

# Warm-up as you come in

Back of last time's activity sheet (also on website by the schedule; in-class poll coming soon...)

(If you've seen this before:) 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: [conitzer@cs.cmu.edu](mailto:conitzer@cs.cmu.edu)

Recitation starting this Friday

- Start P0 before recitation to make sure Python 3.10 is working for you
- Choose any section for first two weeks
- We'll have you commit to a recitation section after the third week
- Priority given to students in the section
- Stay tuned to Piazza for form to informally switch sections if space

# Announcements

## Assignments:

- P0: Python & Autograder Tutorial
  - Due Friday 9/1, 10 pm
  - No pairs, submit individually
  - No OH on Fridays!
- HW1 (online)
  - Out
  - Due Tuesday 9/5, 10 pm
- P1 out soon!

Remaining programming assignments may be done in pairs

Note about course grading...

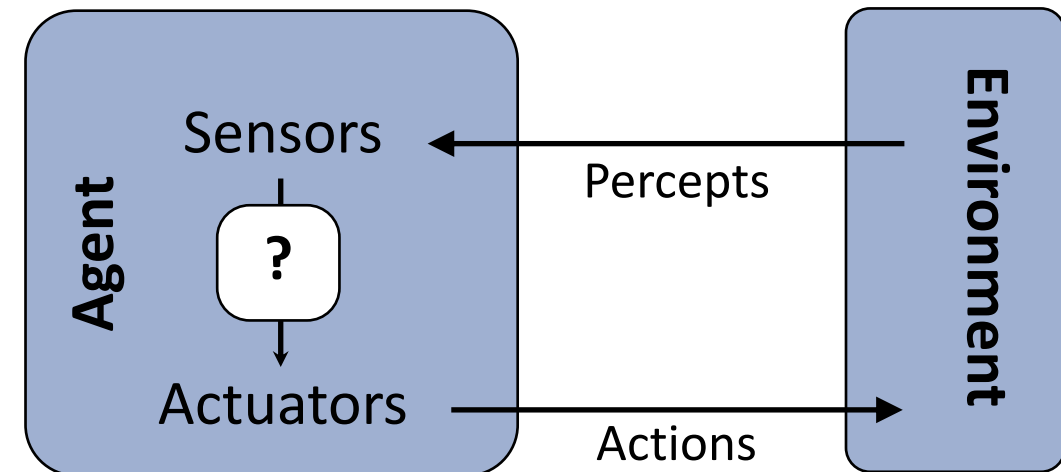
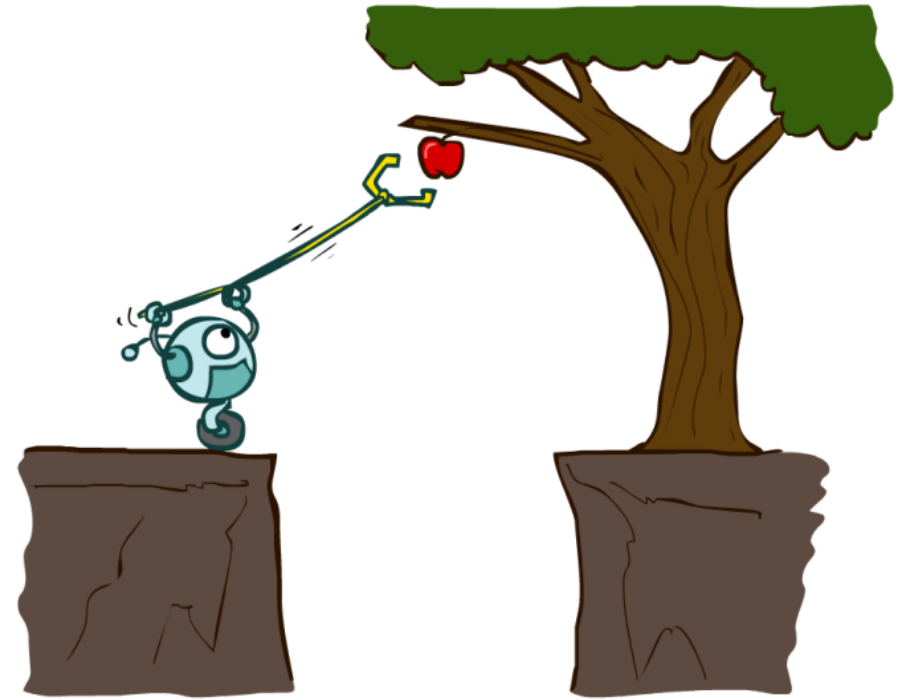
# Designing Agents

An **agent** is an entity that *perceives* and *acts*.

Characteristics of the **percepts and state, environment, and action space** dictate techniques for selecting actions

This course is about:

- General AI techniques for a variety of problem types
- Learning to recognize when and how a new problem can be solved with an existing technique



# Missed Lecture 1 Activity

- 1) An agent controls the elevator in a 10-story building. On each floor, the doors can be open or closed. The elevator can also be ``moving'' between floors. How many states could the agent be in?

Single or Multi-agent?

Discrete or Continuous states?

Static or Dynamic environment?

Deterministic or Stochastic actions?

Fully observable or partially states?

# Missed Lecture 1 Activity

Hopscotch is a game where **10 squares** are drawn and labeled 1-10. There is also a “**start state**” to stand on to throw a stone. A player **throws the stone** and then **hops the squares** in order, **avoiding the one with the stone** in it. Other players watch.

Consider the **states where two players** – scotcher and observer - **and the stone are situated** in the middle of the game. Ignore the state where the player is holding the stone, but do consider when they have not started jumping yet. Assume the game is played on a flat surface.

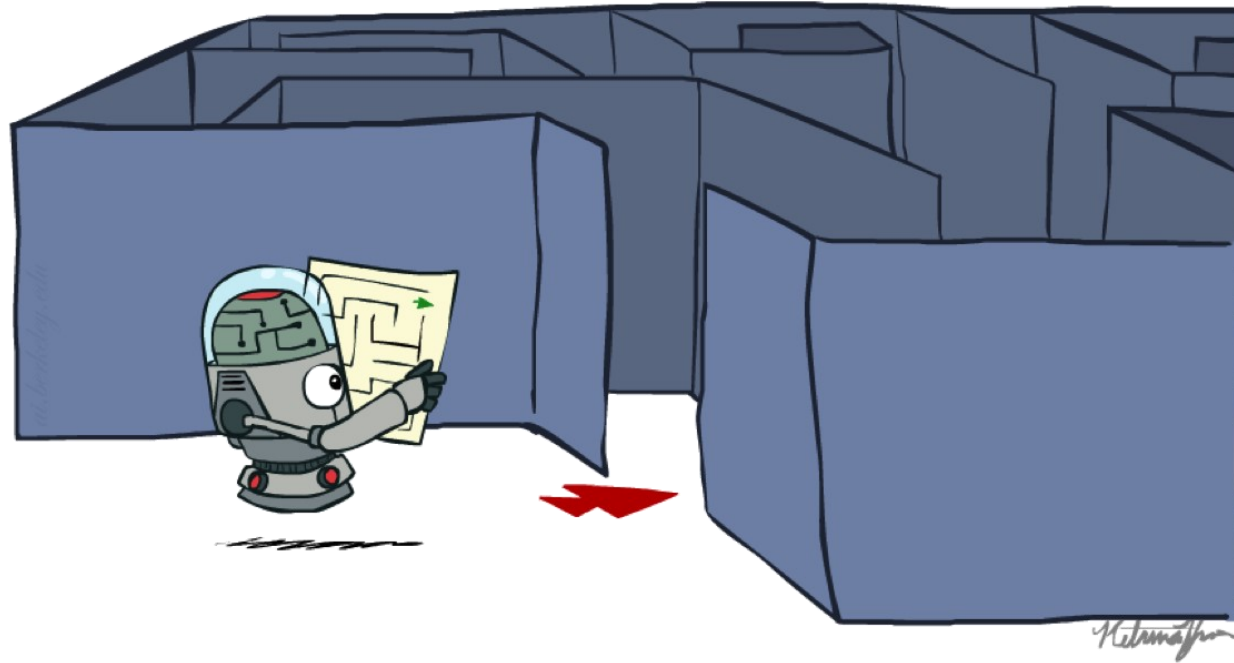
How many states are there in hopscotch?

Continuous or discrete states?



# AI: Representation and Problem Solving

## Agents and Search



Instructors: Vincent Conitzer and Aditi Raghunathan

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

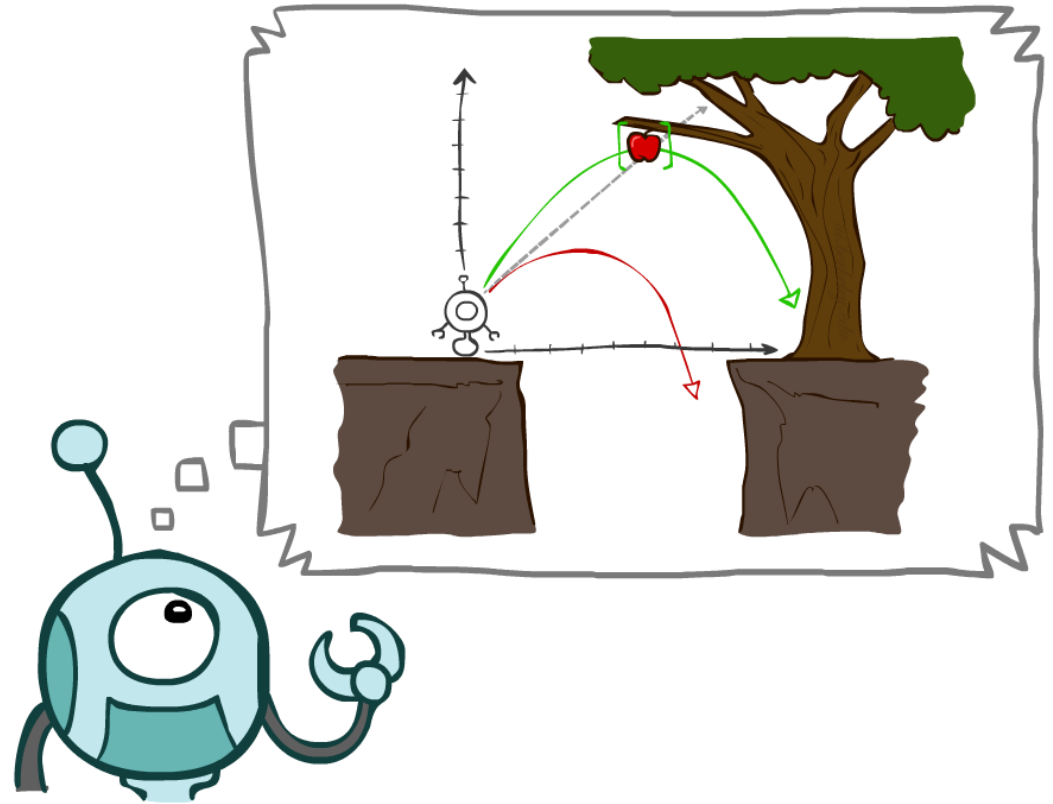
# Today

## Reflex vs Planning Agents

## Search Problems

## Uninformed Search Methods

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

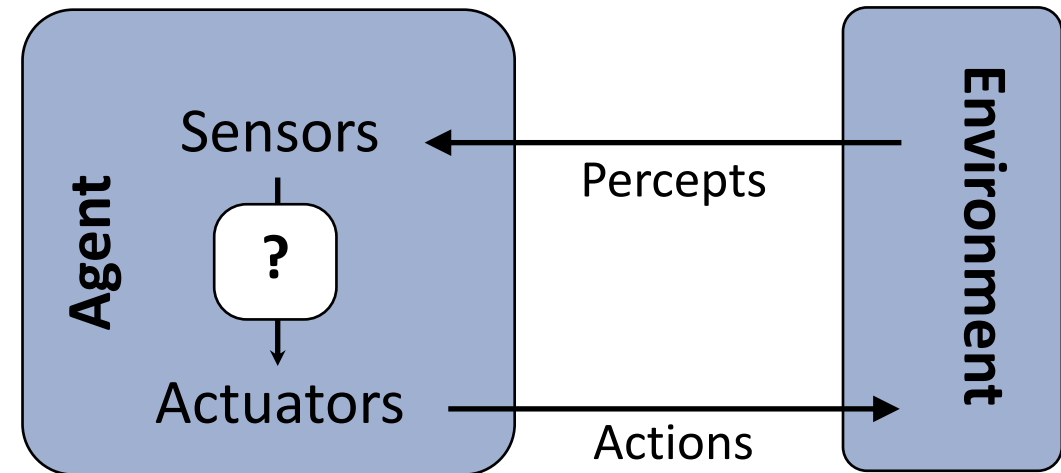
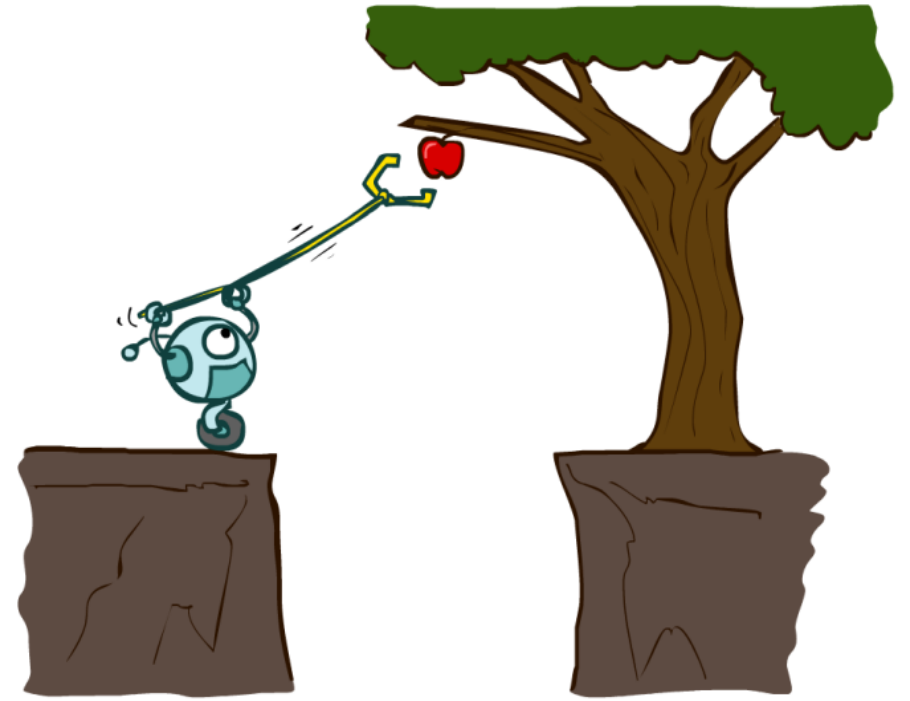




# Designing Agents

An **agent** is an entity that *perceives* and *acts*.

Characteristics of the **percepts and state, environment, and action space** dictate techniques for selecting actions

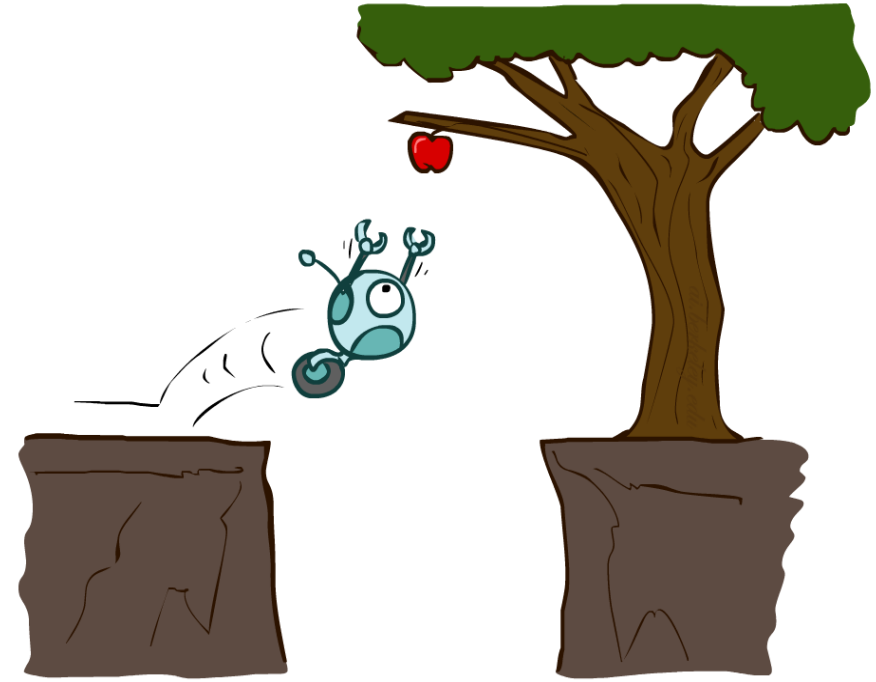


# Reflex Agents

## Reflex agents:

- Choose actions based on current/historic state
- Do not consider the future consequences of their actions
- May have memory or a model of the world's current state
- Consider how the world IS

Can a reflex agent be rational?



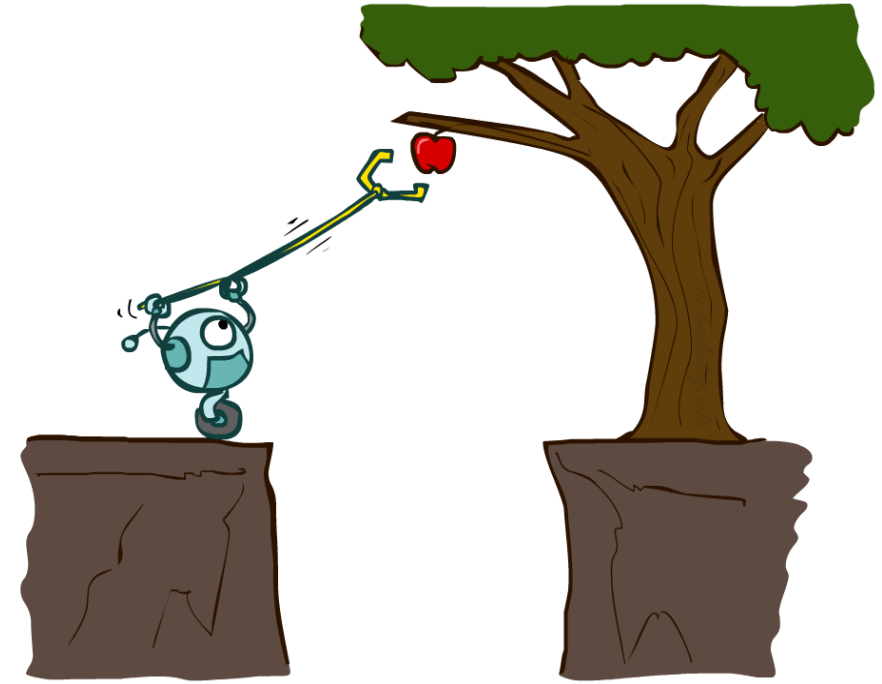
# Agents that Plan Ahead

## 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:

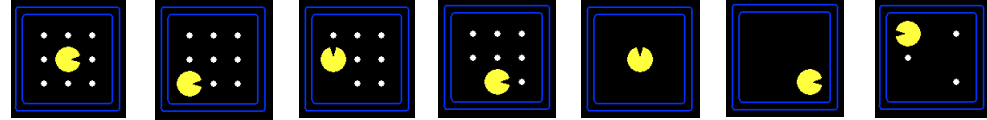
- Generate complete, optimal plan offline, then execute
- Generate a simple, greedy plan, start executing, replan when something goes wrong



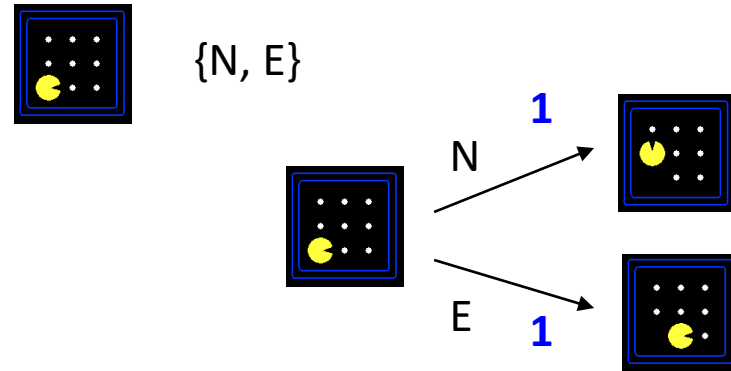
# Search Problems

A **search problem** consists of:

- A state space



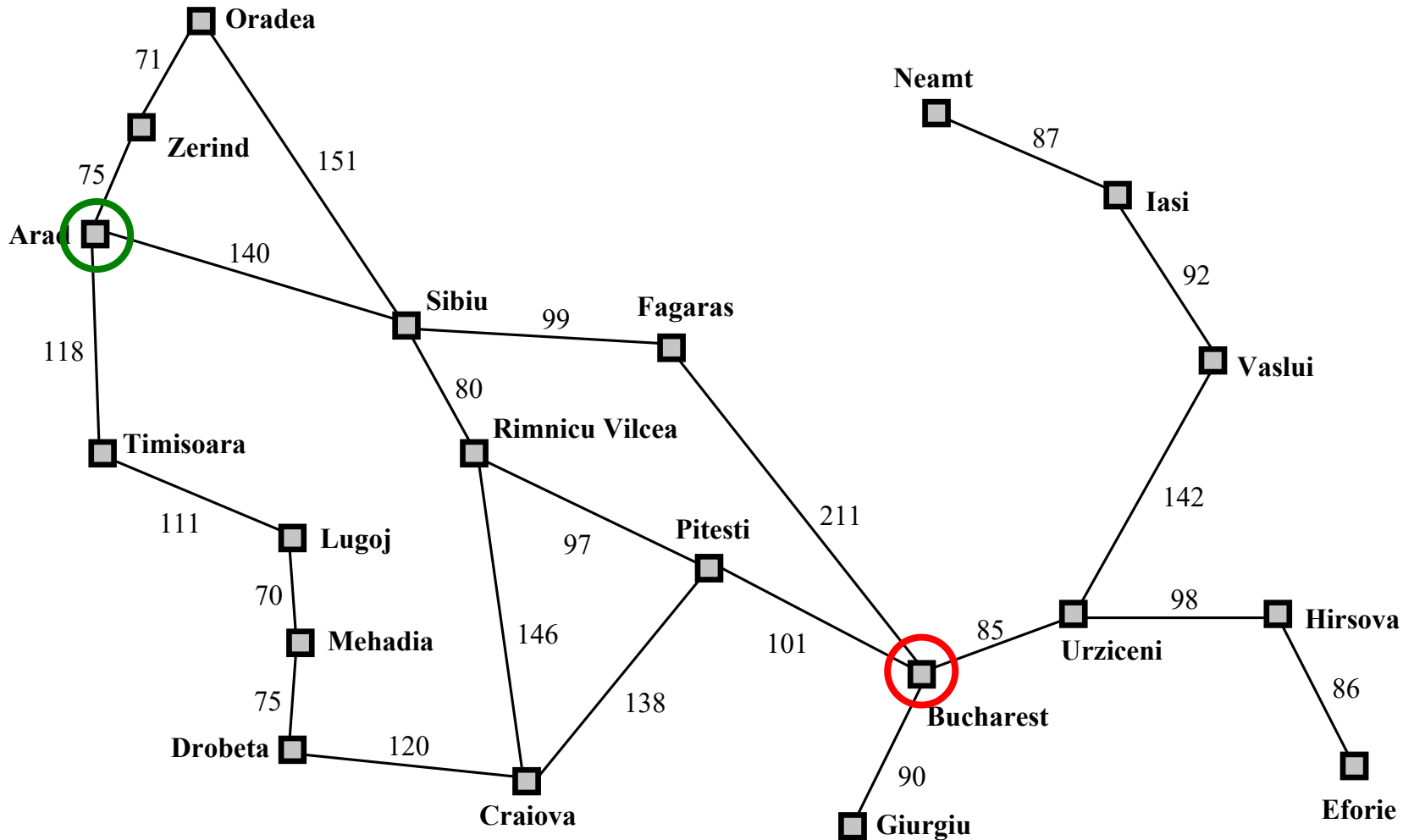
- For each state, a set  
Actions(s) of allowable actions



- 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

# 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:

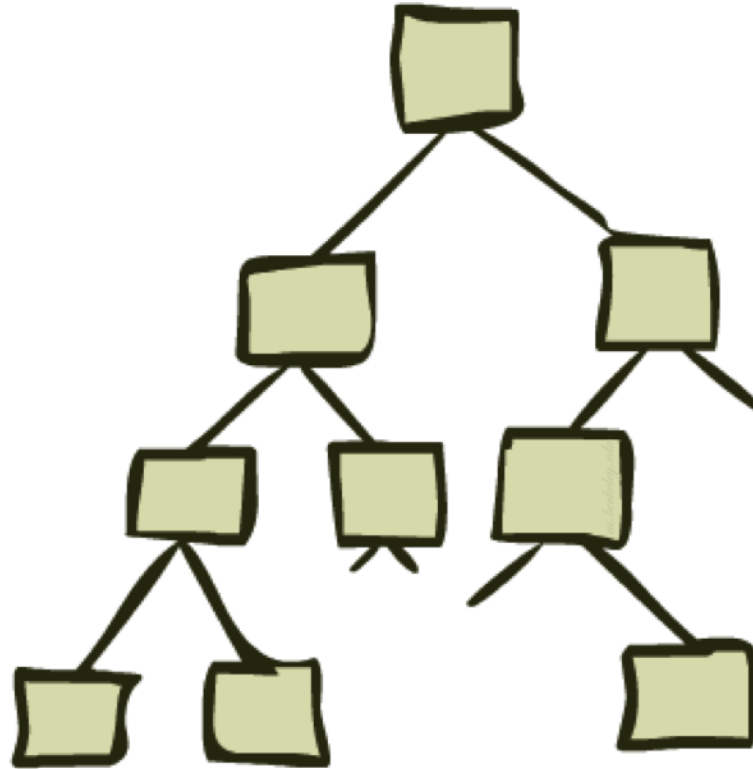
- Arad

Goal test:

- Is state == Bucharest?

Solution?

# State Space Graphs and Search Trees



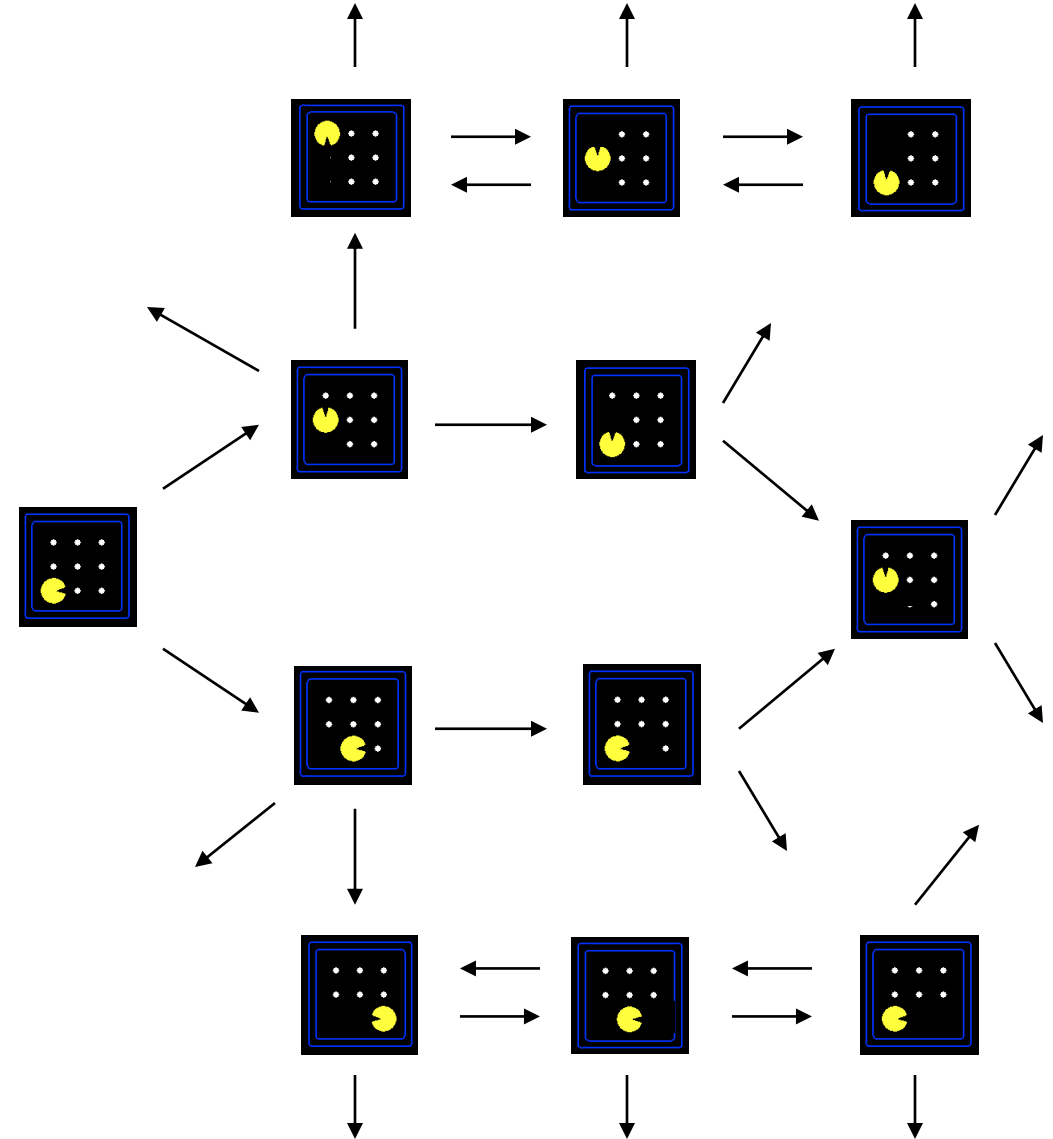
# State Space Graphs

State space graph: A mathematical representation of a search problem

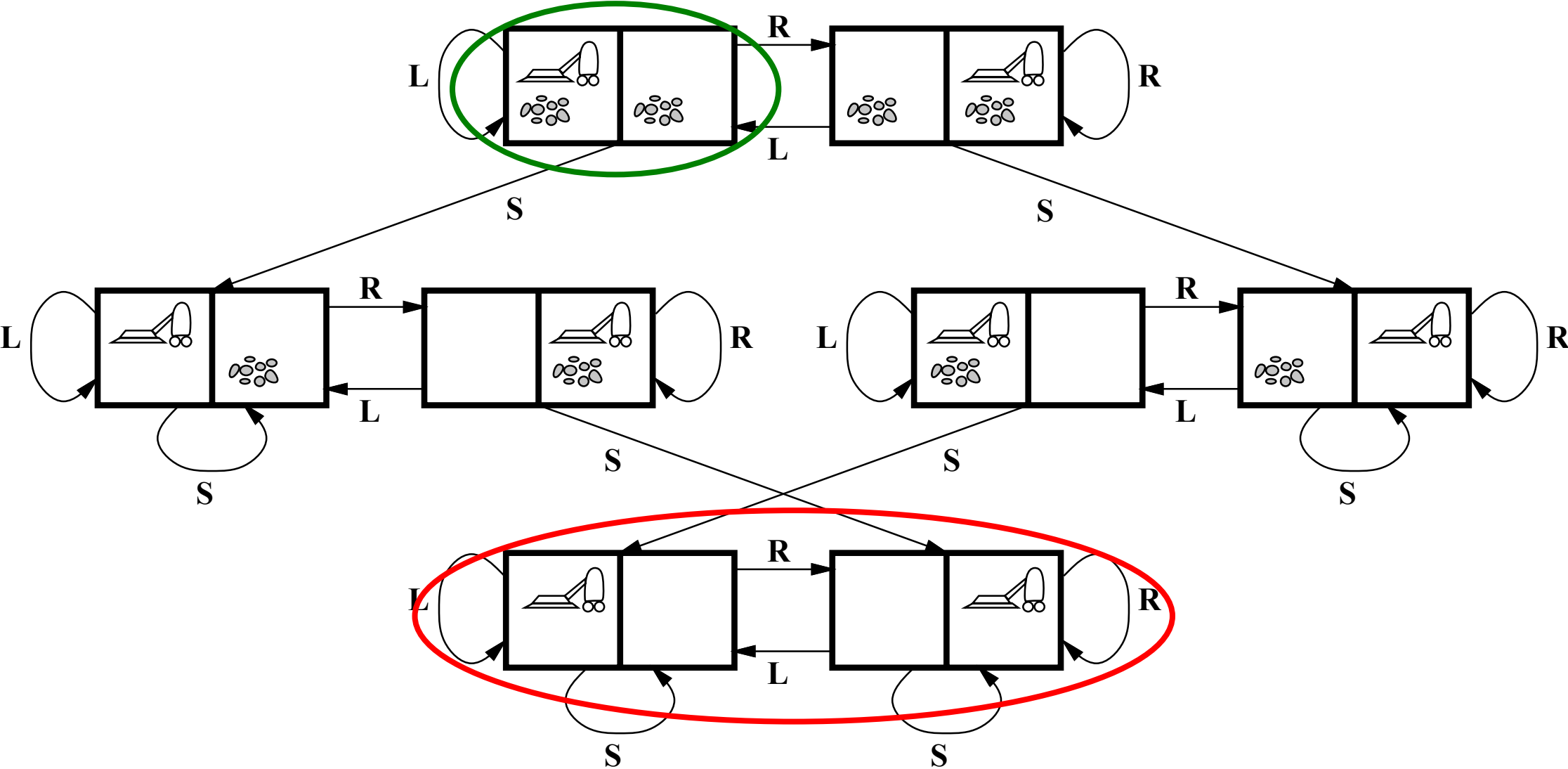
- Nodes are (abstracted) world configurations
- Arcs represent transitions resulting from actions
- The goal test is a set of goal nodes (maybe only one)

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



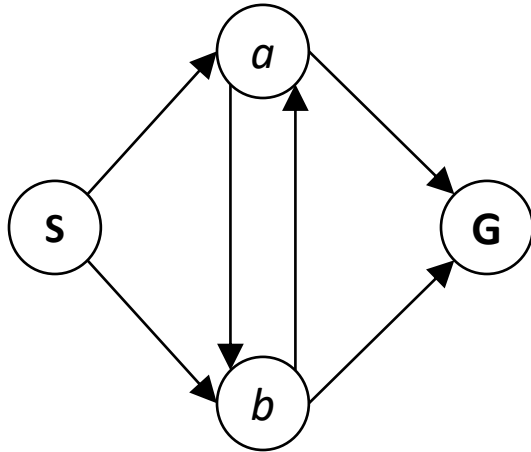
# More Examples



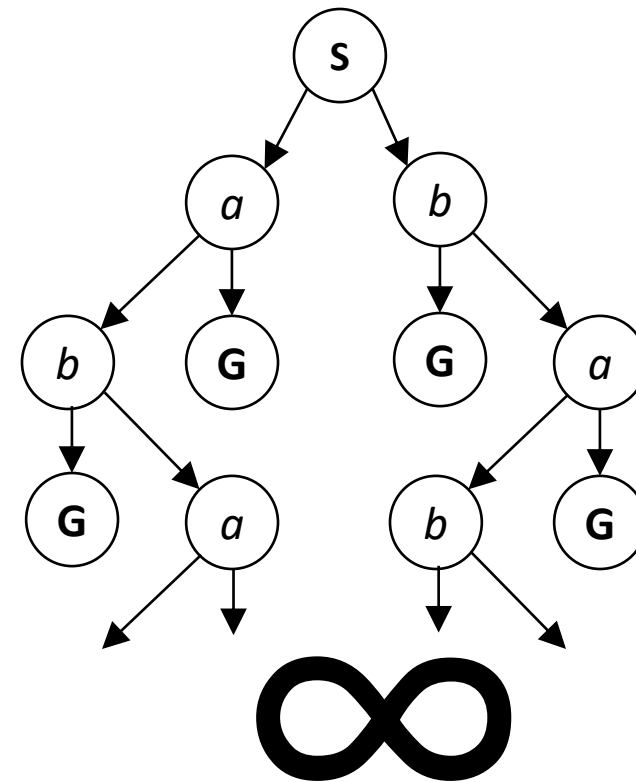


# State Space Graphs vs. Search Trees

Consider this 4-state graph:

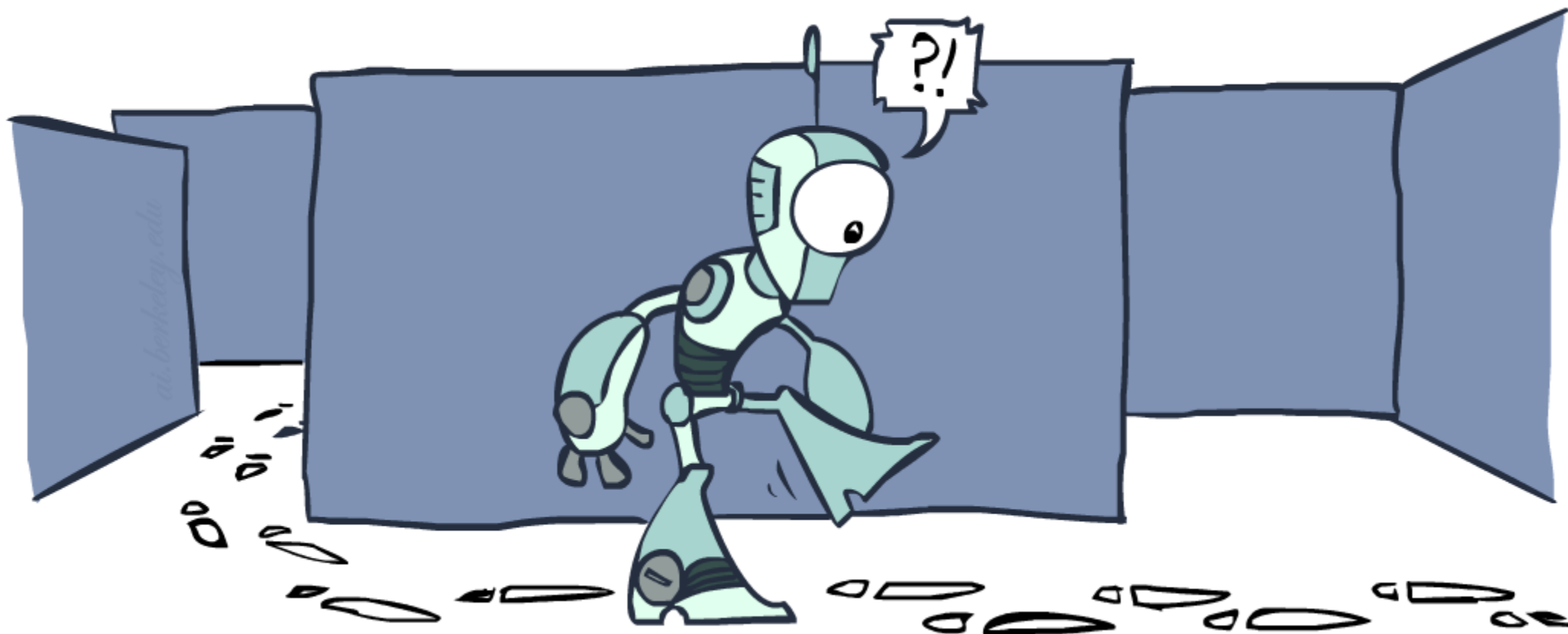


How big is its search tree (from S)?



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

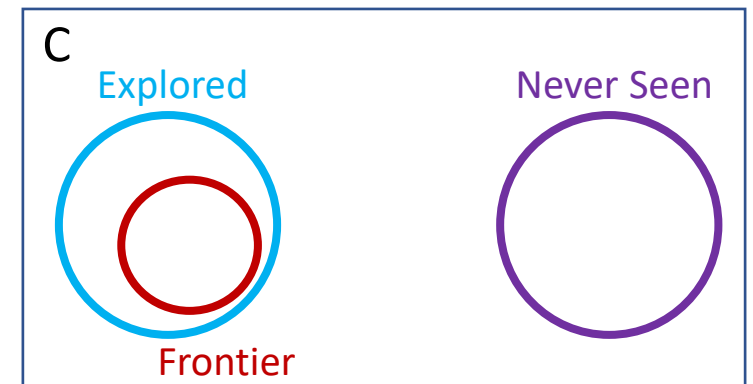
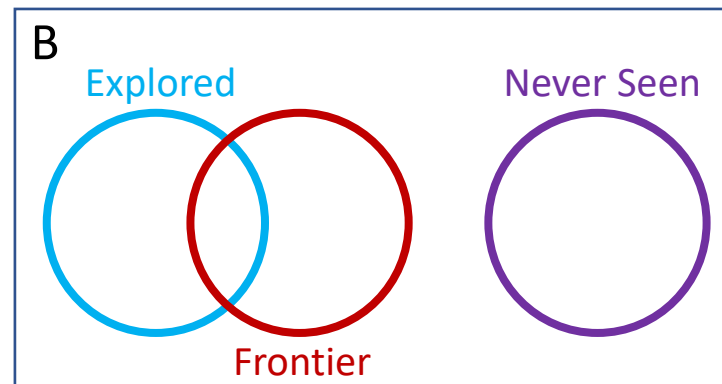
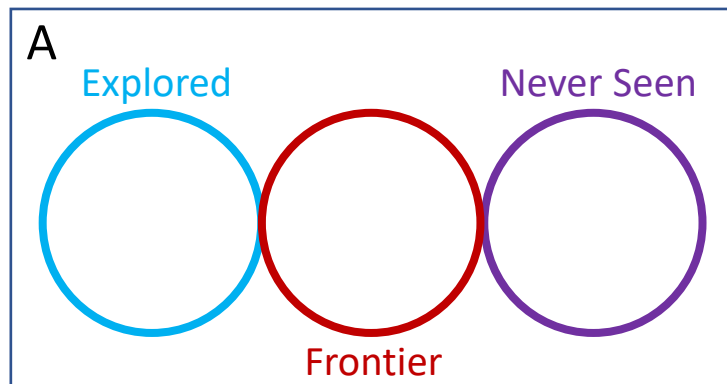
**if** the **child** state is not already in the **frontier** or **explored set** **then**

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

# Poll 1

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

(Loop invariants!!!)



# Poll 1

**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**

# A Note on Implementation

Nodes have

*state, parent, action, path-cost*

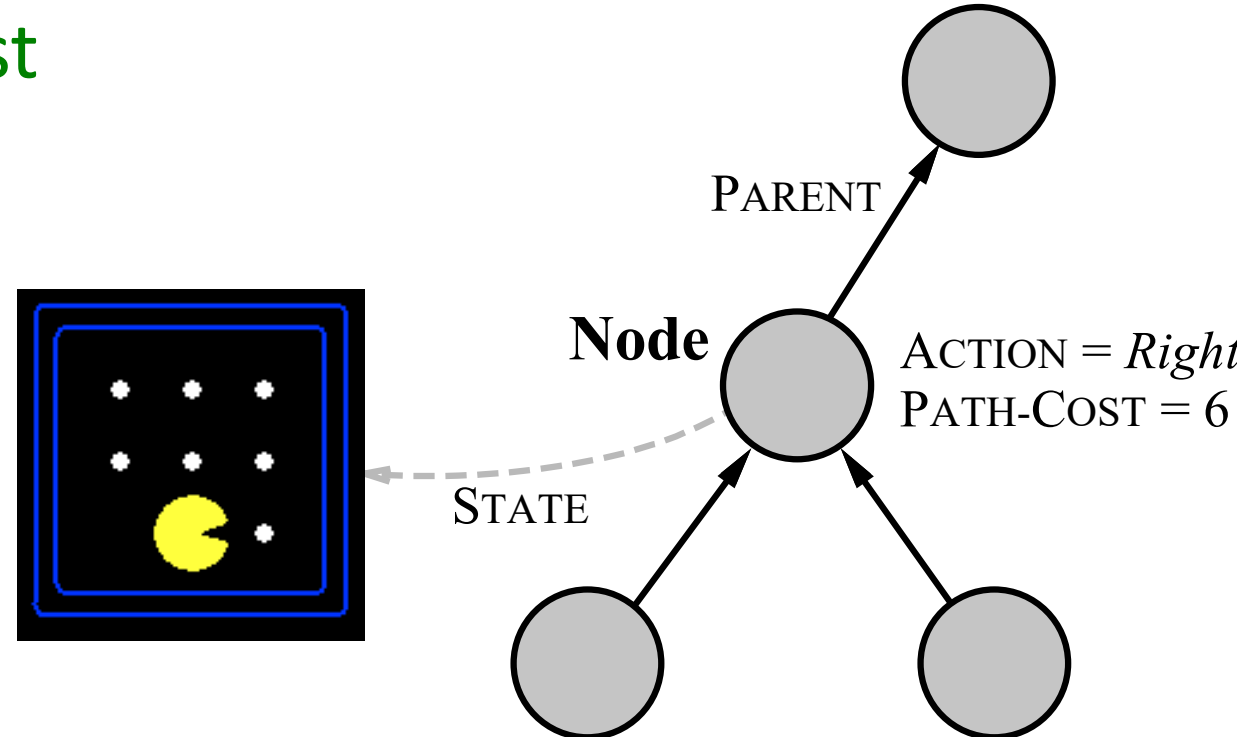
A child of *node* by action *a* has

*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





# BFS vs DFS



# Walk-through BFS Graph Search

Frontier

~~S~~

~~S~~ d e

~~S~~ p

~~S~~

~~S~~ S d b

S d c

S e h

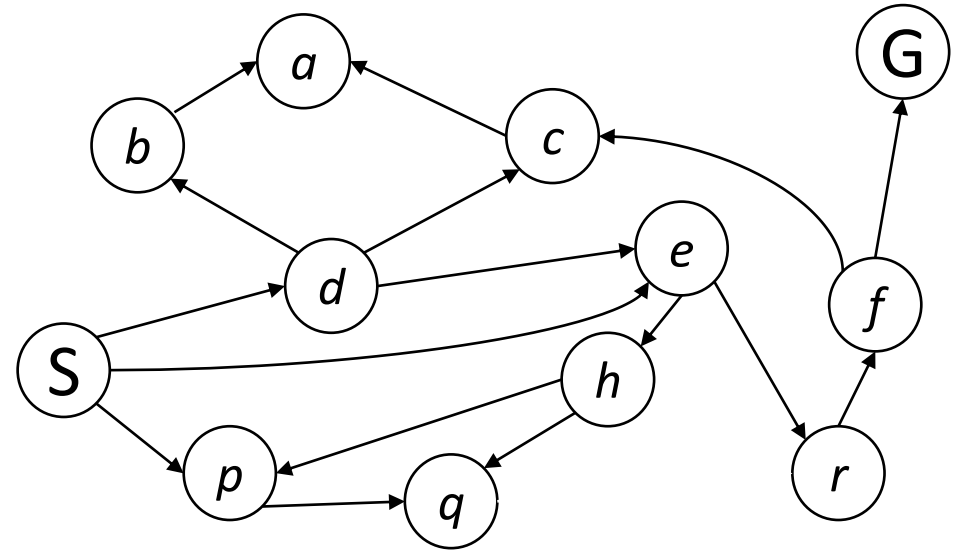
S e r

Explored

S

d

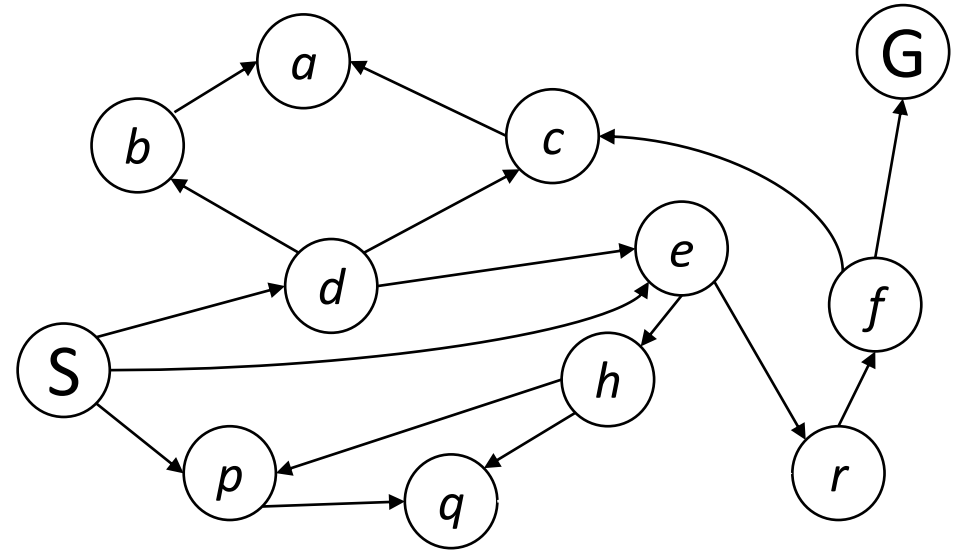
e



# Walk-through DFS Graph Search

Fr  
f  
~~fat~~  
se  
sp  
sab  
sdc  
sdba

Ex  
s  
d  
b

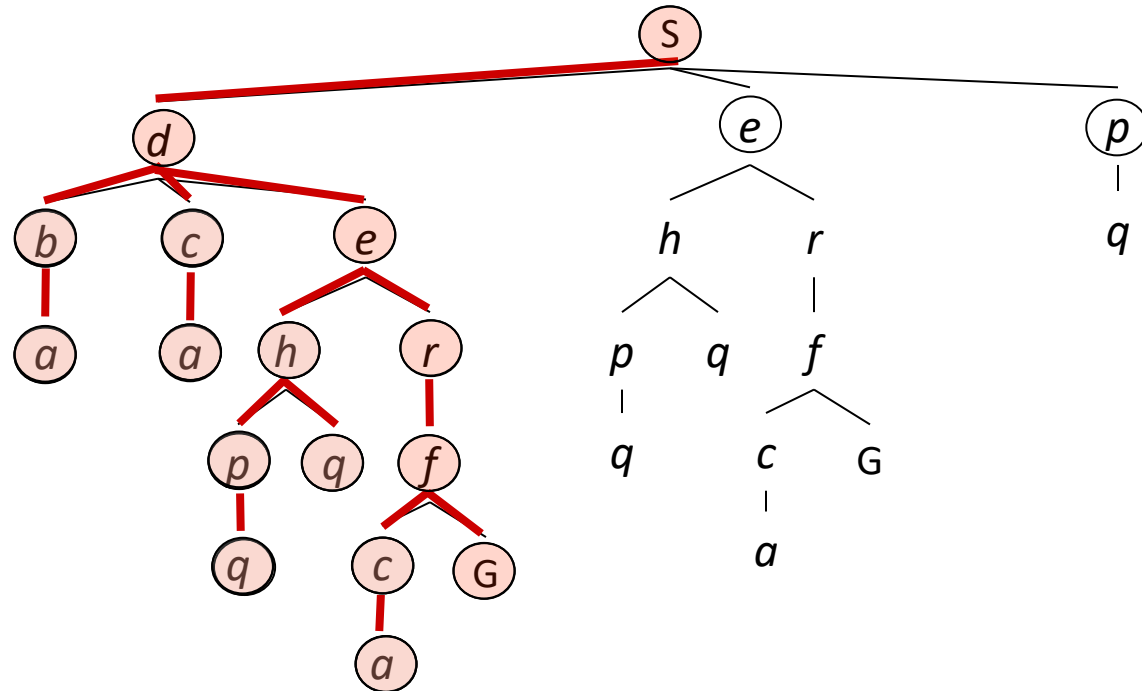
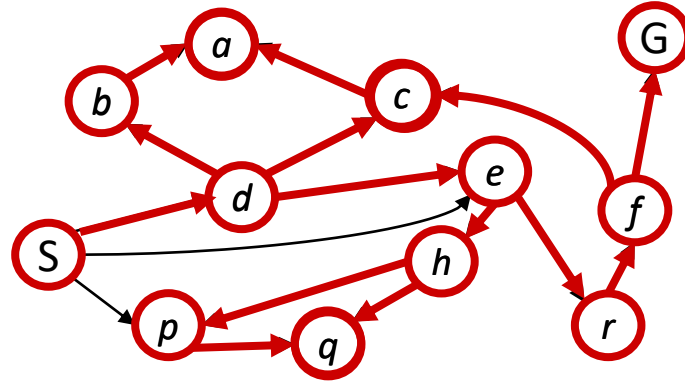


# Depth-First (Tree) Search

Strategy: expand a  
deepest node first

Implementation:

**Frontier is a LIFO stack**

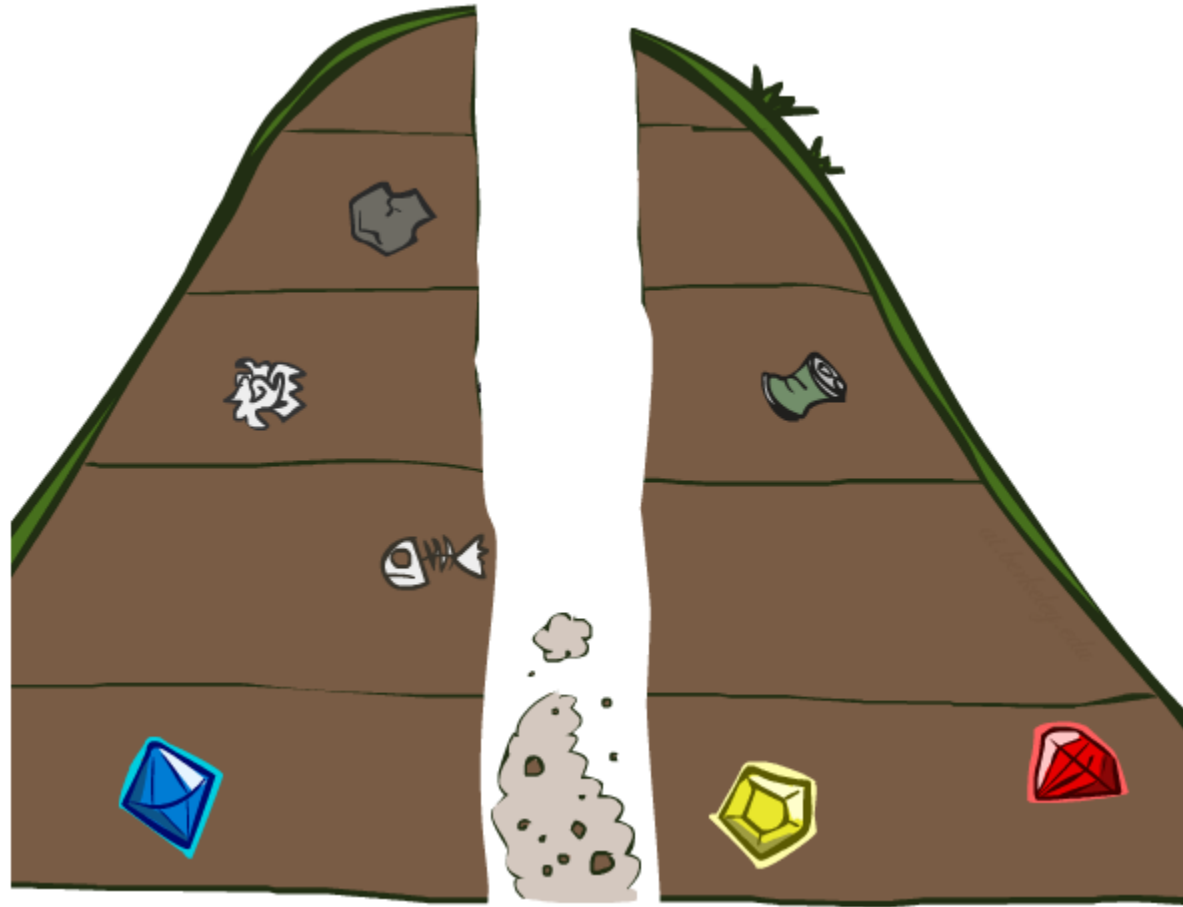


# BFS vs DFS

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?

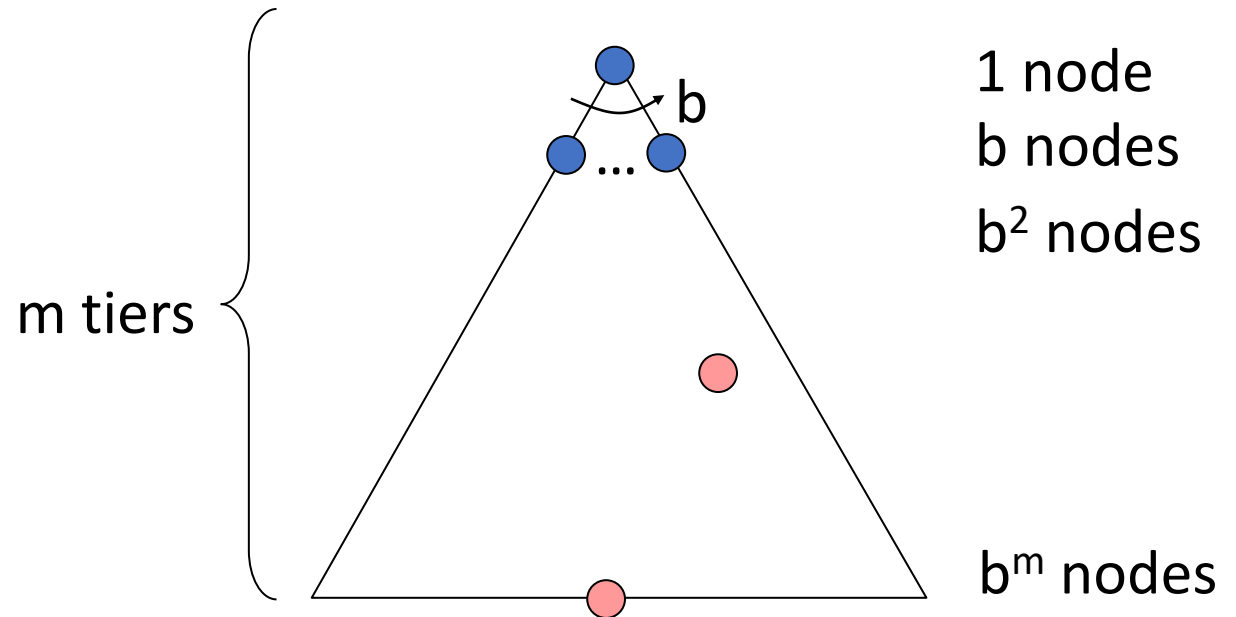
Space complexity?

Cartoon of search tree:

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

Number of nodes in entire tree?

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



# Search Algorithm Properties

Complete: Guaranteed to find a solution if one exists?

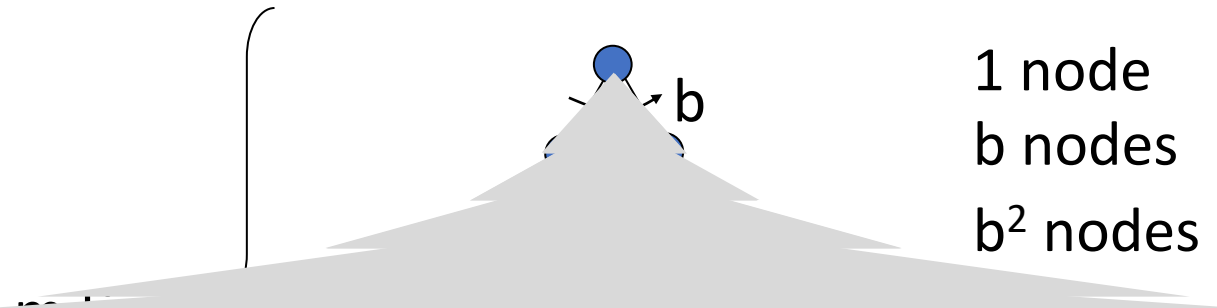
Optimal: Guaranteed to find the least cost path?

Time complexity?

Space complexity?

Cartoon of search tree:

- $b$  is the branching factor

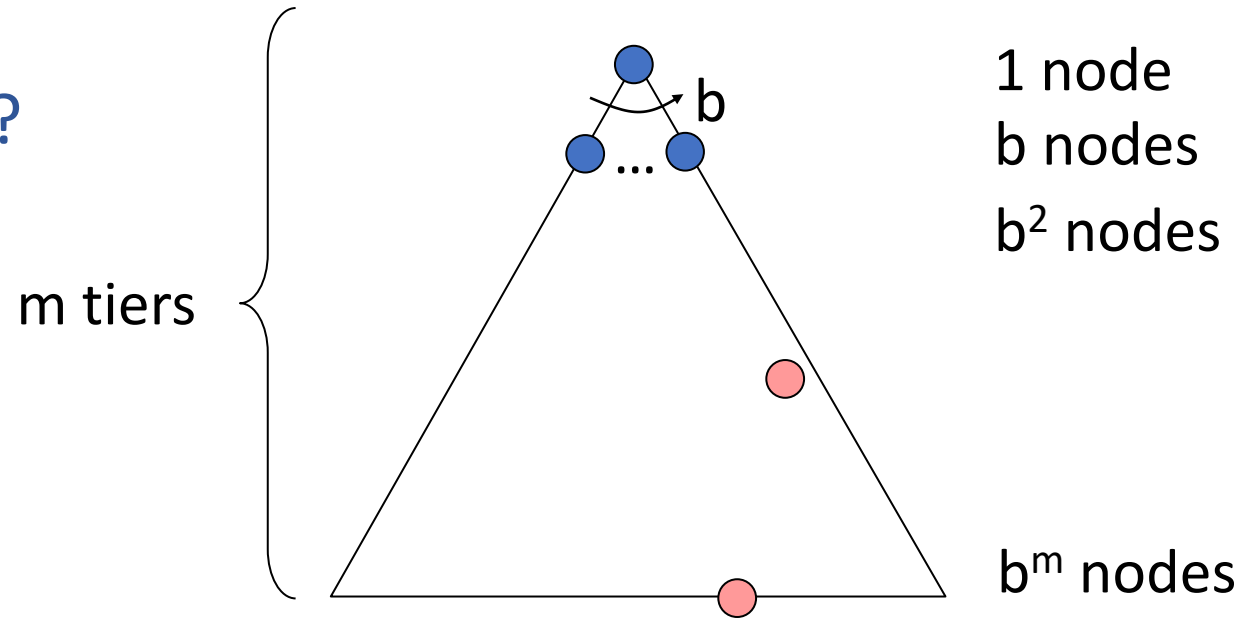




# Think about it...

Are these the properties for BFS or DFS?

- Takes  $O(b^m)$  time
- Uses  $O(bm)$  space on frontier
- Complete with graph search & finite number of states
- 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?

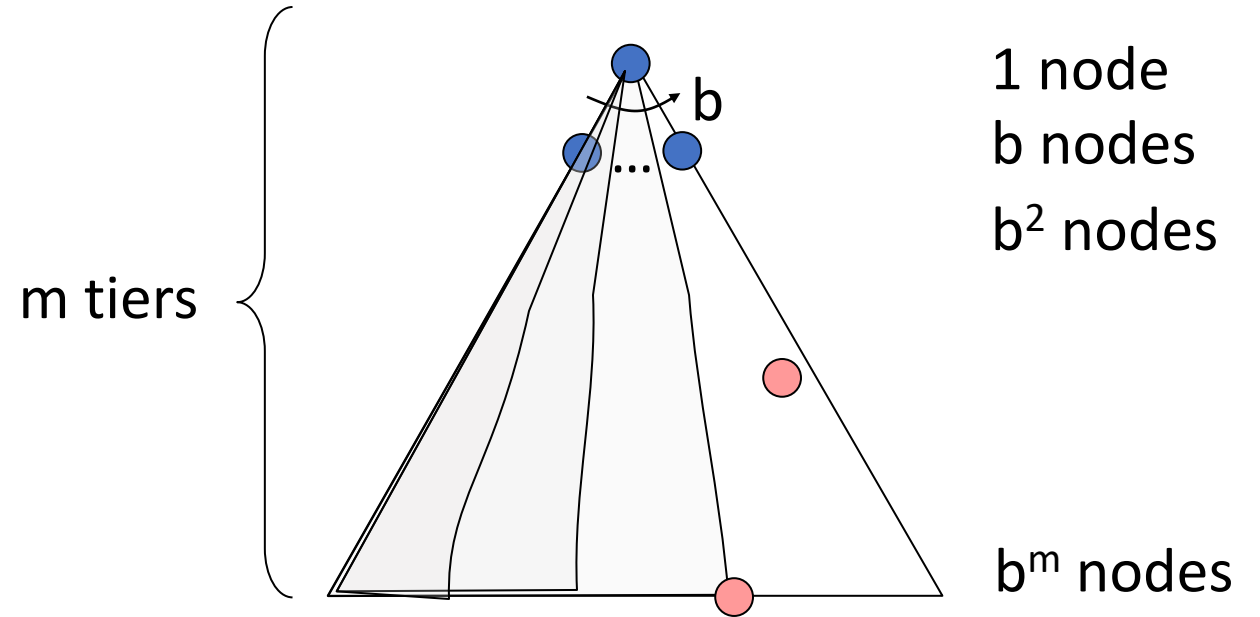
- Only has siblings on path to root, so  $O(bm)$

## Is it complete? (always find a solution)

- $m$  could be infinite, so only if there are finitely many possible states and we prevent cycles (graph search)

## Is it optimal? (solution is “best”)

- 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?

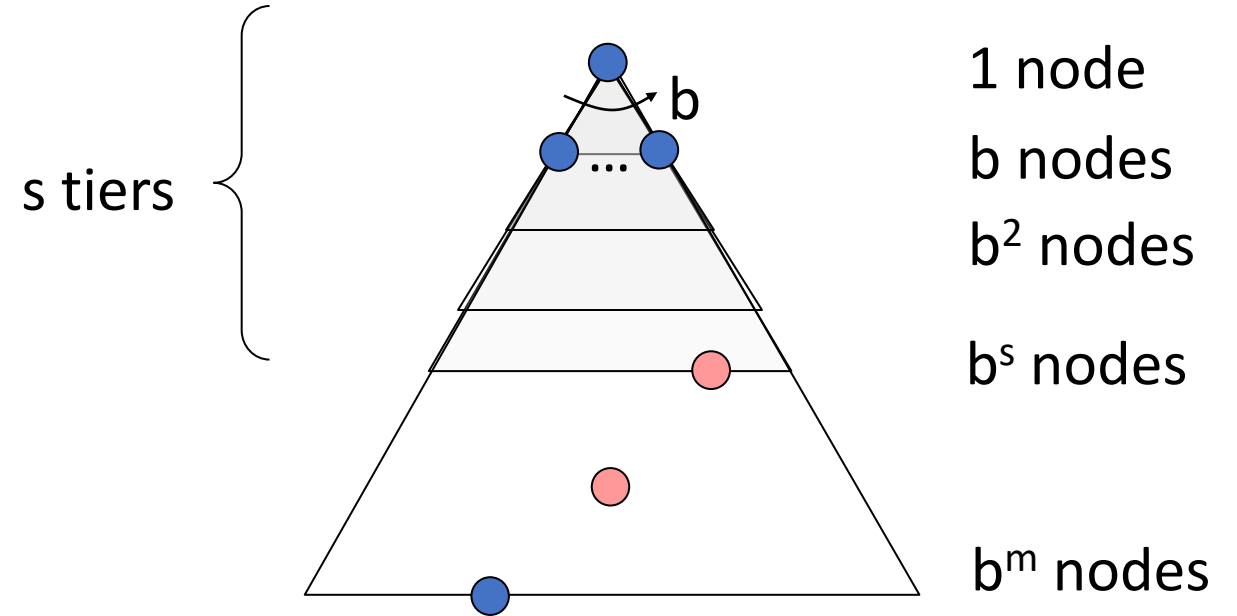
- Has roughly the last tier, so  $O(b^s)$

## 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)



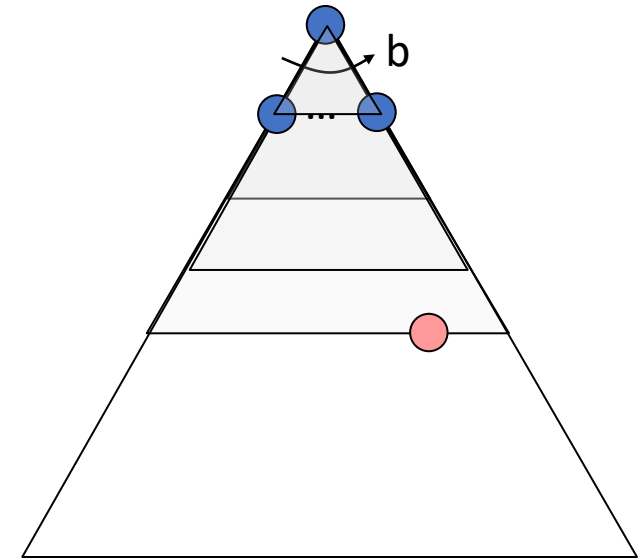
# Iterative Deepening

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!

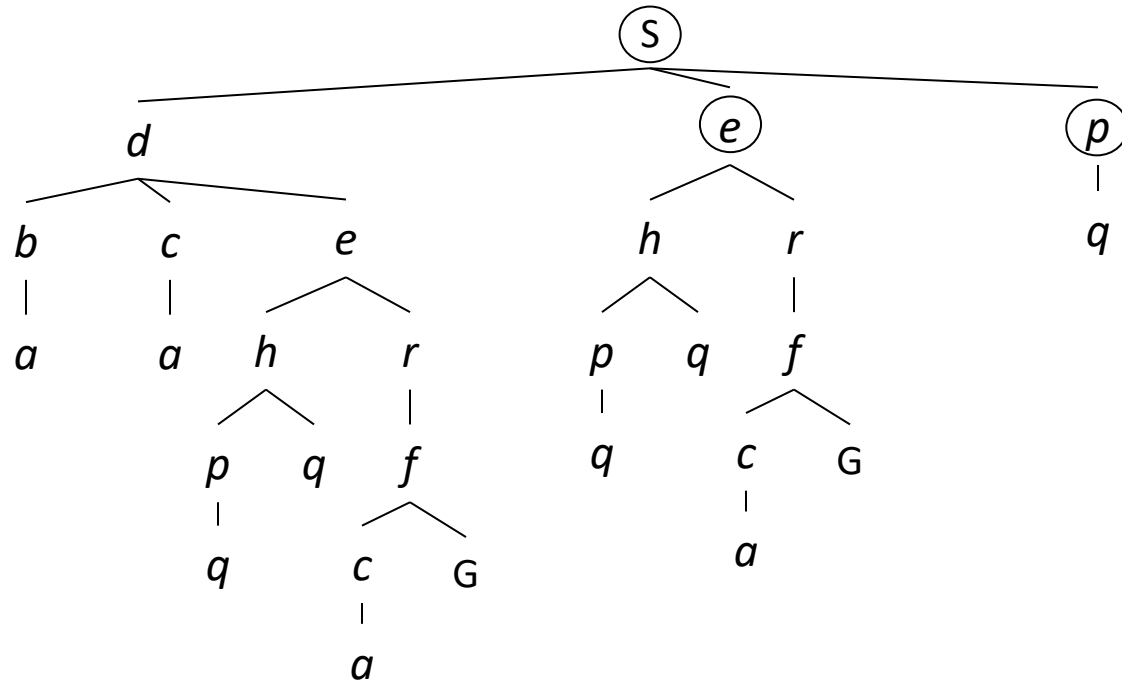
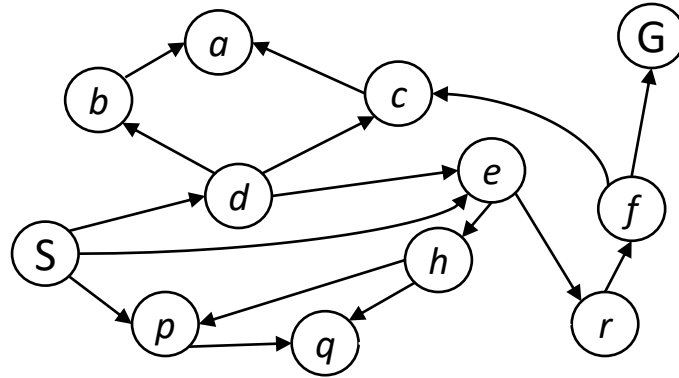


# Iterative Deepening

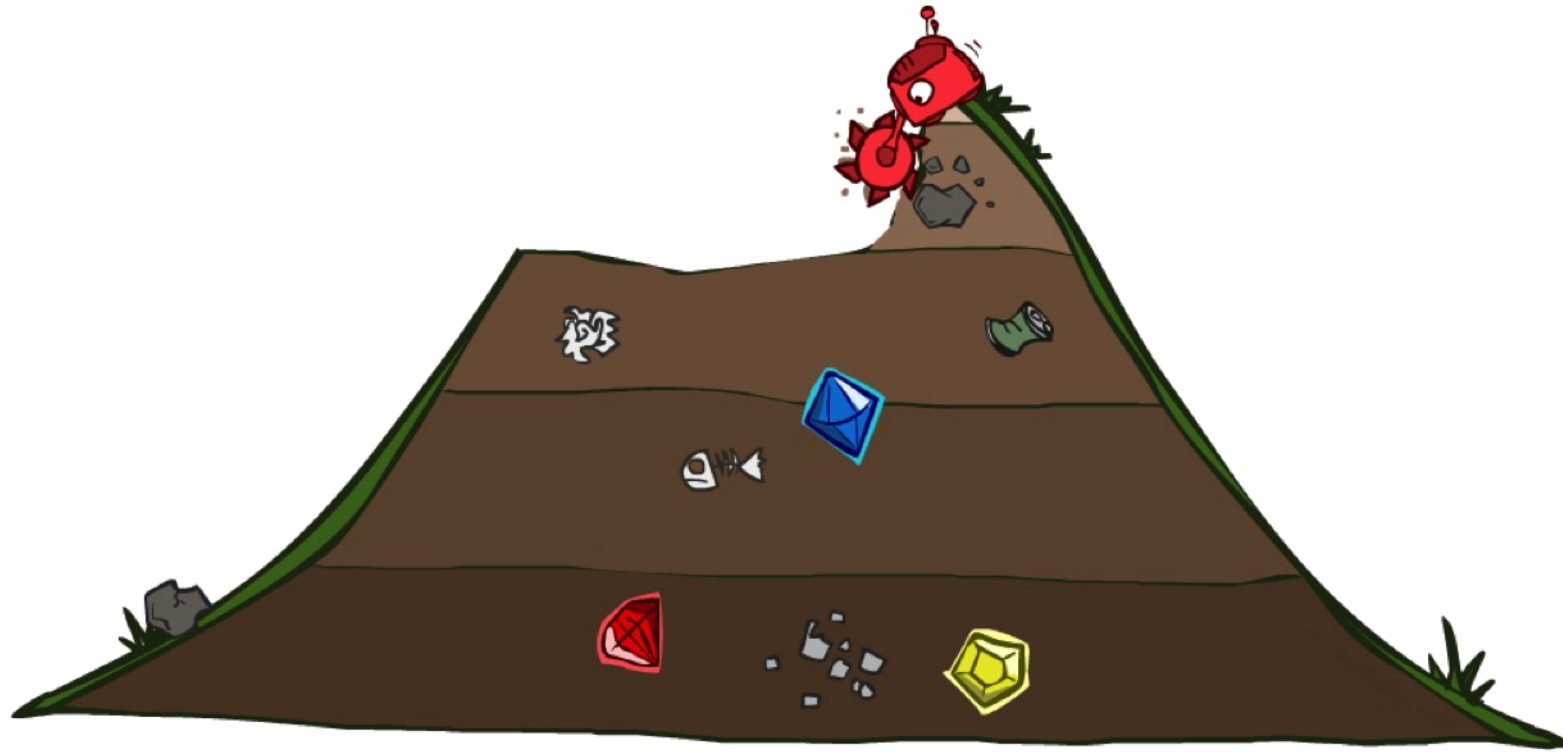
Strategy: expand a deepest node first to a max depth, iteratively increase the depth

Implementation:

**Frontier is a LIFO stack**



# 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**

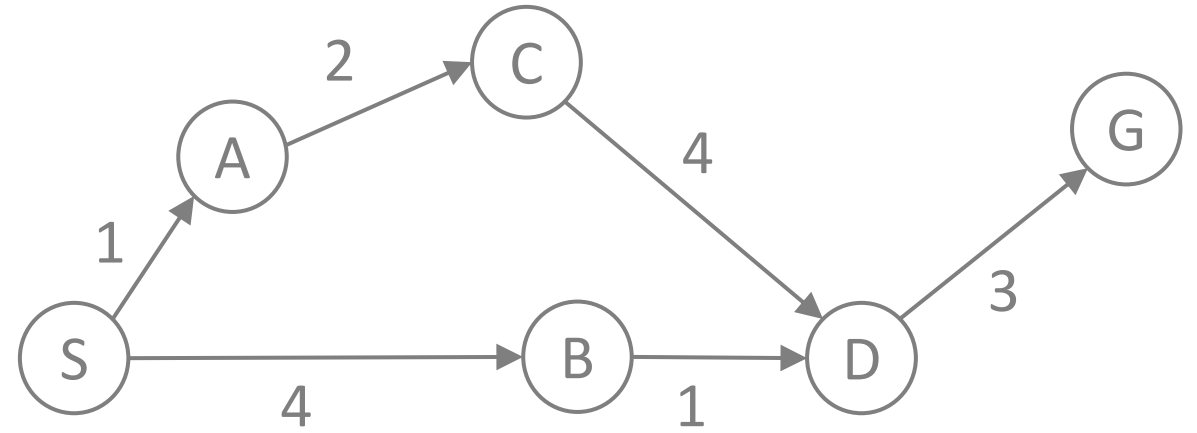
replace that **frontier** node with **child**



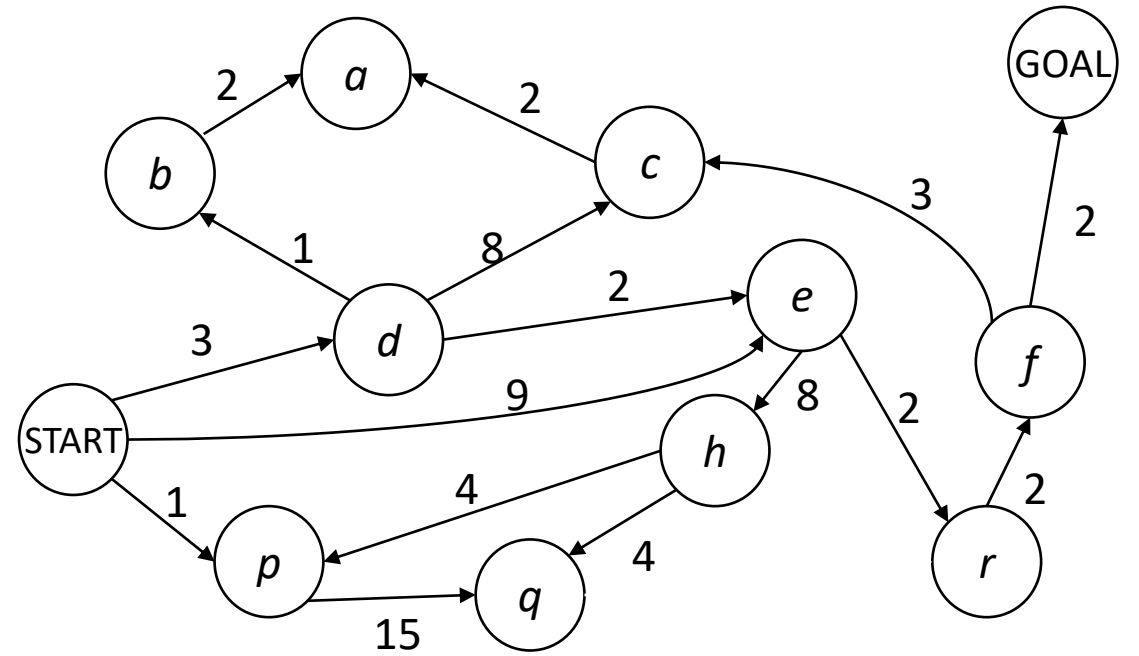
# Walk-through UCS

~~Fr~~  
~~S~~  
~~Sa~~  
~~S<sub>5</sub>~~  
~~Sac~~  
~~Sacd~~  
SBD 5

~~fx~~  
~~S~~  
~~a~~  
~~c~~  
~~B~~



# Walk-through UCS



# In Class Activity!

Q1 – practice running graph search. What gets added to the explored list in what order?

Q2 - Amazon warehouses use robots to transport items to packers along the outside edge of the warehouse to reduce the amount of walking those packers must do. These robots need to plan their paths to their goals without hitting each other.

Think about how we would apply graph search to this multi-robot problem...

# Summary

- Reflex vs Planning Agents
- Modeling state based on the problem you're trying to solve
- Tree vs Graph Search
- BFS, DFS, UCS
- Branching factor, Search space (size of frontier)
- Completeness of search is whether it will always find A solution
- Optimality of search is whether it always finds the BEST solution

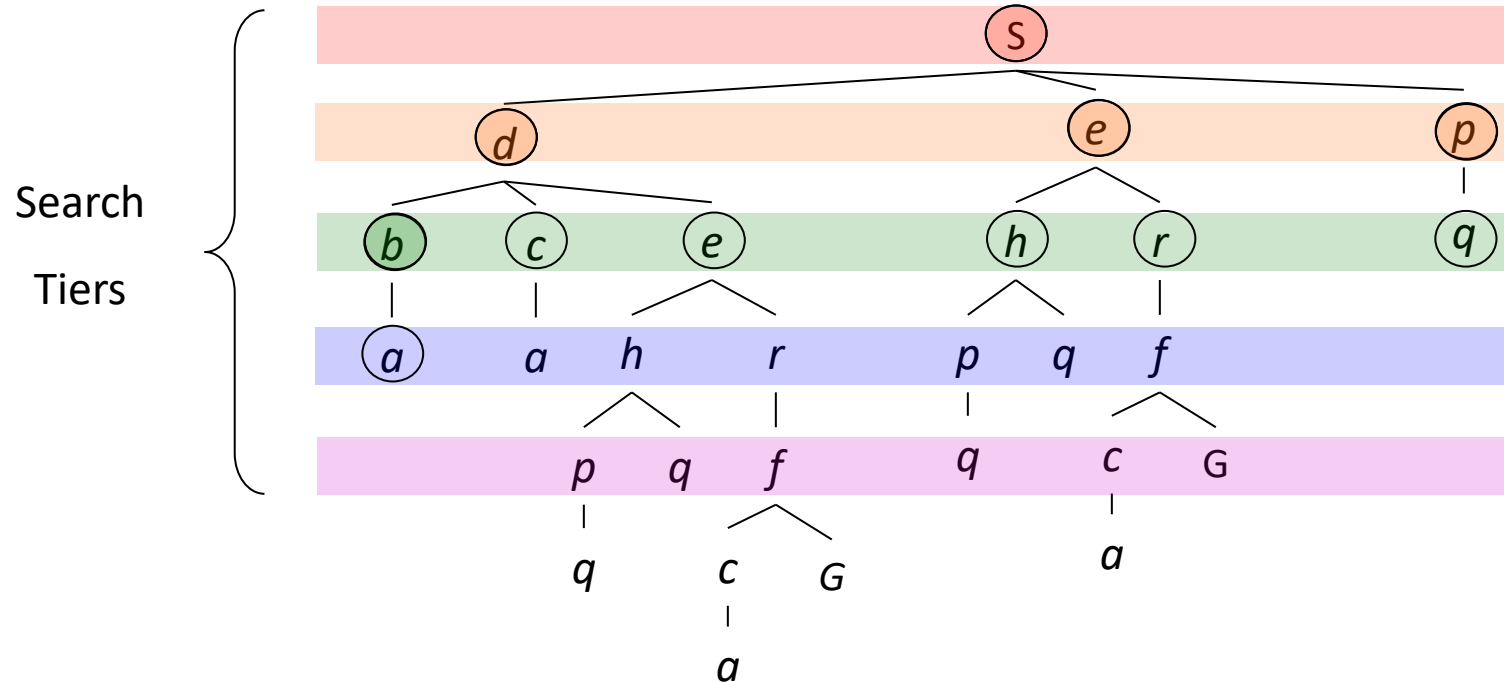
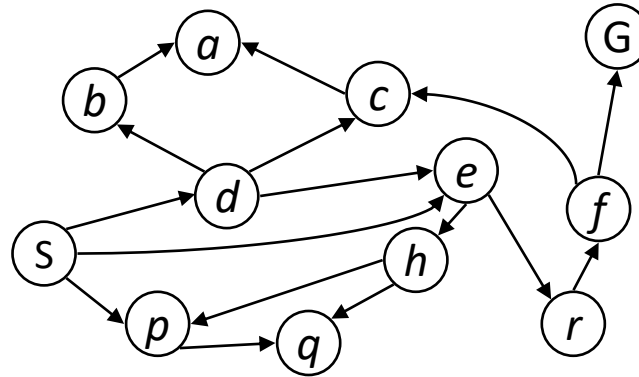
Extra slides below on search properties and iterative deepening

# Breadth-First (Tree) Search

Strategy: expand a shallowest node first

Implementation:

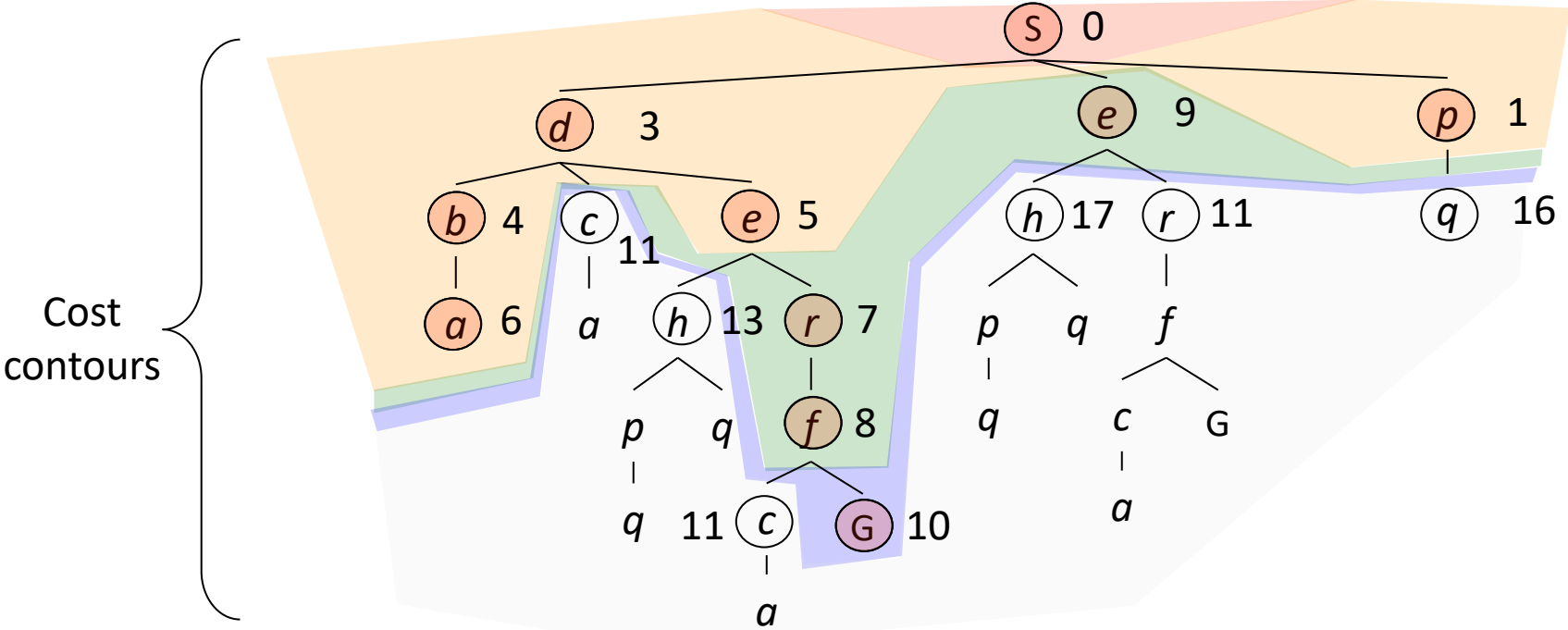
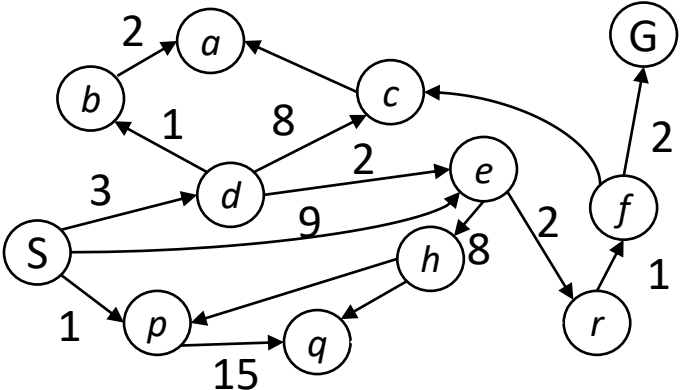
**Frontier is a FIFO queue**



# Uniform Cost (Tree) Search

Strategy: expand a cheapest node first:

Frontier is a priority queue (priority: cumulative cost)



# Uniform Cost Search (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?

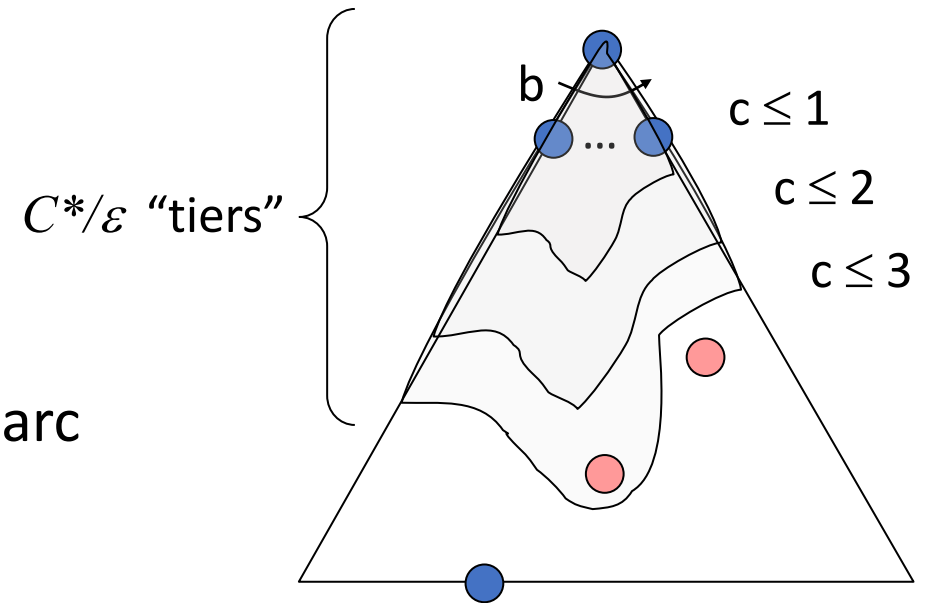
- Has roughly the last tier, so  $O(b^{C^*/\varepsilon})$

## 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!

