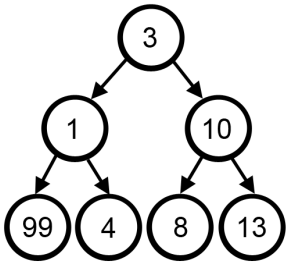
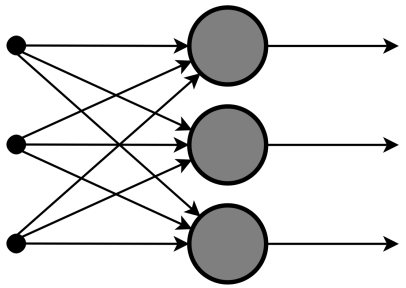
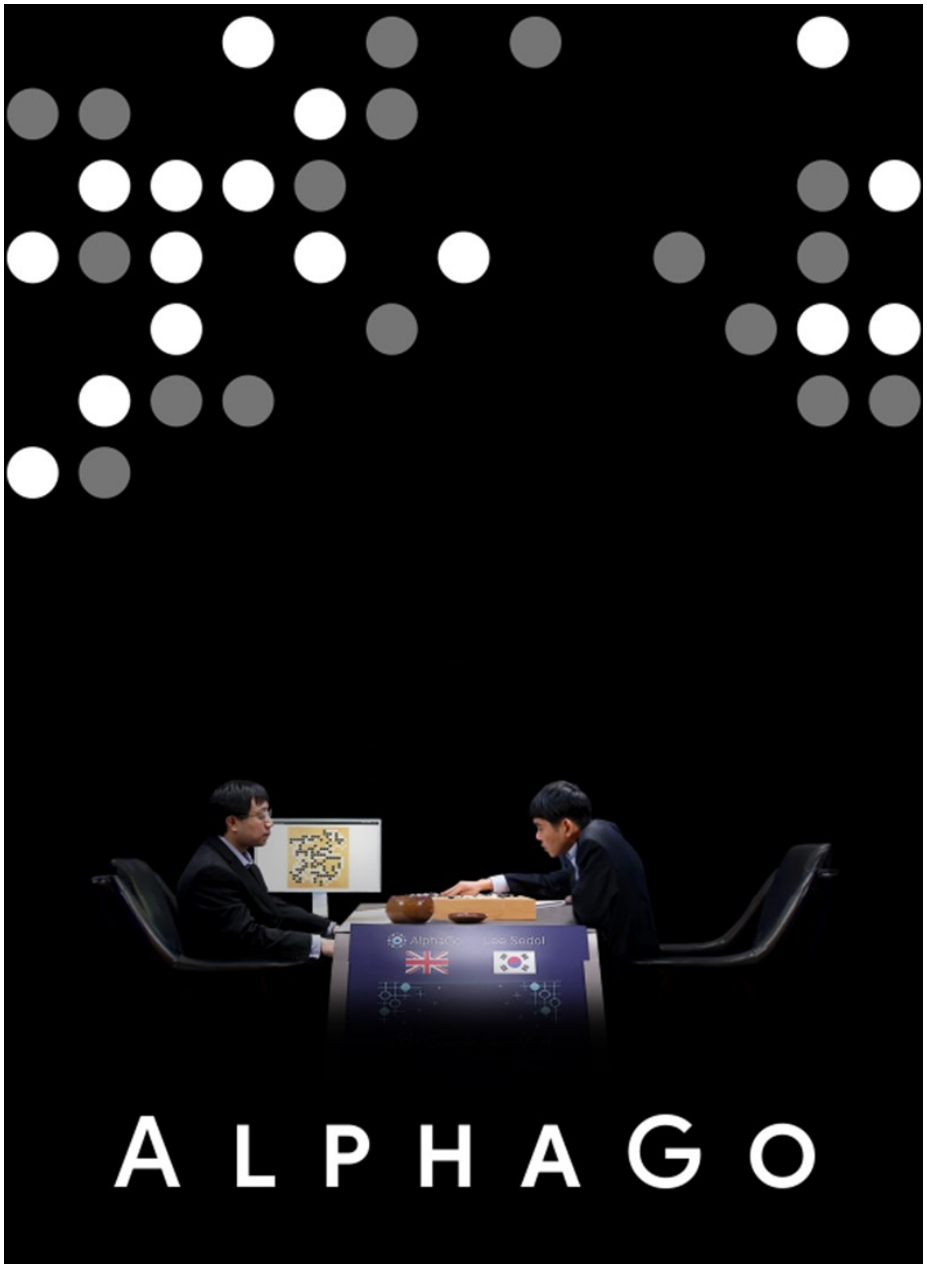
# CSE 40171:
# Artificial Intelligence



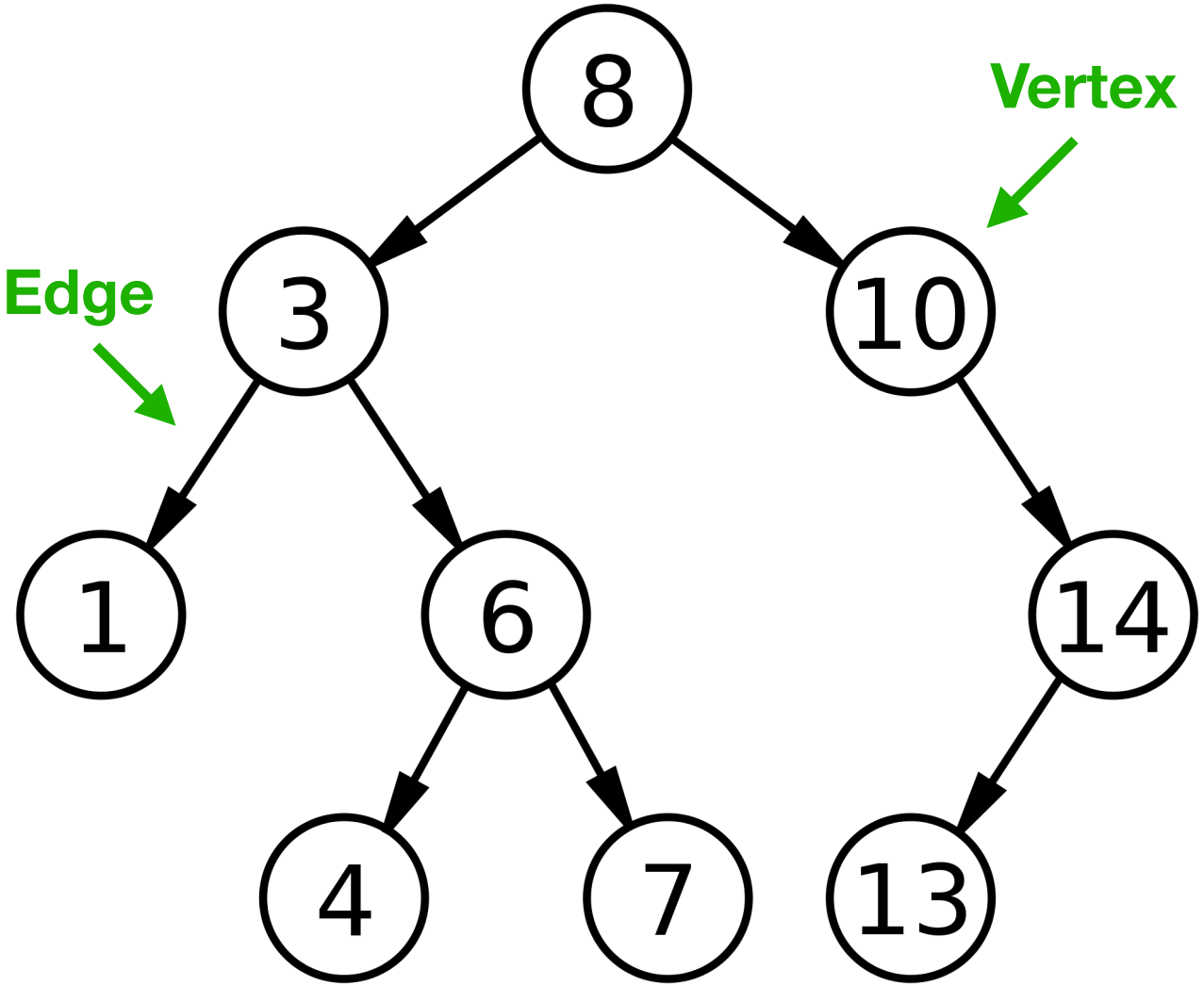
Uninformed Search: Search Trees



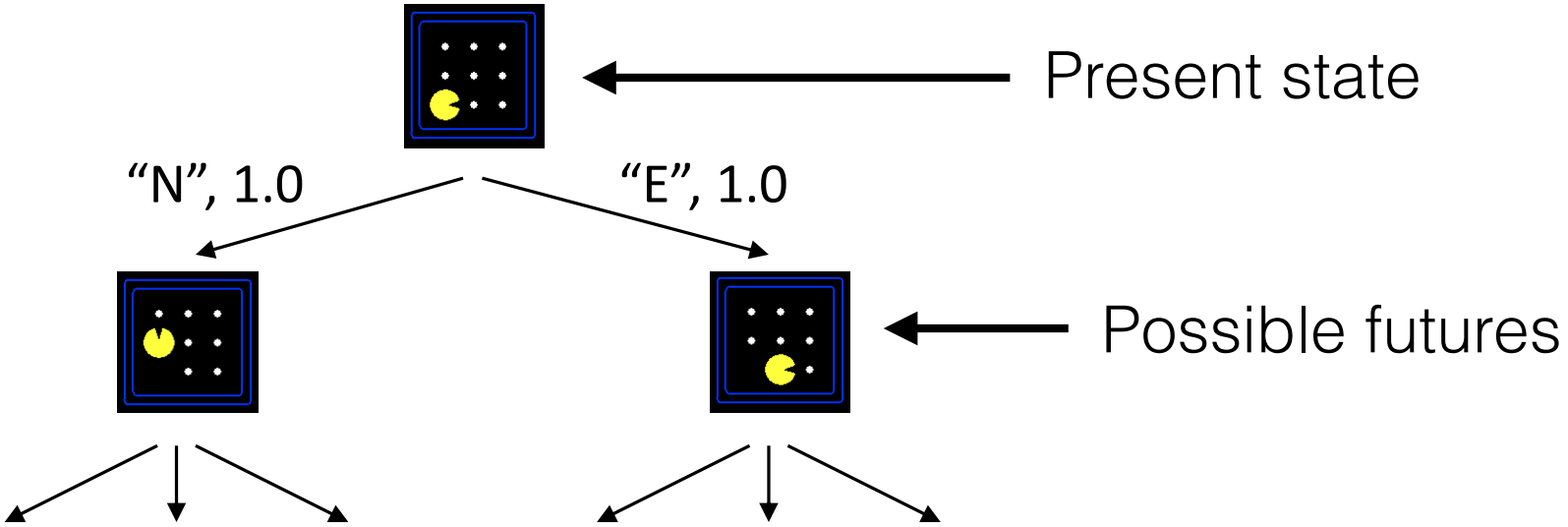

**Film Response Activity**
**Due Tonight**

Homework #2 has been released
It is due at 11:59PM on 9/30

# Search Trees



# Search Trees

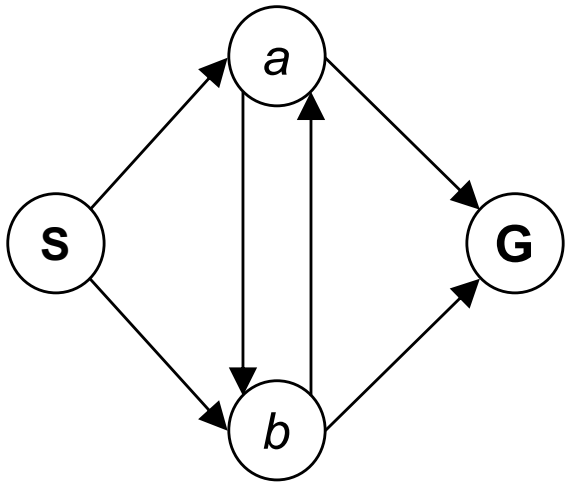



- A “what if” tree of plans and their outcomes
- The start state is the root vertex
- Children correspond to successors
- Vertices show states, but correspond to plans that achieve them
- For most problems, we can never actually build the whole tree

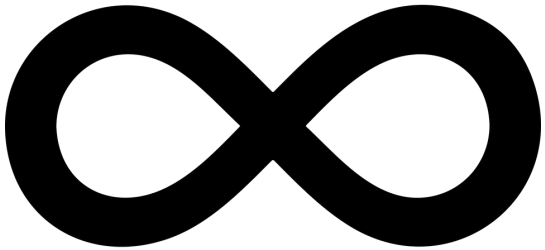
Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188

# From Graphs to Trees

Consider this 4-state graph:




How large is its search tree?
(starting from S)



Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188

# From Graphs to Trees

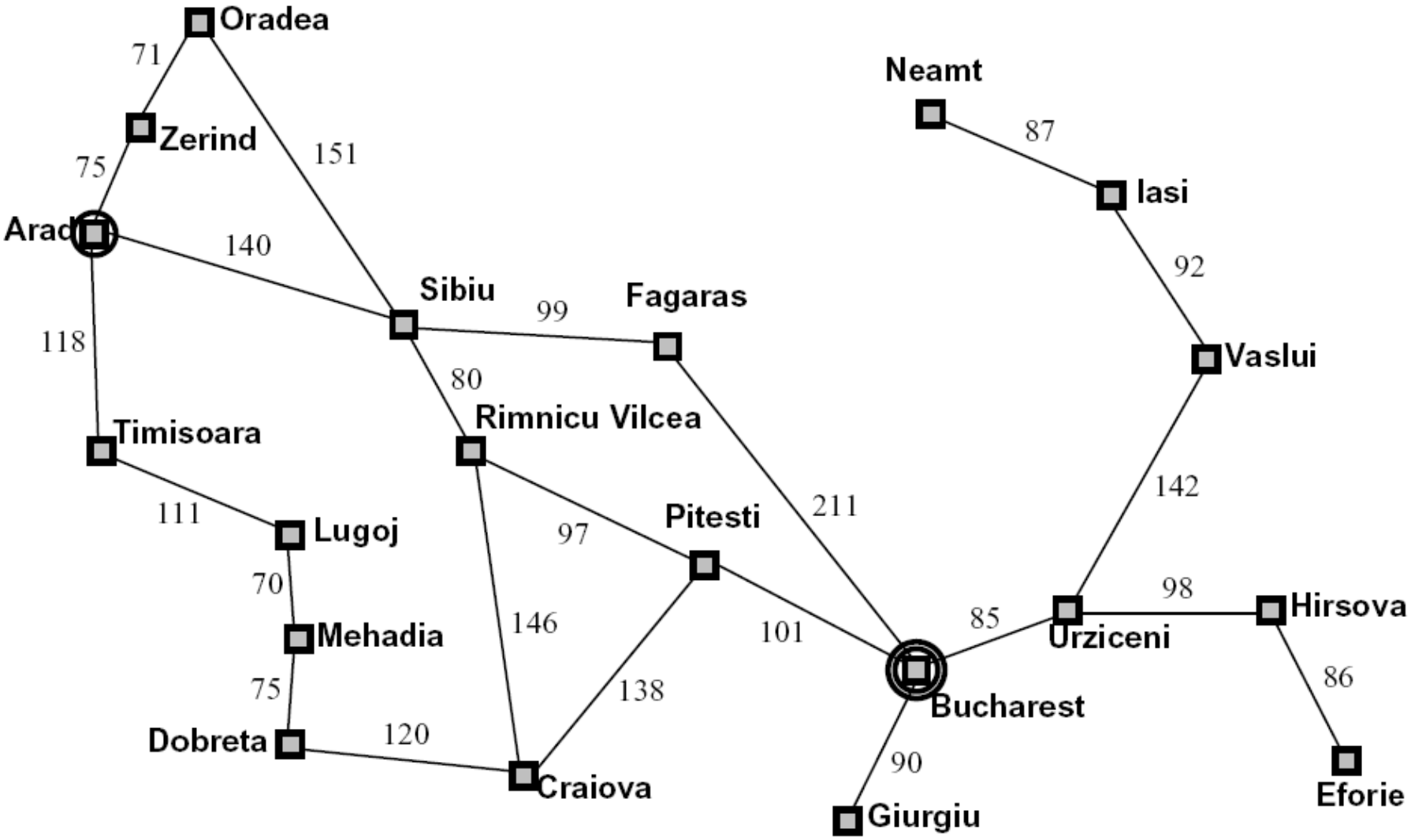
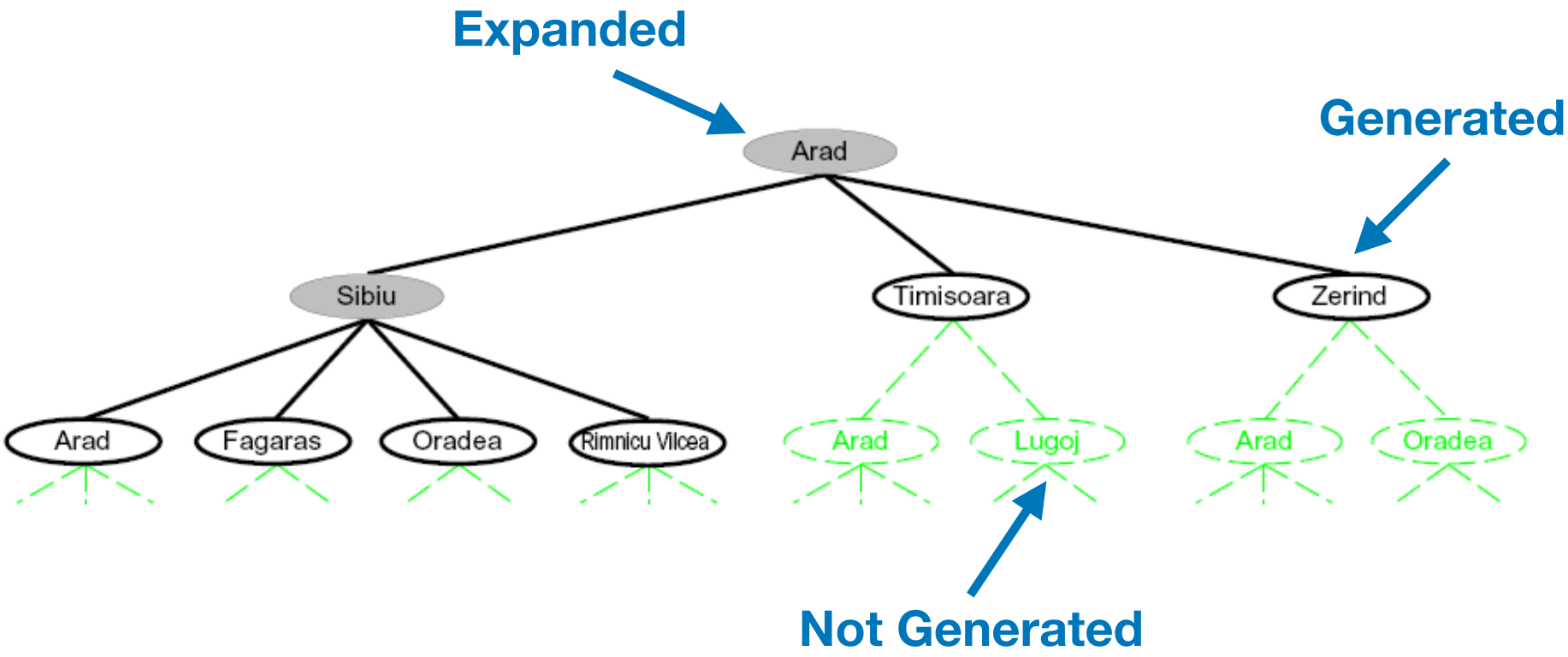



Image Credit: Russel and Norvig

# Partial search tree for finding a route between two cities




**Frontier: vertices with bold outlines**

Image Credit: Russel and Norvig

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

initialize the frontier using the initial state of *problem*

**loop do**

**if** the frontier is empty **then return** failure

choose a leaf vertex and remove it from the frontier

**if** the vertex contains a goal state **then return** the corresponding solution

expand the chosen vertex, adding the resulting vertices to the frontier

What is the problem with TREE-SEARCH?

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

initialize the frontier using the initial state of *problem*

***initialize the explored set to be empty***

**loop do**

**if** the frontier is empty **then return** failure

choose a leaf vertex and remote it from the frontier

**if** the vertex contains a goal state **then return** the corresponding solution

***add the vertex to the explored set***

expand the chosen vertex, adding the resulting vertices to the frontier

***only if not in the frontier or explored set***

# Each vertex has a structure that contains four components

$n$.STATE: the state in the state space to which the vertex corresponds

$n$.PARENT: the vertex in the search tree that generated this vertex

$n$.ACTION: the action that was applied to the parent to generate the vertex

$n$.PATH-COST: the cost of the path from the initial state to the vertex

# Keeping track of vertices

The right data structure for this is a **queue**

EMPTY?(*queue*): returns true only if there are no more elements in the queue

POP(*queue*): removes the first element of the queue and returns it

INSERT(*element*, *queue*): inserts an element and returns the resulting queue

# Algorithm Performance Evaluation

**Completeness:** Is the algorithm guaranteed to find a solution when there is one?

**Optimality:** Does the strategy find the optimal solution (lowest path cost)?

**Time complexity:** How long does it take to find a solution?

**Space complexity:** How much memory is needed to perform the search?

# Breadth-first search

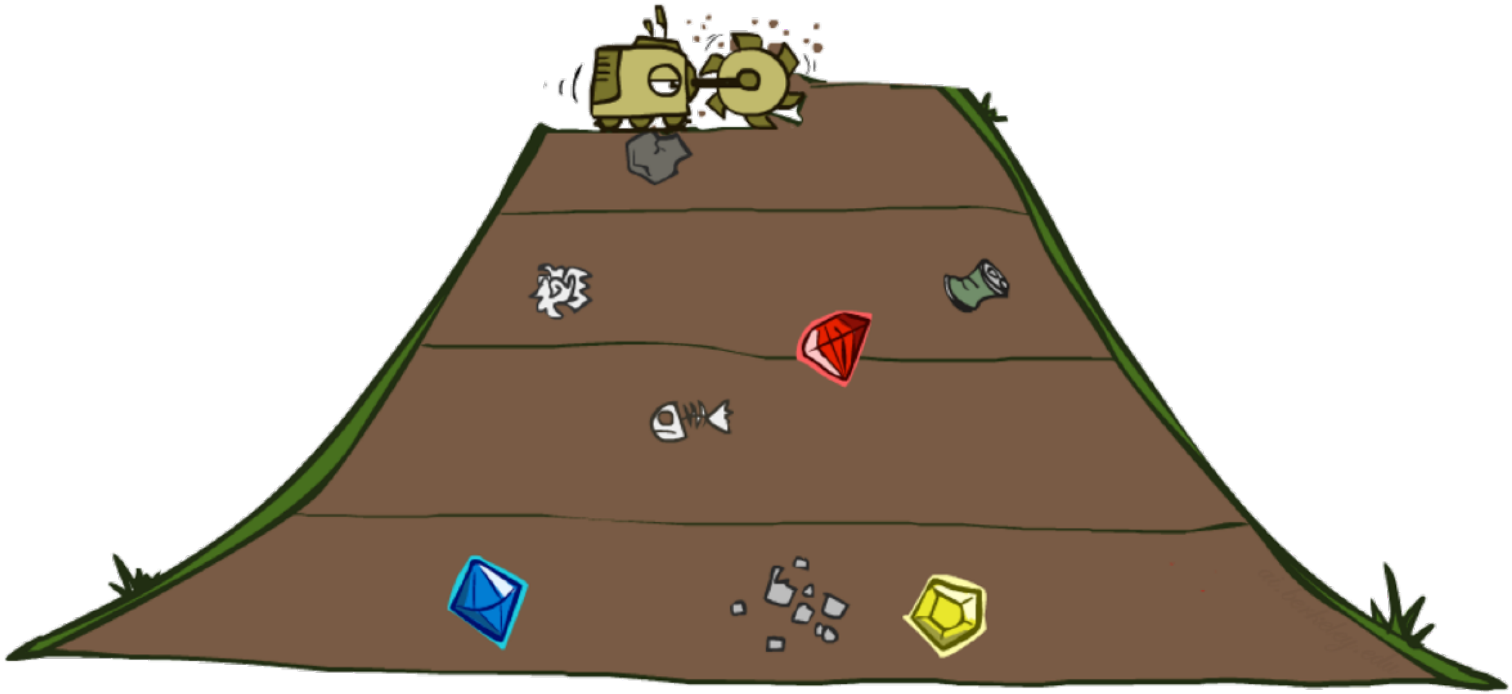


Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188

```
function BREADTH-FIRST-SEARCH( problem ) returns a solution, or failure

    vertex ← a vertex with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST( vertex.STATE ) then return SOLUTION( vertex )
    frontier ← a FIFO queue with vertex as the only event
    explored ← an empty set
    loop do
        if EMPTY?( frontier ) then return failure
        vertex ← POP( frontier )  // chooses the shallowest vertex in frontier
        add vertex.STATE to explored
        for each action in problem.ACTIONS( vertex.STATE ) do
            child ←  CHILD-VERTEX( problem, vertex, 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 )
```

# Breadth-first Search Performance

**The good:**

**Completeness:** if the shallowest goal vertex is at some finite depth $d$, it will be found

**Optimality:** optimal if the path cost is a non-decreasing function of the depth of the node

- Most common such scenario: all actions have the same cost

# Breadth-first Search Performance

**The bad:**

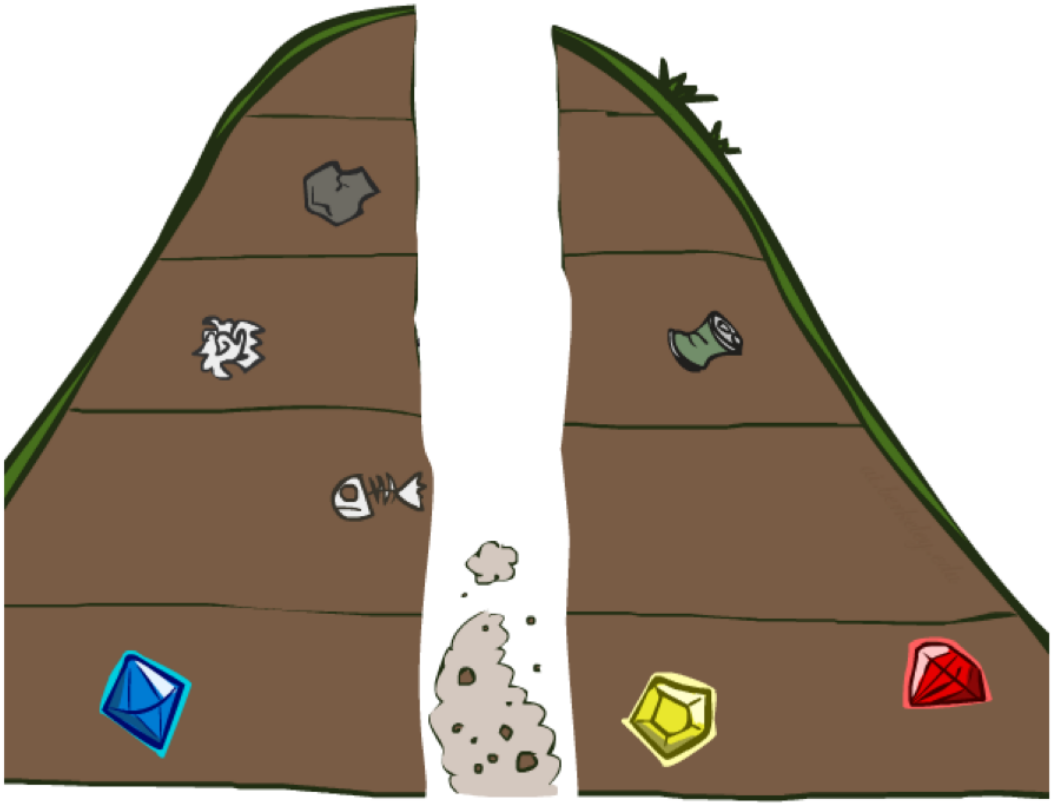
Assume a uniform tree where every state has $b$ successors

**Time complexity**: Worst case when the solution is at depth $d$, in the last vertex generated at that level

- $b + b^2 + b^3 + \ldots b^d = \mathrm{O}(b^d)$

**Space complexity:** always within a factor of $b$ of the time complexity

# Depth-first Search



What happens if we have infinite state spaces?

```
function DEPTH-LIMITED-SEARCH( problem, limit ) returns a solution, or failure/cutoff
    return RECURSIVE-DLS( MAKE-NODE( problem.INITIAL-STATE ), problem, limit)

function RECURSIVE-DLS( vertex, problem, limit ) returns a solution, or failure/cutoff
    if problem.GOAL-TEST( vertex.STATE) then return SOLUTION( vertex )
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS( vertex.STATE ) do
            child ← CHILD-NODE( problem, vertex, action )
            result ← RECURSIVE-DLS( child, problem, limit - 1 )
            if results = cutoff then cutoff_occured? ← true
            else if result ≠ failure then return result
        if cutoff_occurred? then return cutoff else return failure
```

# Depth-limited search performance

Choose a depth limit $l$

Assume a uniform tree where every state has $b$ successors

**Completeness:** incomplete if we choose $l < d$

**Optimality:** non-optimal if $l > d$

**Time complexity:** $O(b^l)$

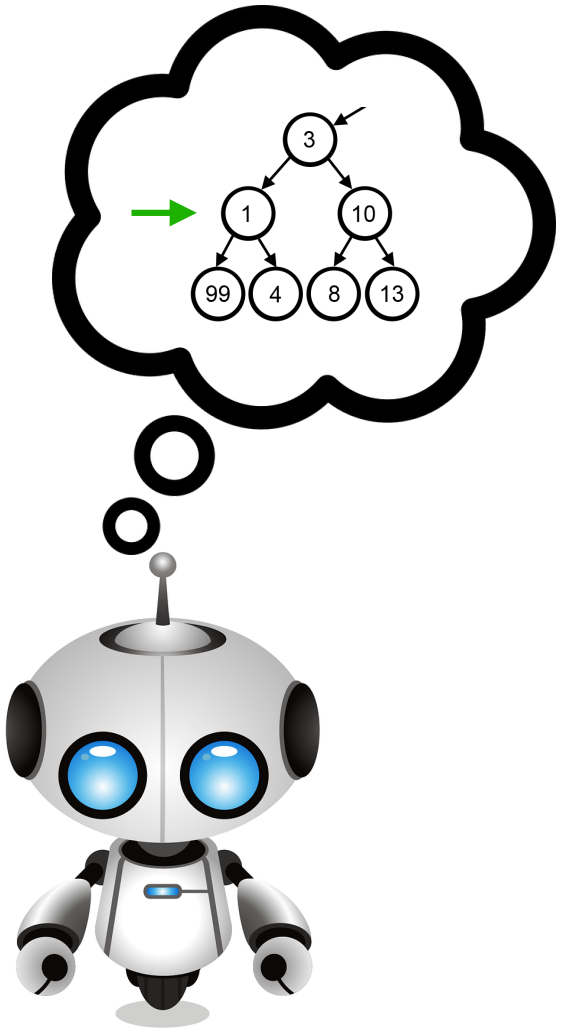
**Space complexity:** $O(bl)$

When will breadth-first search outperform depth-first search?

When will depth-first search outperform breadth-first search?

# Search and Models

- Search operates over models of the world
  - The agent doesn't try all of the plans in the real world
  - Planning is all in simulation
  - Therefore, your search is only as good as your models




Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188