# Python Programming

```python
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
```

```
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```python
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))  # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]

portfolio = [
   {'name': 'IBM', 'shares': 100, 'price': 91.1},
   {'name': 'AAPL', 'shares': 50, 'price': 543.22},
   {'name': 'FB', 'shares': 200, 'price': 21.09},
   {'name': 'HPQ', 'shares': 35, 'price': 31.75},
   {'name': 'YHOO', 'shares': 45, 'price': 16.35},
   {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
```

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
```

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```python
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}

e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

```python
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)

d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Python Programming

```python
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

```python
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

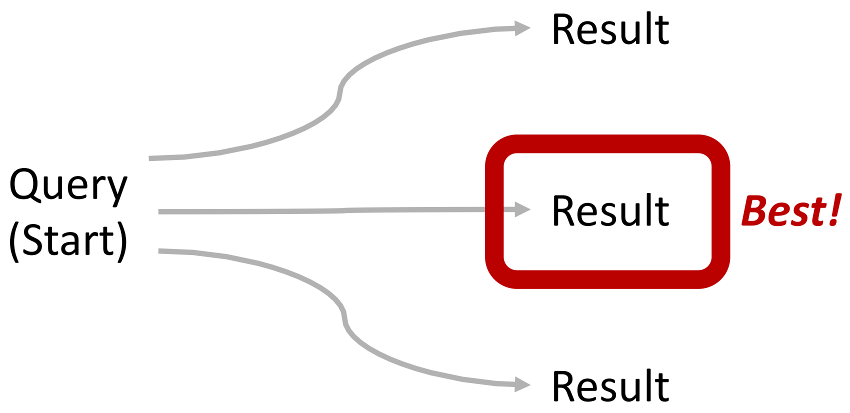
Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

# Search Problems

# Search?

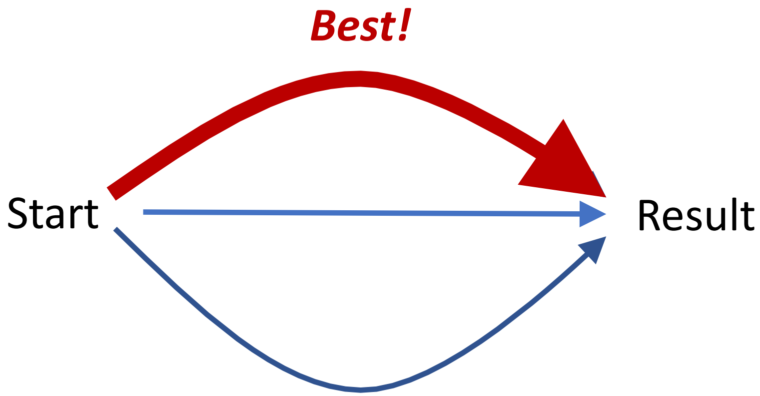
# Information Retrieval vs Search

*(Information Retrieval)*




## Search

*(Problem-Solving)*



# Definition of Search

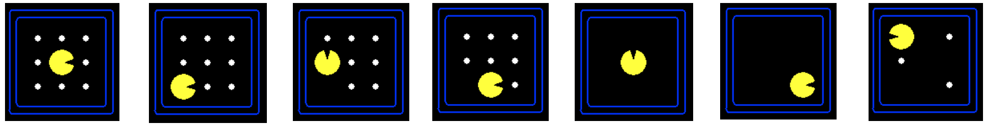
**Finding a (best) sequence of actions to solve a problem**

For now, assume the problem is

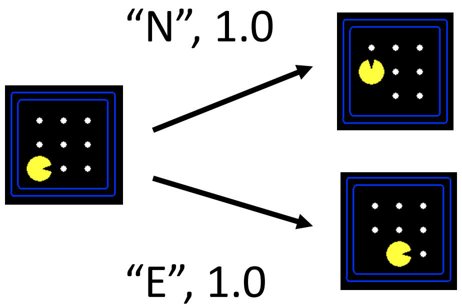
- Deterministic
- Fully observable
- Known

# Search Problem Mechanics

- A search problem consists of:
  - A state space



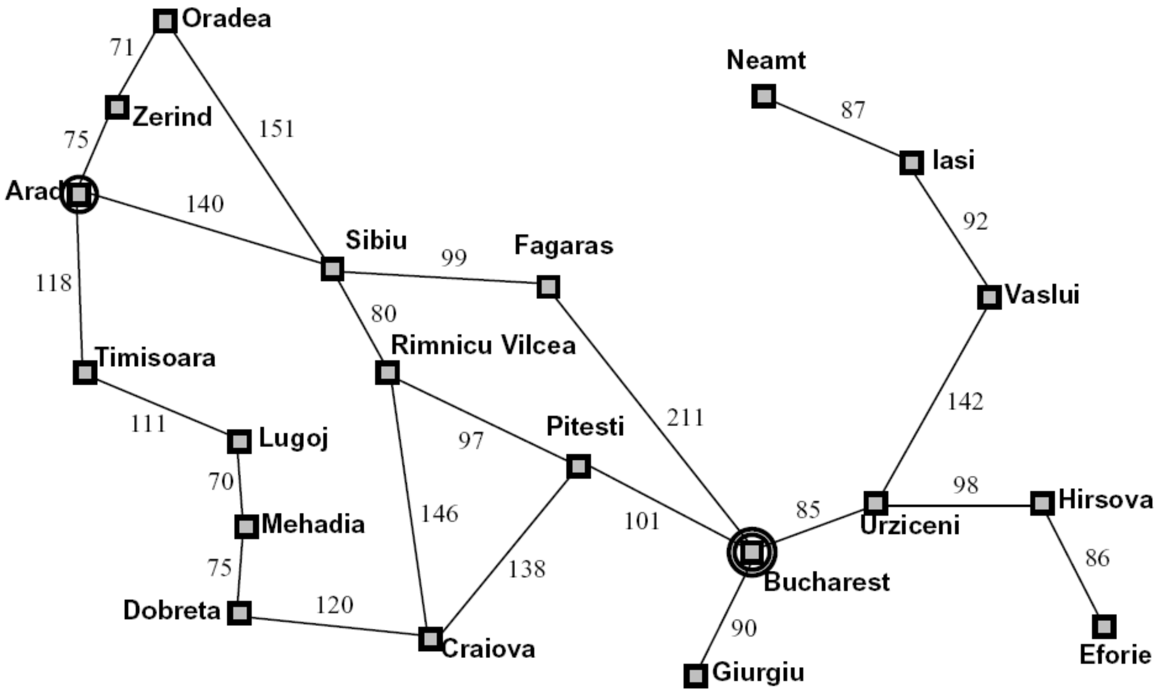
  - A successor function (with actions, costs)




  - 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

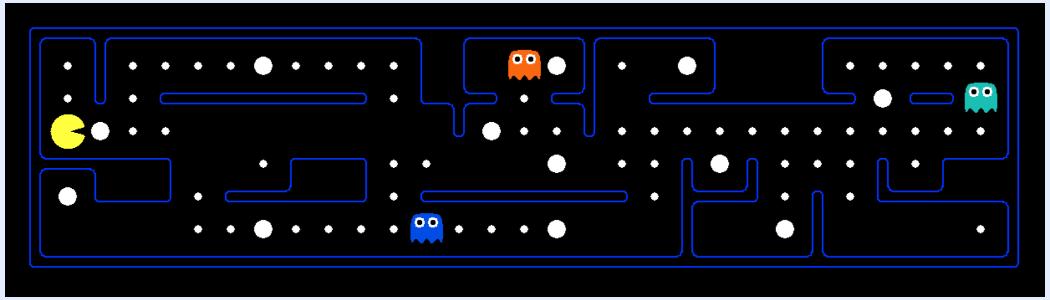
# Example: Traveling in Romania




- 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



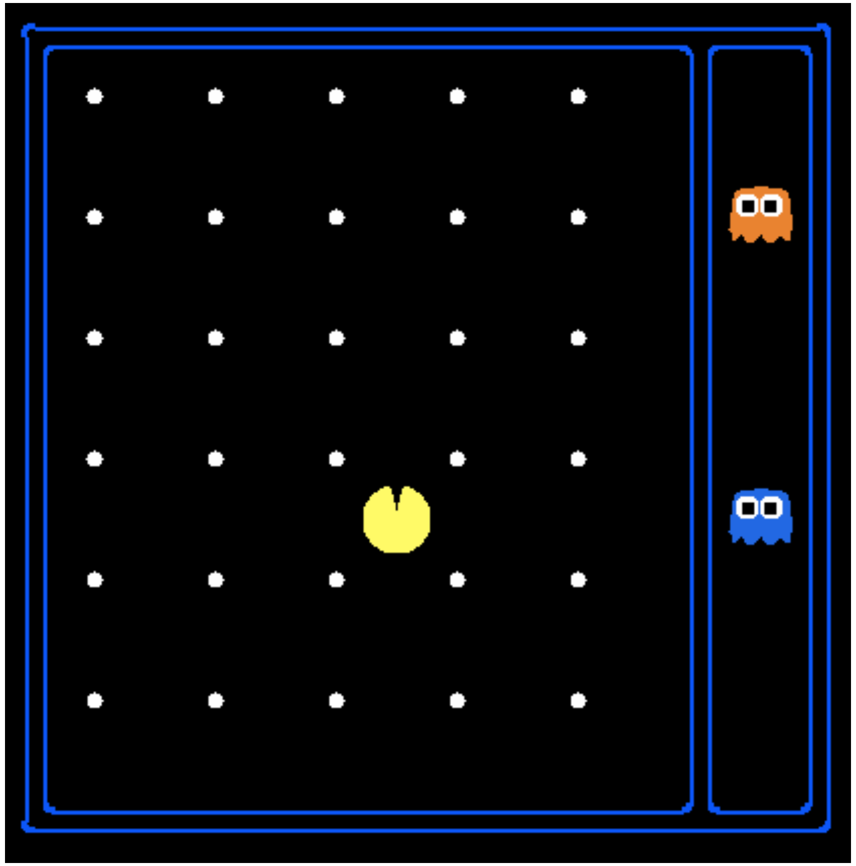
A search state keeps only the details needed for planning (abstraction)

- 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), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

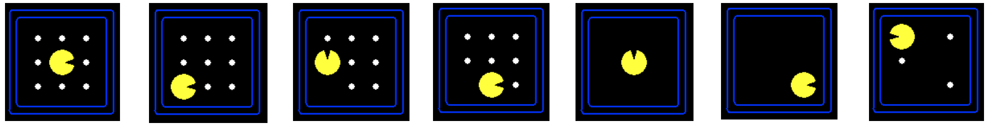
# State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW
- How many
  - World states?
    $120x(2^{30})x(12^2)x4$
  - States for pathing?
    120
  - States for eat-all-dots?
    $120x(2^{30})$

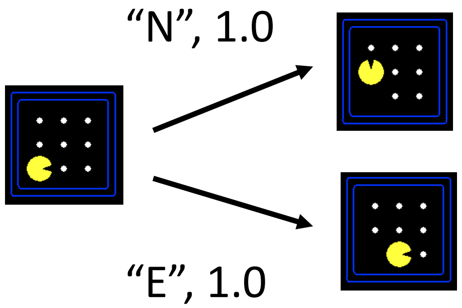


# Search Problem Mechanics

- A search problem consists of:
  - A state space



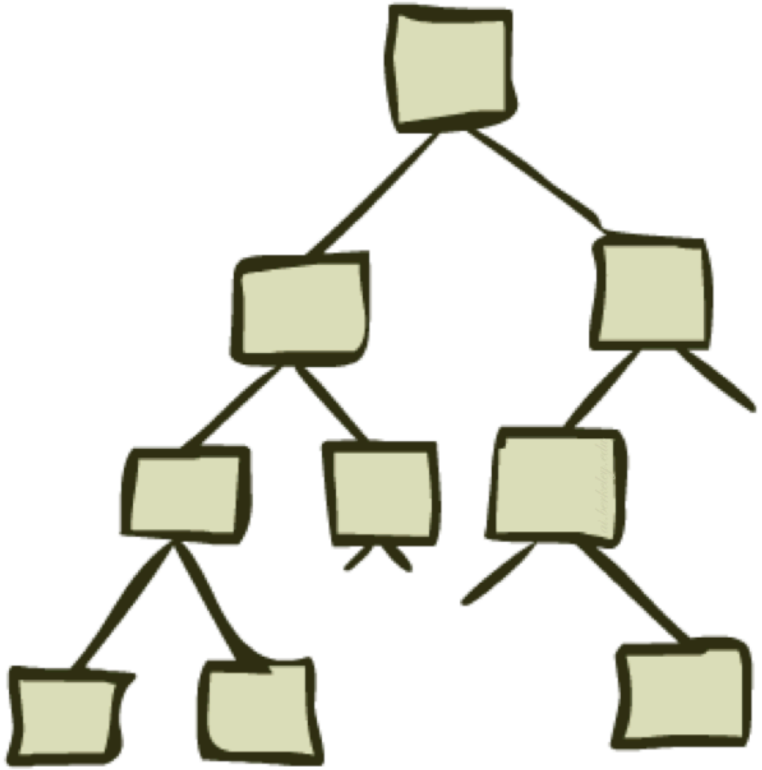
  - A successor function (with actions, costs)
  - A start state and a goal test




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

What are some problems that <u>can’t</u> be formulated as search?

# State Space Graphs and Search Trees

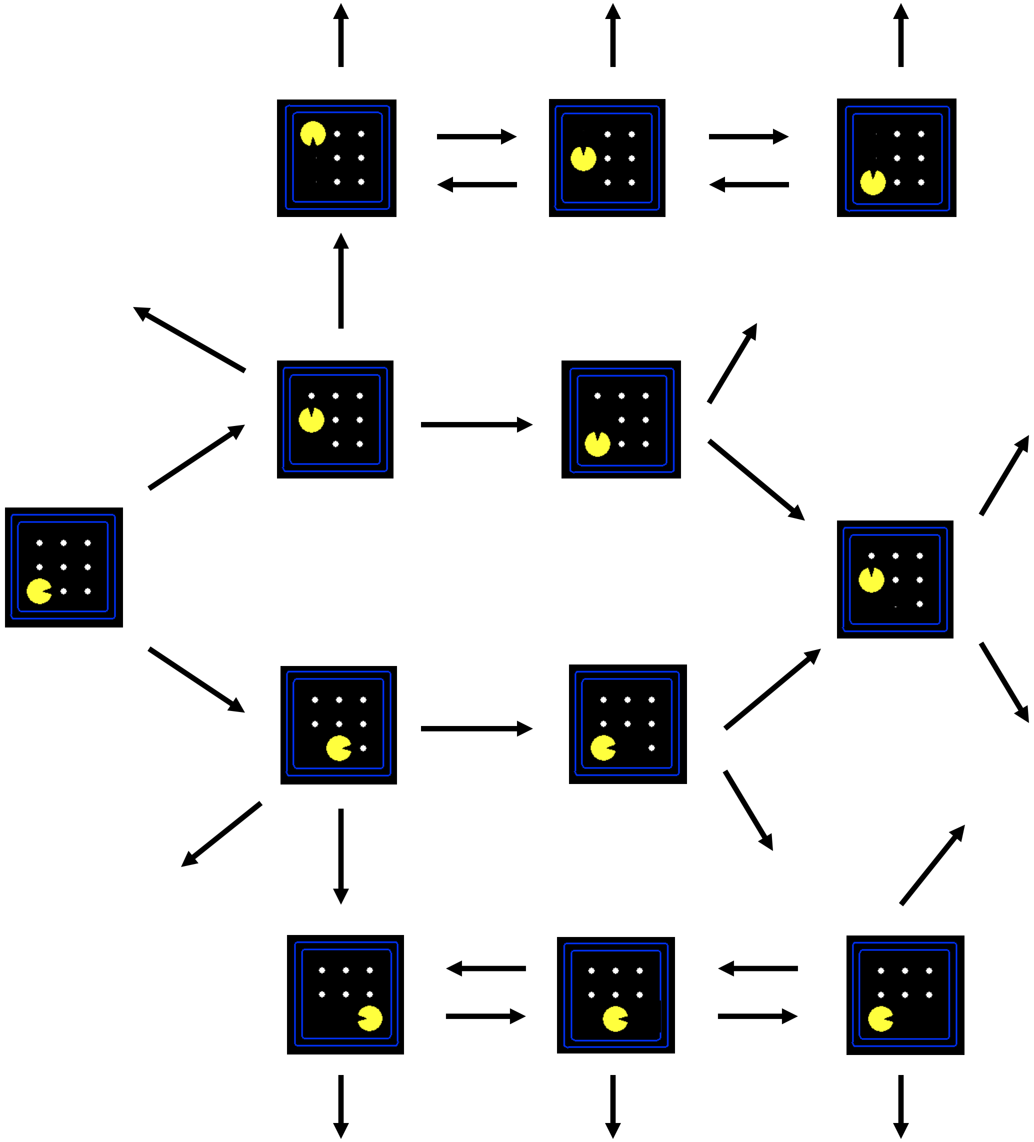


# State Space Graphs

- 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

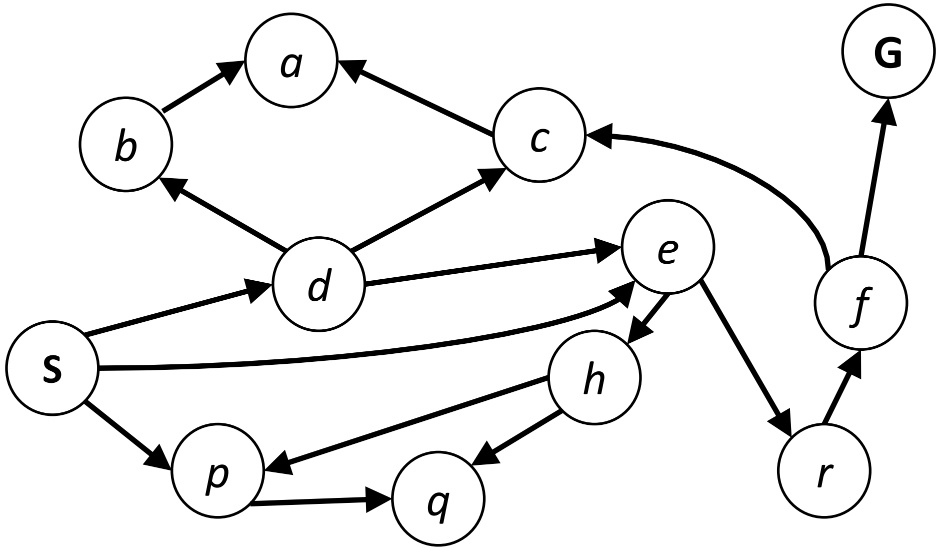


# State Space Graphs

- 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)

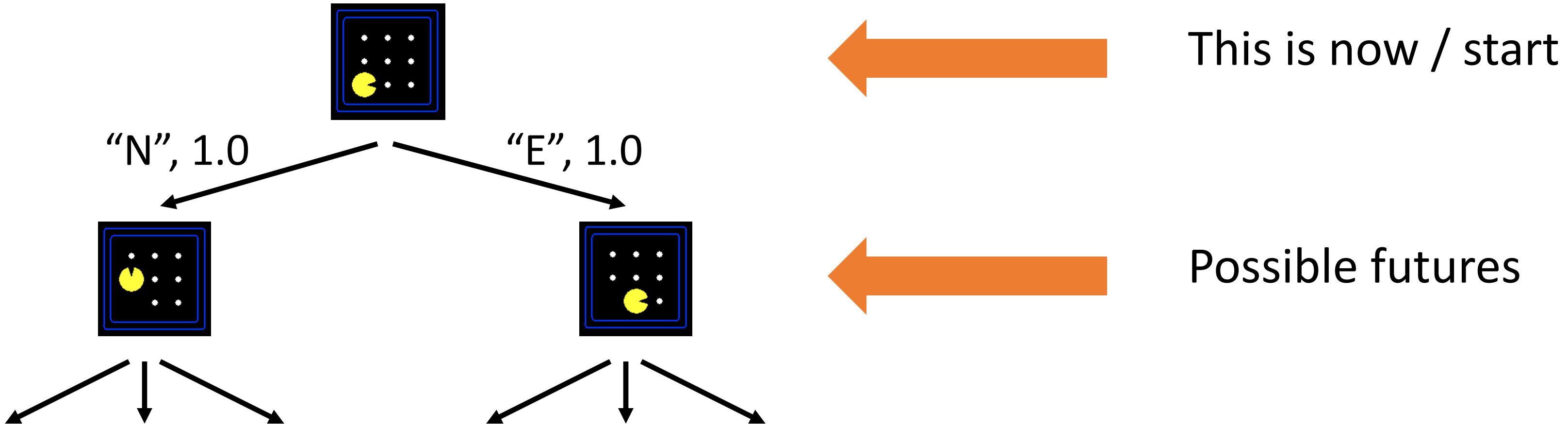
- In a state space graph, each state occurs only once!

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




*Tiny search graph for a tiny search problem*

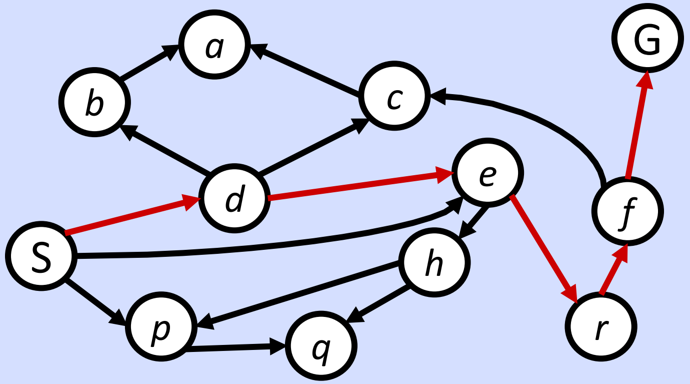
# Search Trees




- 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 ACTION SEQUENCES that achieve those states
  - For most problems, we can never actually build the whole tree

# State Space Graphs vs. Search Trees

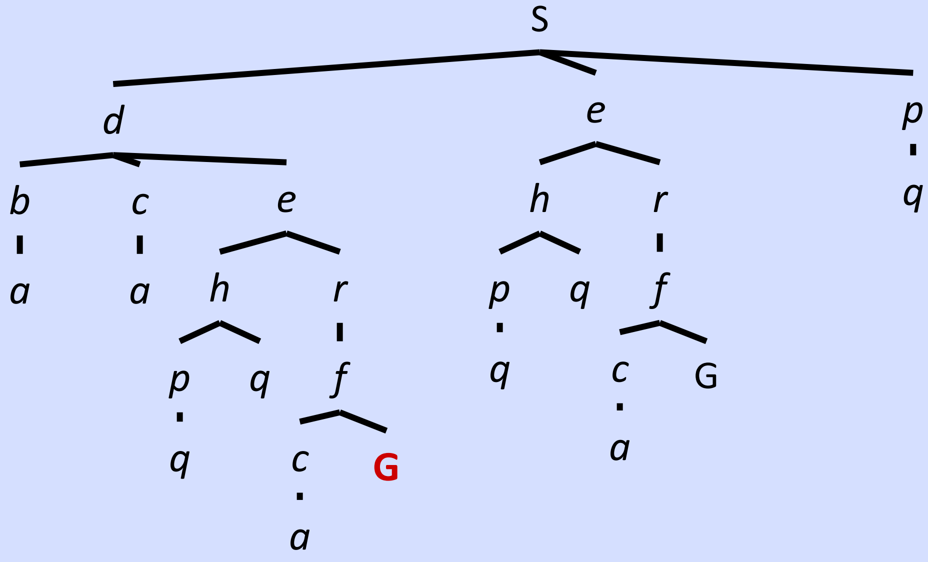
## State Space Graph




*Each NODE in in the search tree corresponds to an entire PATH in the state space graph.*

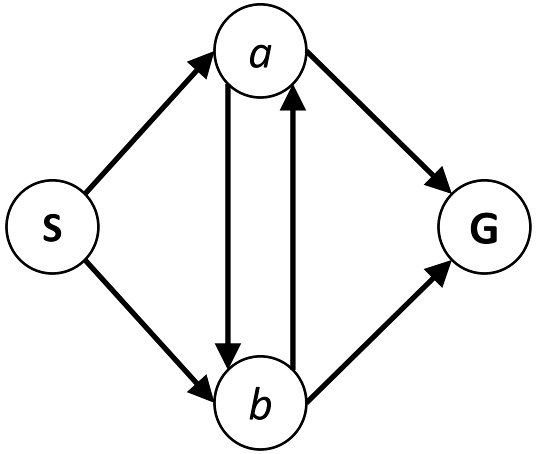
*We construct both on demand – and we construct as little as possible.*

## Search Tree



# Quiz: State Space Graphs vs. Search Trees

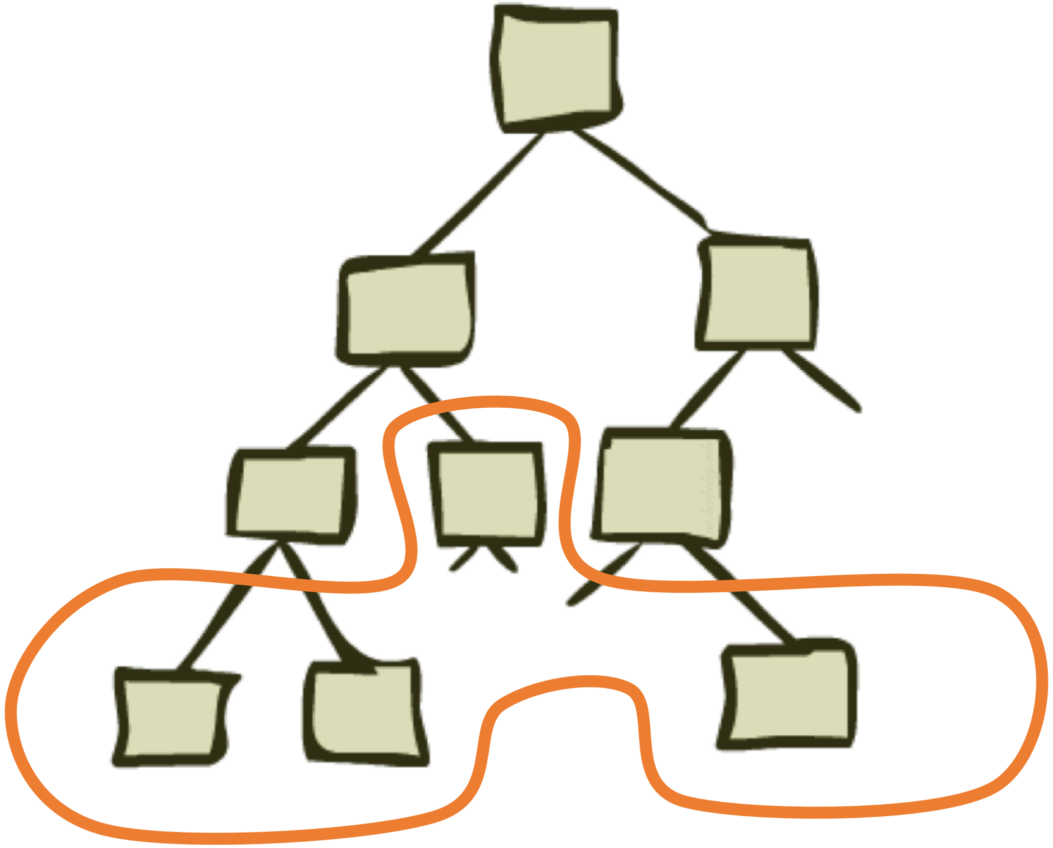
Consider this 4-state graph:



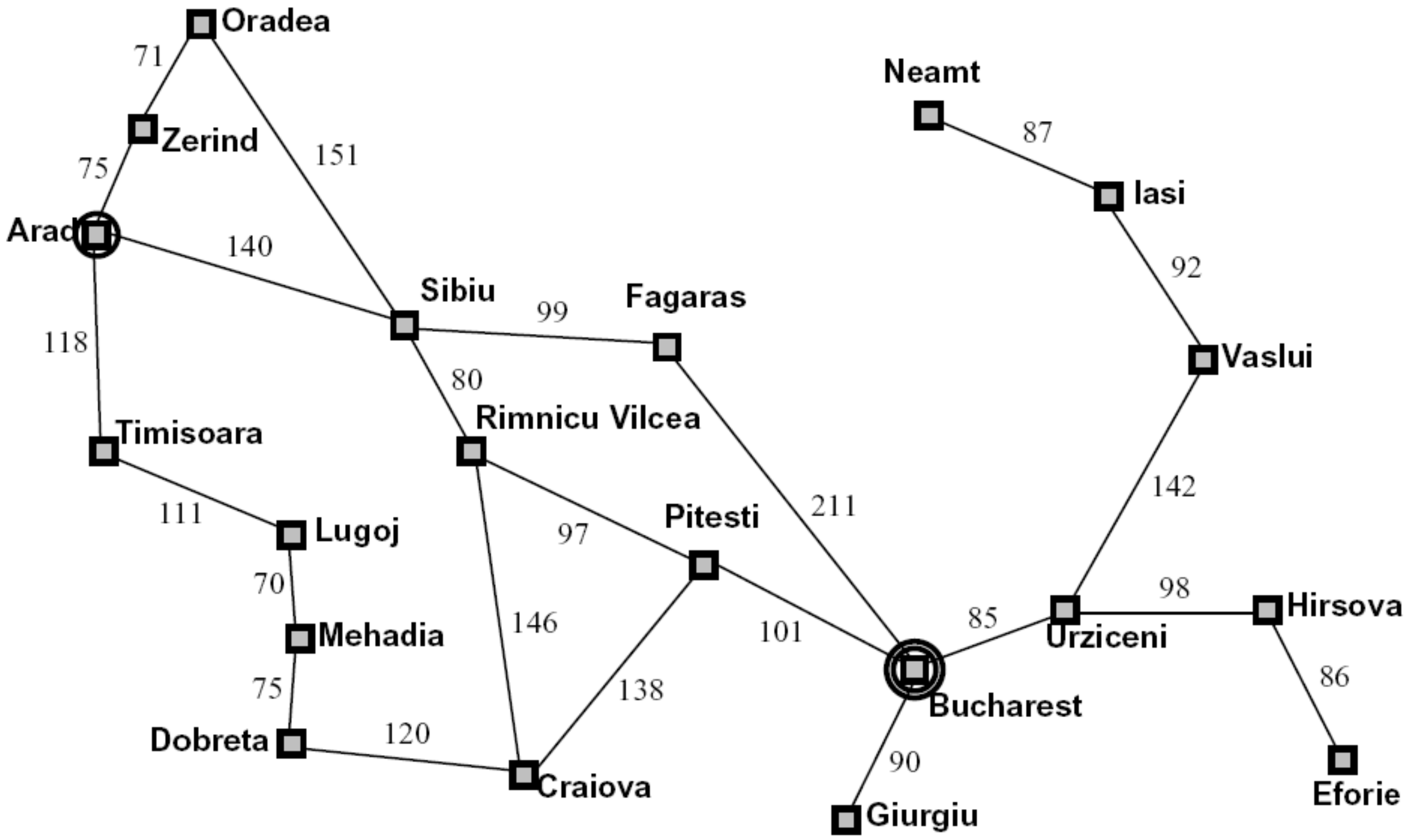

How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!

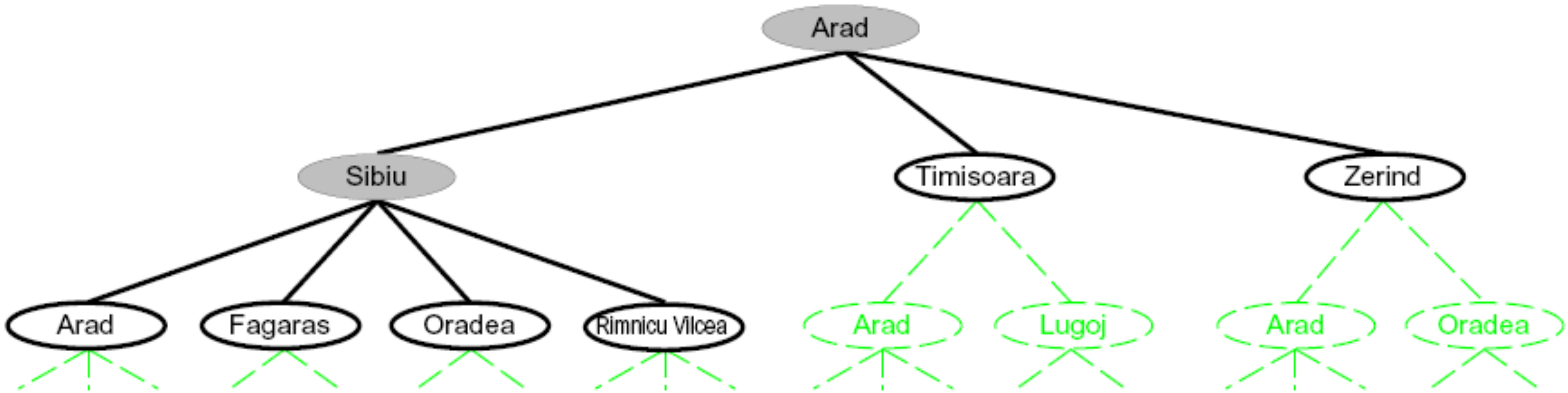
# Tree Search



# 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

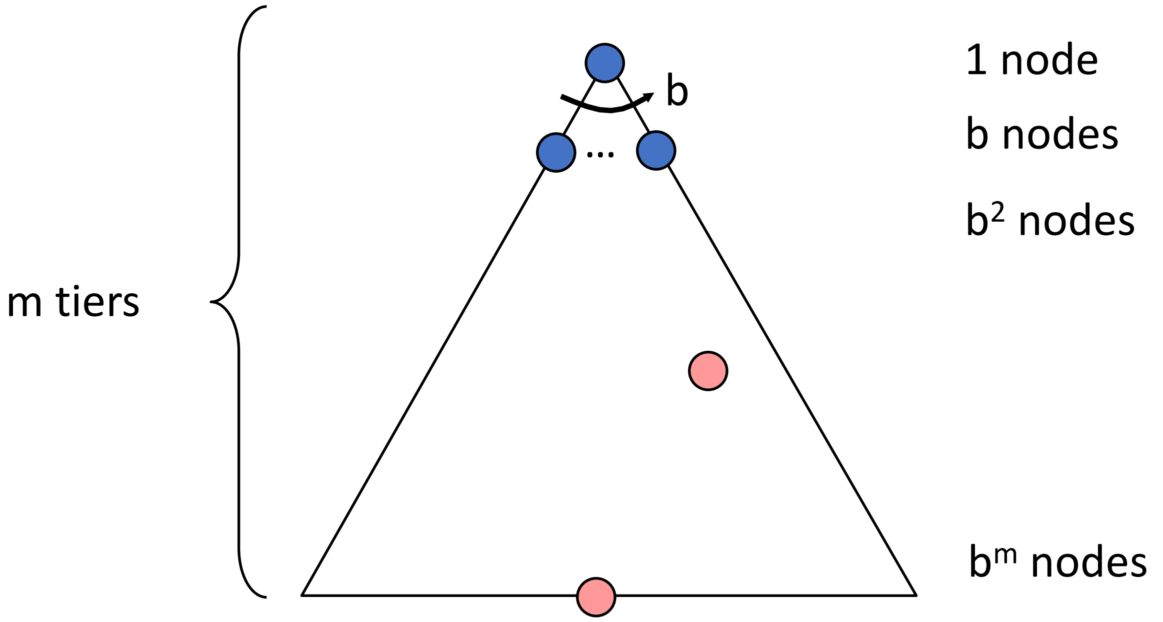
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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

# Search Algorithm Properties

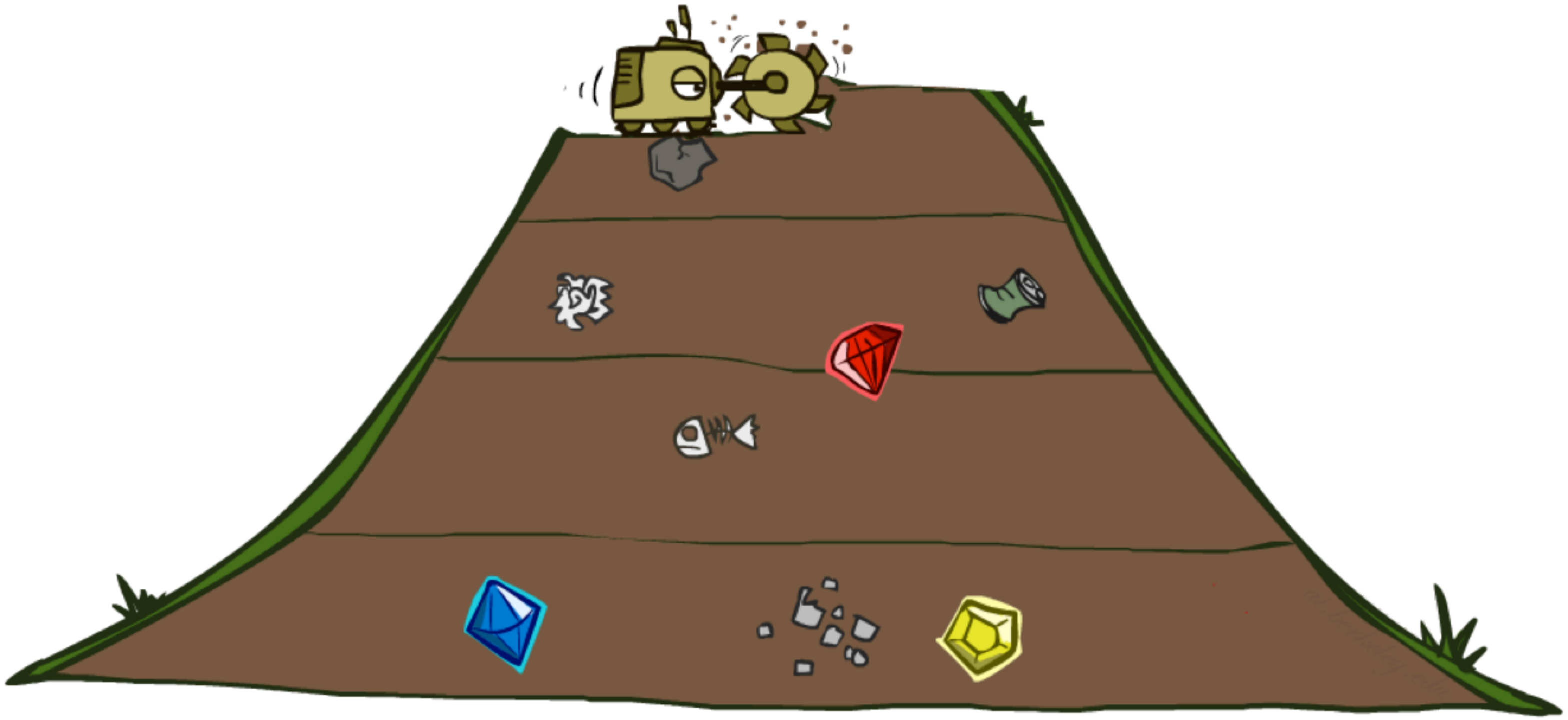
- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

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

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



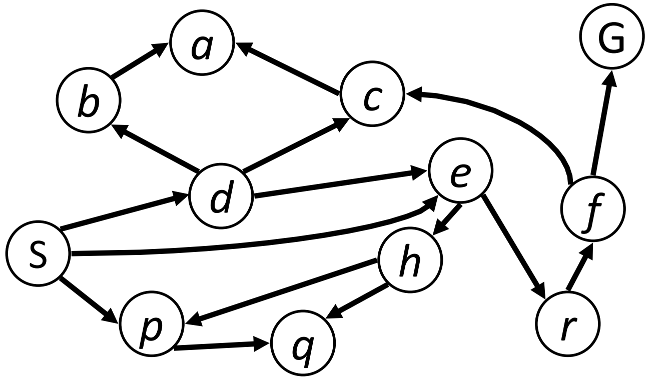
# Breadth-First Search

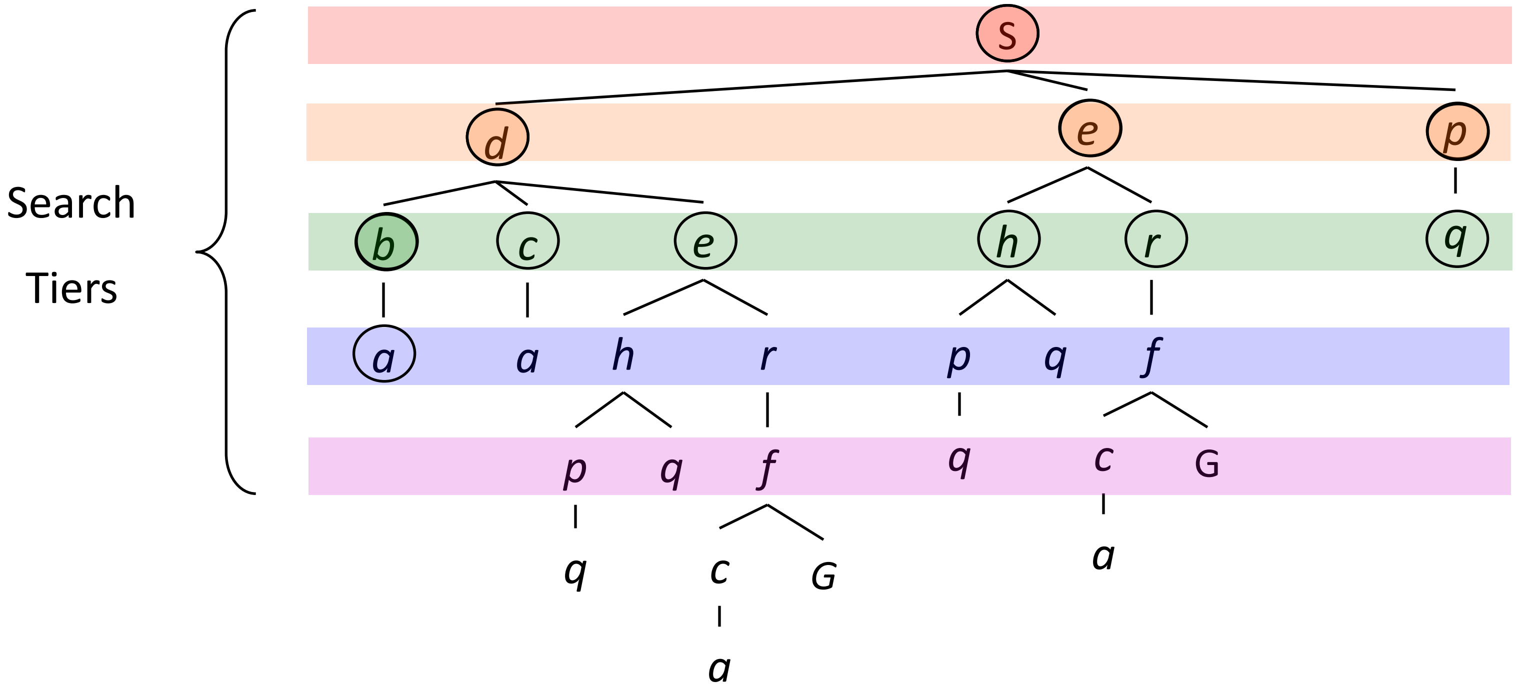


# Breadth-First Search

*Strategy: expand a shallowest node first*

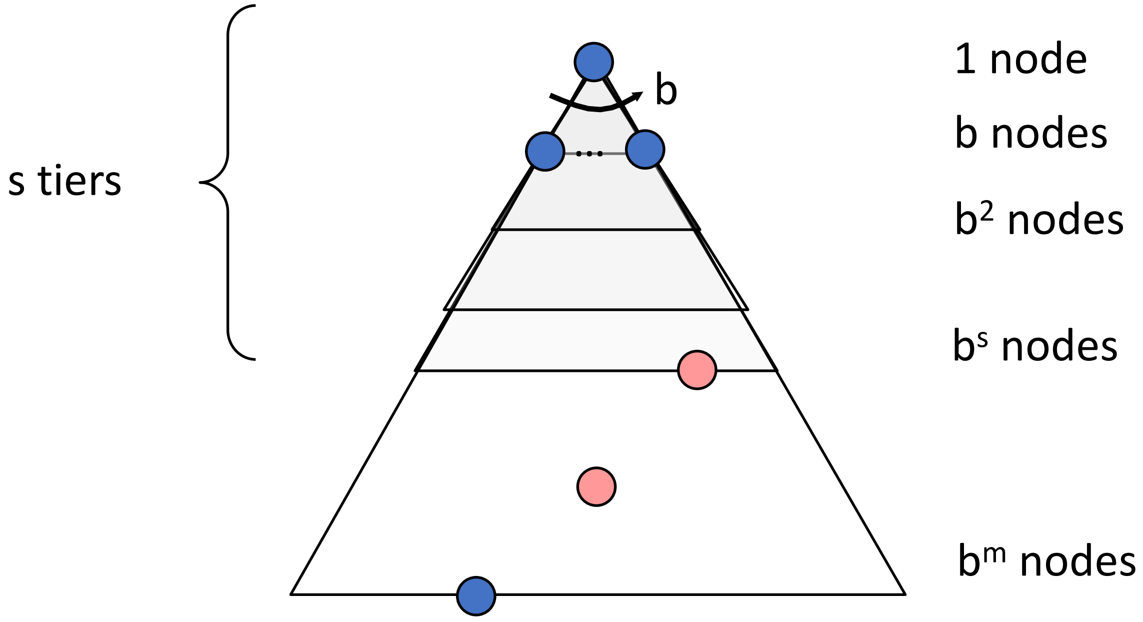
*Implementation: Fringe is a FIFO queue*



# Breadth-First Search (BFS) Properties

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



# Python code for BFS

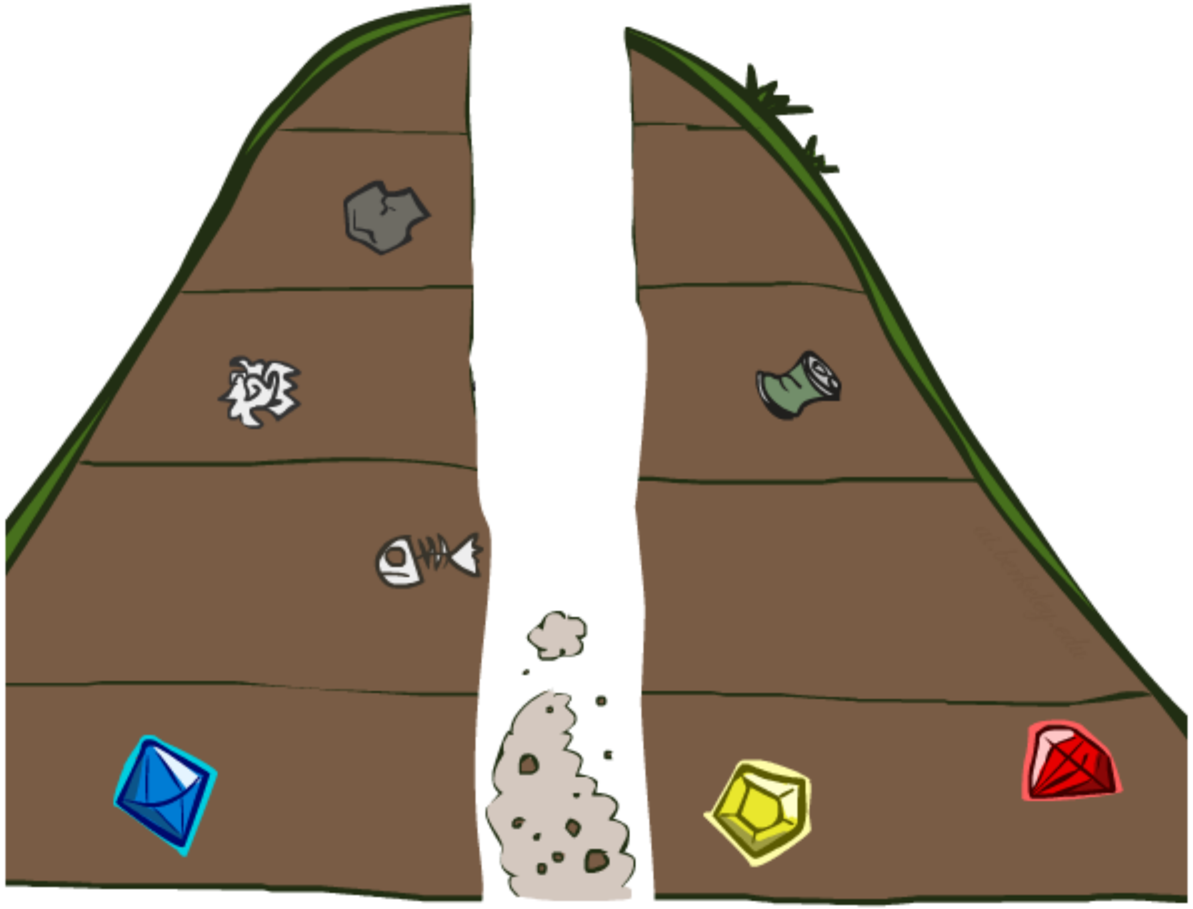
```python
import collections

def bfs(graph, root):
    seen, queue = set([root]), collections.deque([root])
    while queue:
        vertex = queue.popleft()
        visit(vertex)
        for node in graph[vertex]:
            if node not in seen:
                seen.add(node)
                queue.append(node)

def visit(n):
    print(n)

if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2, 0], 2: []}
    bfs(graph, 0)
```

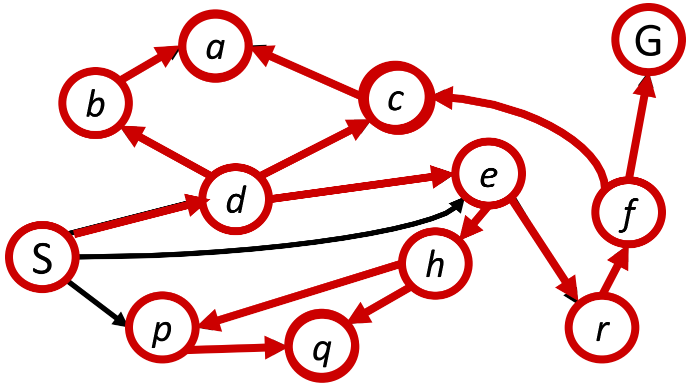
# Depth-First Search

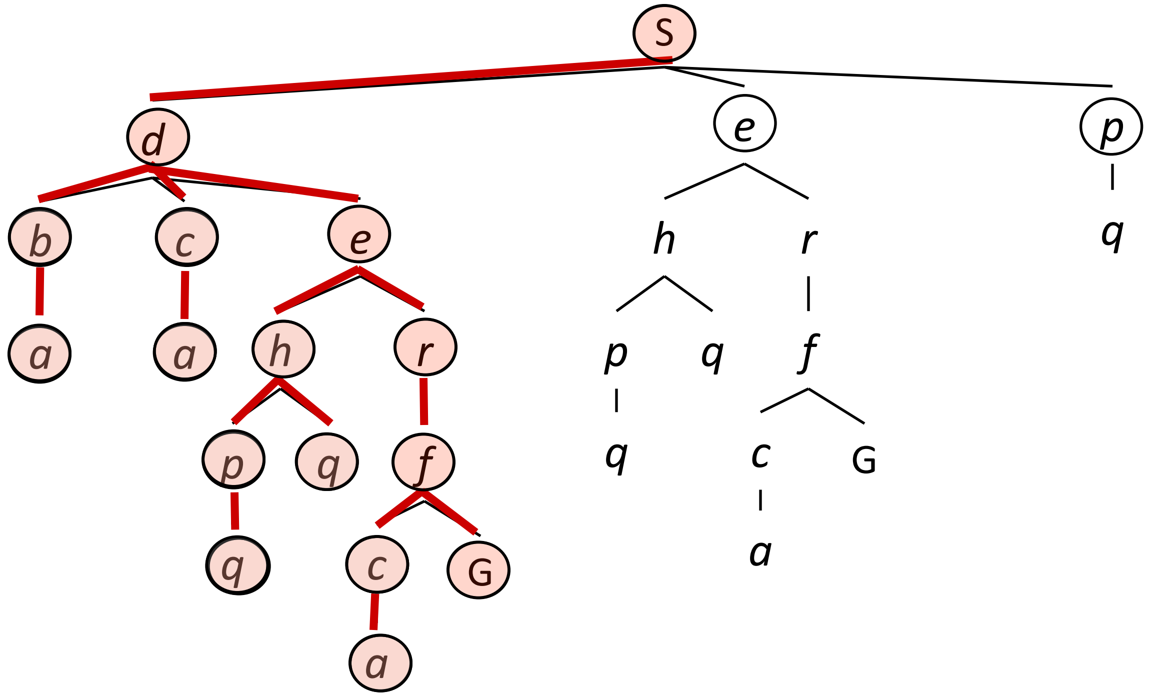


# Depth-First Search

*Strategy: expand a deepest node first*

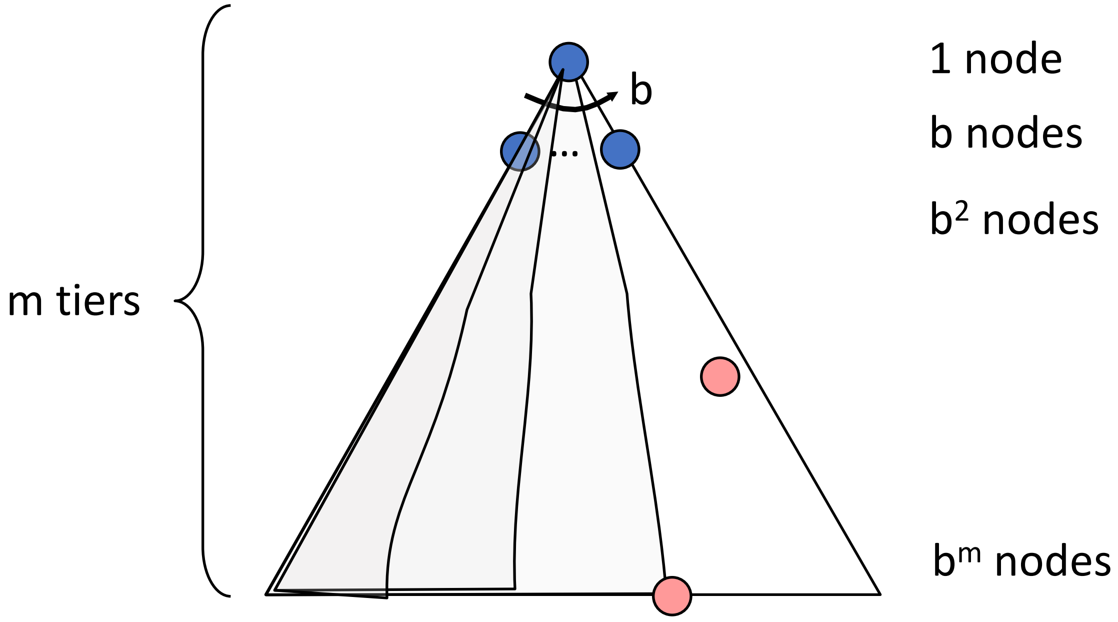
*Implementation: Fringe is a LIFO stack*



# 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



# Python code for DFS

```
def dfs_recursive(graph, vertex, path=[]):
    path += [vertex]

    for neighbor in graph[vertex]:
        if neighbor not in path:
            path = dfs_recursive(graph, neighbor, path)

    return path


adjacency_matrix = {1: [2, 3], 2: [4, 5],
                    3: [5], 4: [6], 5: [6],
                    6: [7], 7: []}

print(dfs_recursive(adjacency_matrix, 1))
# [1, 2, 4, 6, 7, 5, 3]
```

# DFS vs BFS

- If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
- If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
- If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.
- If solutions are frequent but located deep in the tree, BFS could be impractical.
- If the search tree is very deep you will need to restrict the search depth for depth first search (DFS), anyway (for example with iterative deepening).