# Agents that Plan Ahead: Search

**Russell and Norvig: Chapter 3.1-3.4, 3.5-3.6**
**CSE 240: Winter 2023**
**Lecture 2**

# Agenda

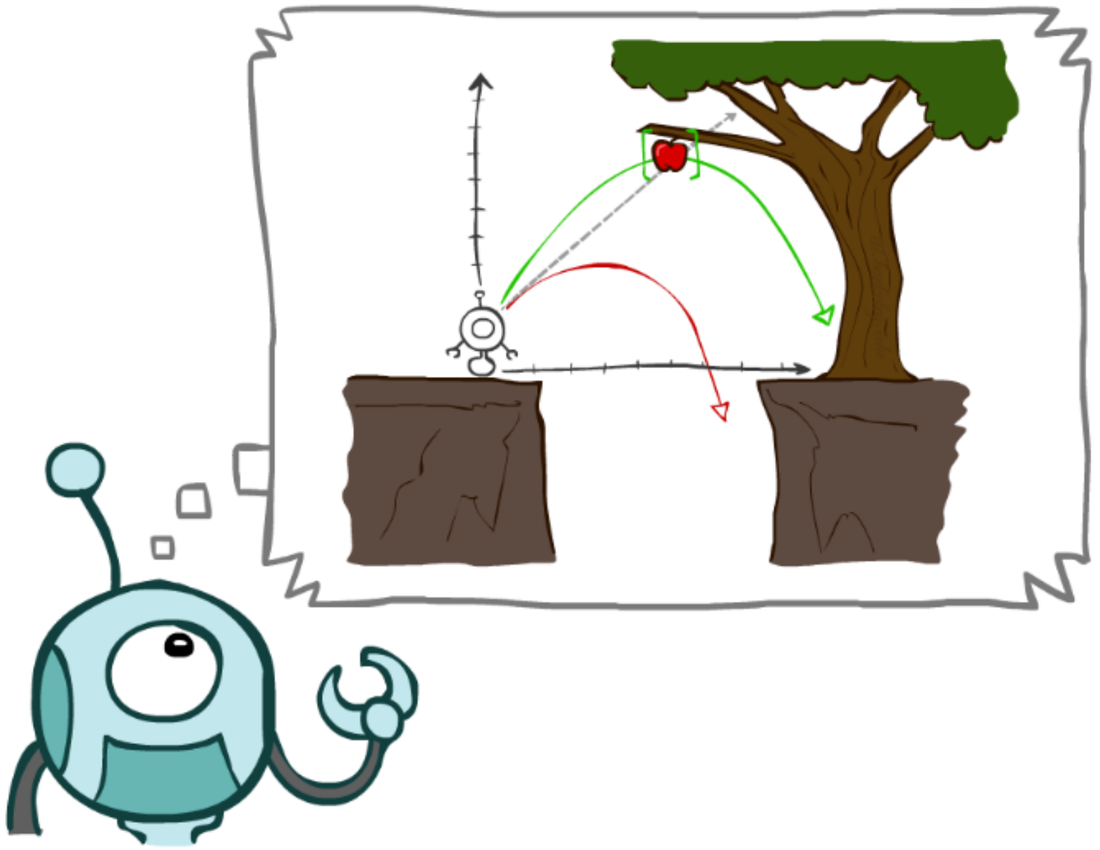
Agents that Plan Ahead

Search Problems

Uninformed Search Methods

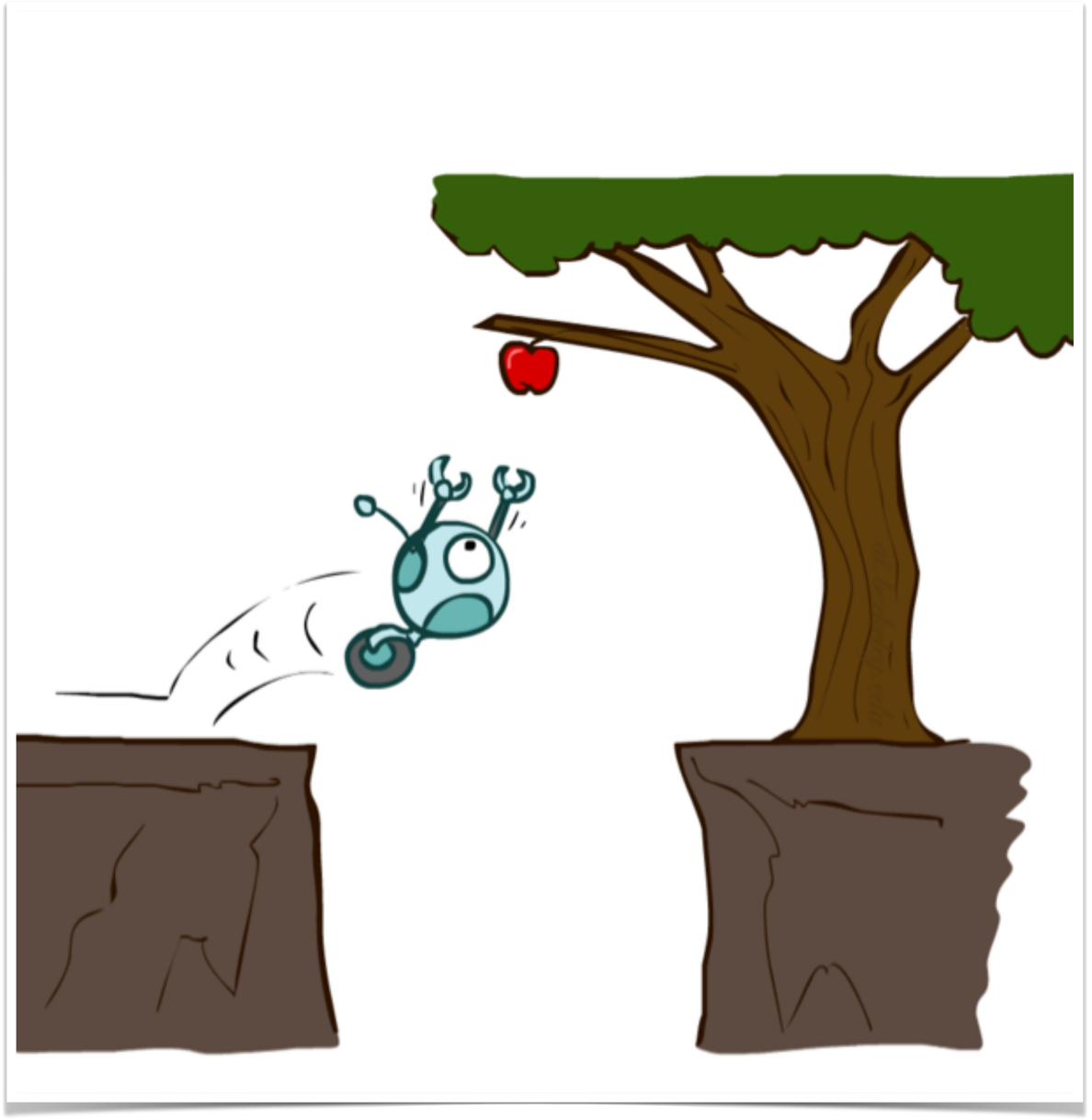
- Backtracking
- Depth-First Search
- Breadth-First Search
- Iterative deepening DFS
- Uniform-Cost Search

# Agents that Plan Ahead



# 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?*



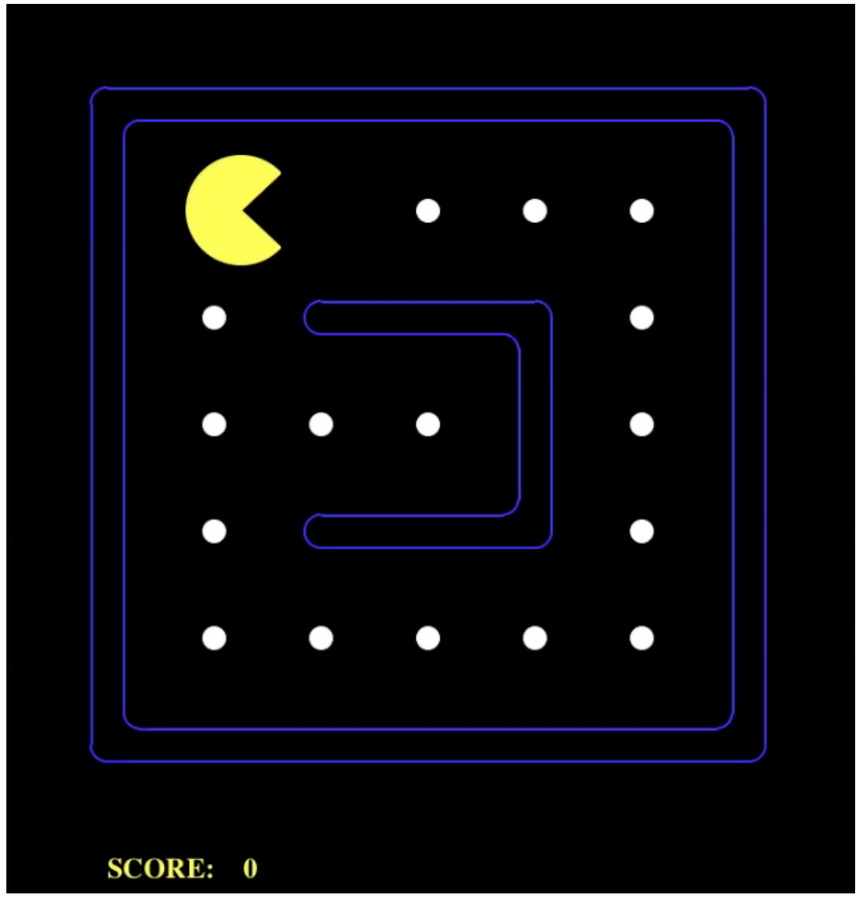
# Reflex agent

Strategy: Collect closest food item

How good is our agent ? Let’s check it out…

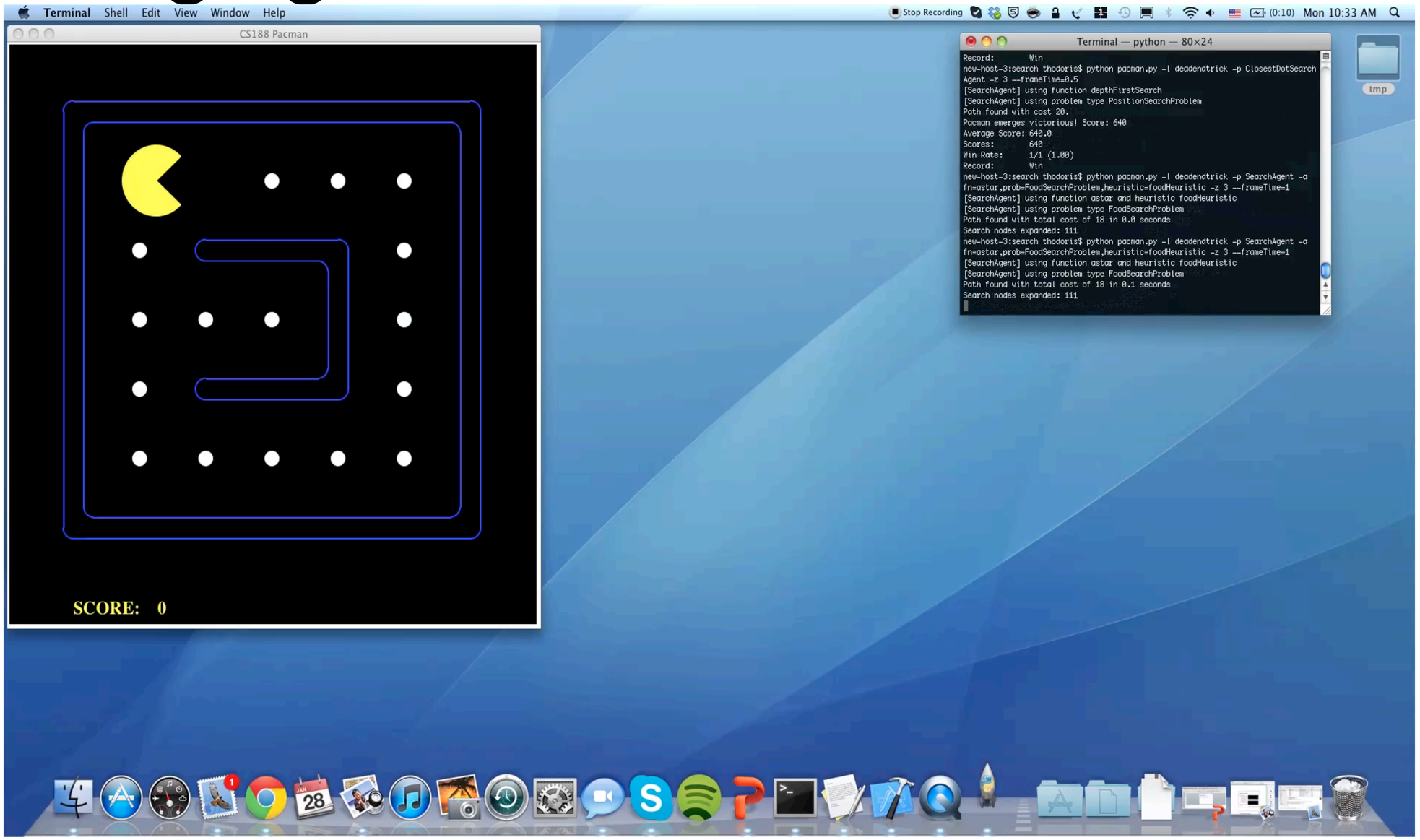
# Reflex agent

## That was easy… What about harder instances?



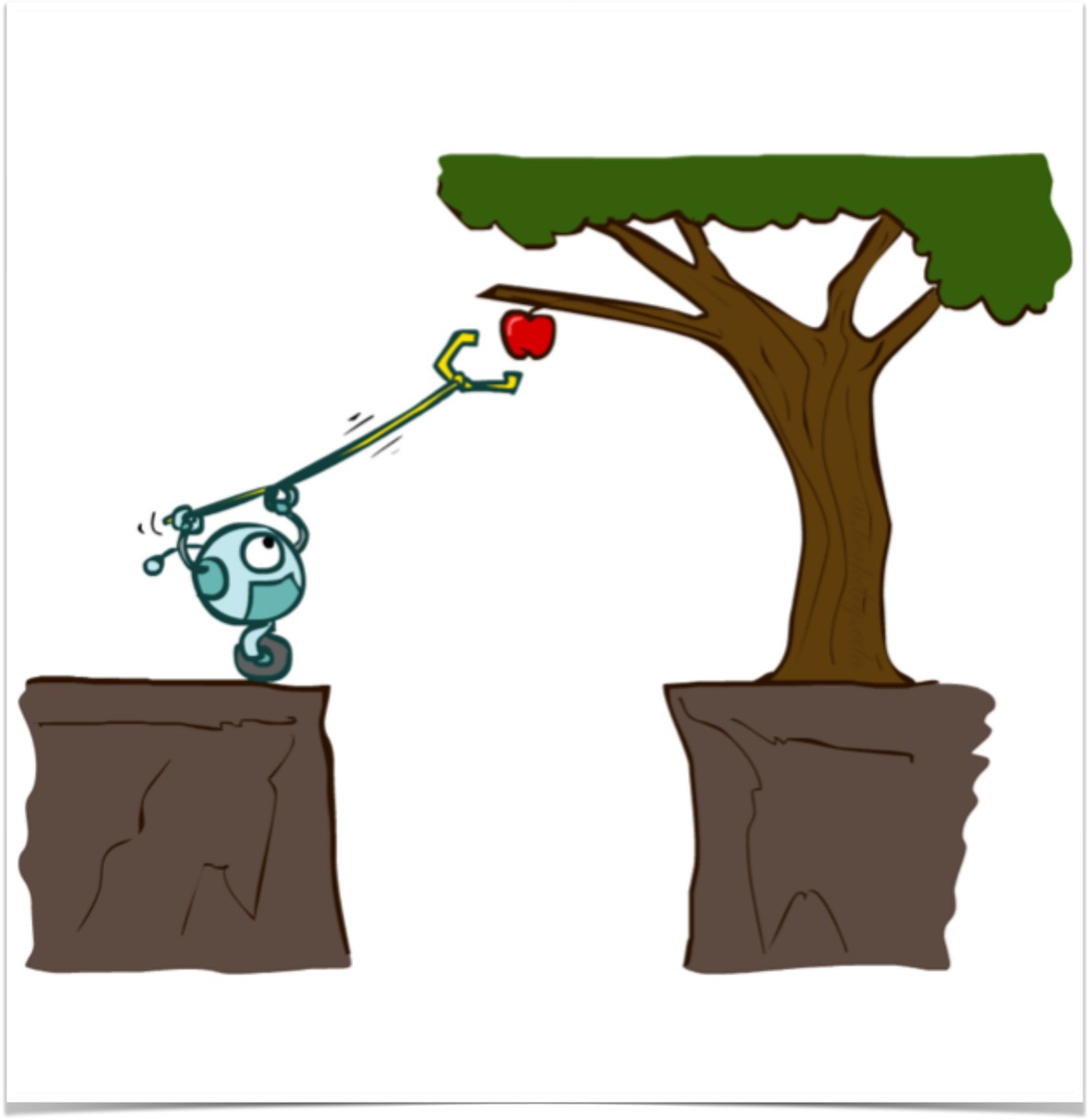
# Reflex agent

# Planning agent



# Planning Agents

- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning
- Planning vs. re-planning

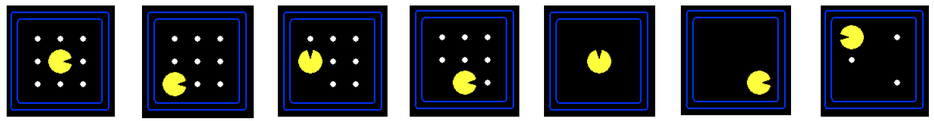


# Search Problems

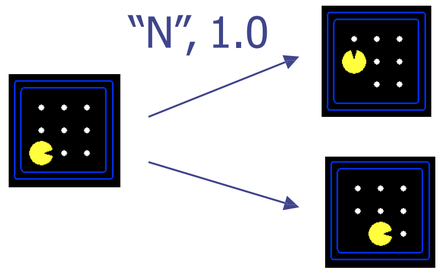
# Search Problems

A search problem consists of:

- (1) A state space



- (2) A successor function



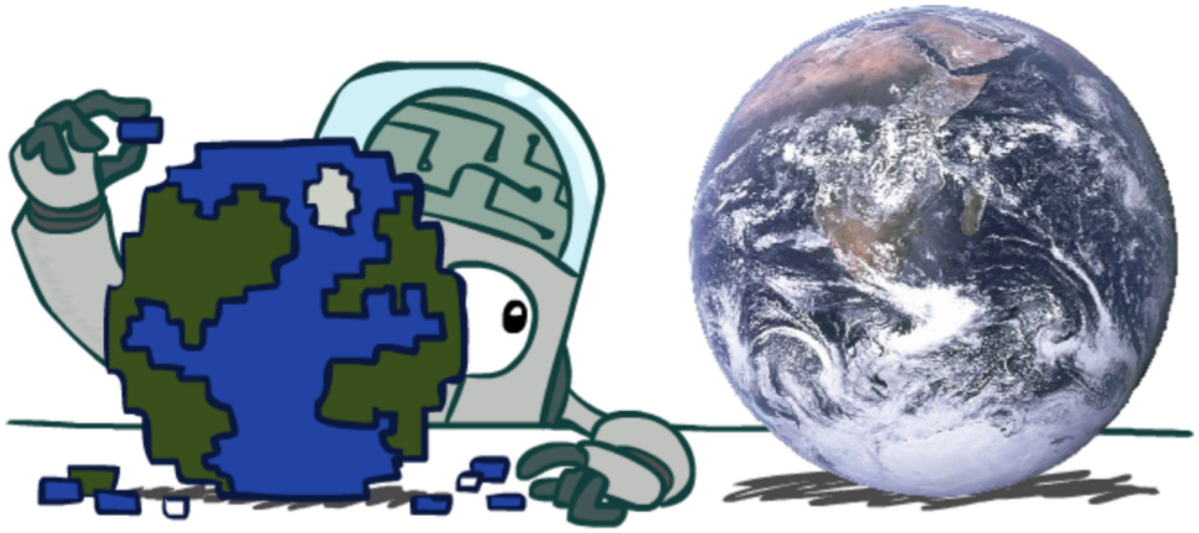

- (3) A start state and (4) a goal test

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

# Formal Definition of Search Problems

- $s_{start}$: starting state
- Actions(s): possible actions
- Cost(s, a): action cost
- Succ(s, a): successor state
- IsGoal(s): reached the goal state?

# Search Problems Are Models



# Example: Traveling in Romania

State space:

- Cities

Successor function:

- Go to adj city with cost = dist

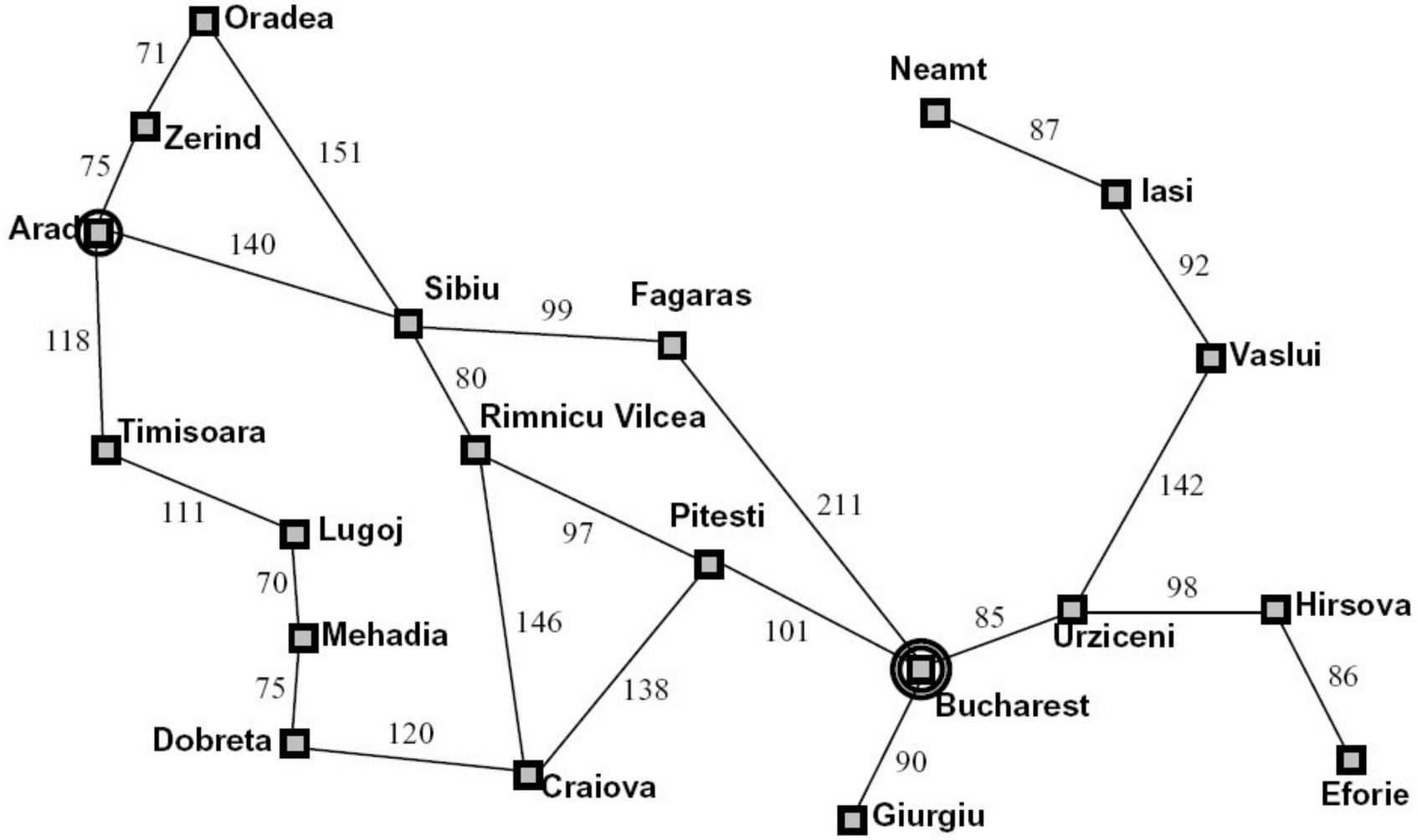
Start state:

- Arad

Goal test:

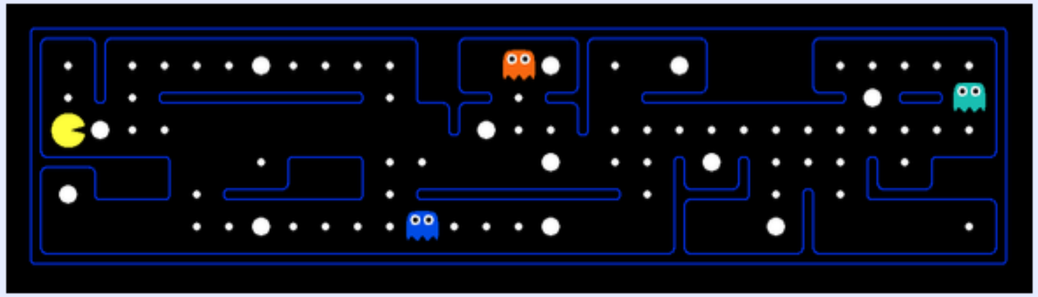
- Is state == Bucharest?

Solution?



# What's in a State Space?

The world state includes every last detail of the environment

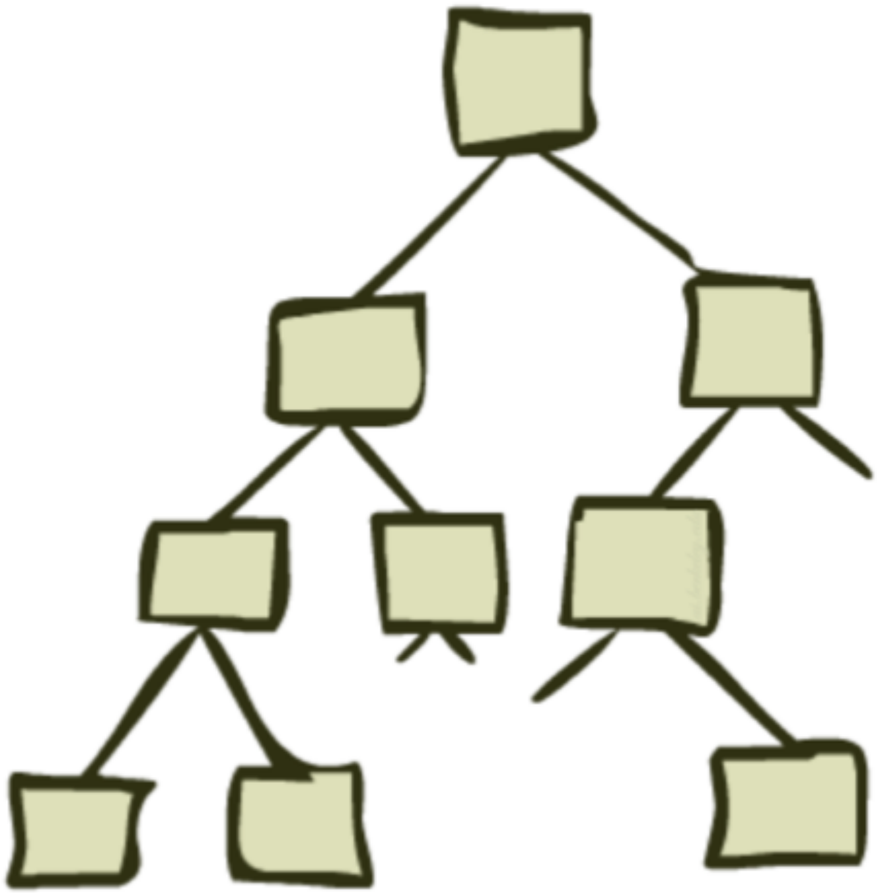


A search state keeps only the details needed for planning (abstraction)

- Problem: Finding a path
  - States: (x,y) location
  - Actions: N, S, E, W
  - Successor: update location only
  - Goal test: is (x,y)=END

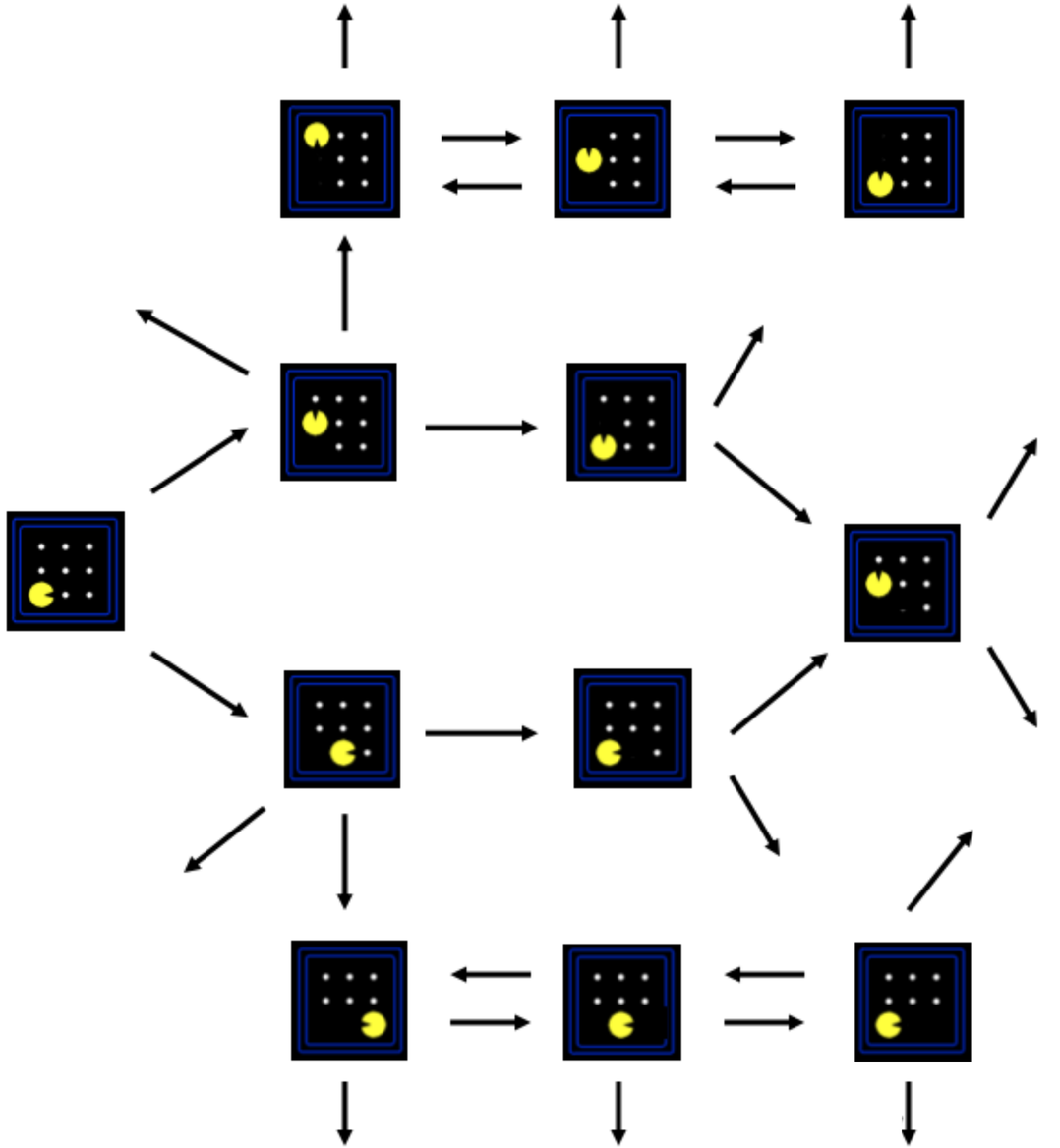
- Problem: Eat-All-Dots
  - States: {(x,y), dot Booleans}
  - Actions: N, S, E, W
  - Successor: update location and possibly a dot Boolean
  - Goal test: dots all false

# State Space Graphs and Search Trees

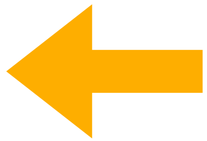


# State Space Graphs

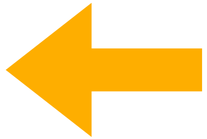
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal node(s)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph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



# Search Trees



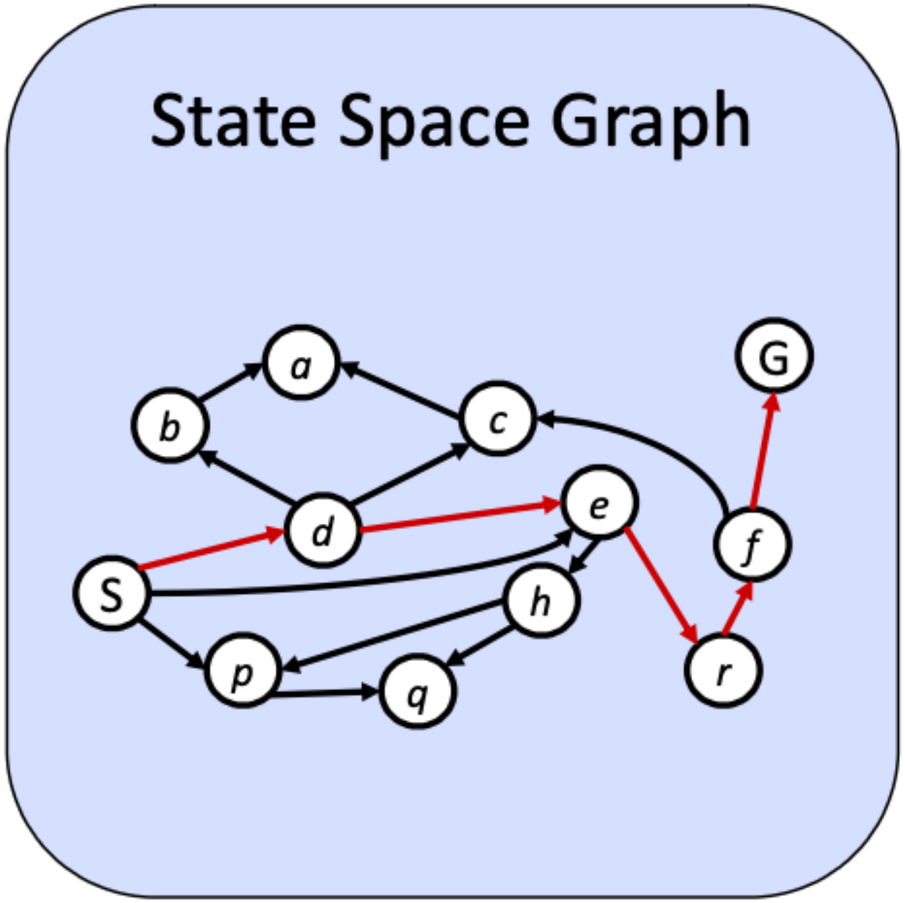
**This is now /start**



**Possible futures**

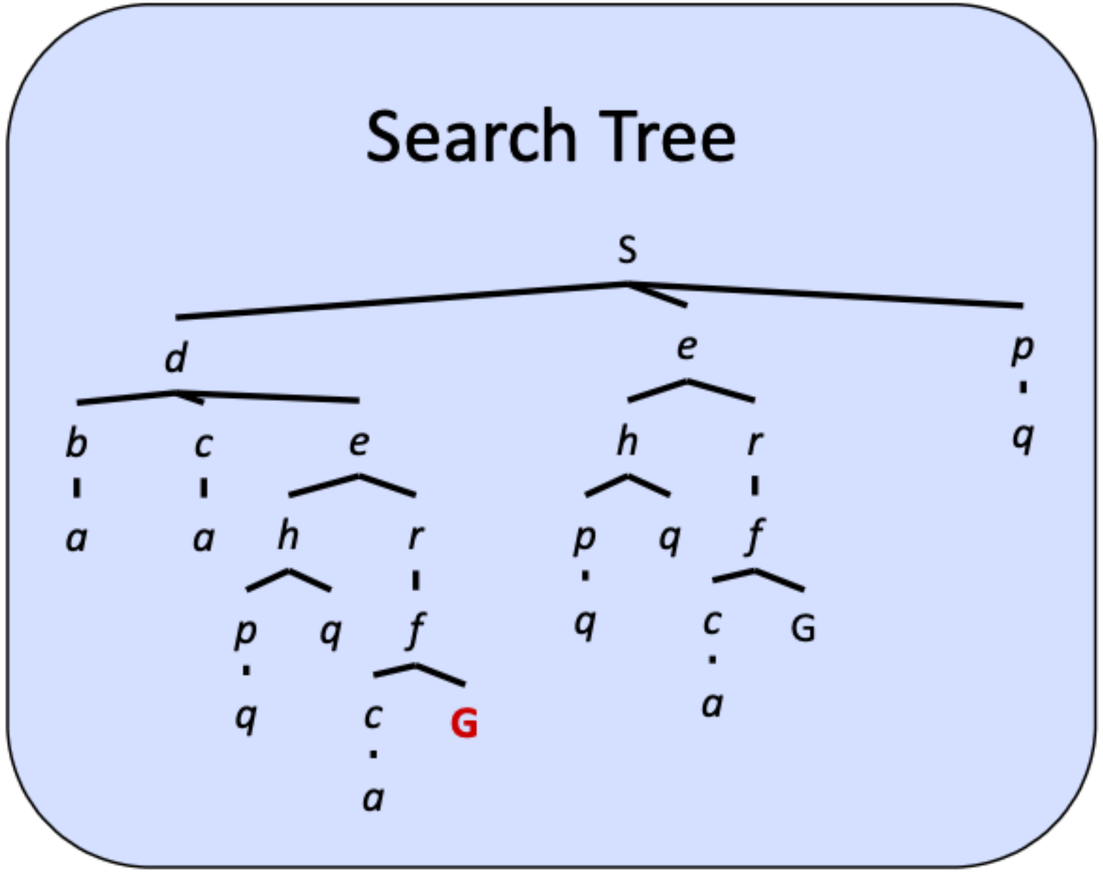
- A search tree:
  - This is a “what if” tree of plans and outcomes
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, paths correspond to PLANS to those states
  - For most problems, we can never actually build the whole tree

# State Space Graph vs. Search Trees




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

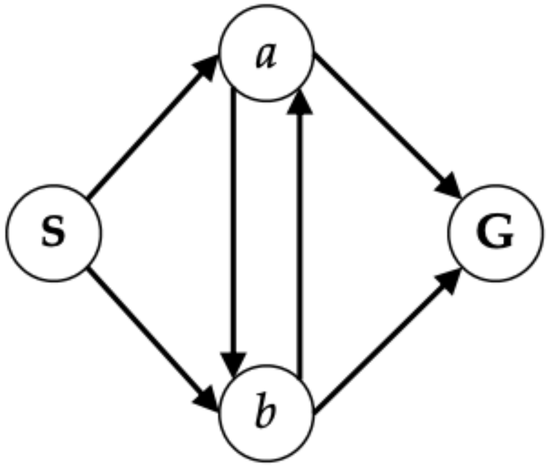
*We construct both on demand — and we construct as little as possible*



# State Space Graph vs. Search Trees

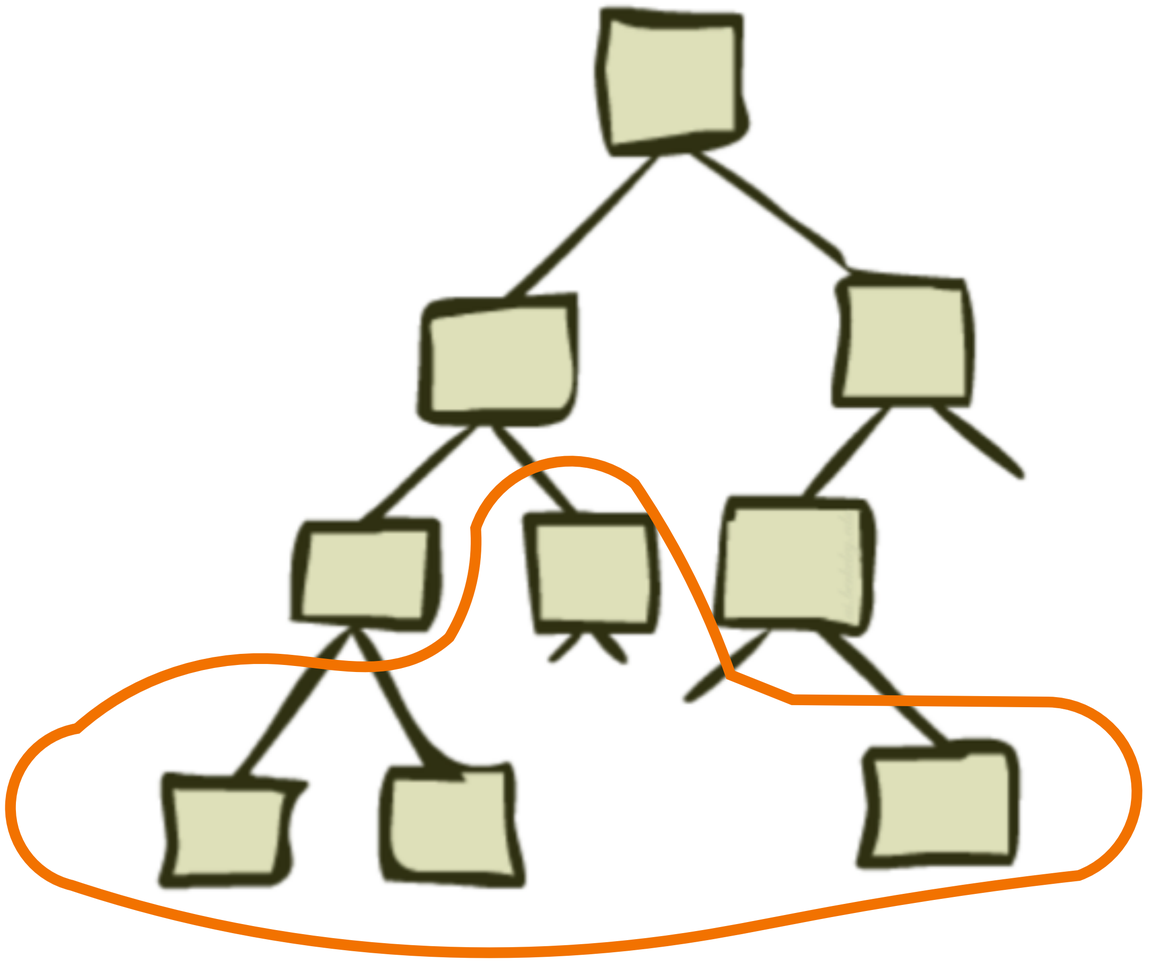
Consider this 4-state graph:

How big is its search tree (from S)?

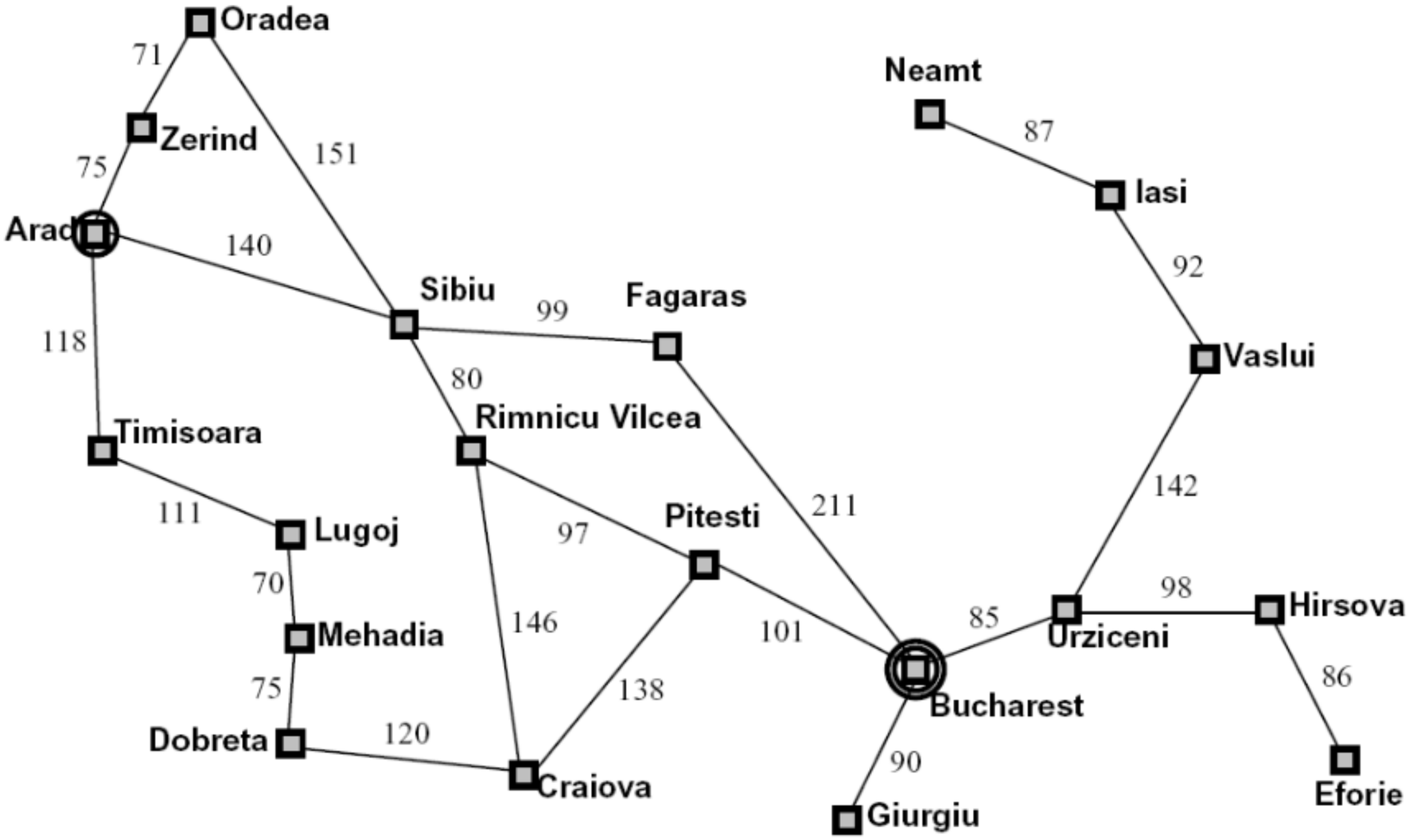



Important: Lots of repeated structure in the search tree!

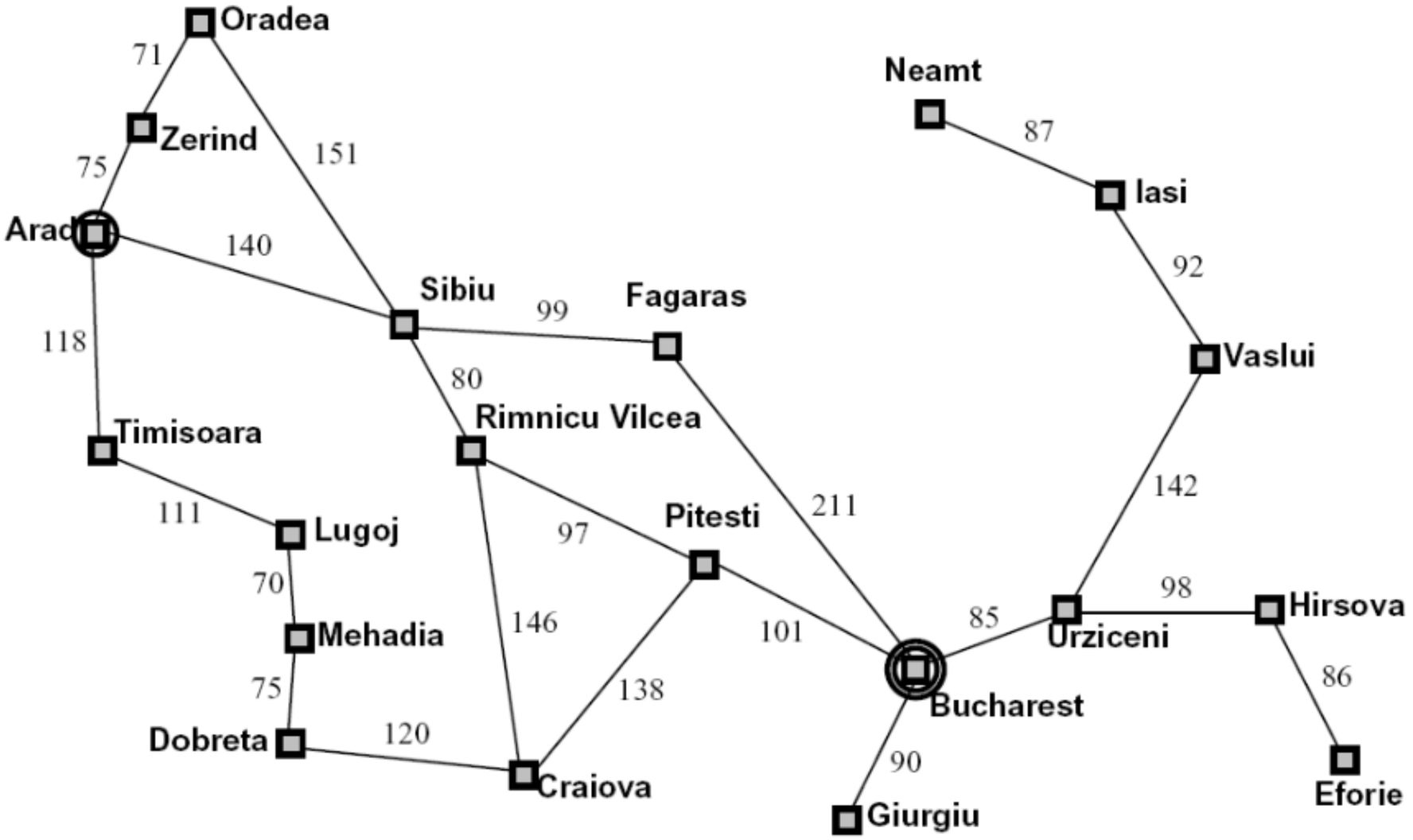
# Tree Search



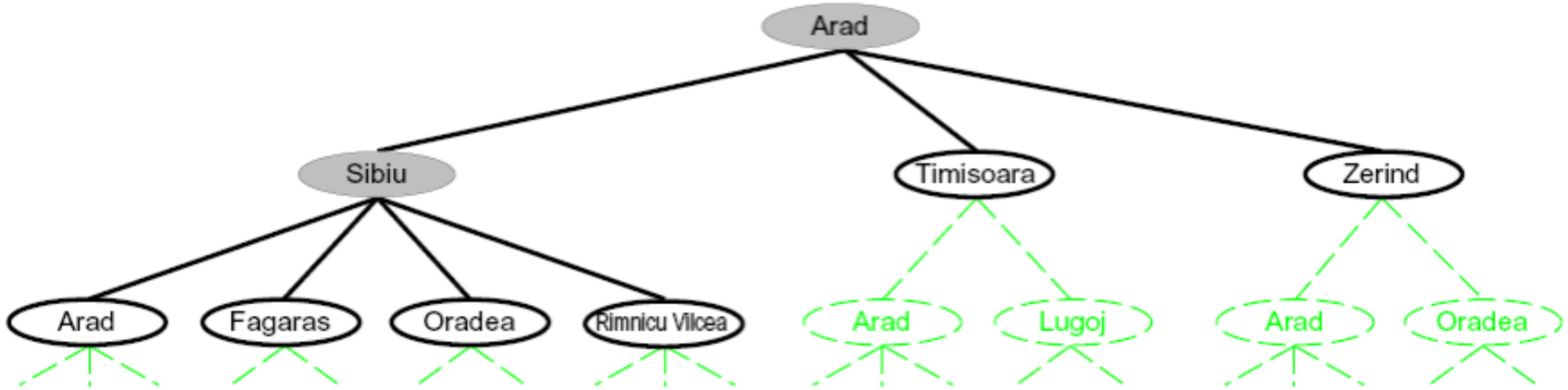
# Search Example: Romania



# CE 2: Draw/write out the search tree



# Another Search Tree




- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - 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
```

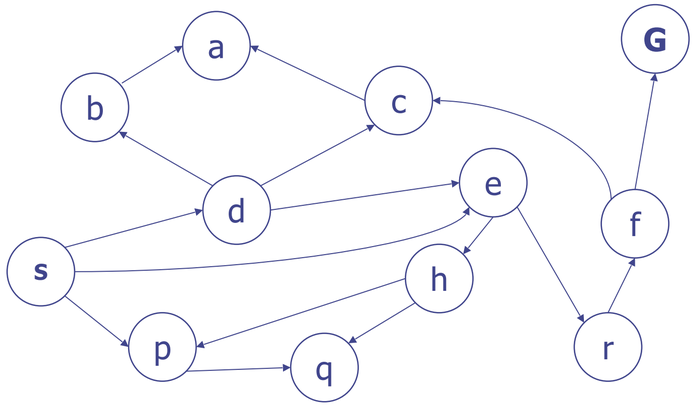
Important ideas:

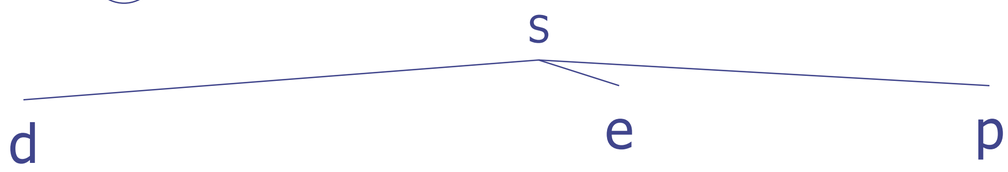
- Fringe or frontier set
- Expansion
- Exploration strategy

**Detailed pseudocode is in the book!**

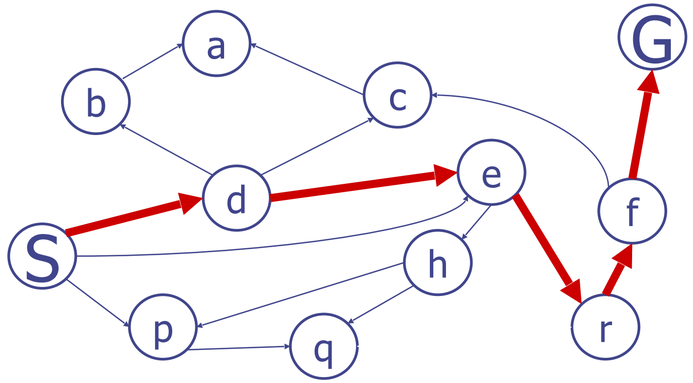
- Main question: which fringe nodes to explore?

# Example: Tree Search



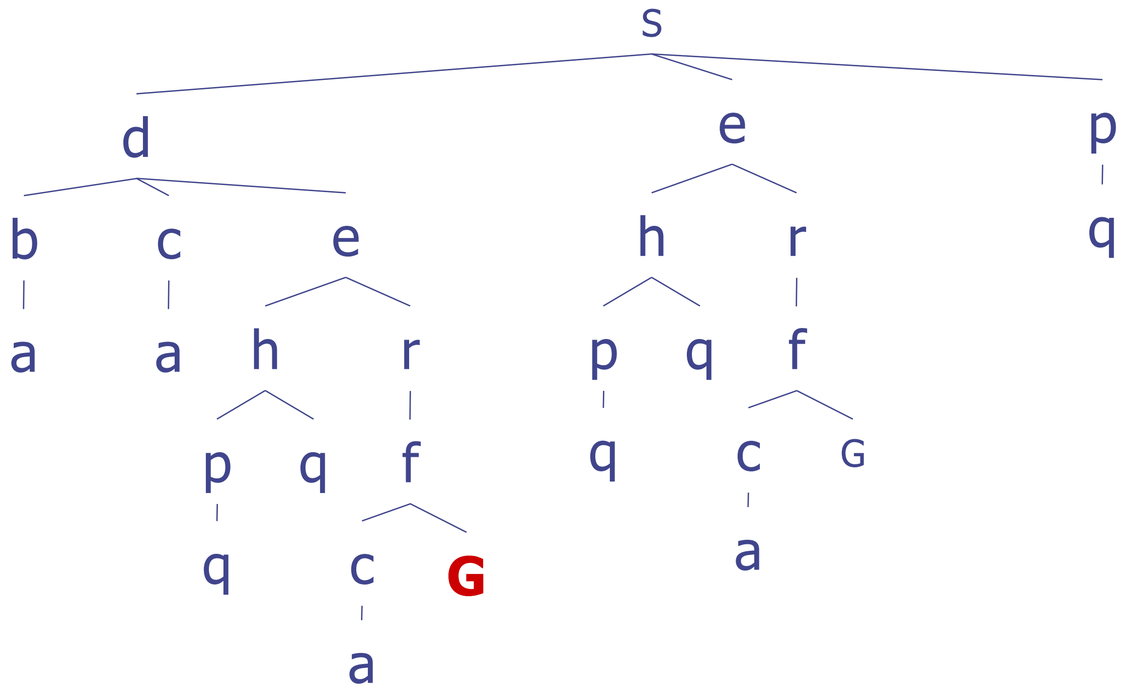



# Example: Tree Search




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

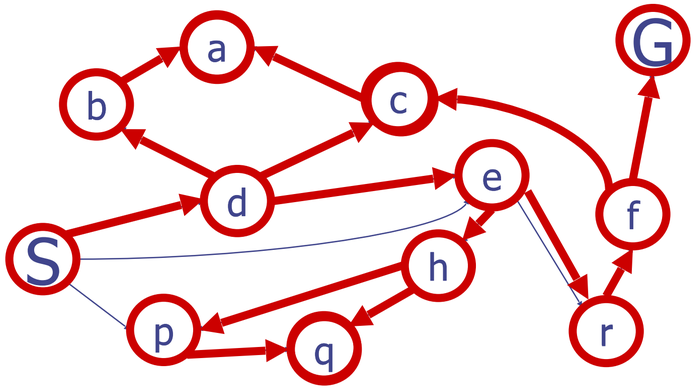
**We construct both on demand – and we construct as little as possible.**

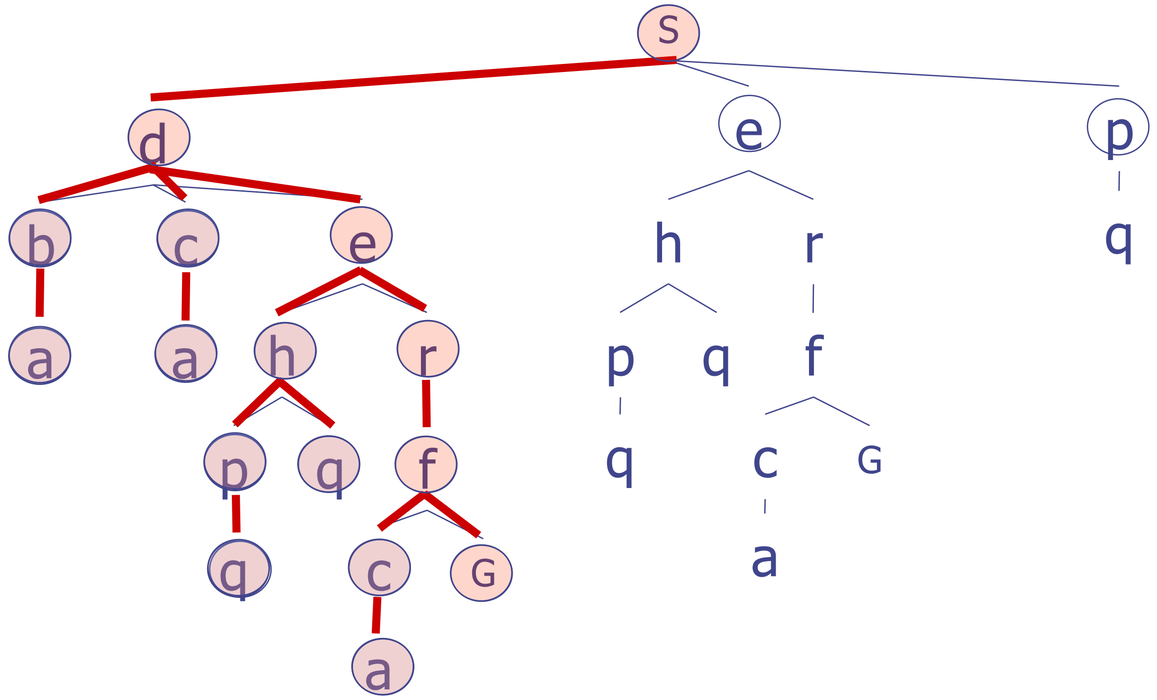


# Depth First Search

# Review: Depth First Search

**Strategy: expand deepest node first**
**Implementation: Fringe is a LIFO stack**



# 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?

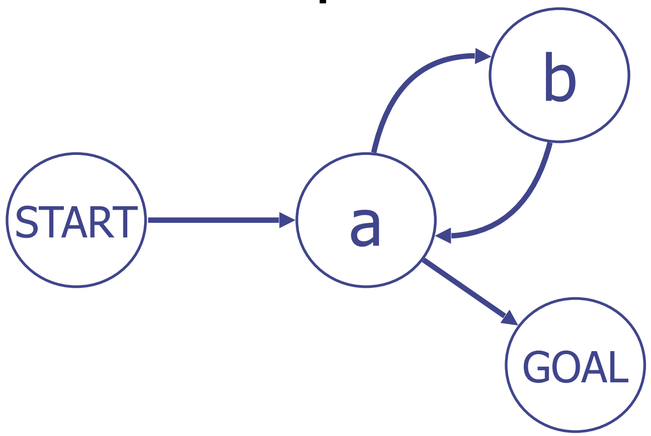
Variables:

| | |
|---|---|
| $n$ | Number of states in the problem |
| $b$ | The average branching factor $B$<br>(the average number of successors) |
| $C^*$ | Cost of least cost solution |
| $s$ | Depth of the shallowest solution |
| $m$ | Max depth of the search tree |

# DFS

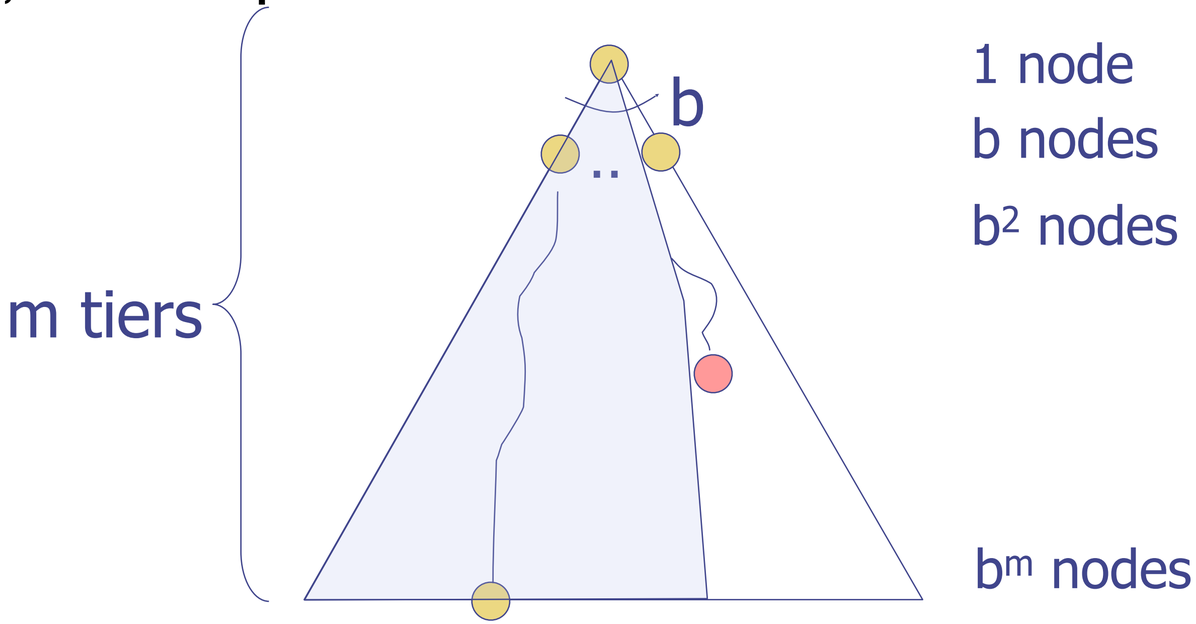
| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | Depth First Search | N | N | Infinite | Infinite |

- Infinite paths make DFS incomplete…
- How can we fix this?



# DFS

- **With cycle checking, DFS is complete.**




| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^{m+1})$ | $O(bm)$ |

- **When is DFS optimal?**

# DFS Assumption

**What assumption about the search problem should we have to be able to use DFS to optimally find the goal state?**

# Backtracking Search

**Algorithm: backtracking search**

def backtrackingSearch($s$, path):
    If IsEnd($s$): update minimum cost path
    For each action $a \in$ Actions($s$):
        Extend path with Succ($s, a$) and Cost($s, a$)
        Call backtrackingSearch(Succ($s, a$), path)
    Return minimum cost path

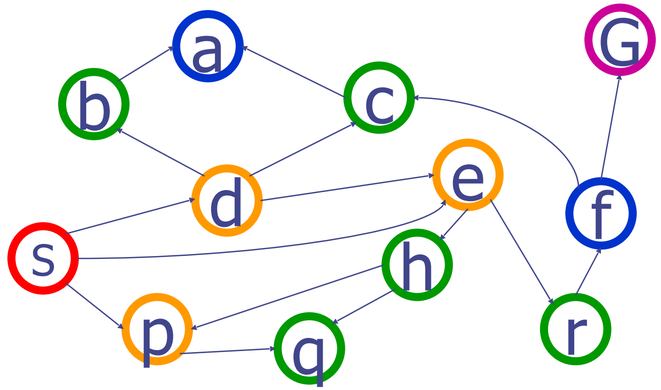
# Backtracking Search Performance

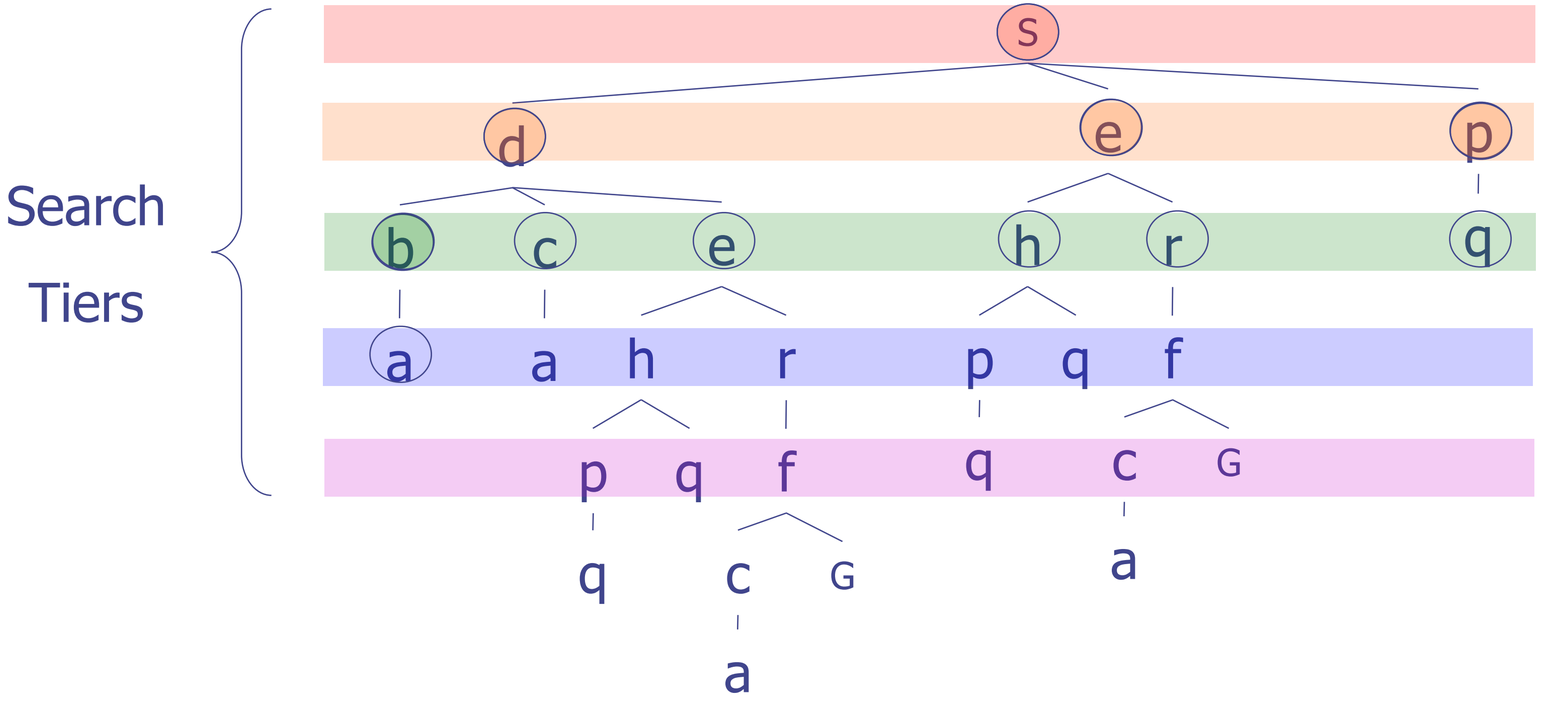
- If ‘b’ actions per state (branching factor) and maximum depth of the tree is ‘m’:
  - Memory complexity: O(bm)
    - In each level at most ‘b’ nodes will be added to the frontier set.
  - Time complexity: $O(b^m)$
    - In worst-case scenario, all nodes in the tree will be evaluated once.

# Breadth-First Search

# Breadth First Search

**Strategy: expand shallowest node first**
**Implementation: Fringe is a FIFO queue**



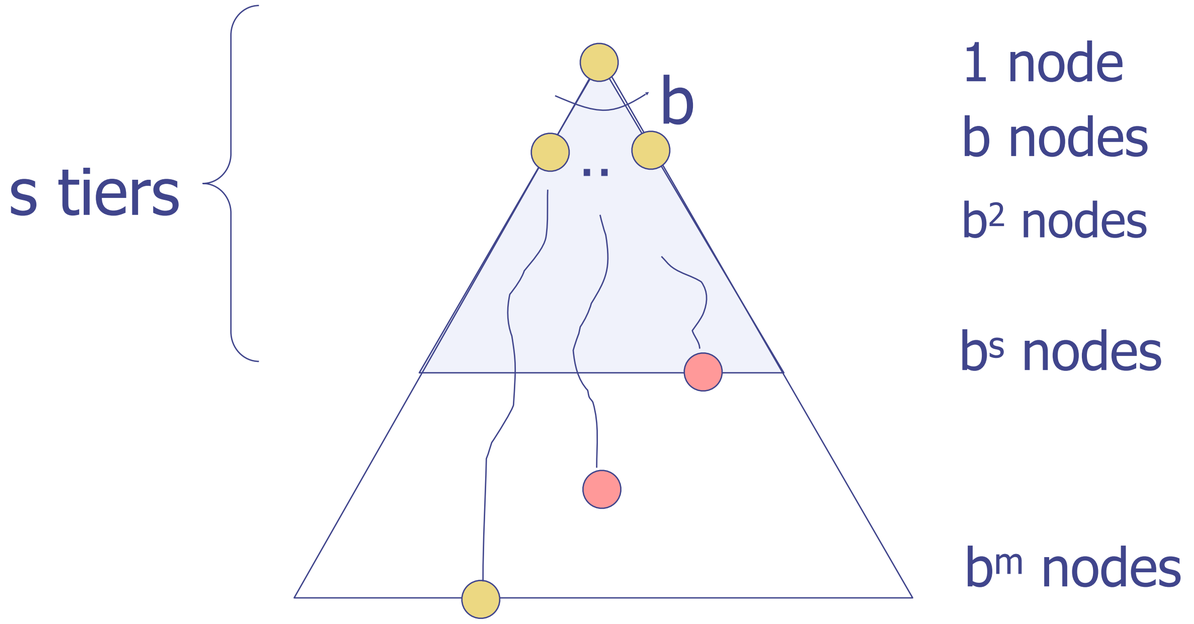



# BFS Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

# BFS

| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^{m+1})$ | $O(bm)$ |
| BFS | | Y | N* | $O(b^{s+1})$ | $O(b^s)$ |




- When is BFS optimal?

# BFS Assumptions?

# Recap and Upcoming

Today

- Solving problems by searching
  - Uninformed search strategies
    - BFS, DFS, ID-DFS, UCS

Next Week

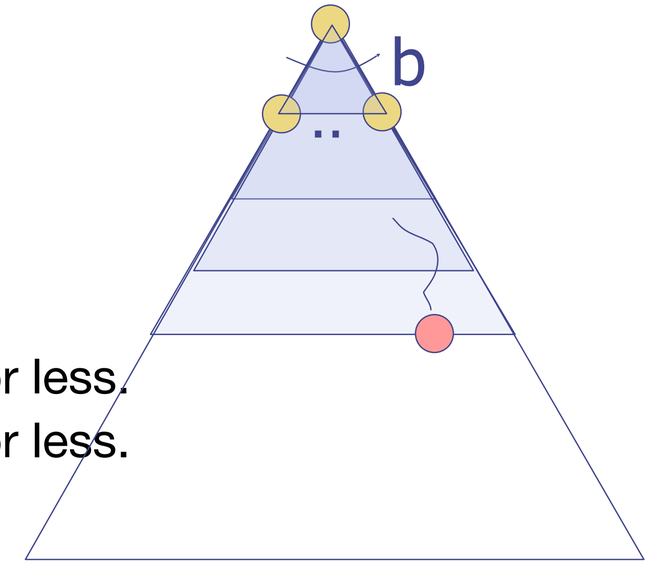
- Solving problems by searching (cont.)
  - Informed search strategies
  - Heuristics functions
- Search in complex environments
  - Hill climbing, simulated annealing, local beam search, evolutionary algorithm.

# Iterative Deepening

Iterative deepening uses DFS as a subroutine:

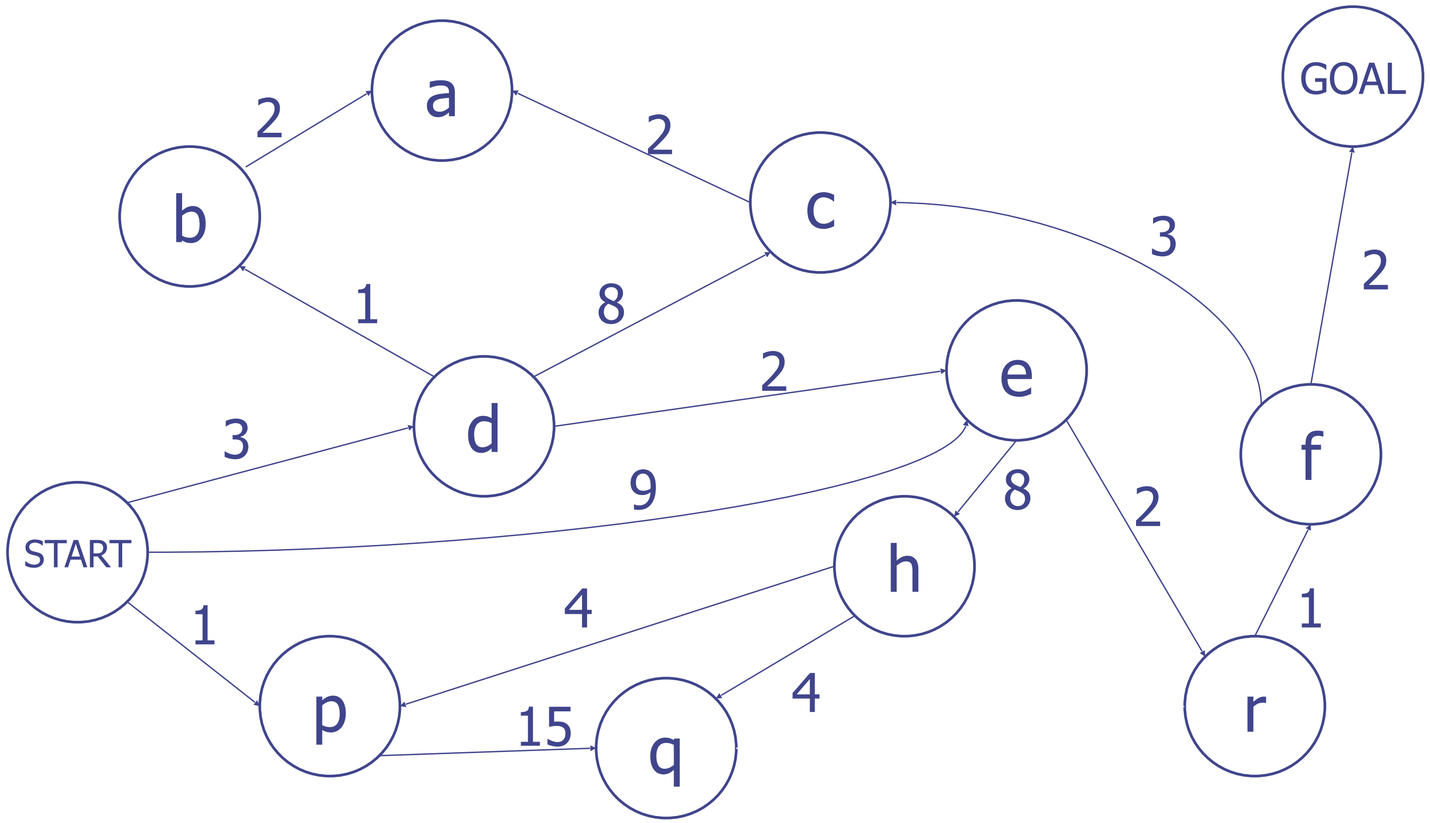
1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.




| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^{m+1})$ | $O(bm)$ |
| BFS | | Y | N* | $O(b^{s+1})$ | $O(b^s)$ |
| ID | | Y | N* | $O(b^{s+1})$ | $O(bs)$ |

# Costs on Actions

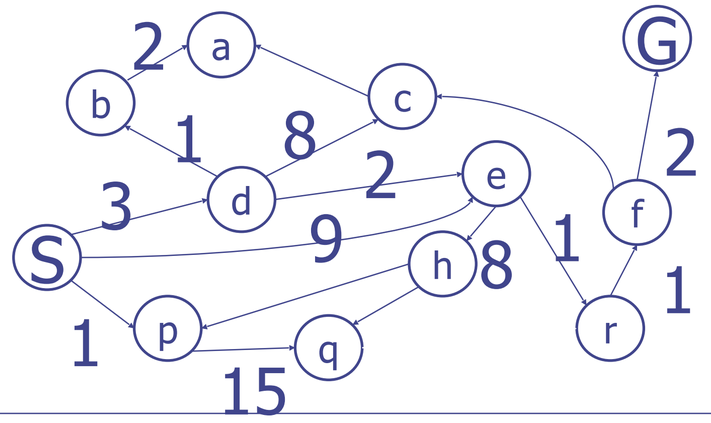


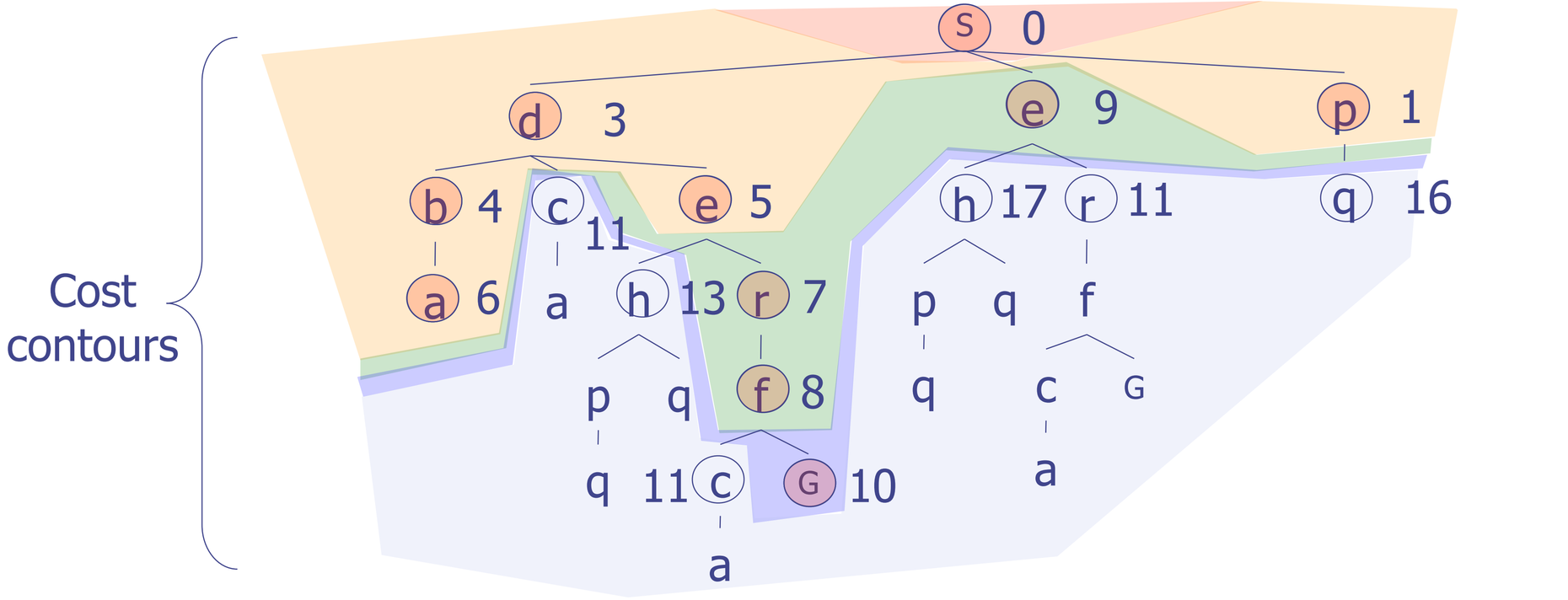

Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

We will quickly cover an algorithm which does find the least-cost path.

# Uniform Cost Search

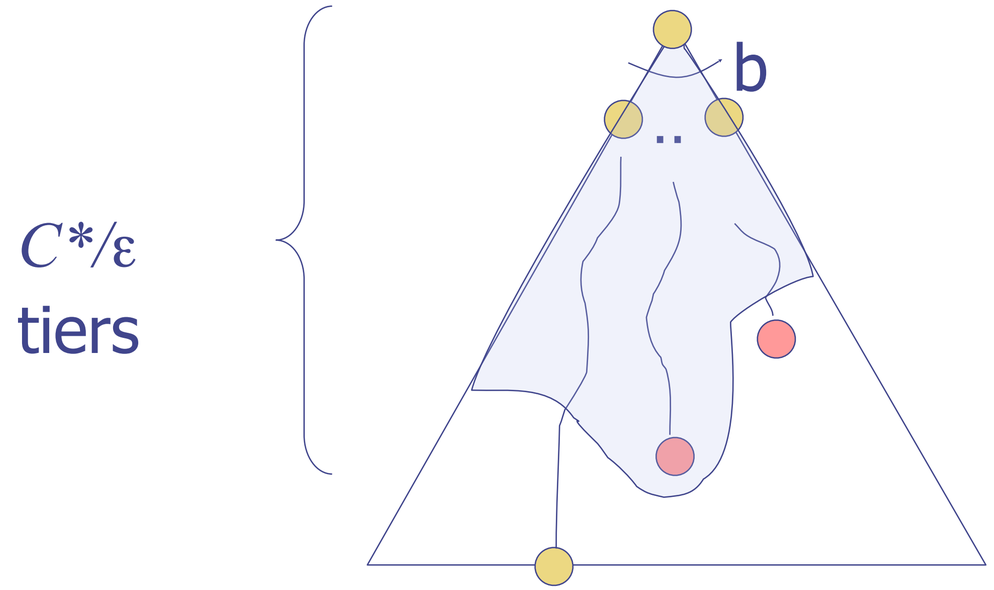
**Expand cheapest node first:**
**Fringe is a priority queue**



# Uniform Cost Search

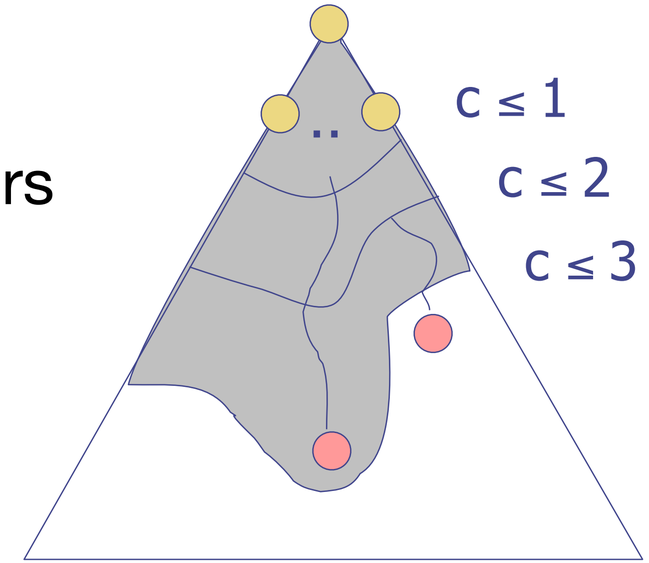
| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^{m+1})$ | $O(bm)$ |
| BFS | | Y | N | $O(b^{s+1})$ | $O(b^{s})$ |
| UCS | | Y* | Y | $O(b^{C^*/\varepsilon})$ | $O(b^{C^*/\varepsilon})$ |




*** UCS can fail if actions can get arbitrarily cheap**

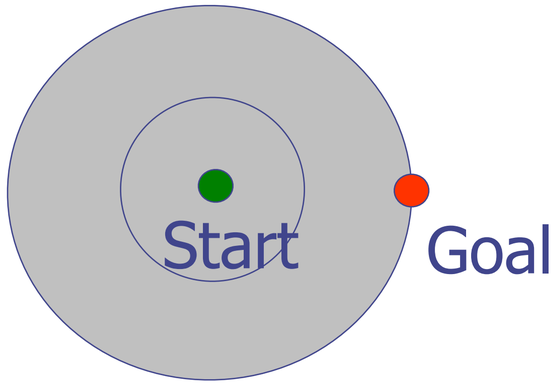
# Uniform Cost Issues

- Remember: explores increasing cost contours




- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location



# Recap and Upcoming

Today

- Solving problems by searching
  - Uninformed search strategies
    - BFS, DFS, ID-DFS, UCS

Net Week

- Solving problems by searching (cont.)
  - Informed search strategies
  - Heuristics functions
- Search in complex environments
  - Hill climbing, simulated annealing, local beam search, evolutionary algorithm.