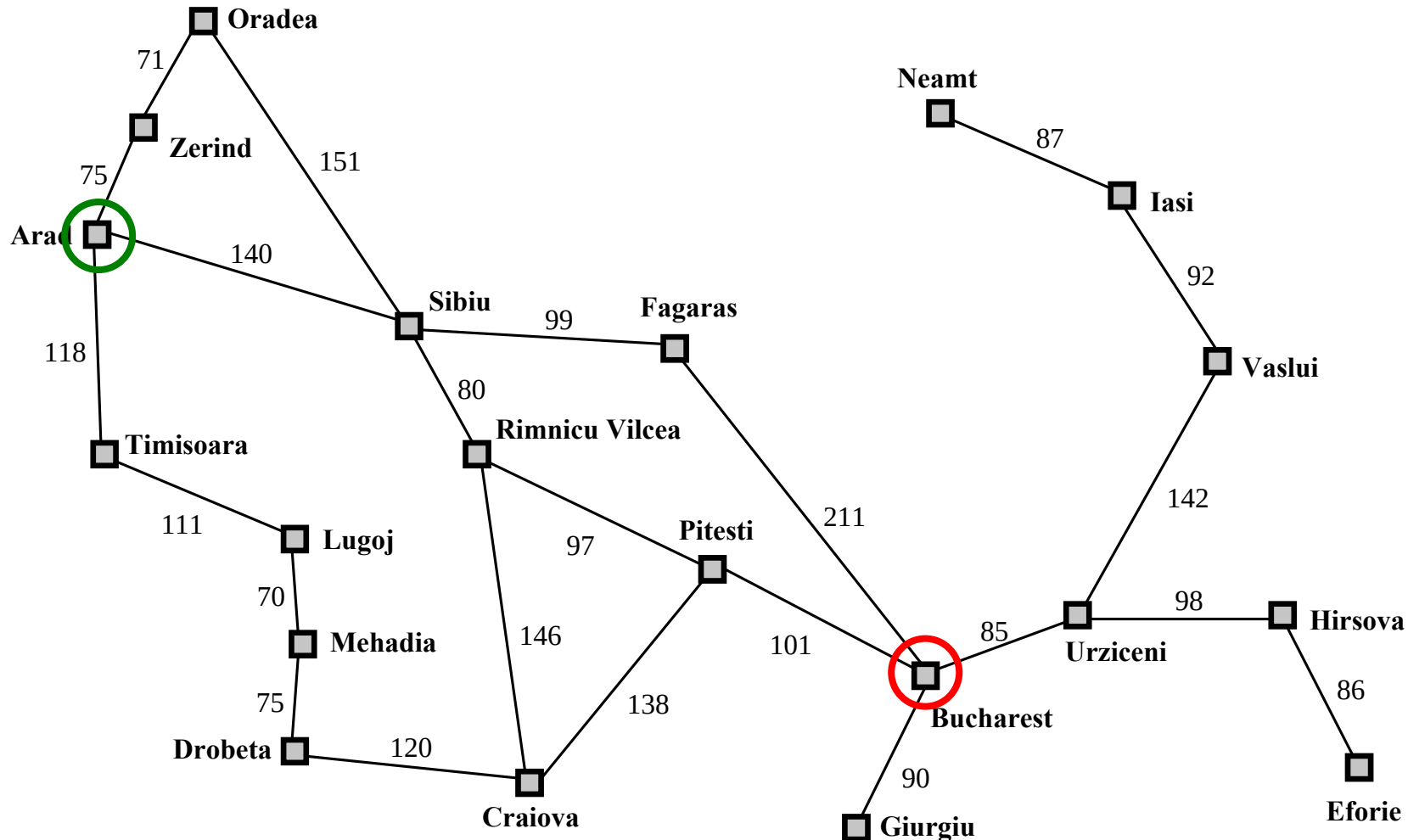
A **solution** is a sequence of actions (a plan) which transforms the start state to a goal state

# Search Problems Are Models





# Example: Travelling in Romania



State space:

- Cities

Actions:

- Go to adjacent city

Transition model

- $\text{Result}(A, \text{Go}(B)) = B$

Step cost

- Distance along road link

Start state:

- Arad

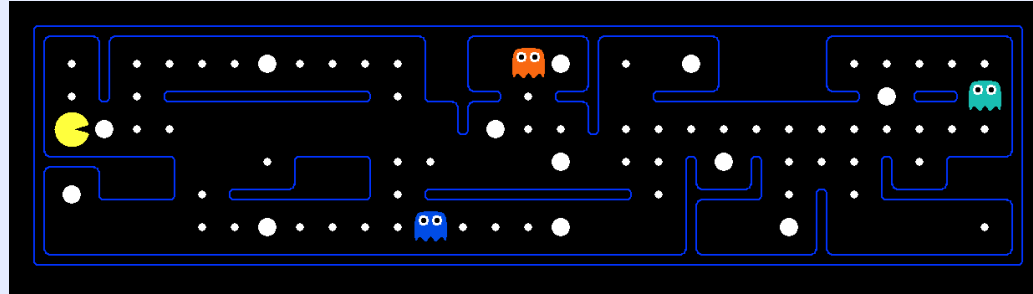
Goal test:

- Is state == Bucharest?

Solution?

# What's in a State Space?

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



A **search state** abstracts away details not needed to solve the problem

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

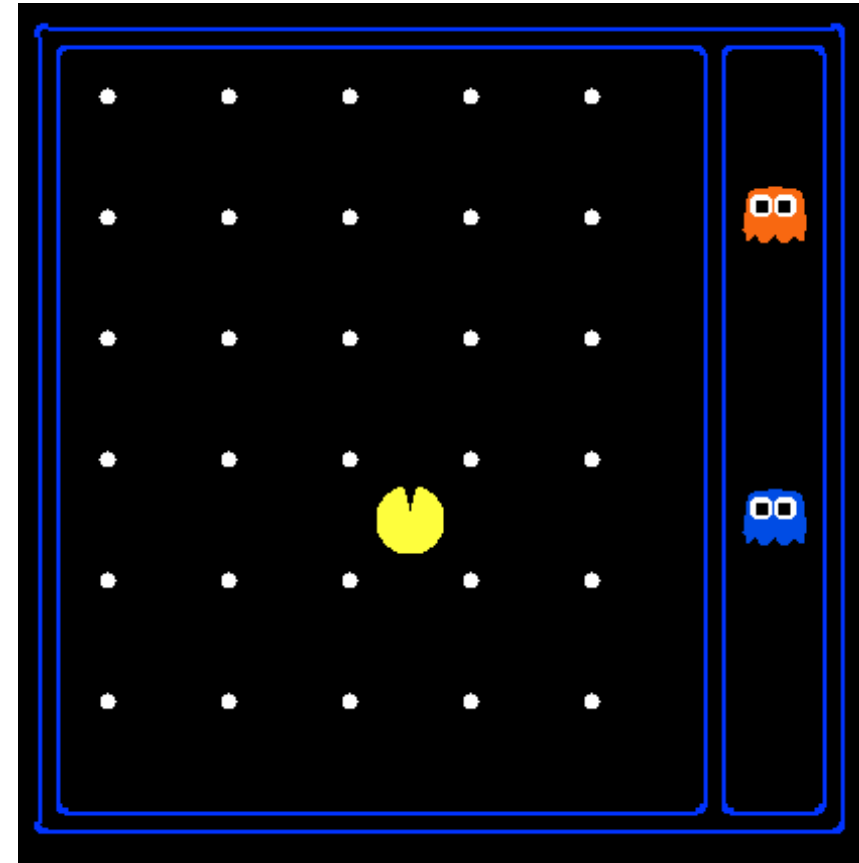
# State Space Sizes?

## World state:

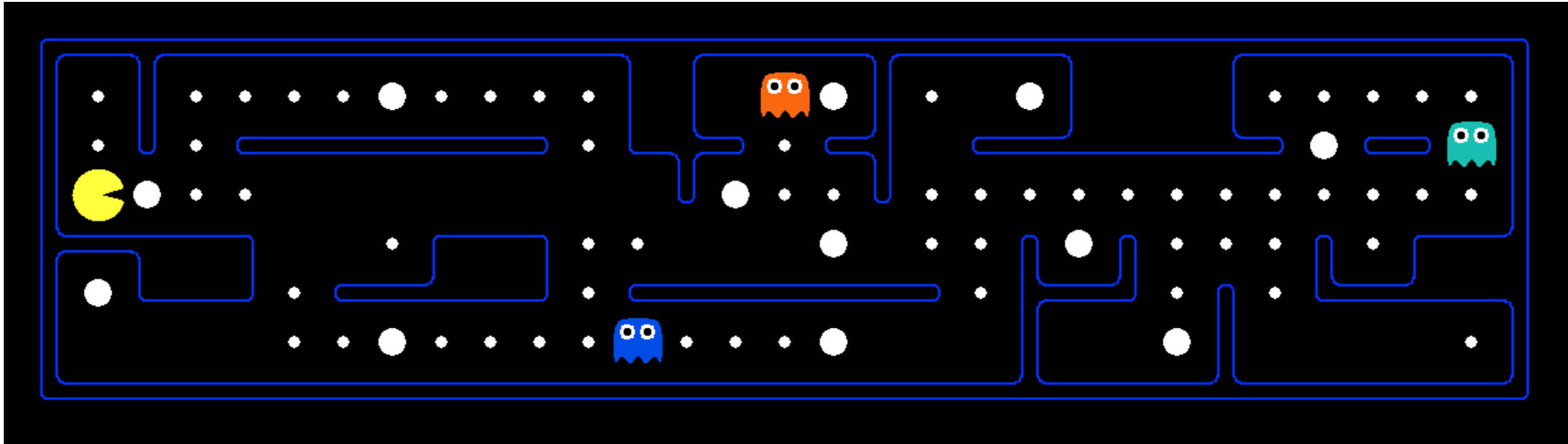
- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: NSEW

## 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

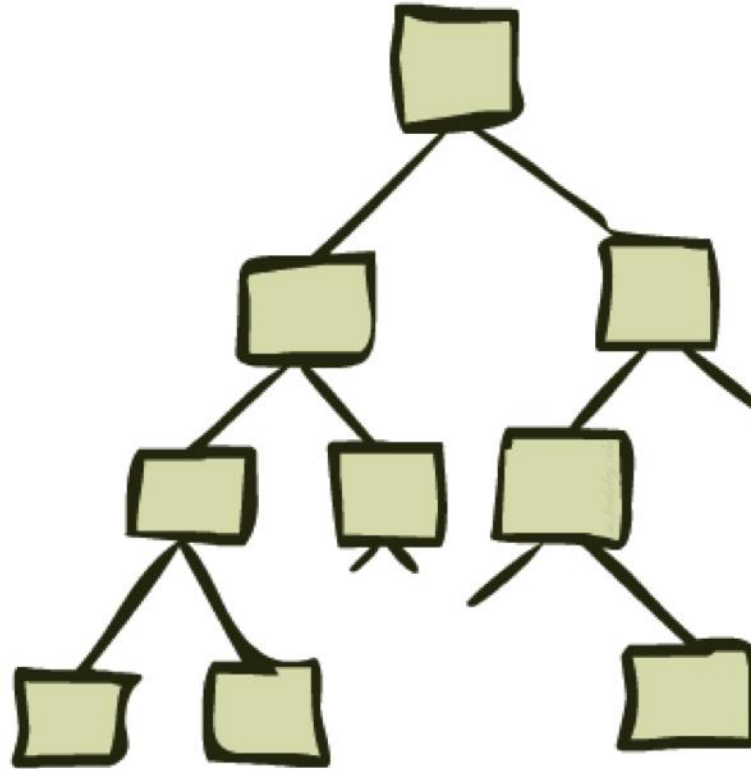


Problem: eat all dots while keeping the ghosts perma-scared

What does the state representation have to specify?

- (agent position, dot booleans, power pellet booleans, remaining scared time)

# State Space Graphs and Search Trees



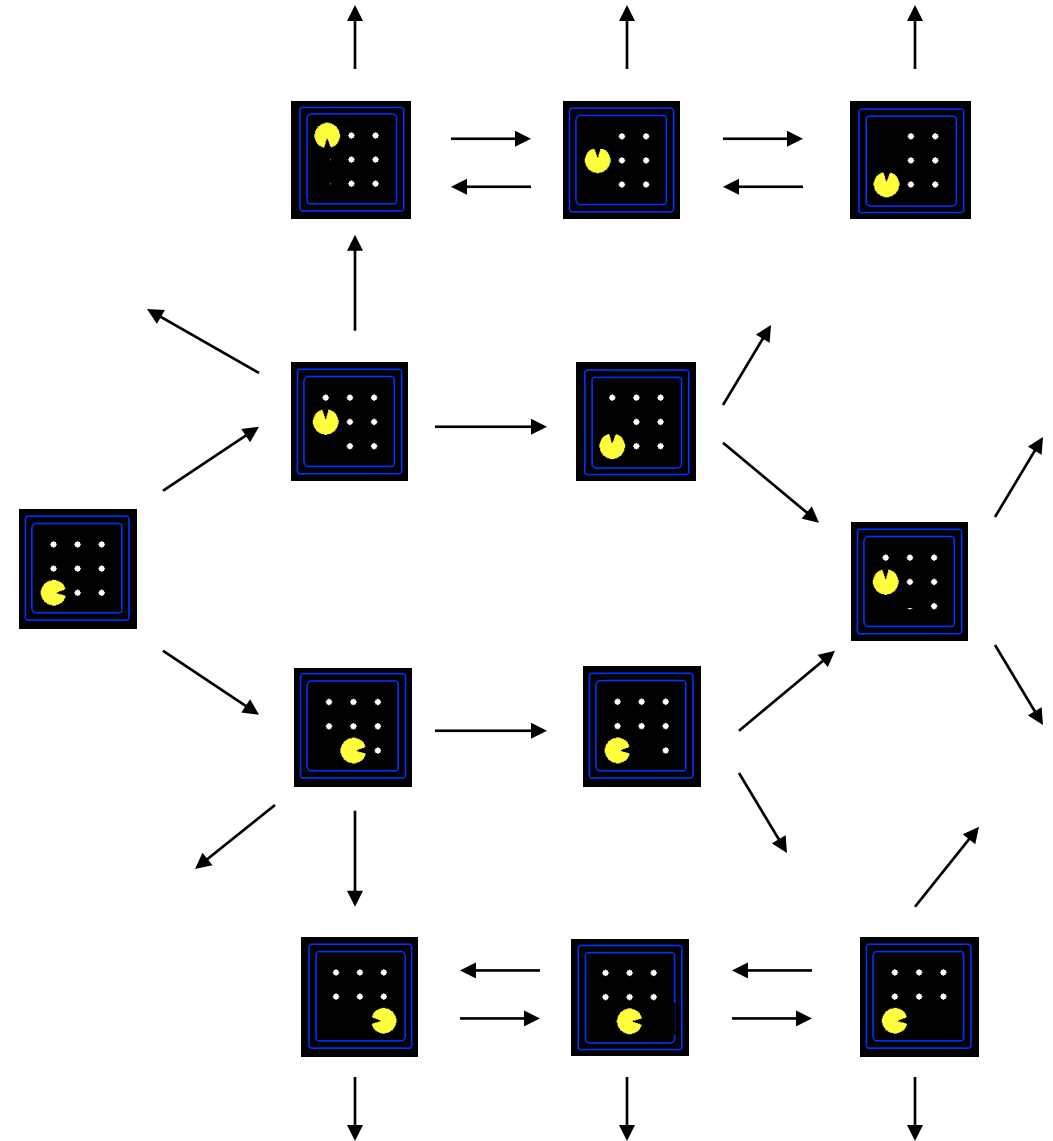
# State Space Graphs

State space graph: A mathematical representation of a search problem

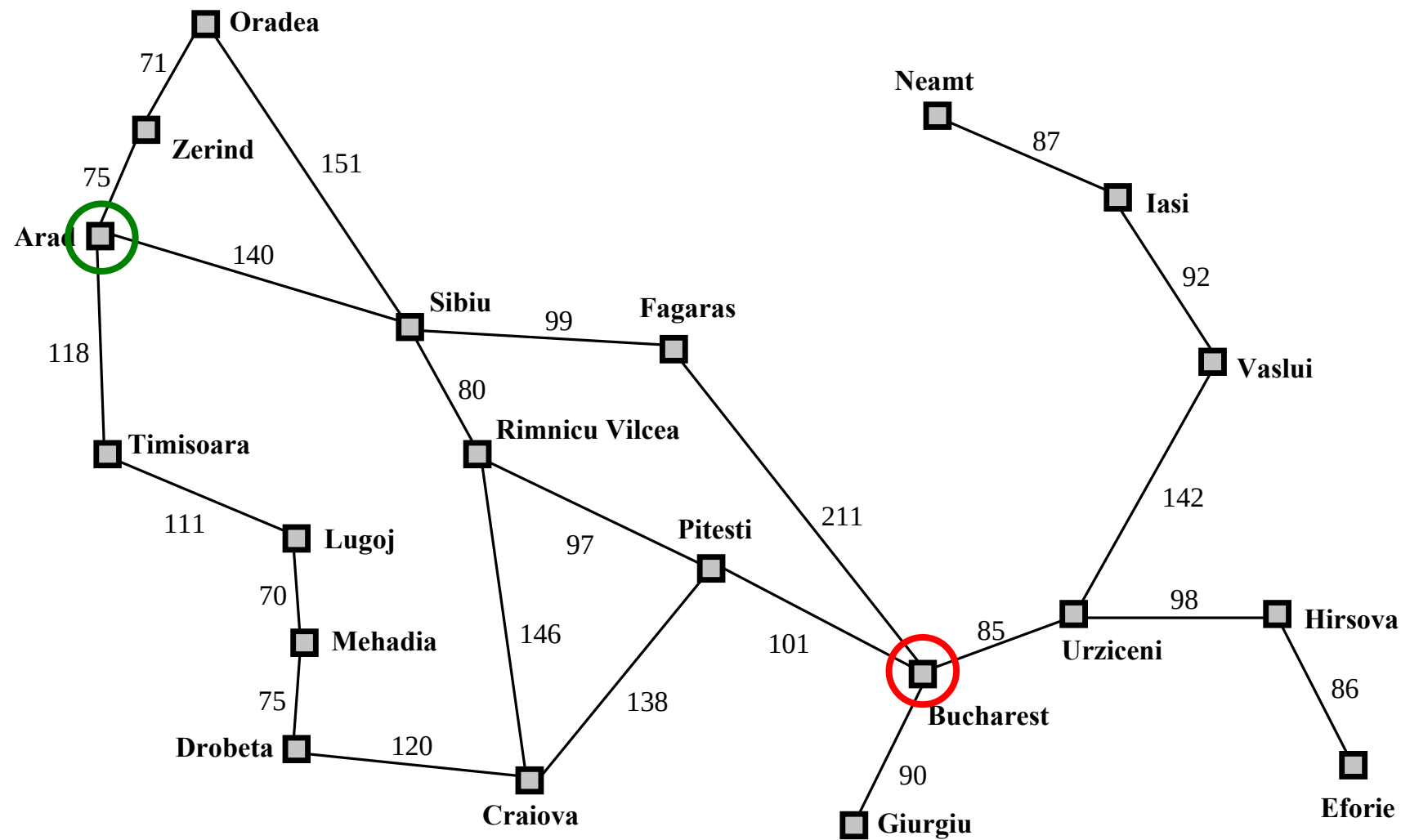
- Nodes are (abstracted) world configurations
- Arcs represent transitions resulting from actions
- The goal test is a set of goal nodes (maybe only one)

In a state space graph, each state occurs only once!

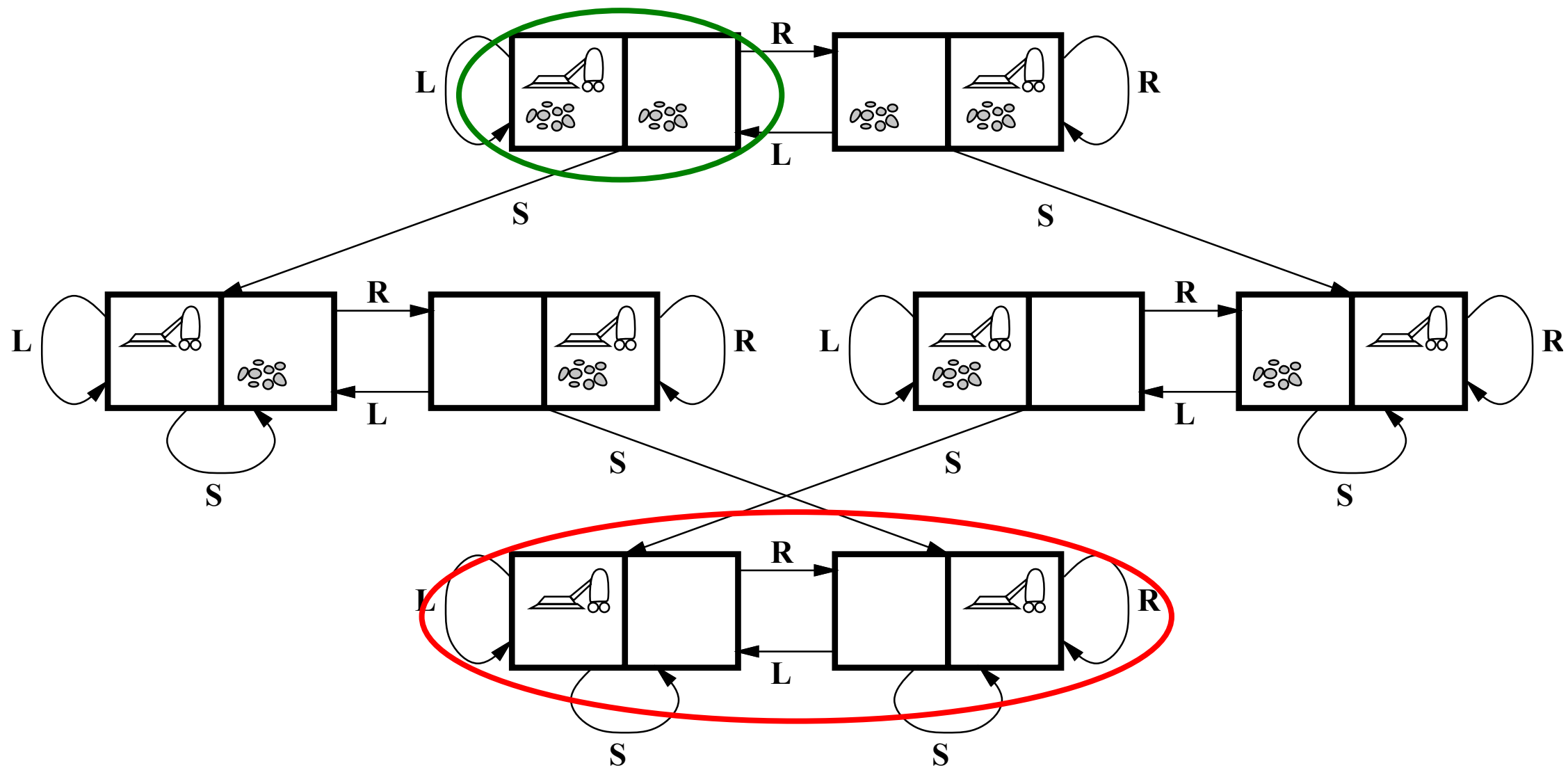
We can rarely build this full graph in memory (it's too big), but it's a useful idea



# More Examples



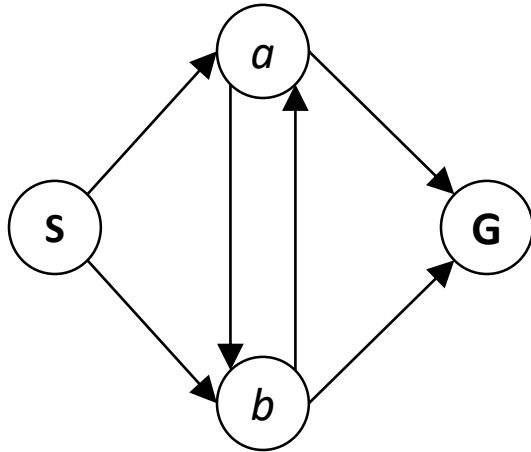
# More Examples



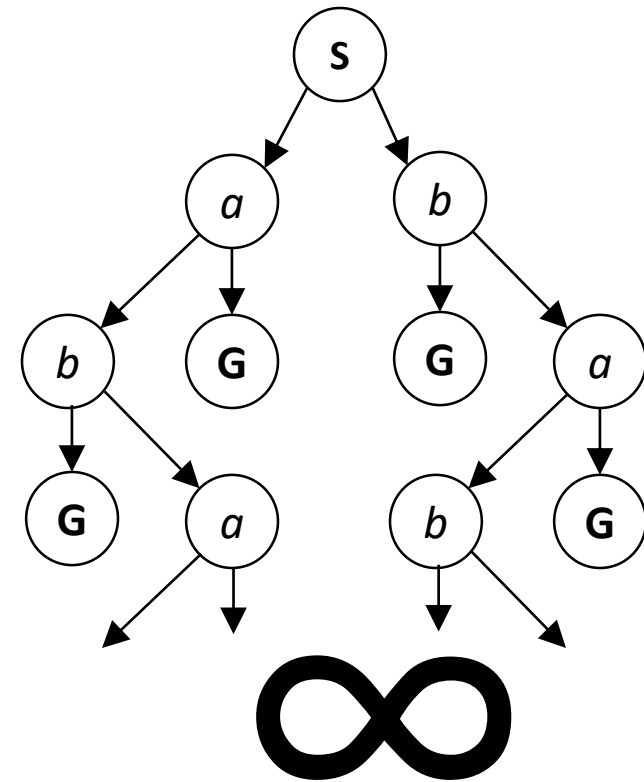


# State Space Graphs vs. Search Trees

Consider this 4-state graph:

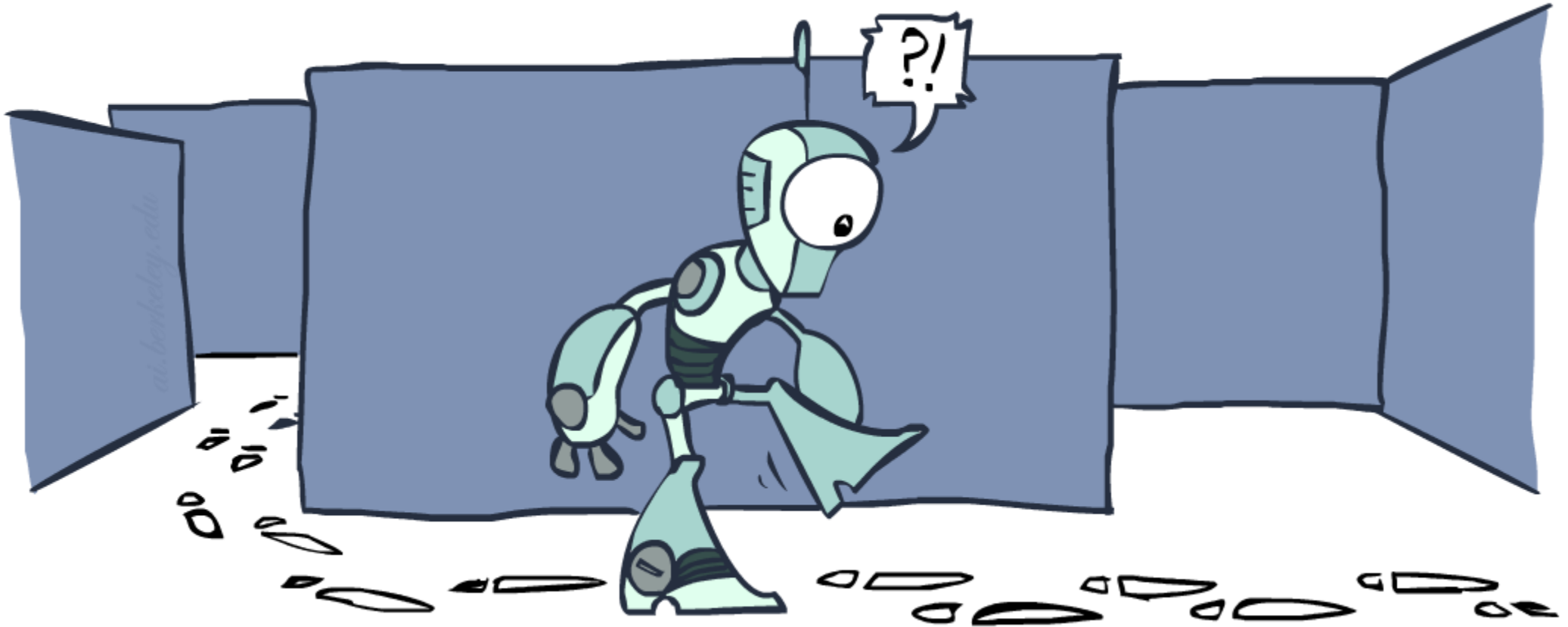


How big is its search tree (from S)?



Important: Lots of repeated structure in the search tree!

# Tree Search vs Graph 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

**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

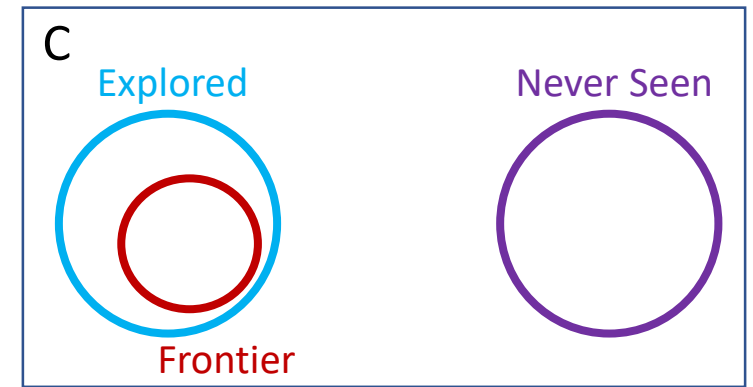
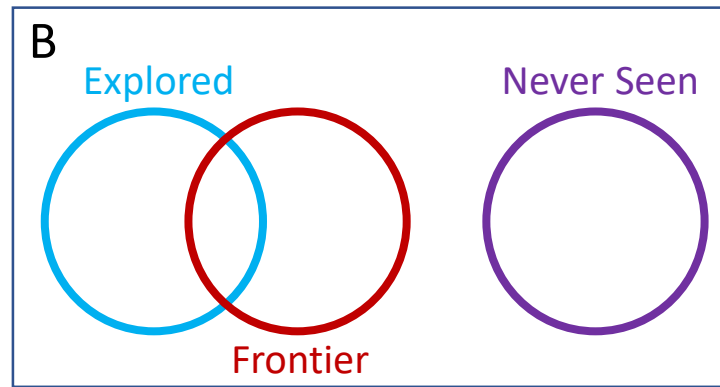
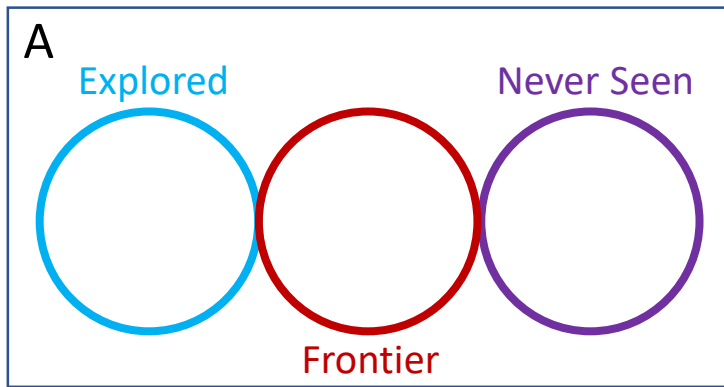
**if** the **child** state is not already in the **frontier** or **explored set** **then**

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

# Piazza Poll

What is the relationship between these sets of states after each loop iteration in **GRAPH\_SEARCH**?

(Loop invariants!!!)



# Piazza Poll

**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

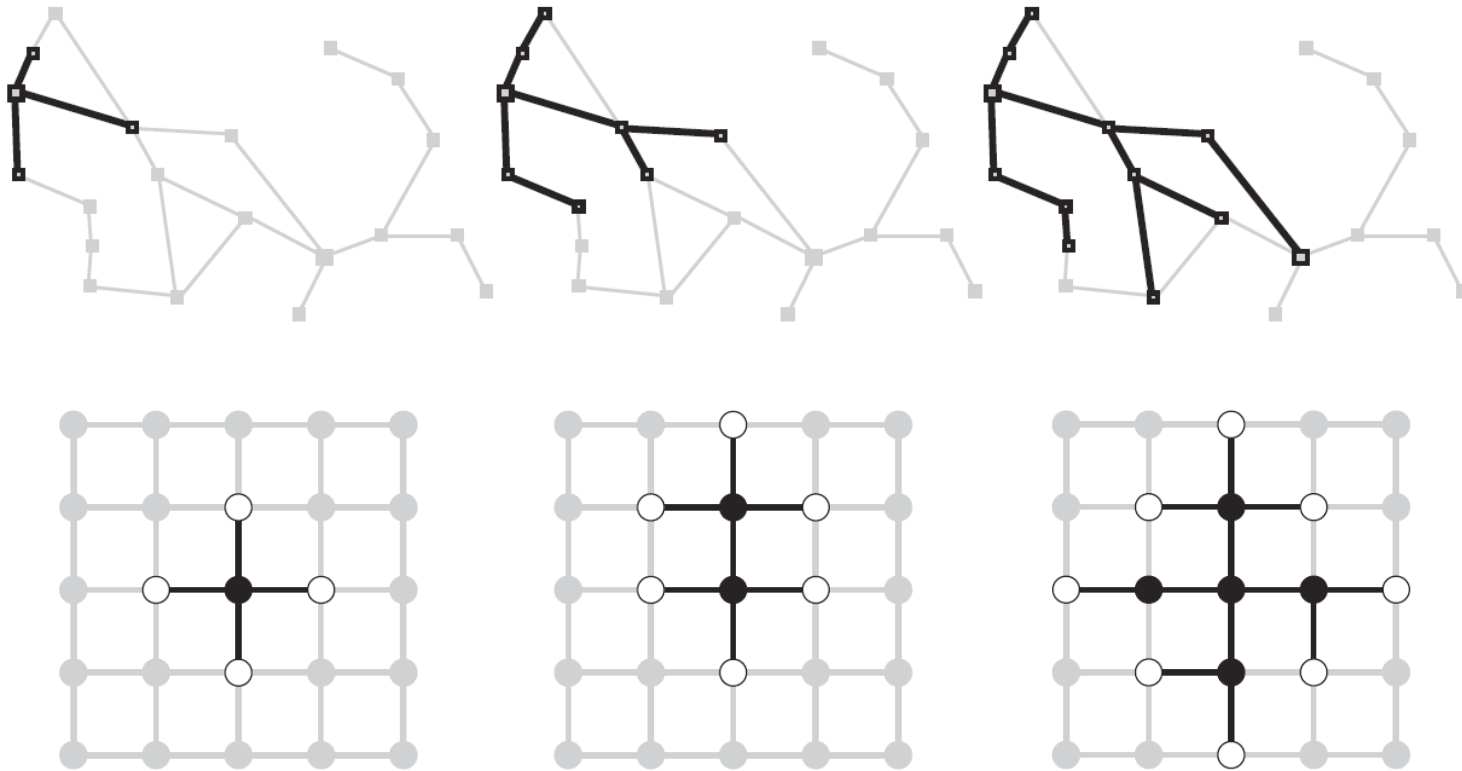
**if** the **child** state is not already in the **frontier** or **explored set** **then**

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

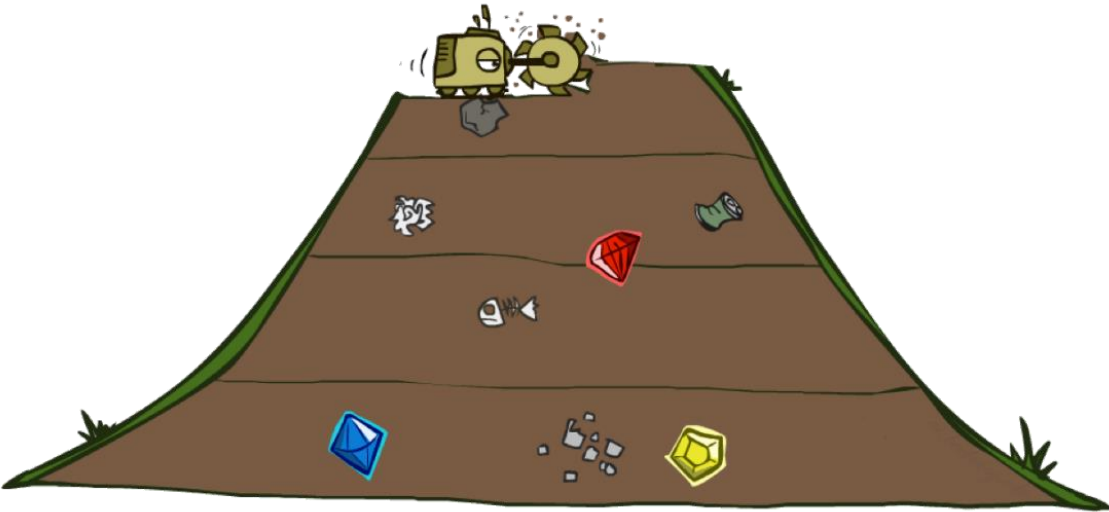
# Graph Search

This graph search algorithm overlays a tree on a graph

The **frontier** states separate the **explored** states from **never seen** states



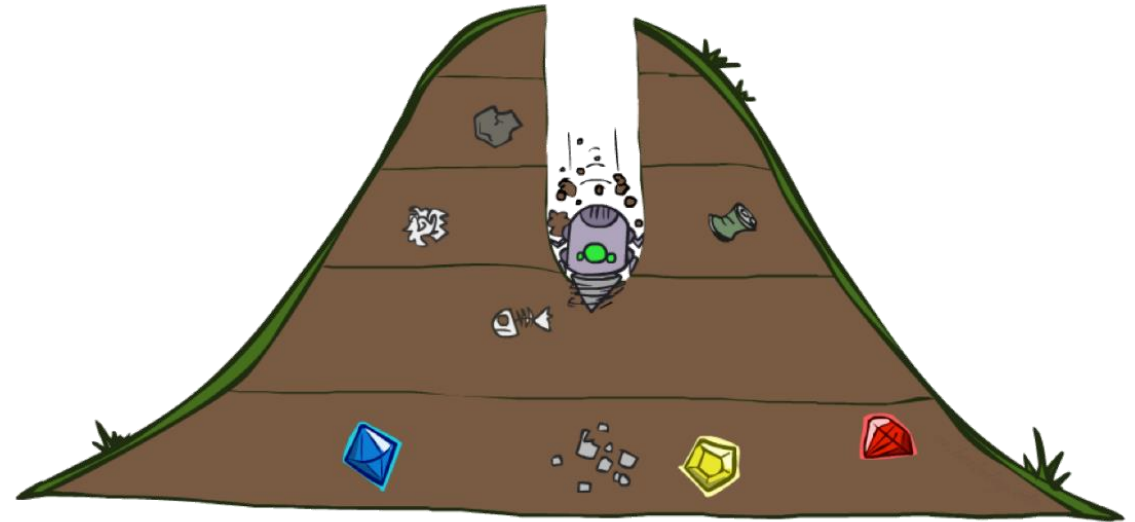
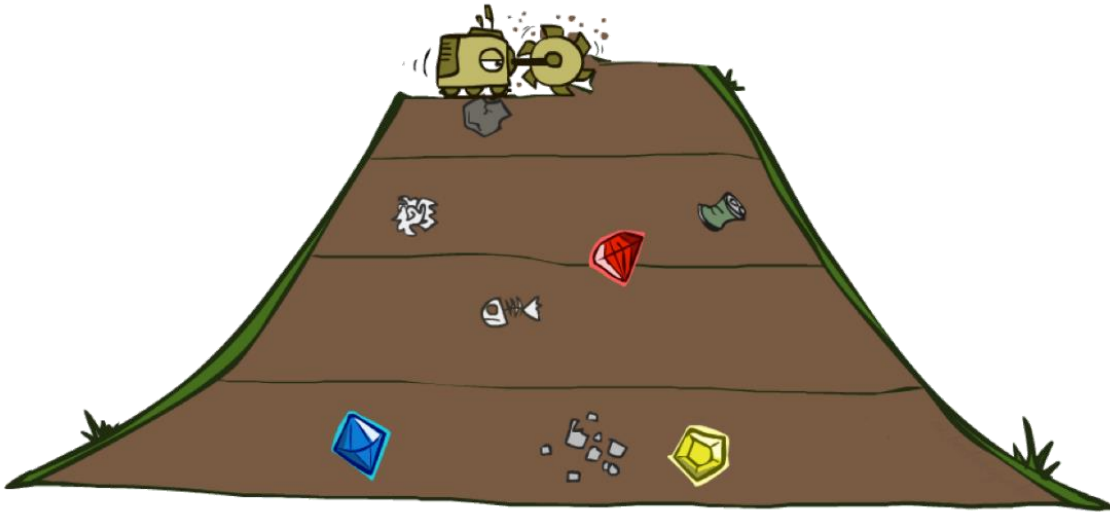
# BFS vs DFS





# Piazza Poll

Is the following demo using BFS or DFS



# A Note on Implementation

Nodes have

state, parent, action, path-cost

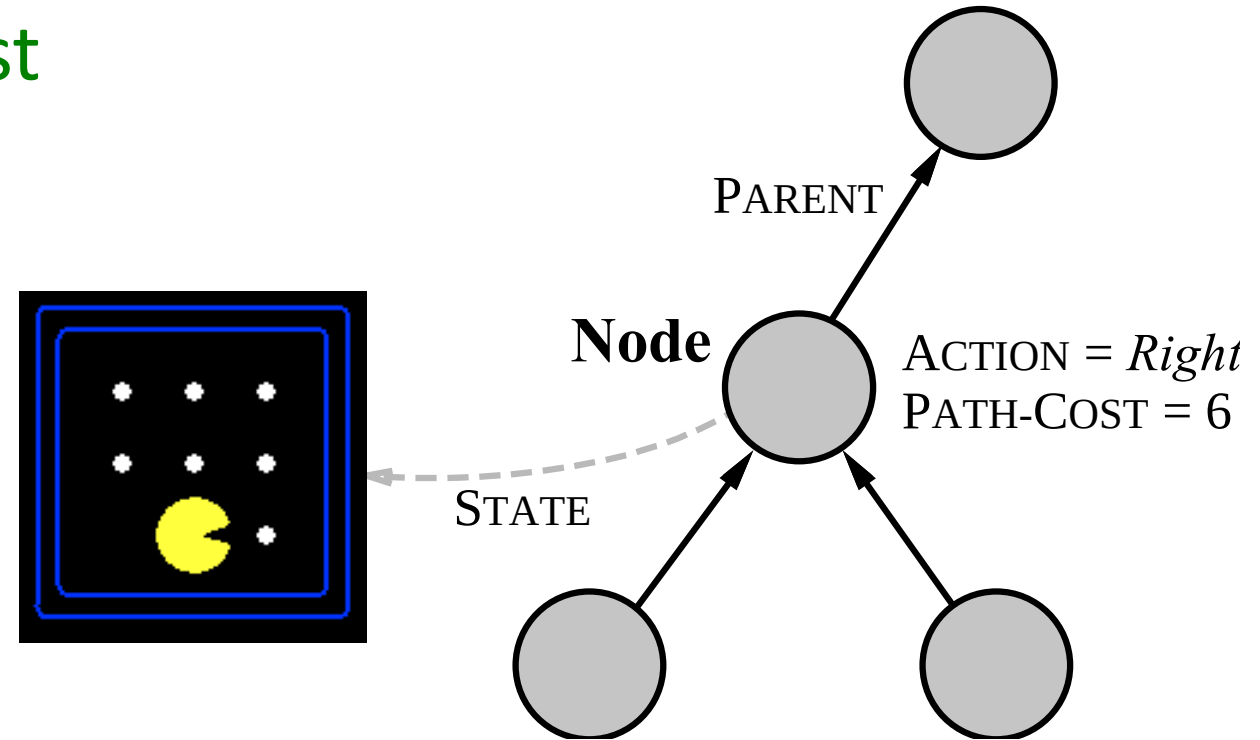
A child of node by action  $a$  has

state = result(node.state,  $a$ )

parent = node

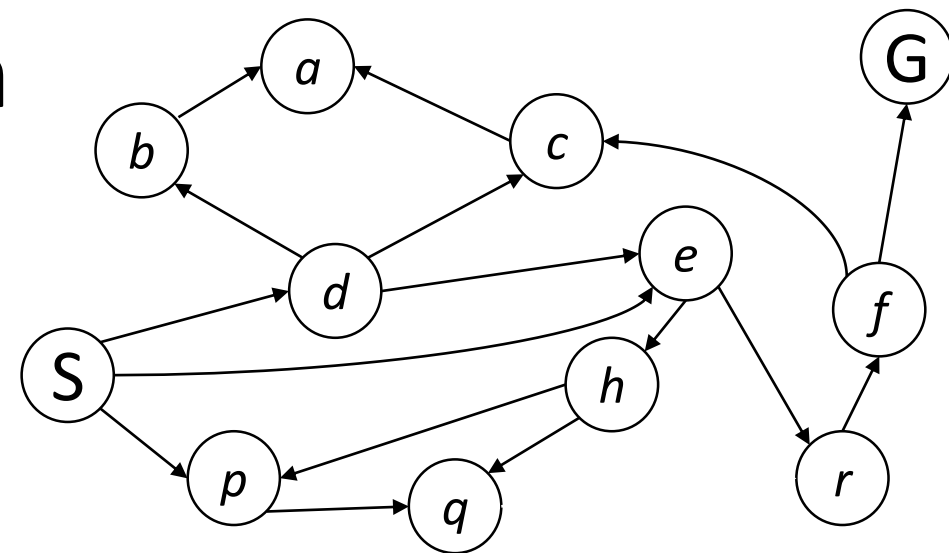
action =  $a$

path-cost = node.path\_cost +  
step\_cost(node.state,  $a$ , self.state)



Extract solution by tracing back parent pointers, collecting actions

# Walk-through DFS Graph Search

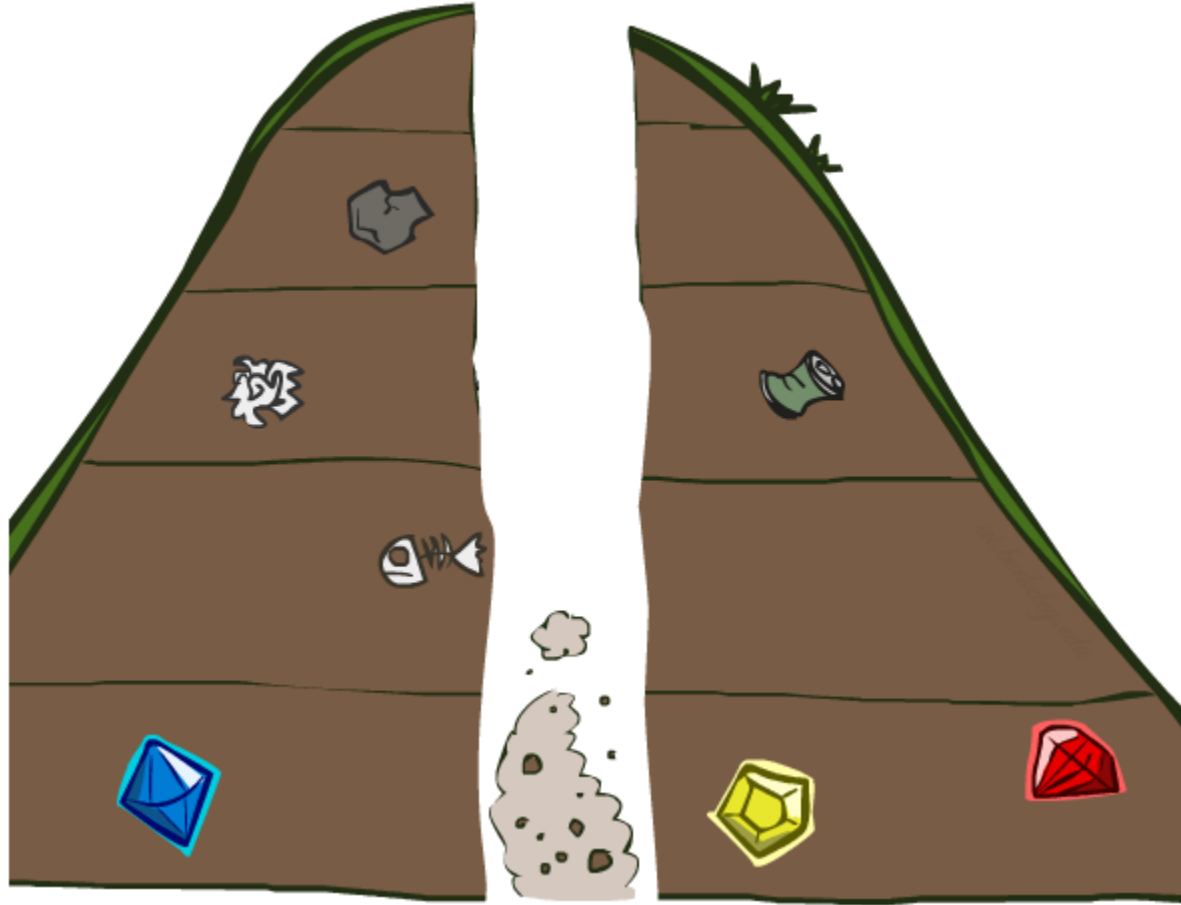


# BFS vs DFS

When will BFS outperform DFS?

When will DFS outperform BFS?

# 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?

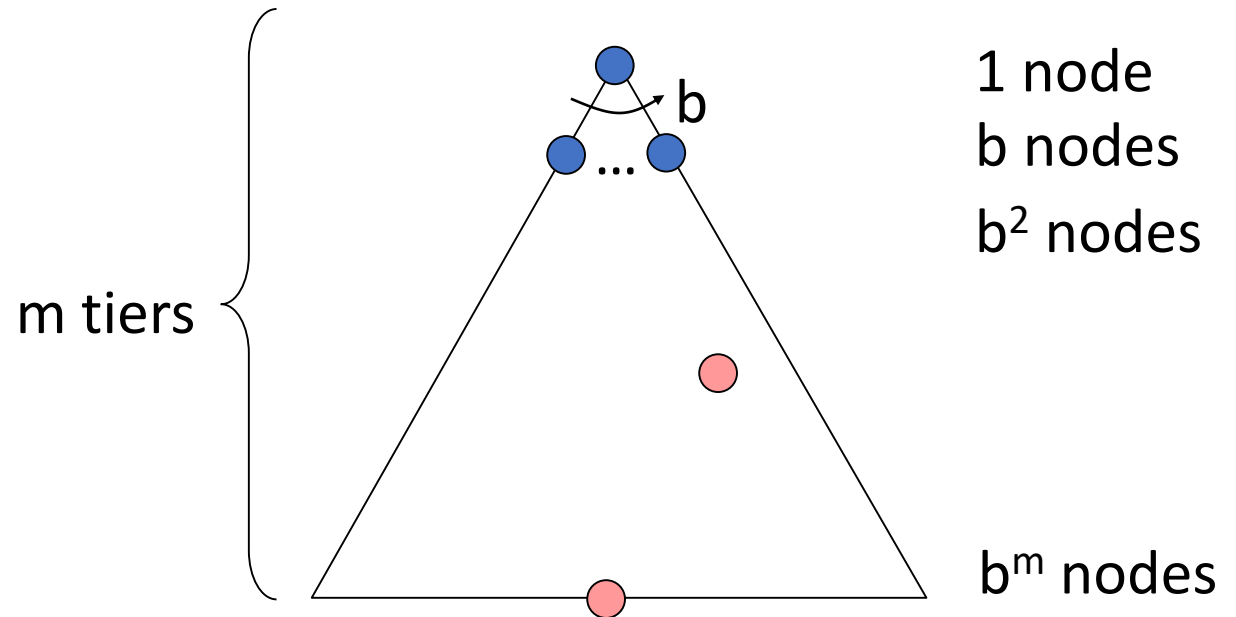
Space complexity?

Cartoon of search tree:

- $b$  is the branching factor
- $m$  is the maximum depth
- solutions at various depths

Number of nodes in entire tree?

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



# Search Algorithm Properties

Complete: Guaranteed to find a solution if one exists?

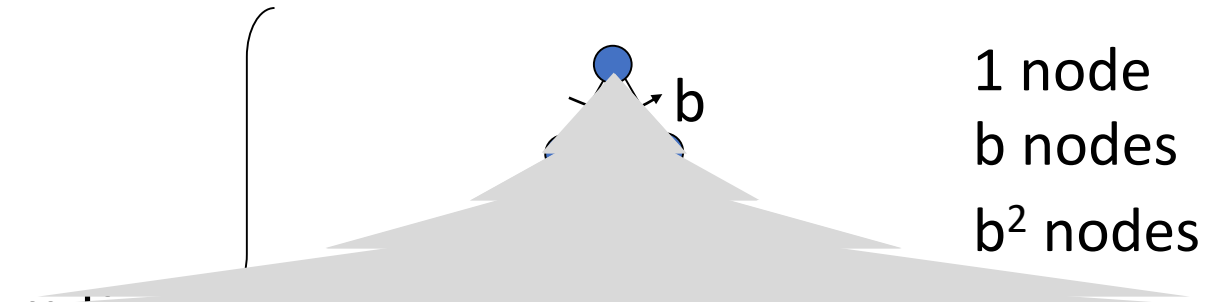
Optimal: Guaranteed to find the least cost path?

Time complexity?

Space complexity?

Cartoon of search tree:

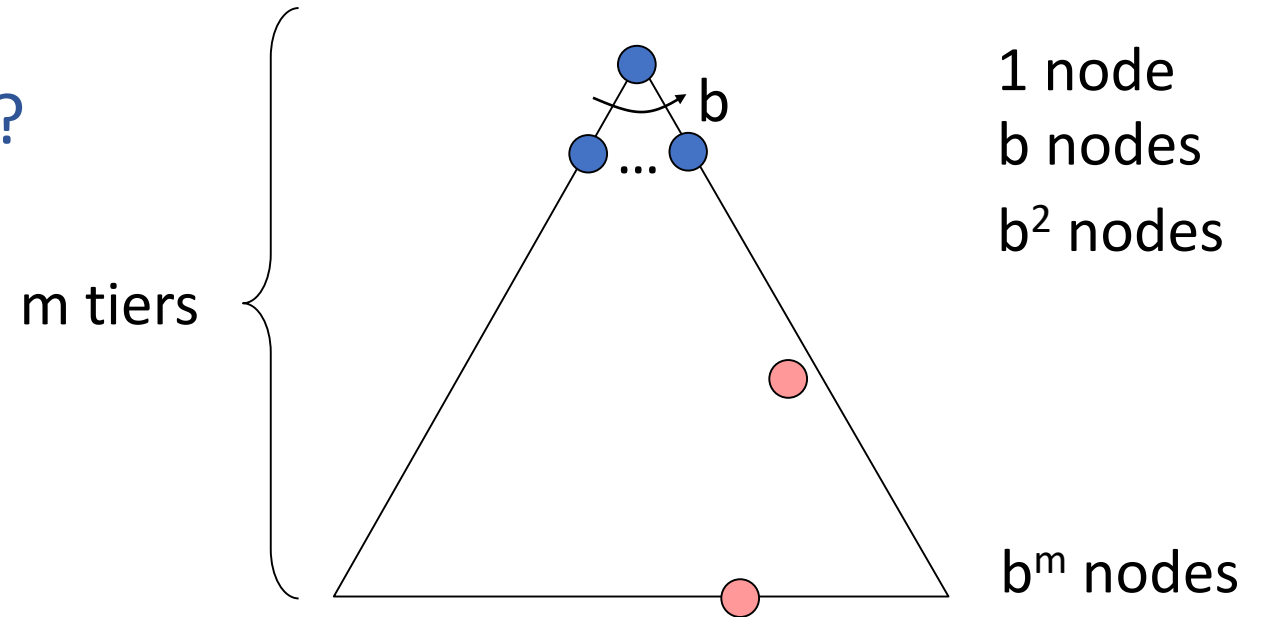
- $b$  is the branching factor



# Piazza Poll

## Are these the properties for BFS or DFS?

- Takes  $O(b^m)$  time
- Uses  $O(bm)$  space on frontier
- Complete with graph search
- Not optimal unless all goals are in the same level (and the same step cost everywhere)





# Depth-First Search (DFS) Properties

## What nodes does 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 frontier take?

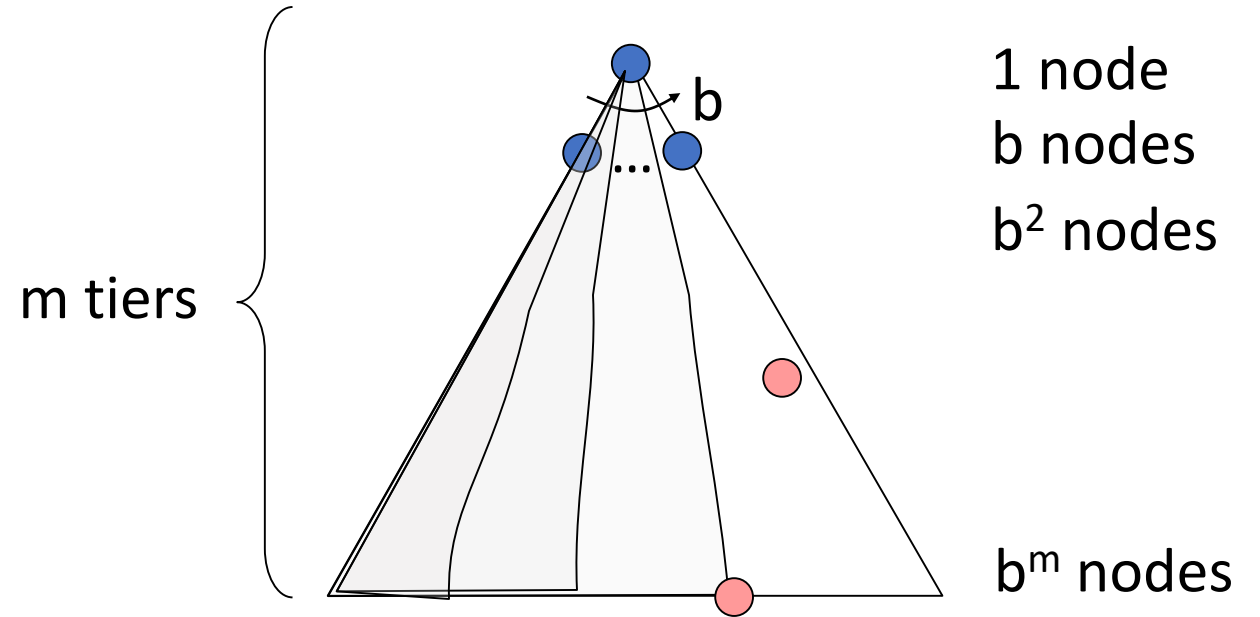
- Only has siblings on path to root, so  $O(bm)$

## Is it complete?

- $m$  could be infinite, so only if we prevent cycles (graph search)

## 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 frontier take?

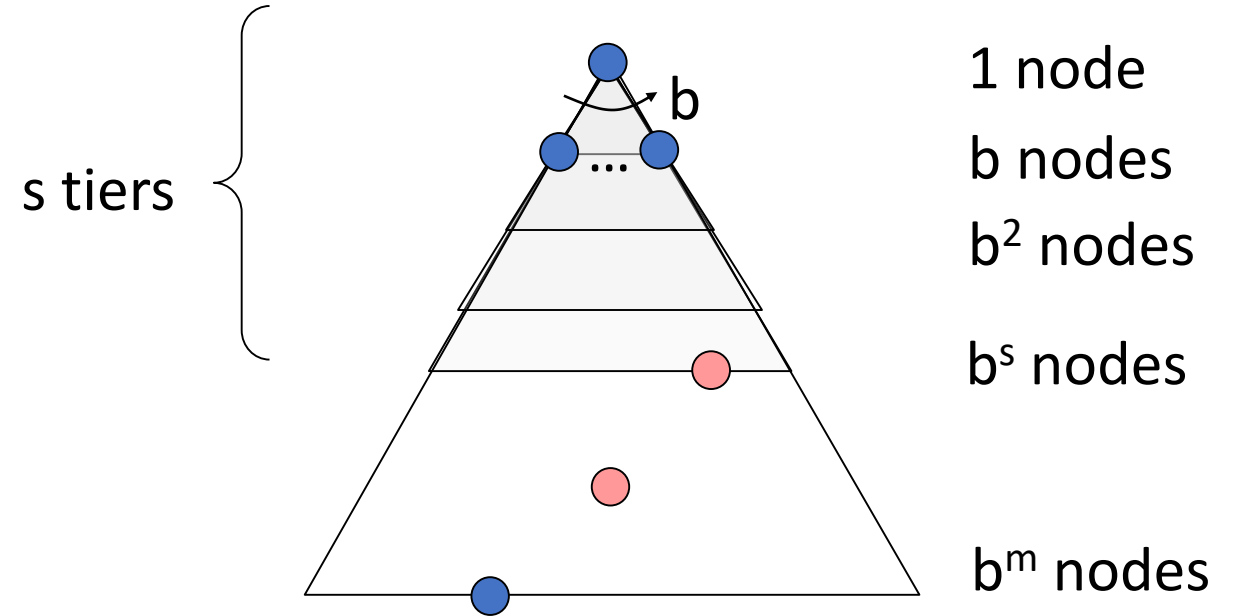
- Has roughly the last tier, so  $O(b^s)$

## Is it complete?

- $s$  must be finite if a solution exists, so yes!

## Is it optimal?

- Only if costs are all the same (more on costs later)



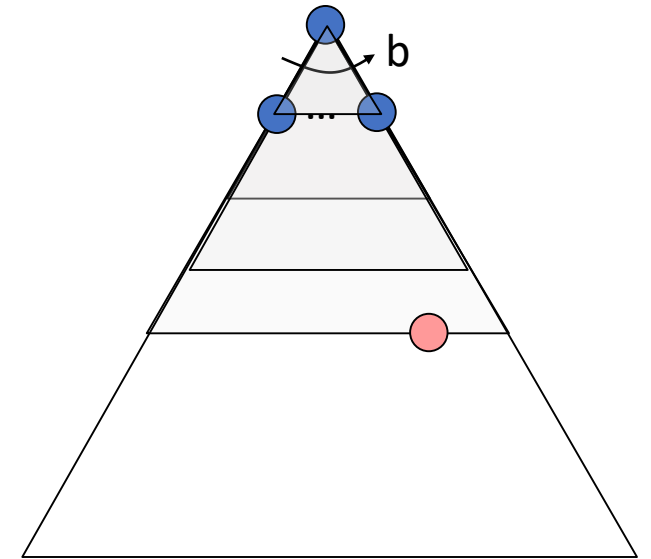
# Iterative Deepening

Idea: get DFS's space advantage with BFS's time / shallow-solution advantages

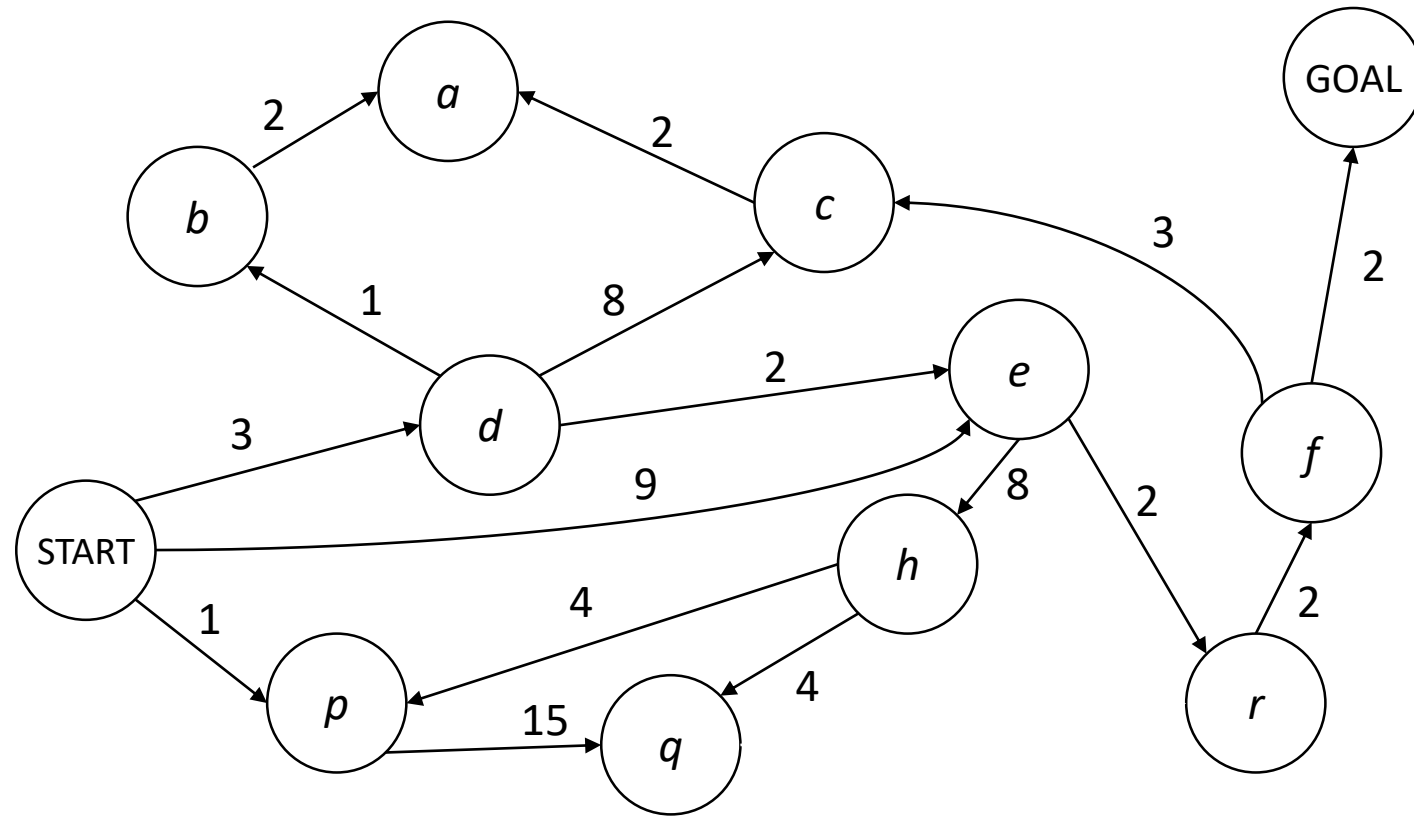
- Run a DFS with depth limit 1. If no solution...
- Run a DFS with depth limit 2. If no solution...
- Run a DFS with depth limit 3. ....

Isn't that wastefully redundant?

- Generally most work happens in the lowest level searched, so not so bad!



# Finding a Least-Cost Path

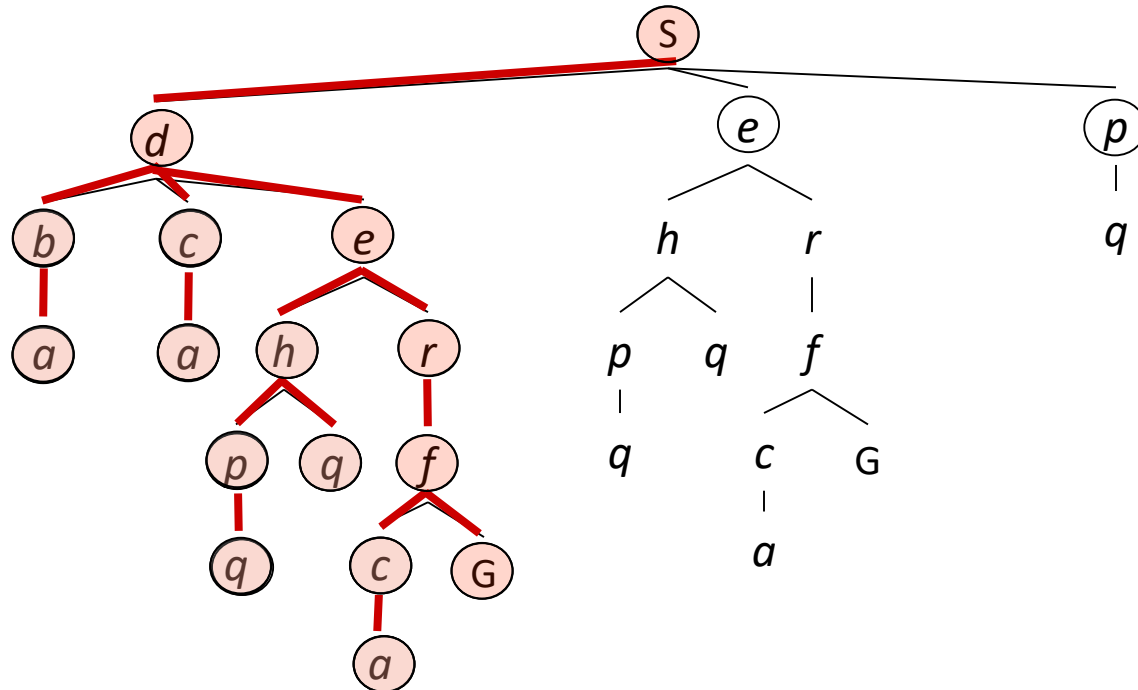
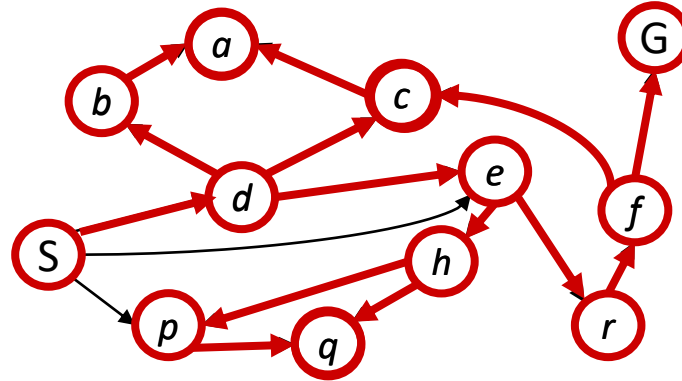


# Depth-First (Tree) Search

*Strategy: expand a deepest node first*

*Implementation:*

## Frontier is a LIFO stack

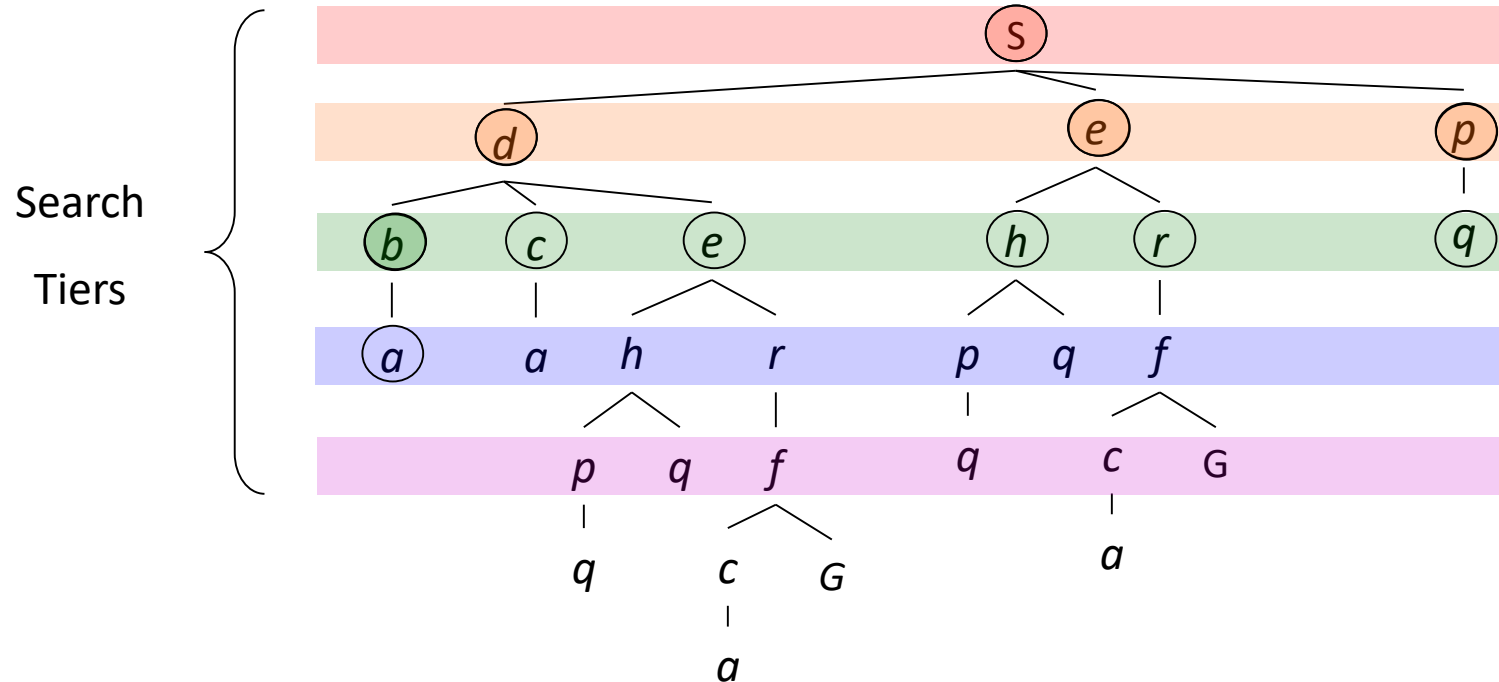
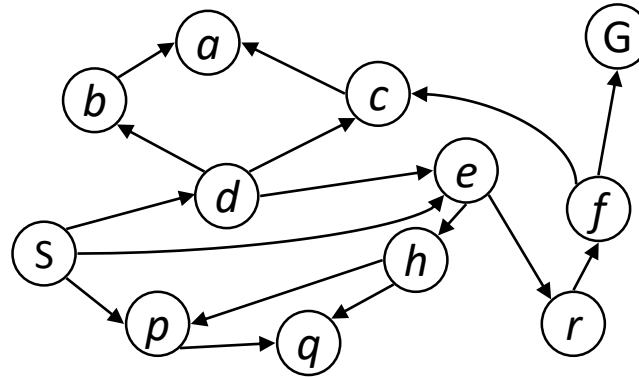


# Breadth-First (Tree) Search

Strategy: expand a  
shallowest node first

Implementation:

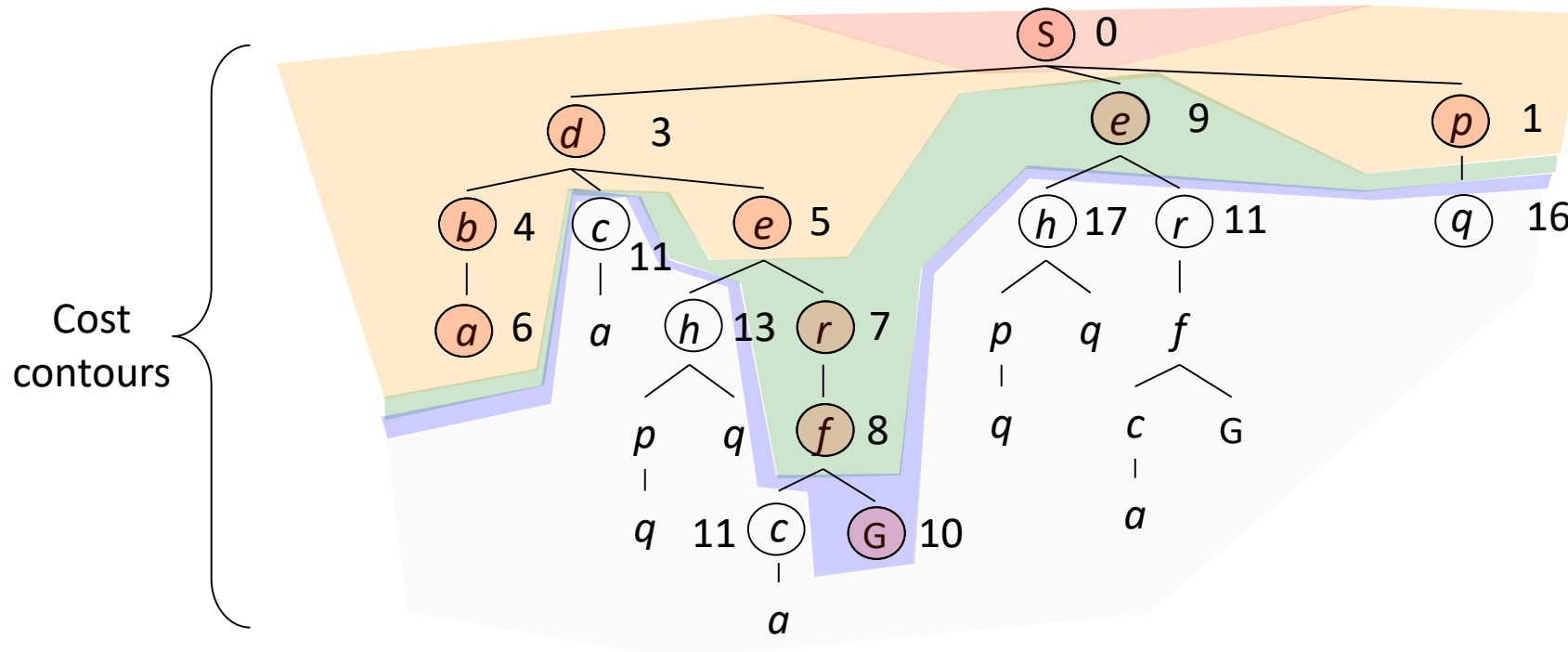
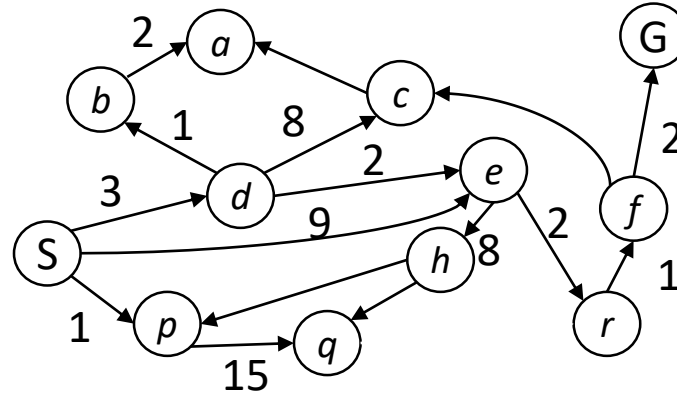
**Frontier is a FIFO queue**



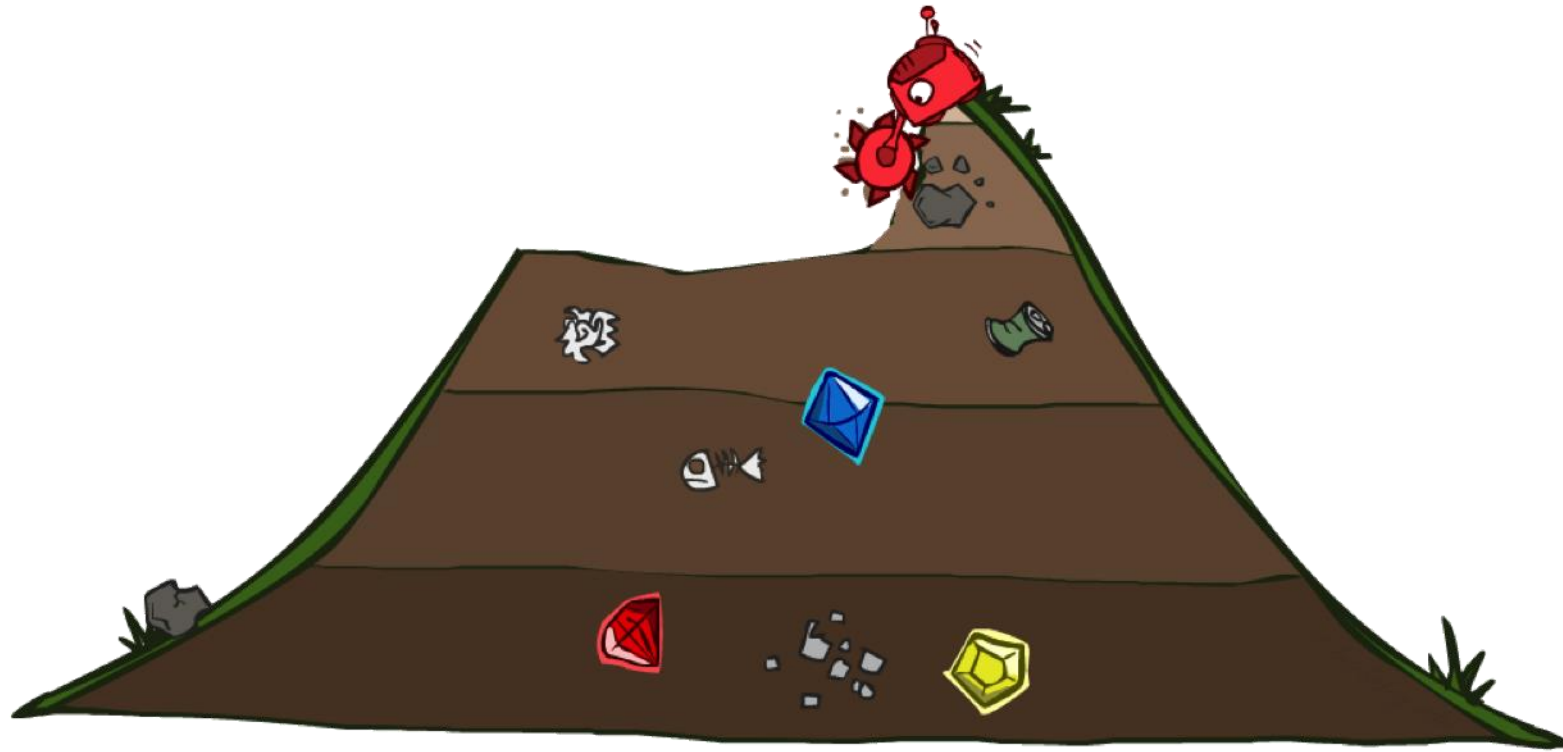
# Uniform Cost (Tree) Search

Strategy: expand a cheapest node first:

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



# Uniform Cost Search





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

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

initialize the **explored set** to be empty

initialize the **frontier** as a **priority queue** using node **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

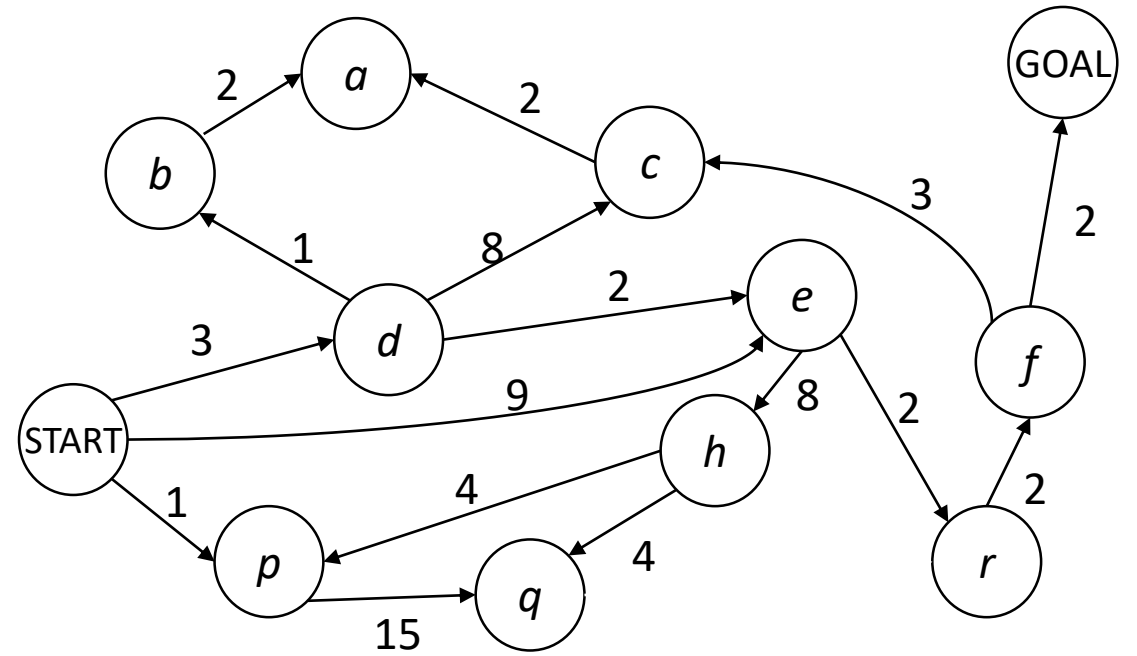
**if** the **child** state is not already in the **frontier** or **explored set** **then**

            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**

# 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  $\varepsilon$ , then the “effective depth” is roughly  $C^*/\varepsilon$
- Takes time  $O(b^{C^*/\varepsilon})$  (exponential in effective depth)

## How much space does the frontier take?

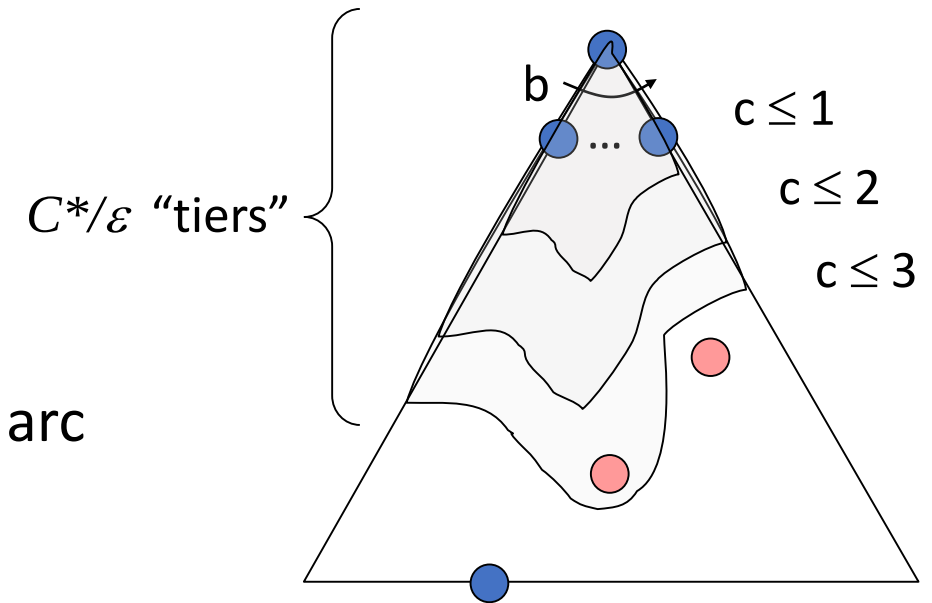
- Has roughly the last tier, so  $O(b^{C^*/\varepsilon})$

## Is it complete?

- Assuming best solution has a finite cost and minimum arc cost is positive, yes!

## Is it optimal?

- Yes! (Proof next lecture 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!

