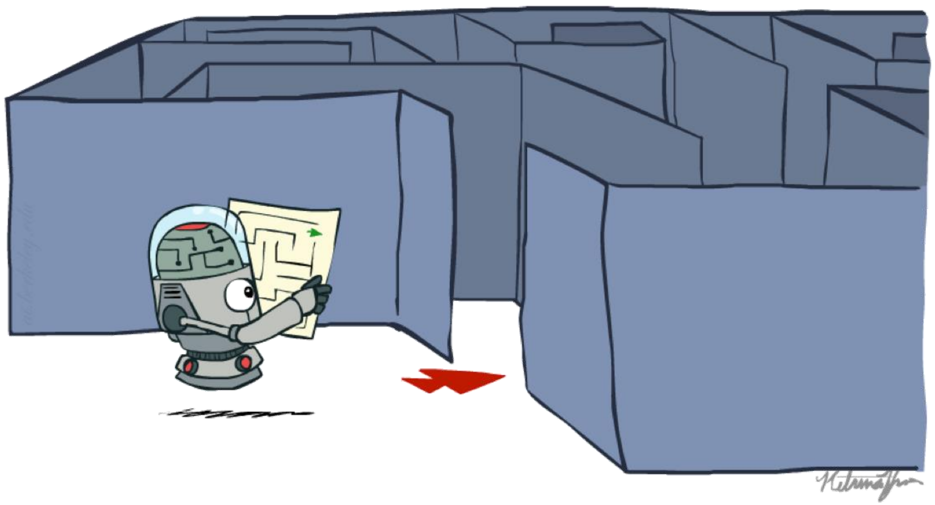
Part 1:
Solving Problems with Search

[Aknowledgment: Some Slides adapted from Dan Klein and Pieter Abbeel]

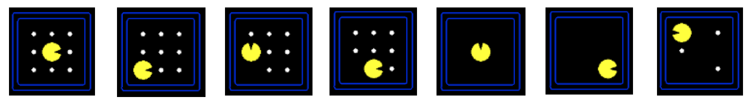
http://ai.berkeley.edu.]

Search, continued

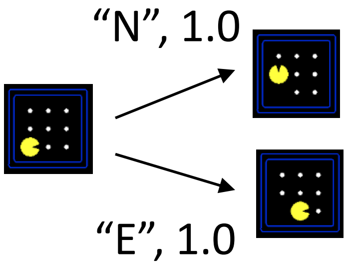


Search Problems

- A search problem consists of:
  - State space



  - Successor function (with actions, costs)



  - Start state and a goal test

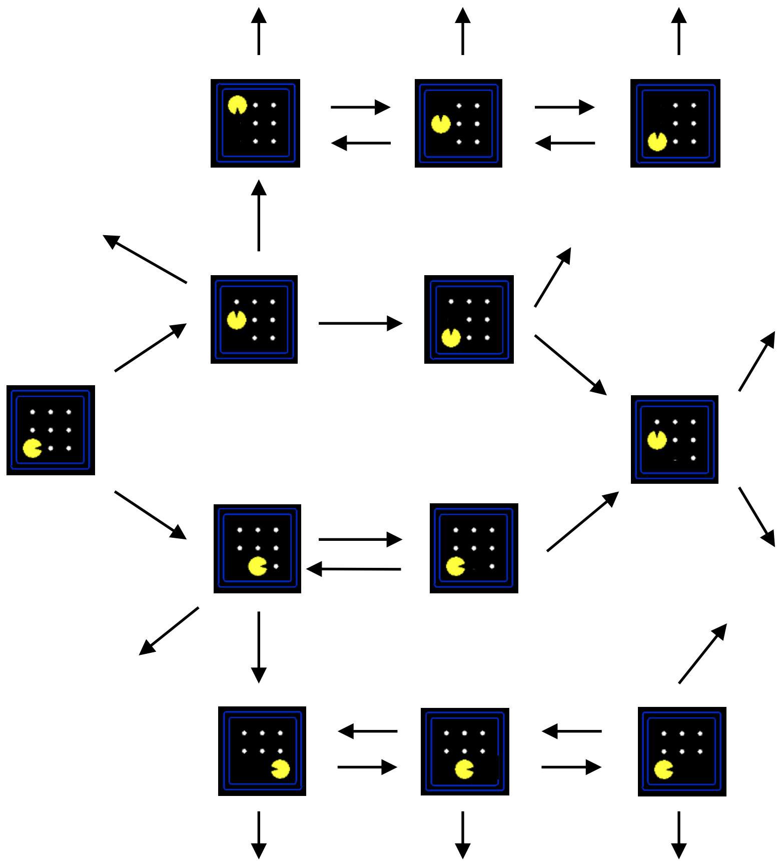
- A **solution** is a sequence of **actions** which transforms the start state to a goal state

State Space Graphs: 1

- **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!
  - Careful: if there are loops in this graph, keep track of visited states

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

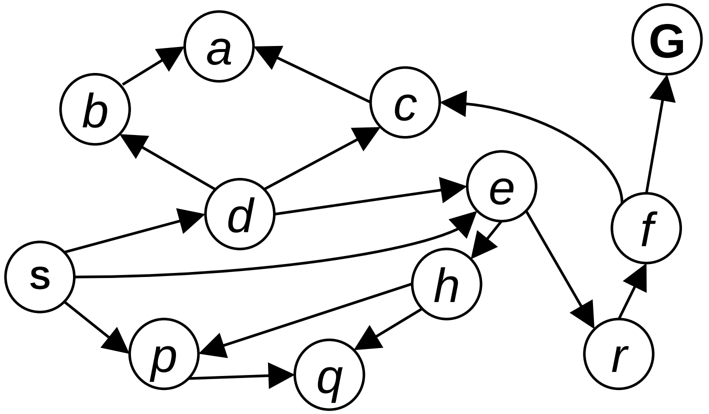


State Space Graphs: 2

- 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)

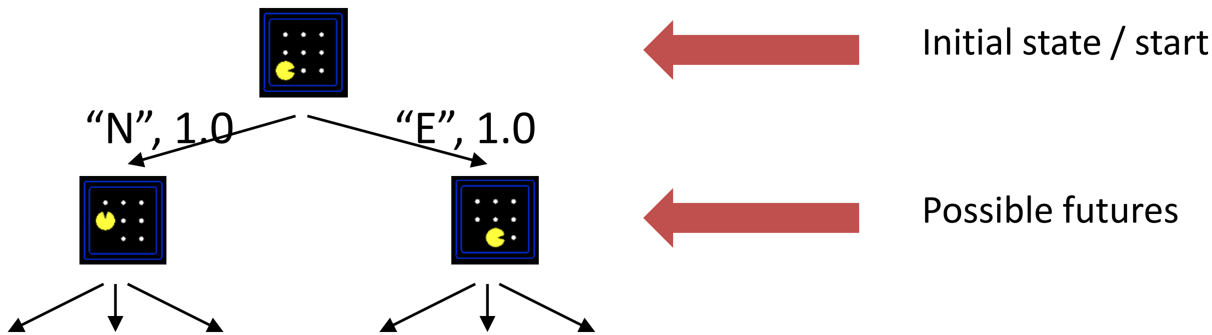
- In a search graph, each state occurs only once!

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



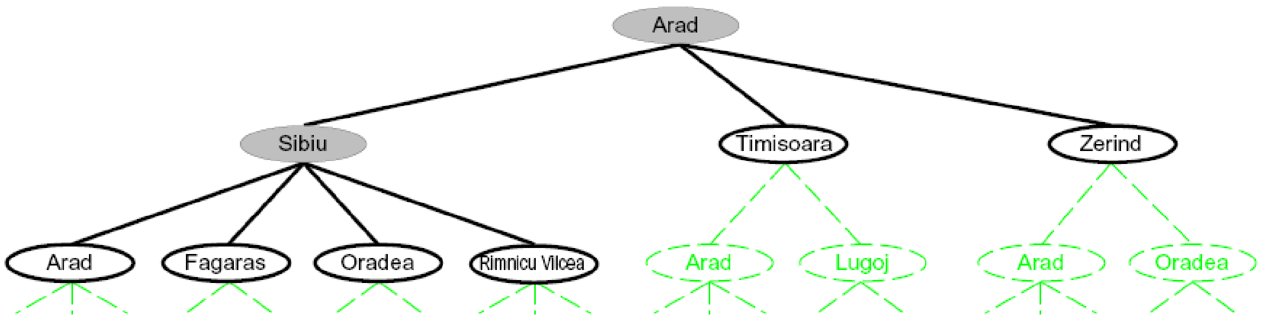
Tiny search graph for a tiny search problem

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 (paths from root to the state)**
  - For most problems, we can never actually build the whole tree (too large)

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, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

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

Python Implementation

```
def tree_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

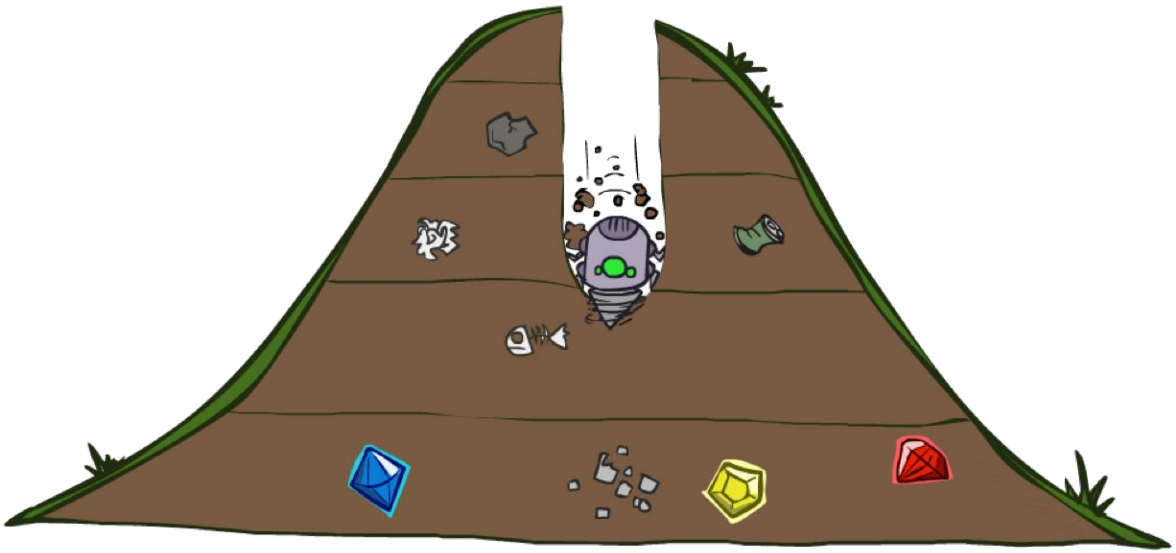
Search Strategies: Notation

- A search strategy is defined by picking the order of node expansion (fringe exploration)

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - ***b***: maximum branching factor of the search tree
  - ***d***: depth of the least-cost solution
  - ***m***: maximum depth of the state space (may be ∞)

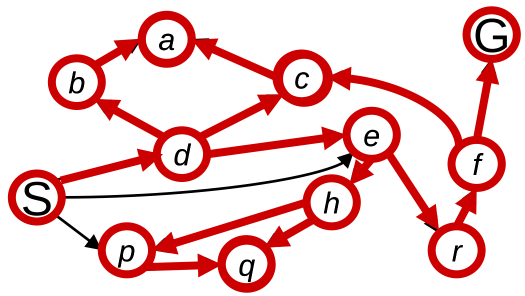
Depth-First Search

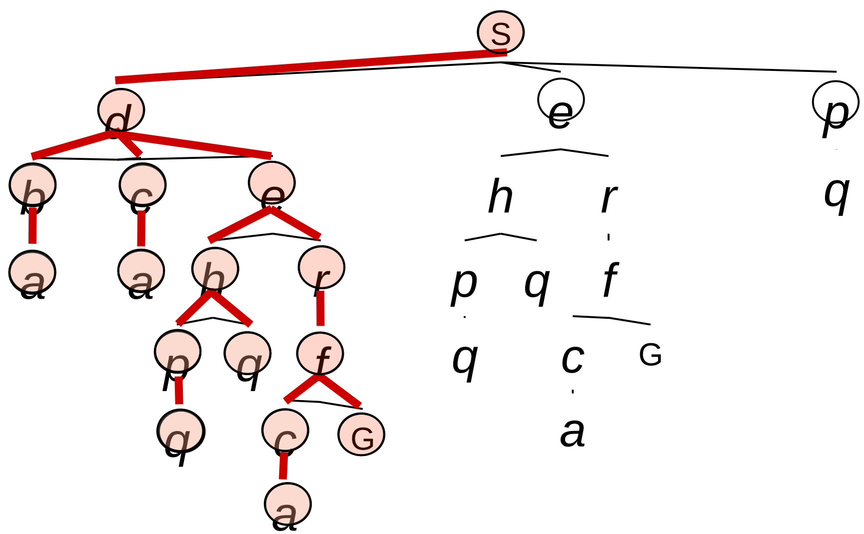


Depth-First Search

Strategy: *expand a deepest node first*

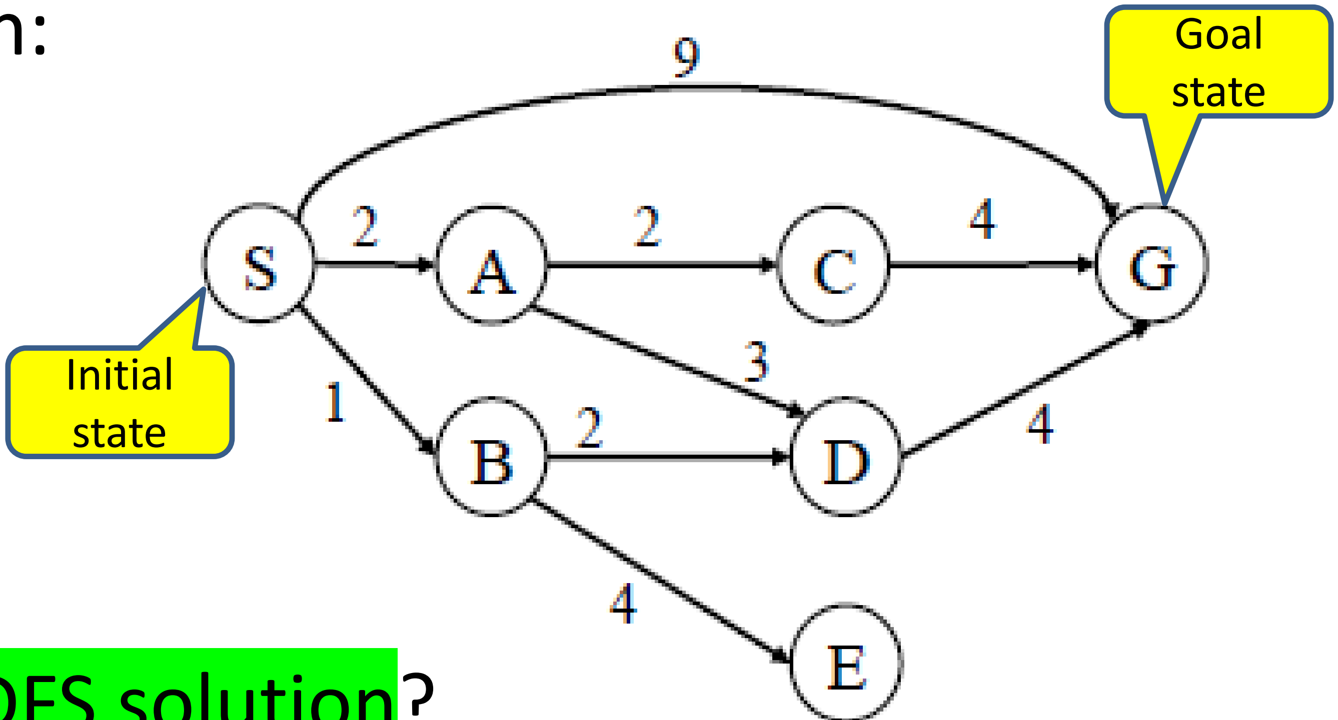
Implementation: *Fringe is a LIFO stack*





Exercise: Solve graph by DFS

- Given a graph:



- What is the DFS solution?
  - Assume children stored in alphabetical order.
- Solution: on board

Exercise 2: make into DFS tree search

```
def tree_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

DFS… in 1 line ☺

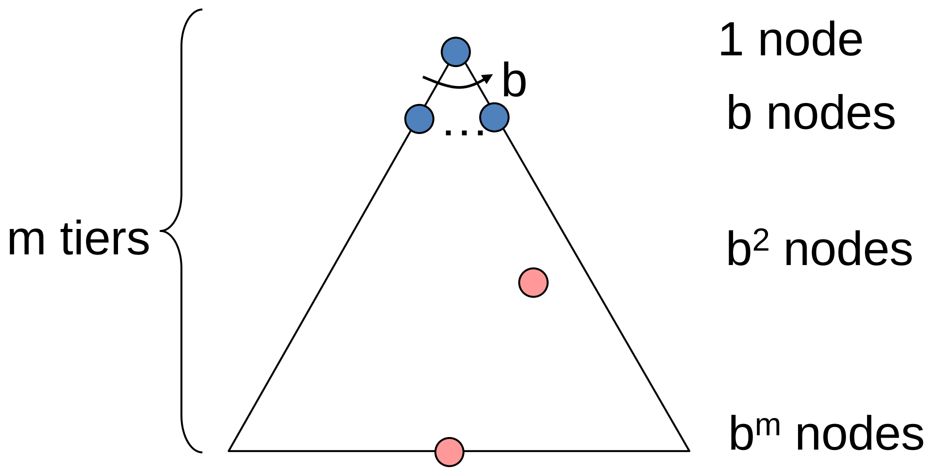
```
def DFS(problem):
    """Search the deepest nodes in the search tree first."""
    return tree_search(problem, Stack())
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

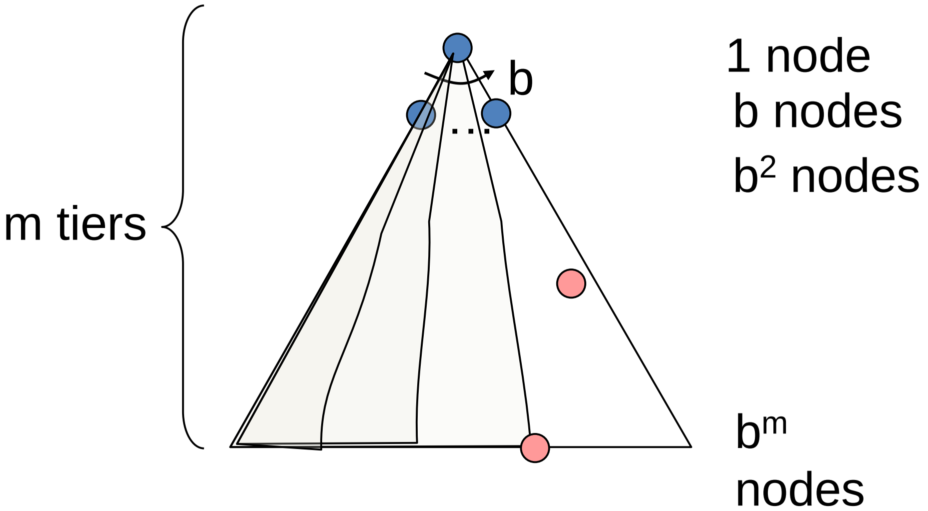
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths



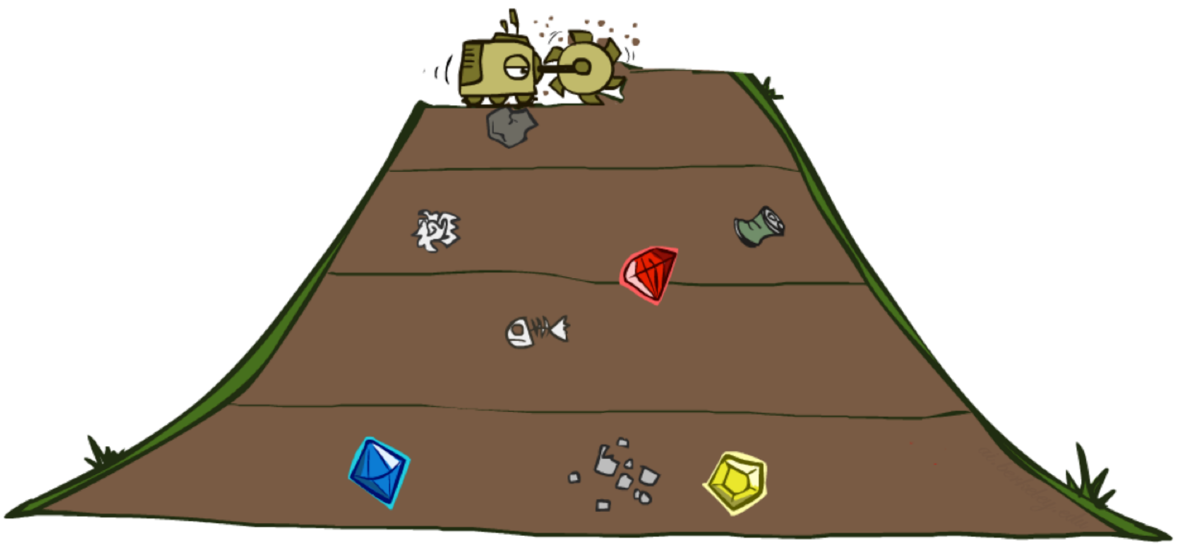
- Number of nodes in entire tree?
  - $1 + b + b^2 + ... b^m = O(b^m)$

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 yes iff we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost



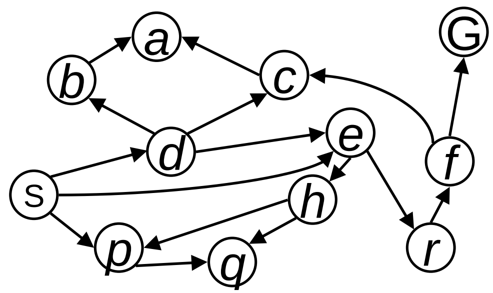
Breadth-First Search



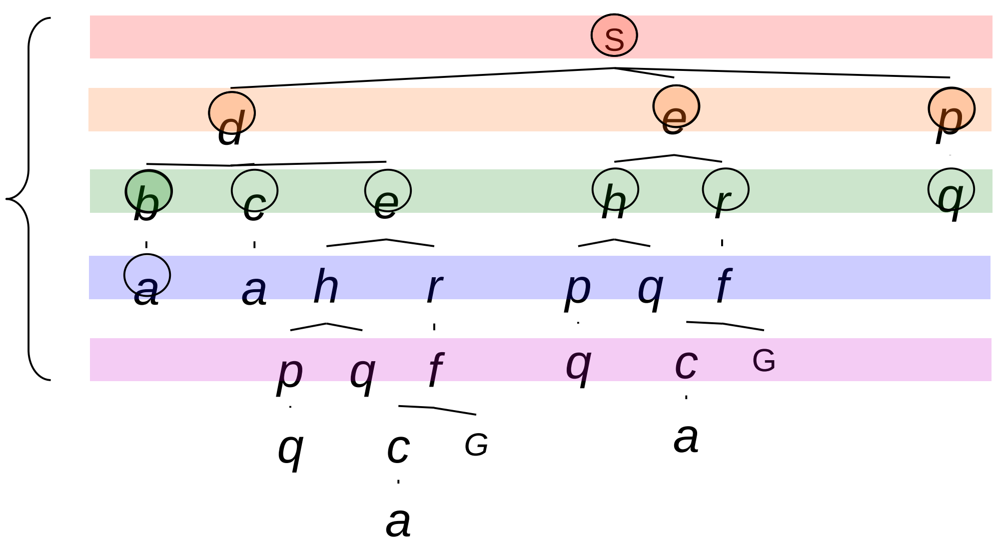
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

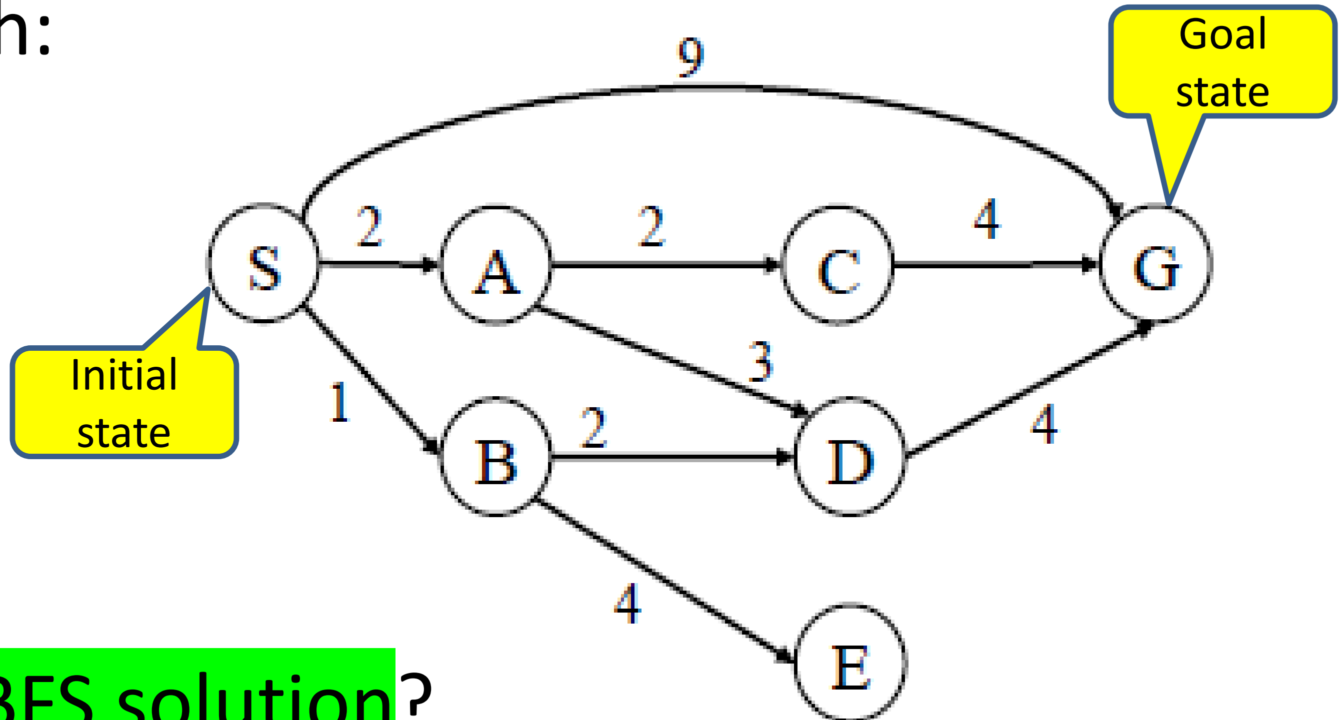


Search Tiers



Exercise: Solve graph by BFS

- Given a graph:



- What is the BFS solution?
  - Assume children stored in alphabetical order.
- Solution: on board

Exercise 3: make into BFS tree search

```
def tree_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

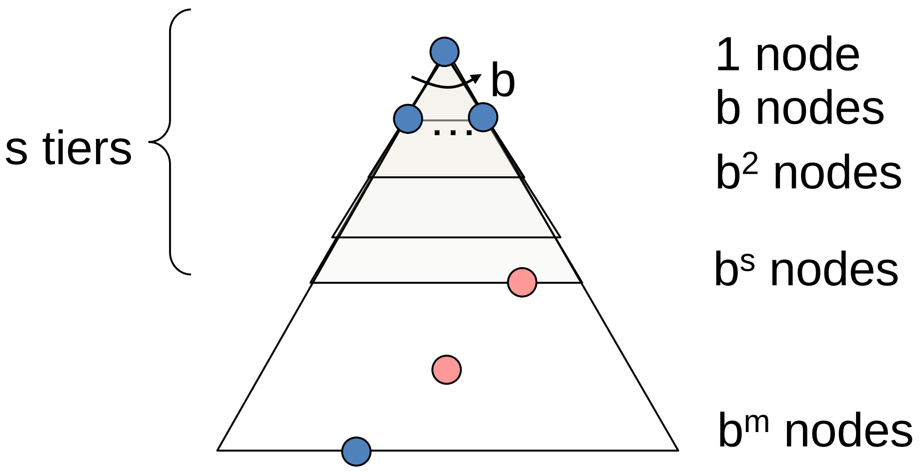
BFS … in 1 line ☺

```
def BFS(problem):
    """Search the shallowest nodes in the search tree first."""
    return tree_search(problem, FIFOQueue() )
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Breadth-First Search (BFS) Properties

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



Memory a Limitation?

- Suppose:
  - 4 GHz CPU
  - 6 GB main memory
  - 100 instructions / expansion
  - 5 bytes / node

- 400,000 expansions / sec
  - Memory filled in 300 sec … 5 min

Remember: BFS needs to keep $O(b^d)$ states (fringe) in memory

Exercise: DFS vs BFS

- When will BFS outperform DFS?

- When will DFS outperform BFS?

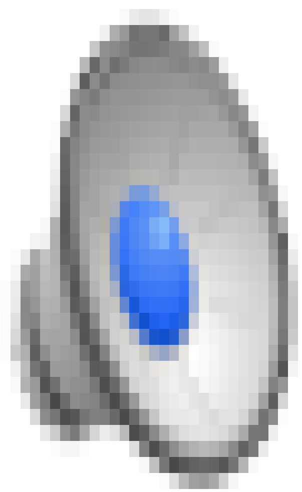
Comparisons

- When will BFS outperform DFS?

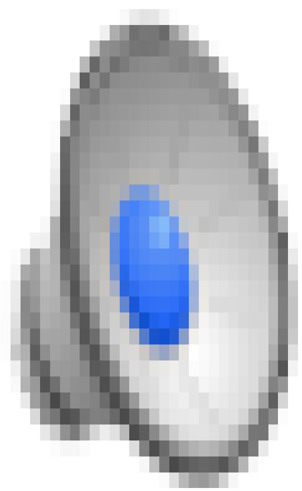
- When will DFS outperform BFS?

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	Y*	$O(b^d)$	$O(b^d)$

Video of Demo Maze Water DFS or BFS? (1)

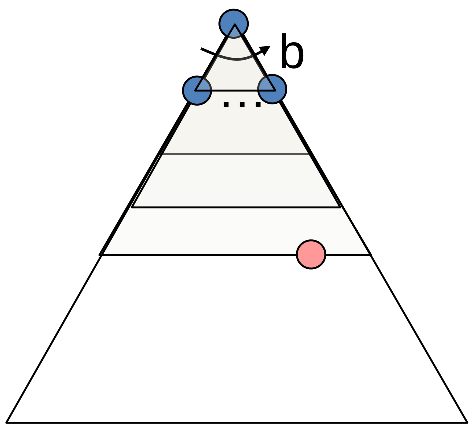


Video of Demo Maze Water DFS or BFS? (2)



BFS+DFS: Iterative Deepening (ID)

- Idea: combine DFS space advantage with BFS 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!

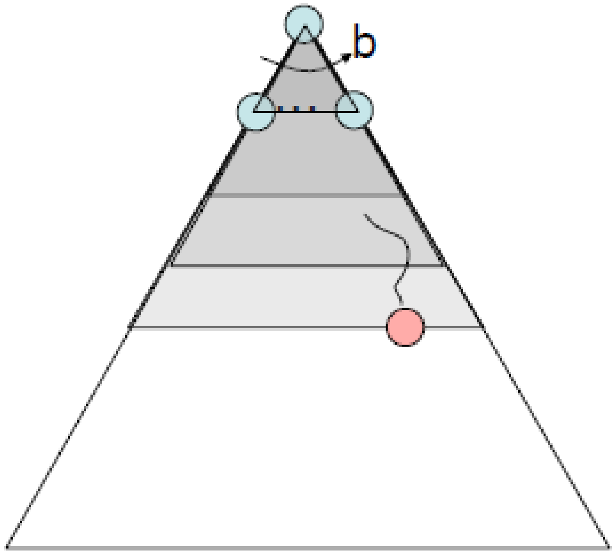


Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

….and so on.



Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	Y*	$O(b^d)$	$O(b^d)$
ID		Y	Y*	$O(b^d)$	$O(bd)$

Speed

Assuming 10M nodes/sec & sufficient memory

	BFS Nodes	BFS Time		Iter. Deep. Nodes	Iter. Deep. Time
8 Puzzle	10^5	.01 sec		10^5	.01 sec
2x2x2 Rubik's	10^6	.2 sec		10^6	.2 sec
15 Puzzle	10^{13}	6 days	1Mx	10^{17}	20k yrs
3x3x3 Rubik's	10^{19}	68k yrs	8x	10^{20}	574k yrs
24 Puzzle	10^{25}	12B yrs		10^{37}	10^{23} yrs

Why the difference?

Rubik has higher branch factor

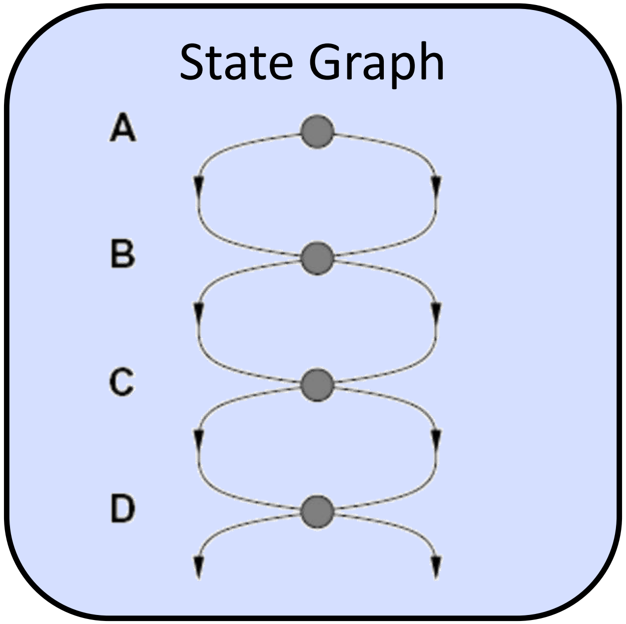
15 puzzle has greater depth

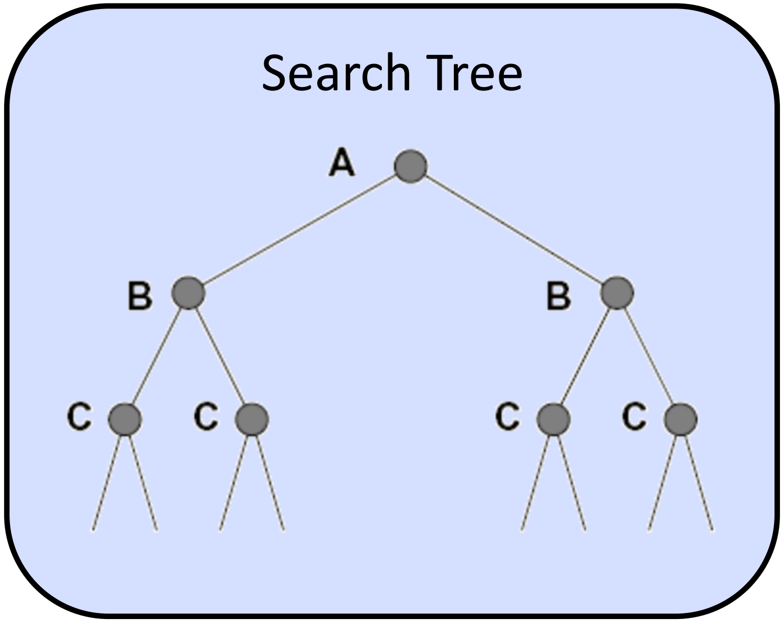
\# of duplicates

Slide adapted from Richard Korf presentation

Danger!

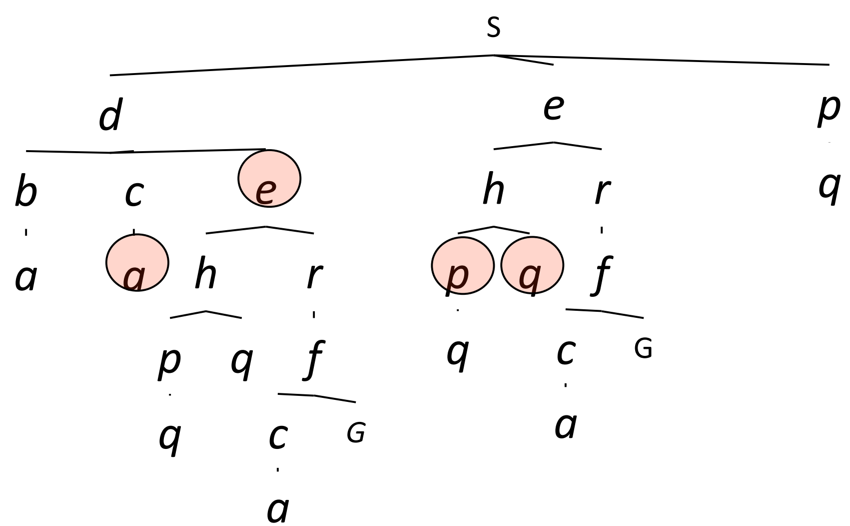
- Failure to detect repeated states can cause exponentially more work.





Graph Search: BFS

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



Graph Search: **Implementation**

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

Graph Search PseudoCode

General code.
Exercise: How to make it behave like BFS? DFS?

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

                        ...

        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Exercise: Python Implementation

```
def graph_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Exercise: Python Implementation

```
def graph_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
   The argument fringe should be an empty queue.
    closed = {}
    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node
        if node.state not in closed:
            closed[node.state]=1
        else continue

        fringe.extend(node.expand(problem))

    return None
```

Node for Search Tree (Python)

```
class Node:

    """node in a search tree. Contains pointer to parent (the node
    that this is a successor of) and to the actual state for this node.
    Note that if a state is arrived at by two paths, then there are two
nodes with the same state.  Also includes the action that got us to
this state, and the total path_cost (also known as g) to reach the
node."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
         self.depth = 0
        if parent:
            self.depth = parent.depth + 1
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Node: methods

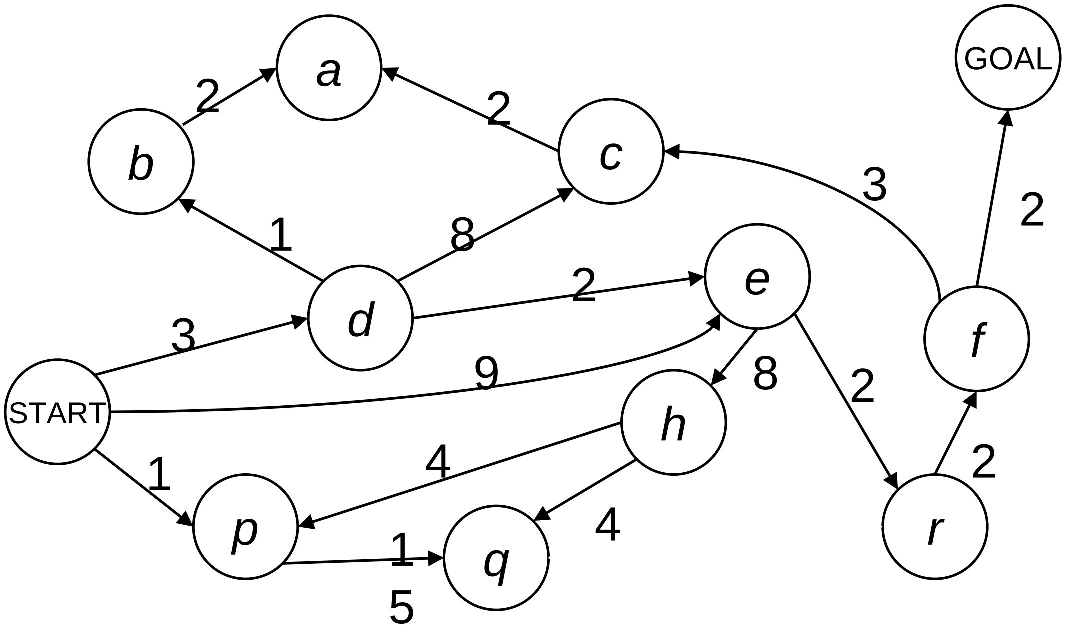
```
class Node:
…. . .
  def expand(self, problem):
     """List the nodes reachable in one step from this node."""
     return [ self.successor(problem, action)
                 for action in problem.actions(self.state) ]

   def successor(self, problem, action):
      next = problem.result(self.state, action)
      return Node(next, self, action)

   def path(self):
      node, path_back = self, []
      while node:
         path_back.append(node)
         node = node.parent
      return list(  reversed(path_back) )
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Node: using in Queue

```
class Node:

    …
    # We want for a queue of nodes in breadth_first_search or
    # DFS to have no duplicated states, so we treat nodes
    # with the same state as equal. [Note: this may not be what you
    # want in other contexts.]

    def __eq__(self, other):
        return isinstance(other, Node) and self.state == other.state

    def __hash__(self):
        return hash(self.state)
```

Cost-Sensitive Search (UCS)

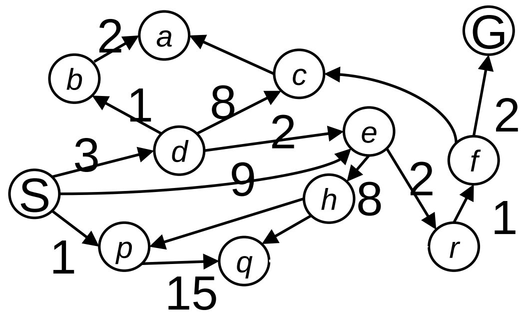


BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path. We will now cover
a similar algorithm which does find the least-cost path.

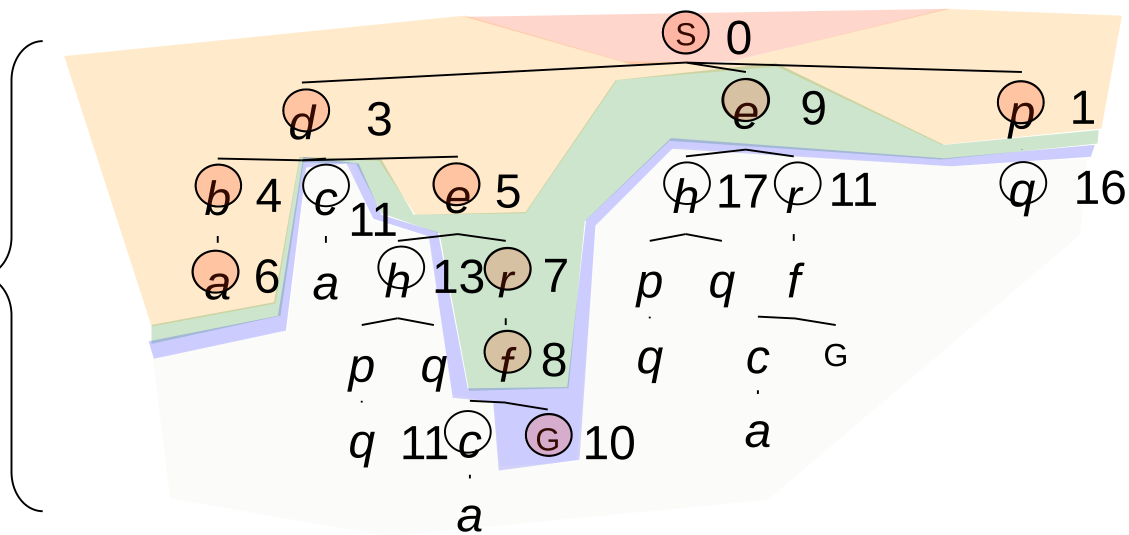
Uniform Cost Search

Strategy: expand a cheapest node first:

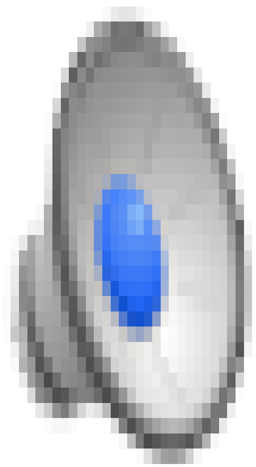
Fringe is ***a priority queue*** *(priority: cumulative cost)*



Cost contours

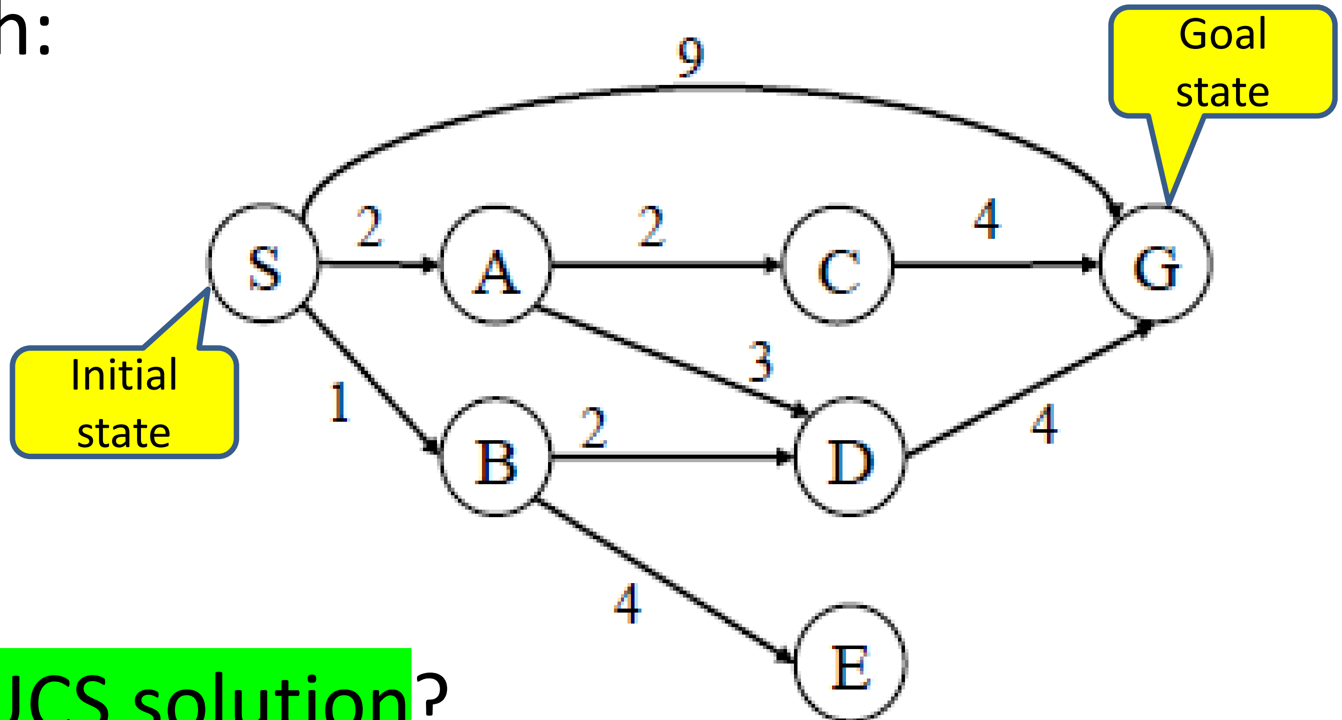


Video of Demo Empty UCS



Exercise: Solve graph by UCS

- Given a graph:



- What is the UCS solution?
  - Assume children stored in alphabetical order.
- Solution: on board

Exercise: make into UCS tree search

```
def tree_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

UCS … in 1 line ☺

```
def BFS(problem):
    """Search the shallowest nodes in the search tree first."""
    return tree_search(problem,
                util.PriorityQueueWithFunction(Node.getCost))
```

HUH?

http://www.mathcs.emory.edu/~eugene/cs425/p1/docs/util.html

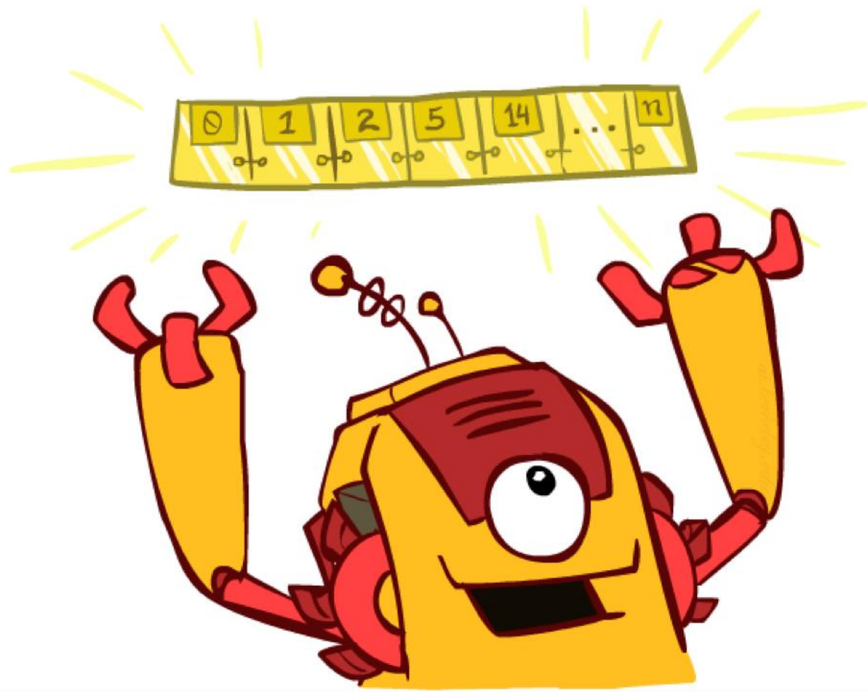
Node for UCS (expanded)

```
class Node:


    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

  def getCost(self):
    return self.cost
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

One Queue to rule them all…

- 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 stack and queues
  - Python Hint: can make one general graph search implementation that takes a variable **Fringe** object as a parameter
  - Use utils.pm for Stack, Queue, PriorityQueue classes.



Priority Queue Refresher

- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

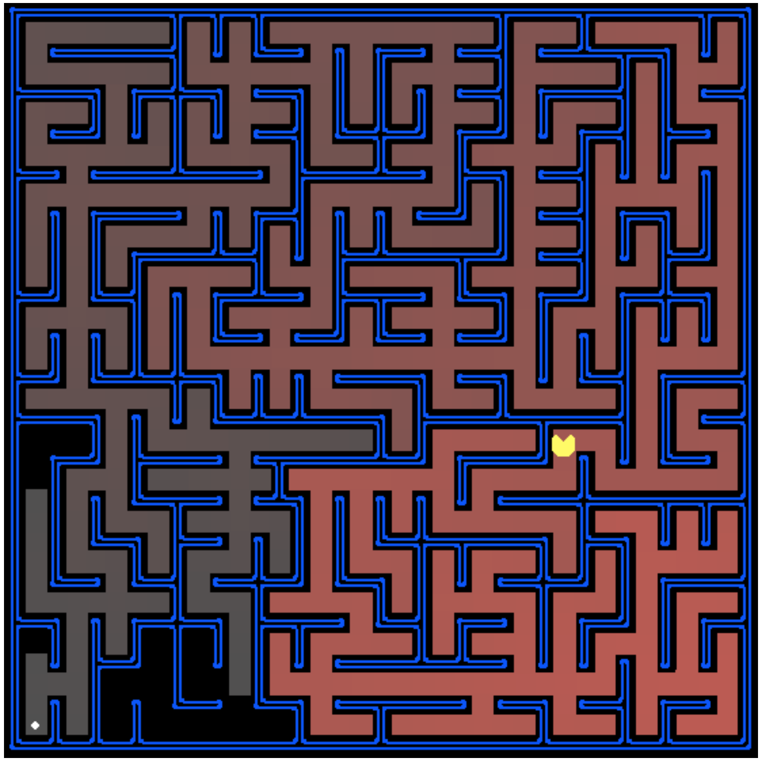
pq.push(key, value)	inserts (key, value) into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

- You can decrease a key's priority by pushing it again
- Unlike a regular queue, insertions aren't constant time, usually O(log n)
- We'll need priority queues for cost-sensitive search methods

Project 1: Pacman Search

http://www.mathcs.emory.edu/~eugene/cs425/p1/

Due: Friday February 9th



Project 1 Tips

- Use Discussions, read FAQ before posting questions:
- **Questions 1-3**: if you develop a correct solution for DFS, the rest will be easy modifications
- Do not use shortcuts: Do use Node class or similar. You will need these for Questions 5-8.
- Questions 5-8: more fun/creative. Leave enough time, start early.
- Most importantly: Don’t Panic! Eat the elephant one question at a time.

To Do: Start solving Q1-3

Files you'll edit:

search.py	Where all of your search algorithms will reside.
searchAgents.py	Where all of your search-based agents will reside.

Files you might want to look at:

pacman.py	The main file that runs Pac-Man games. This file describes a Pac-Man GameState type, which you use in this project.
game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	**Useful** data structures for implementing search algorithms