COE 4213564

Introduction to Artificial Intelligence

SOLVING PROBLEMS BY SEARCHING

Chapter 3

Fall 2025

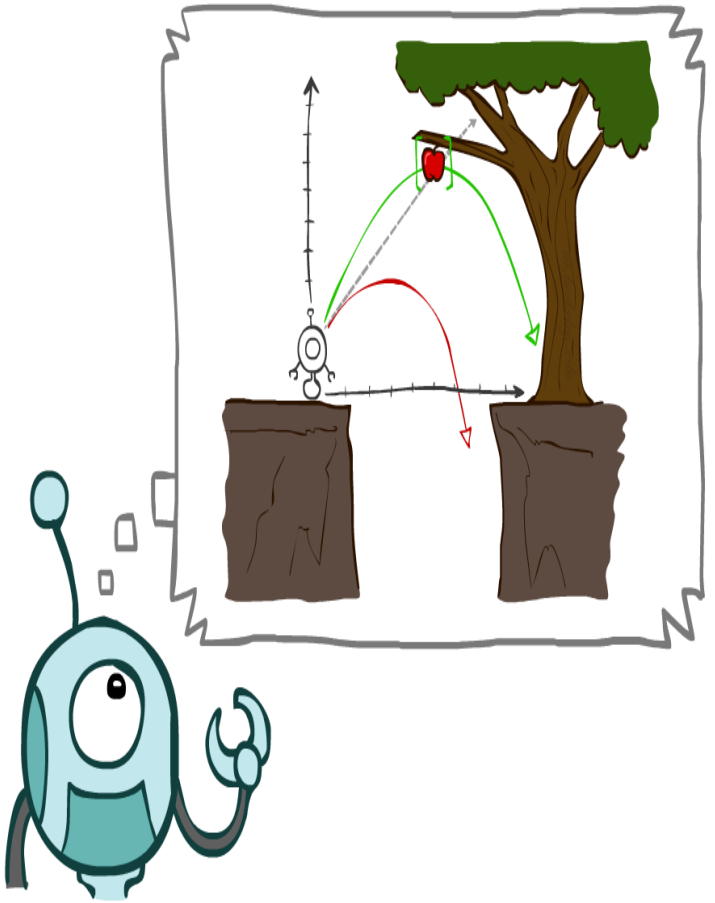
Many slides are adapted from CS 188 (http://ai.berkeley.edu), CIS 521, CS 221.

Problem solving agents and uninformed search methods

- A simple reflex agent is one that selects an action based only on the **current percept**. It **ignores** the rest of the **percept history**.
- A problem-solving agent must plan ahead.
- Agents that Plan Ahead: When the correct action to take is not immediately obvious,
  - an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state.
  - Such an agent is called a problem-solving agent, and
  - the computational process it undertakes is called search.
- In this chapter, we consider only the simplest environments: episodic, single agent, fully observable, deterministic, static, discrete, and known.

- Search Problems: We distinguish between
  - informed algorithms, in which the agent can estimate how far it is from the goal, and
  - uninformed algorithms, where no such estimate is available

- Uninformed Search Methods
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search



Search Problems

Example search problem: Holiday in Romania

- Imagine an agent enjoying a touring vacation in Romania

Problem-Solving Agent follow four-phase problem-solving process

- Goal formulation: The agent adopts the goal of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.
- Problem formulation: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world. For our agent, one good model is to consider the actions of traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.
- Search: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a solution.
  - The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution (such as going from Arad to Sibiu to Fagaras to Bucharest), or it will find that no solution is possible.
- Execution: The agent can now execute the actions in the solution, one at a time.

Search problems and solutions

Classifier (reflex-based models):

Search problem (state-based models):

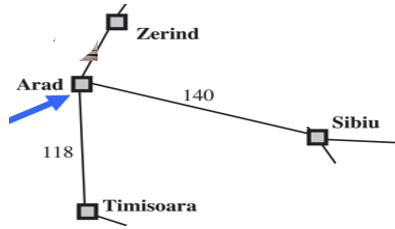
Key: need to consider future consequences of an action!

- Reflex-based models in machine learning (e.g., linear predictors and neural networks) that output either a or (for binary classification) or a real number (for regression).
- While reflex-based models were appropriate for some applications such as sentiment classification or spam filtering, the applications we will look at today, such as solving puzzles, demand more.
- To tackle these new problems, we will introduce search problems, our first instance of a state-based model.
- In a search problem, in a sense, we are still building a predictor which takes an input , but will now return an entire action sequence, not just a single action.
- Of course you should object: can't I just apply a reflex model iteratively to generate a sequence? While that is true, the search problems that we're trying to solve importantly require reasoning about the consequences of the entire action sequence, and cannot be tackled by myopically predicting one action at a time.
- Of course, saying "cannot" is a bit strong, since sometimes a search problem can be solved by a reflex-based model. You could have a massive lookup table that told you what the best action was for any given situation.

Search problems and solutions

- A search problem can be defined formally as follows:
  - A set of possible states that the environment can be in. We call this the state space.
  - A set of one or more goal states. Sometimes there is one goal state (e.g., Bucharest), goal states sometimes, there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states.
  - The actions available to the agent. Given a state s, ACTIONS(s) returns a finite set of Action actions that can be executed in s. We say that each of these actions is applicable in s. An example:

    ACTIONS(Arad) = {ToSibiu,ToTimisoara,ToZerind}.



  - A transition model, which describes what each action does. RESULT(s, a) returns the Transition model state that results from doing action a in state s. For example,

    RESULT(Arad, ToZerind) = Zerind

  - An action cost function, denoted by ACTION-COST(s,a,s') when we are programming or c(s,a,s') when we are doing math, that gives the numeric cost of applying action a in state s to reach state s'. . A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length in miles (as seen in Figure 3.1), or it might be the time it takes to complete the action.
- A sequence of actions forms a path, and a solution is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- An optimal solution has the lowest path cost among all solutions.
- The state space can be represented as a graph in which the vertices are states and the directed edges between them are actions.

Formal Definition

1. ***States***: a set S
2. An *initial state* $s_i \in S$
3. ***Actions:*** *a set* A

   $\forall s$ $Actions(s)$ = *the set of actions that can be executed in* s, *that are applicable* in s.
4. ***Transition Model:*** $\forall s \forall a \in Actions(s)$ $Result(s, a) \rightarrow s_r$

   s_r *is called a successor of* s

   $\{s_i\} \cup Successors(s_i)^*$ = *state space*
5. ***Path cost*** *(Performance Measure):* Must be additive, e.g. sum of distances, number of actions executed, ...

   $c(x,a,y)$ is the step cost, assumed ≥ 0
   - (where action a goes from state x to state y)
6. ***Goal test:*** $Goal(s)$

   Can be implicit, e.g. $checkmate(s)$

   s is a *goal state* if $Goal(s)$ is *true*

Abstraction: Formulating problems

- Our formulation of the problem of getting to Bucharest is a model—an abstract mathematical description—and not the real thing.
  - Compare the simple atomic state description Arad to an actual cross-country trip, where the state of the world includes so many things:
    - the traveling companions,
    - the current radio program,
    - the scenery out of the window,
    - the proximity of law enforcement officers,
    - the distance to the next rest stop,
    - the condition of the road,
    - the weather,
    - the traffic, and so on.
  - All these considerations are left out of our model because they are irrelevant to the problem of finding a route to Bucharest.
- The process of removing detail from a representation is called abstraction.
- The abstraction is valid if we can elaborate any abstract solution into a solution in the more detailed world. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

Art: Formulating a Search Problem

- Decide:

  Which properties matter & how to represent

  - *Initial State, Goal State, Possible Intermediate States*

  Which actions are possible & how to represent

  - *Operator Set: Actions and Transition Model*

  Which action is next

  - *Path Cost Function*
- **Formulation greatly affects combinatorics of search space and therefore speed of search.**
- **Hard subtask: Selecting a state space**

  Real world is absurdly complex

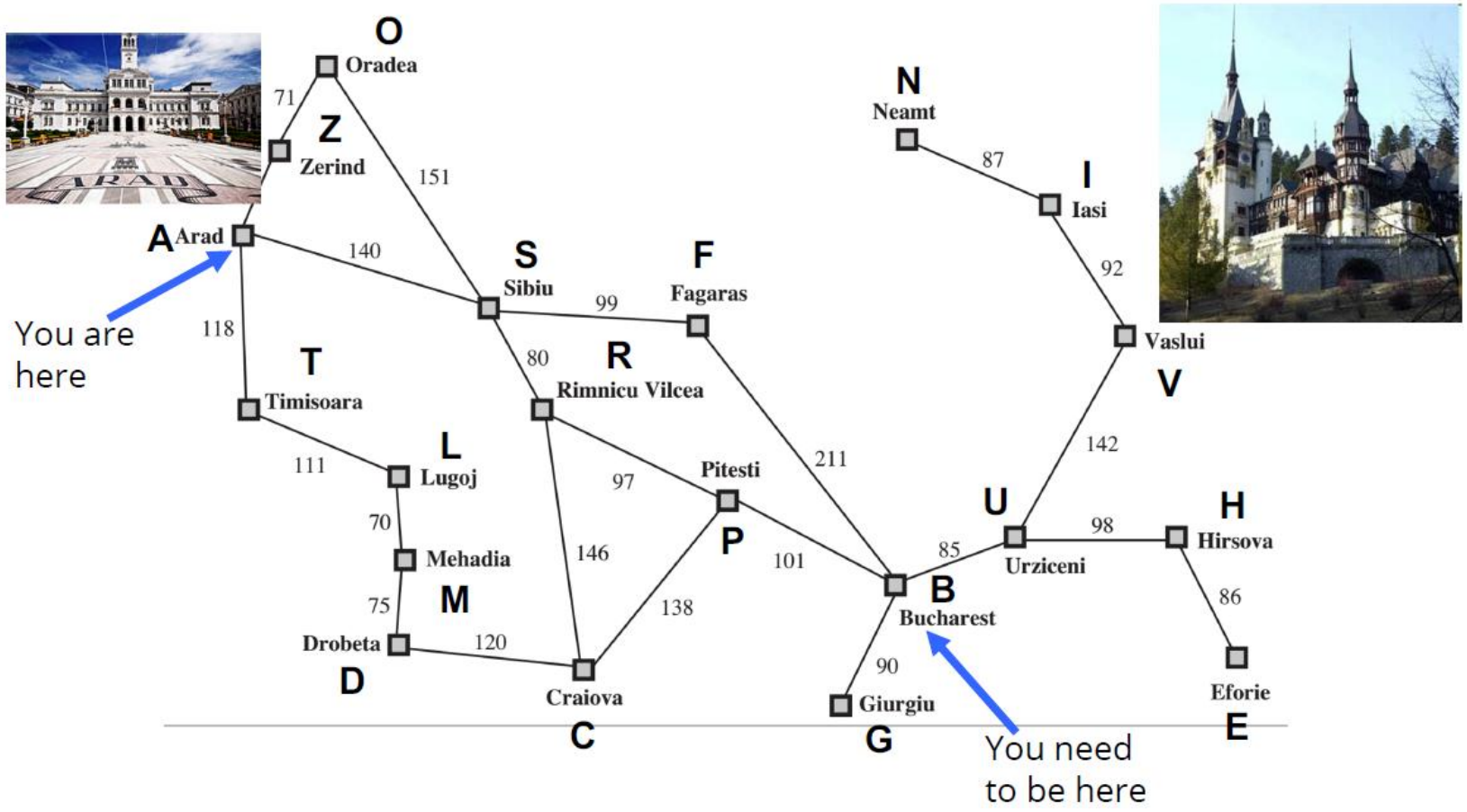
  State space must be **abstracted** for problem solving

  (abstract) **State** = set (equivalence class) of real-world states

  (abstract) **Action** = equivalence class of combinations of real-world action

  - e.g. *Arad* → *Zerind* represents a complex set of possible routes, detours, rest stops, etc
  - The abstraction is valid if the path between two states is reflected in the real world
- Each abstract action should be "easier" than the real problem

Example: Traveling in Romania



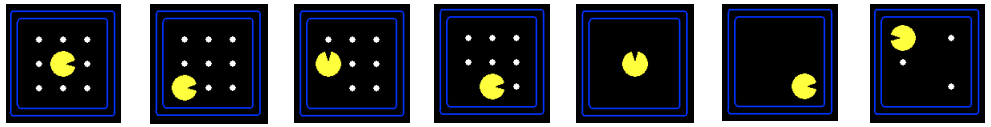
- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with
  - cost = distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

Example: Pac-Man Game

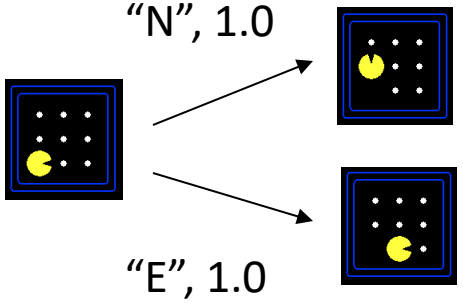
Pac-Man is an action[7] maze chase video game;

- the player controls the eponymous character through an enclosed maze.
- The objective of the game is to eat all of the dots placed in the maze
- while avoiding four colored ghosts — Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange)

- 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

Another example: vacuum world

- States: **integer dirt and robot locations** (ignore dirt amounts etc.)
- Actions: **Left**, **Right**, **Suck**, **NoOp**
- Goal test: **no dirt**
- Path cost: **1 per action (0 for NoOp)**

States: A state of the world says which objects are in which cells.

In a simple two cell version,

- the agent can be in one cell at a time
- each cell can have dirt or not,
- 2 positions for agent * 2^2 possibilities for dirt = **8 states**.

- With n cells, there are $n*2^n$ states.

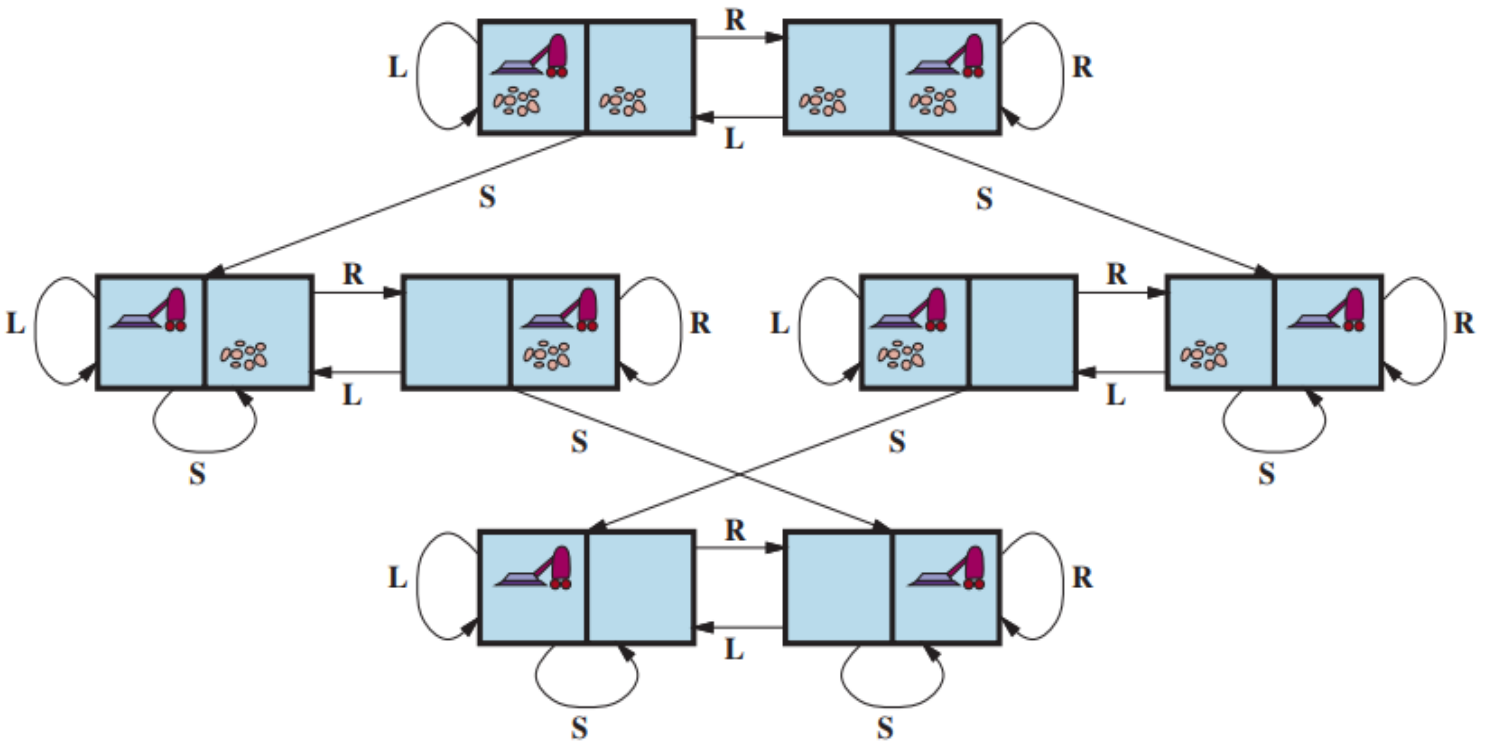
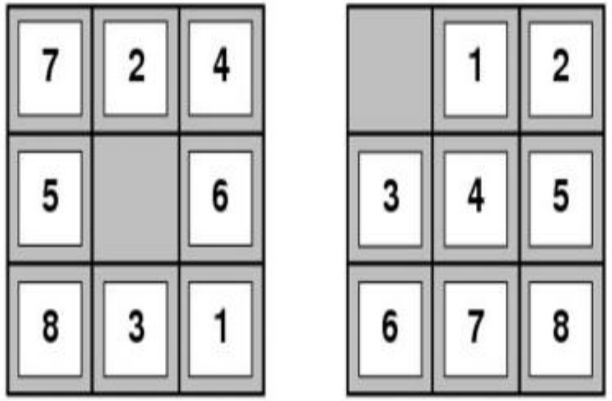


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Example search problem: 8-puzzle

Formulate *goal*

- Pieces to end up in order as shown...



Start State

Goal State

Formulate *search problem*

- ***States:*** configurations of the puzzle (9! configurations)
- ***Actions***: Move one of the movable pieces (≤4 possible)
- ***Performance measure***: minimize total moves

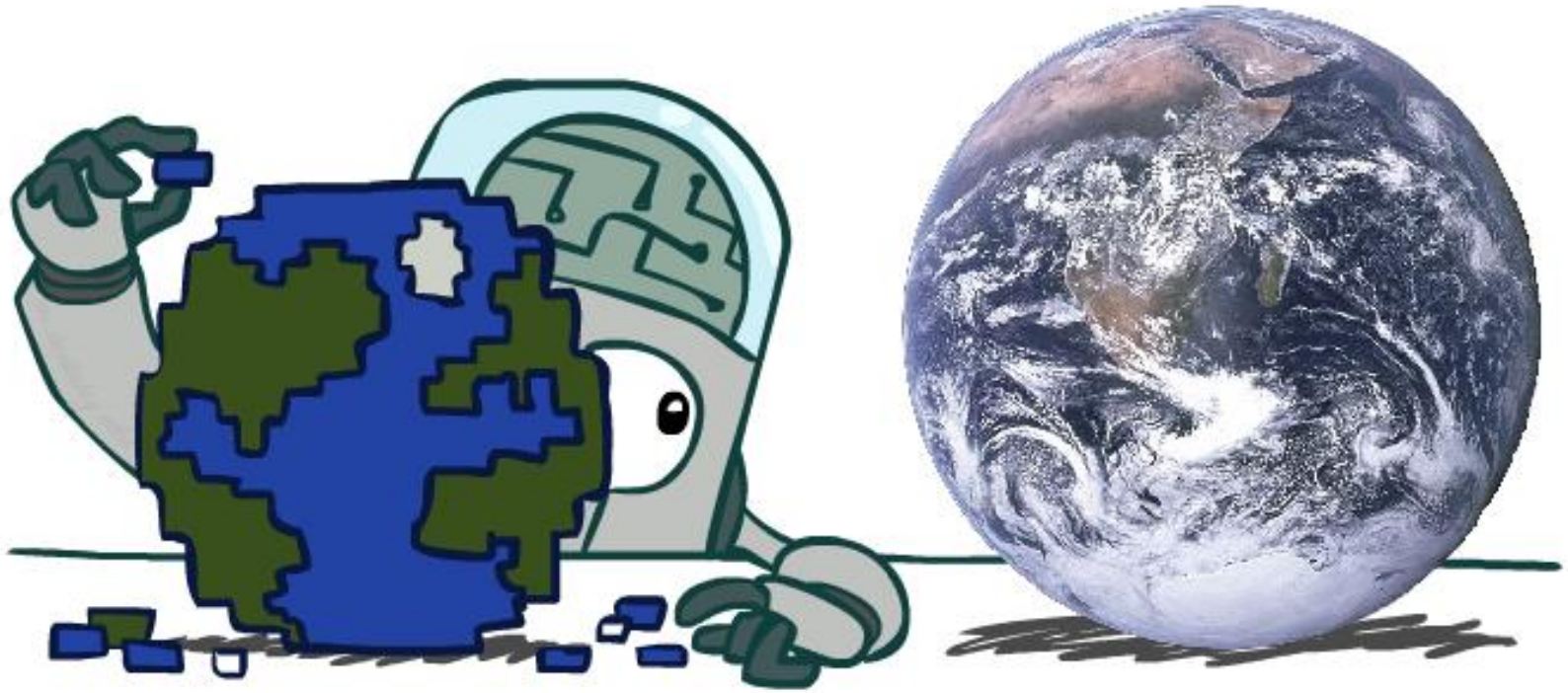
Find *solution*

- Sequence of pieces moved: 3,1,6,3,1,...

Other search problems

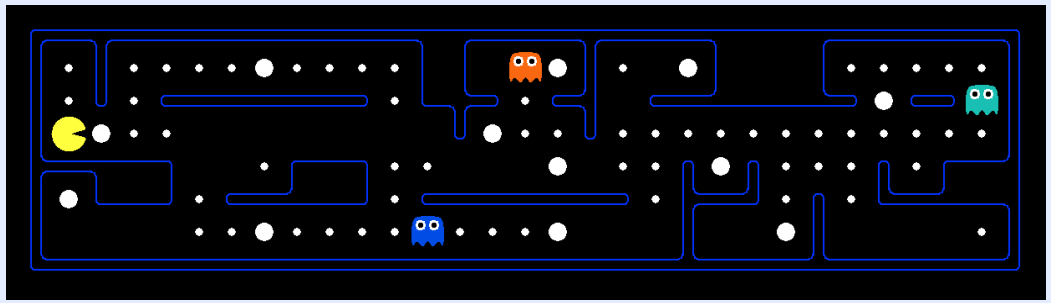
- Route-finding problem is defined in terms of specified locations and transitions along edges between them.
- Touring problems describe a set of locations that must be visited, rather than a single goal destination.
- A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- Robot navigation is a generalization of the route-finding problem described earlier. Rather than following distinct paths (such as the roads in Romania), a robot can roam around, in effect making its own paths.
- Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost.

Search Problems Are (Absract) Models



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?

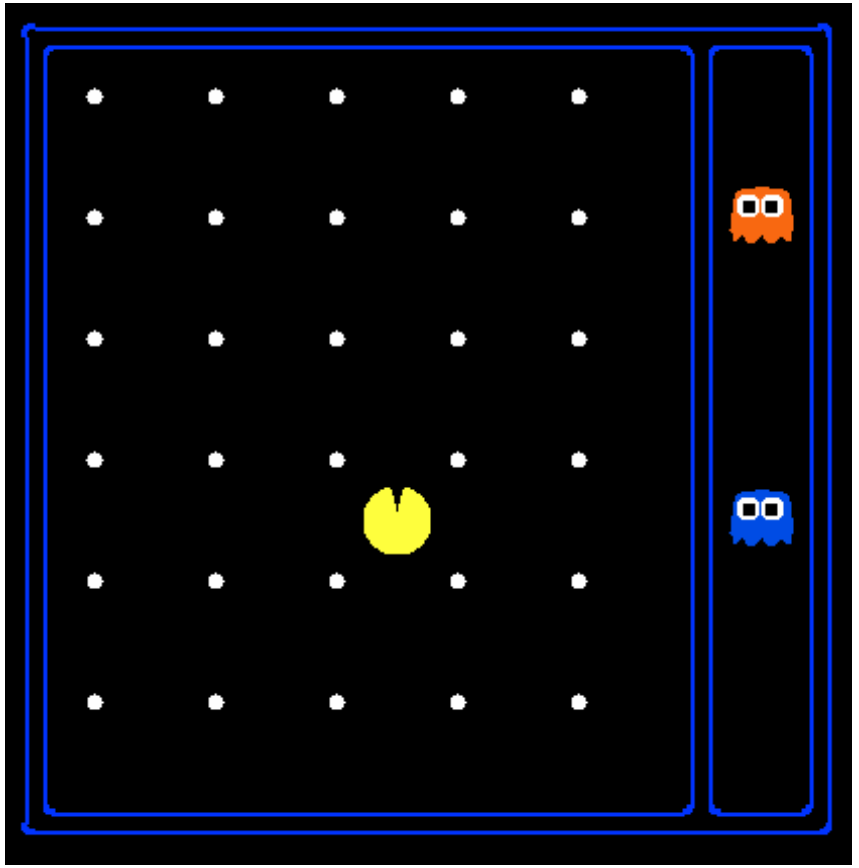
- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?

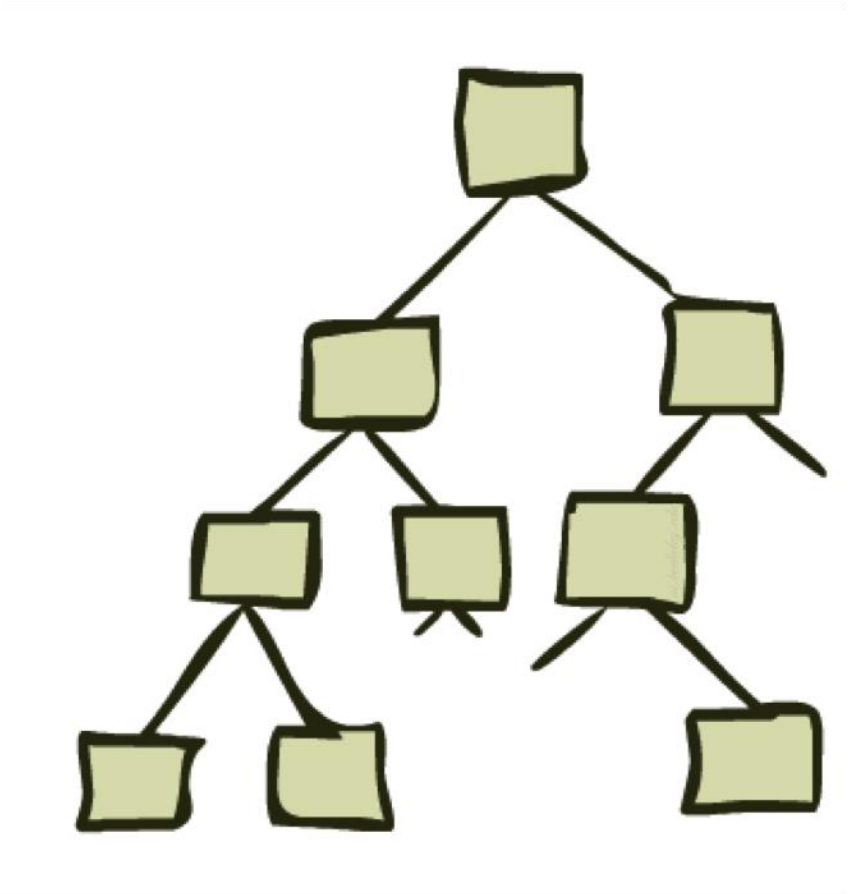
    $120x(2^{30})x(12^2)x4$
  - States for pathing?

    120
  - States for eat-all-dots?

    $120x(2^{30})$

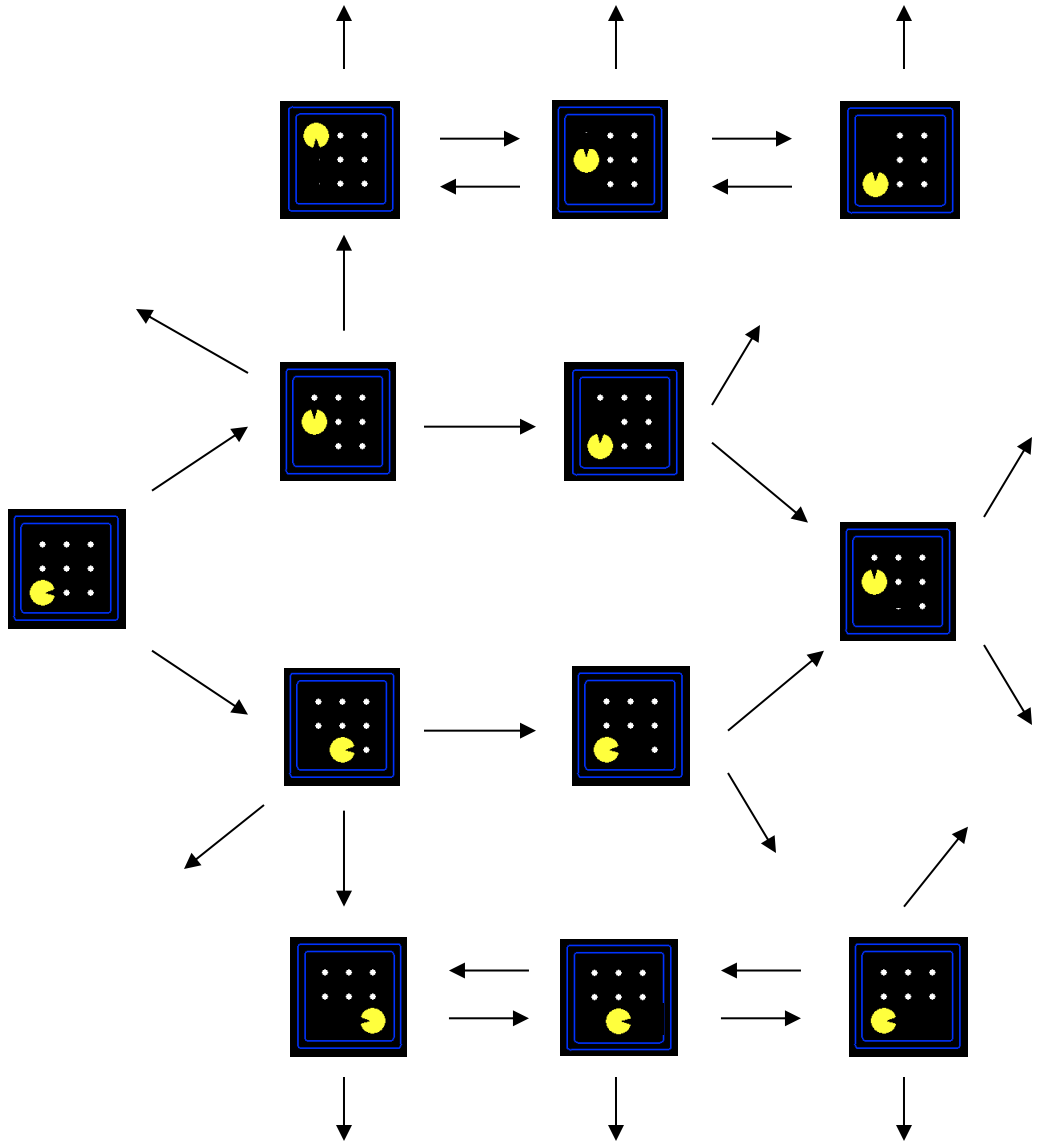


State Space Graphs and Search Trees



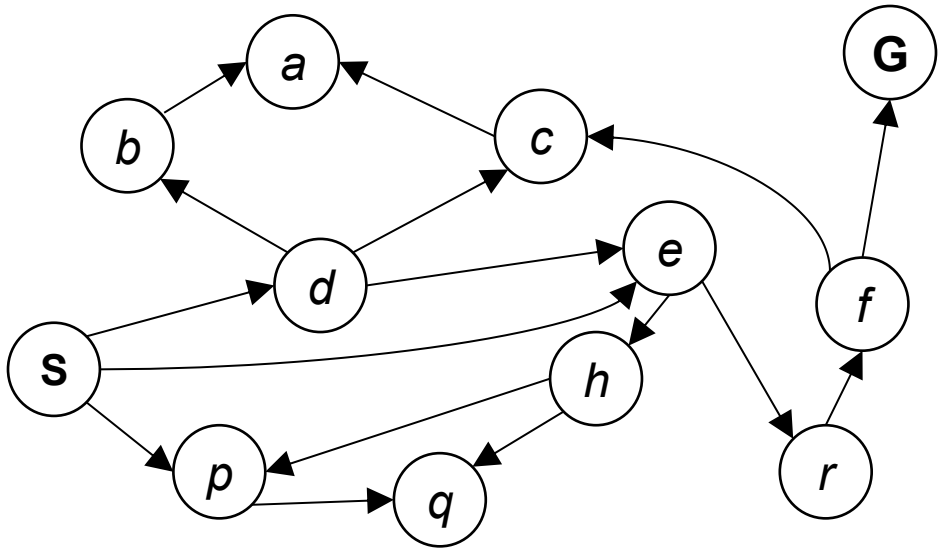
State Space Graphs

- 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



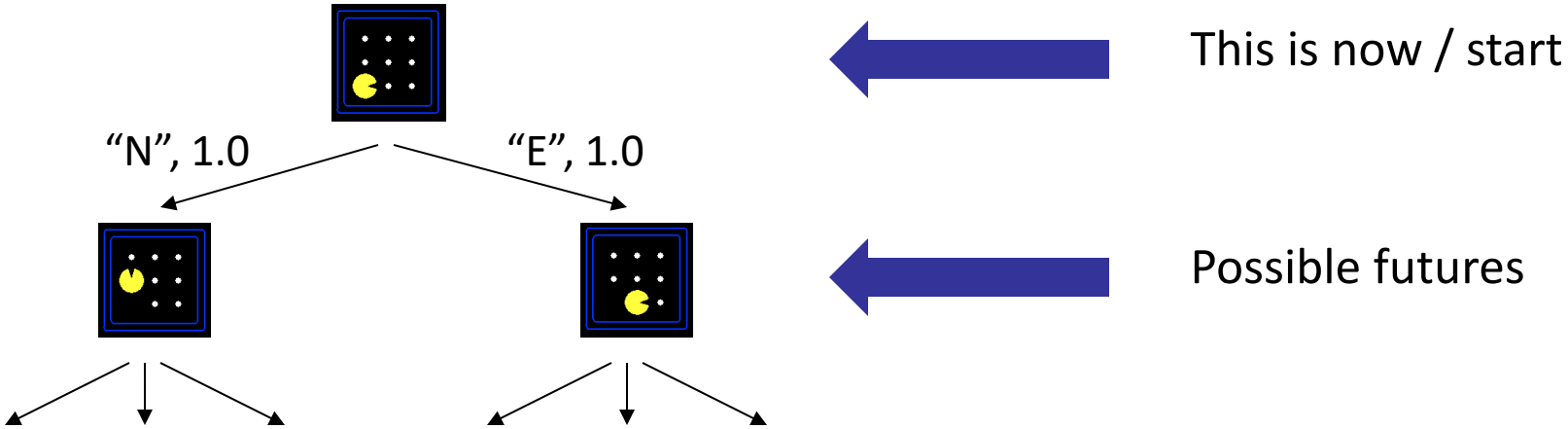
State Space Graphs

- 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 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

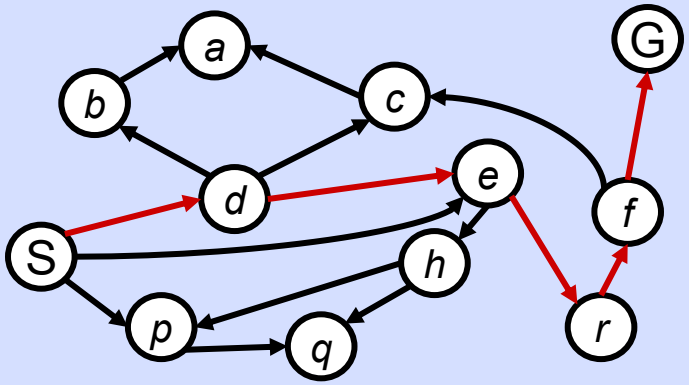
Search Trees



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

State Space Graphs vs. Search Trees

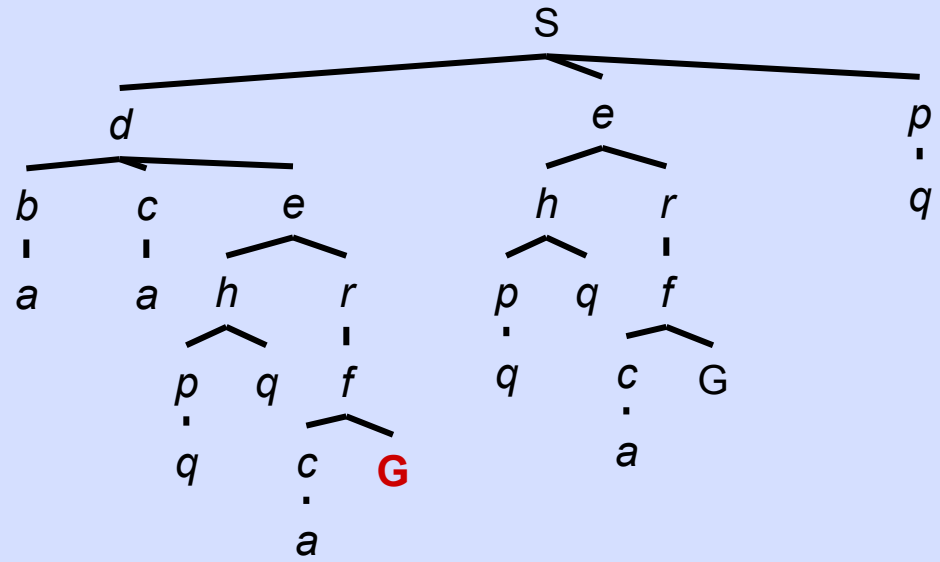
State Space Graph



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

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

Search Tree

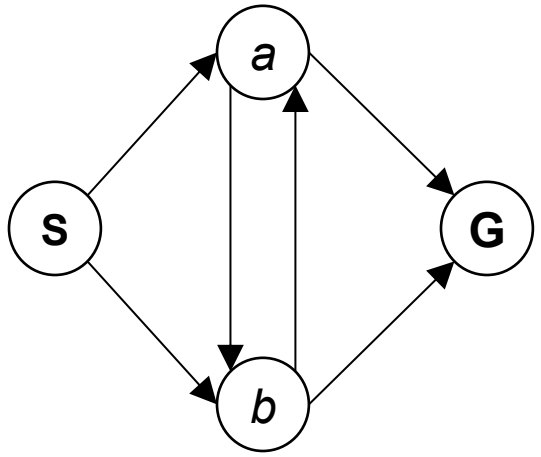


State Space Graphs vs. Search Trees

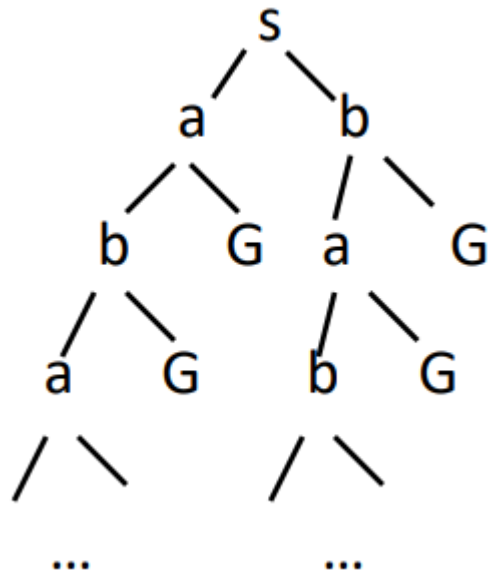
- It is important to understand the distinction between the state space and the search tree.
- The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.
- The search tree describes paths between these states, reaching towards the goal.
- The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

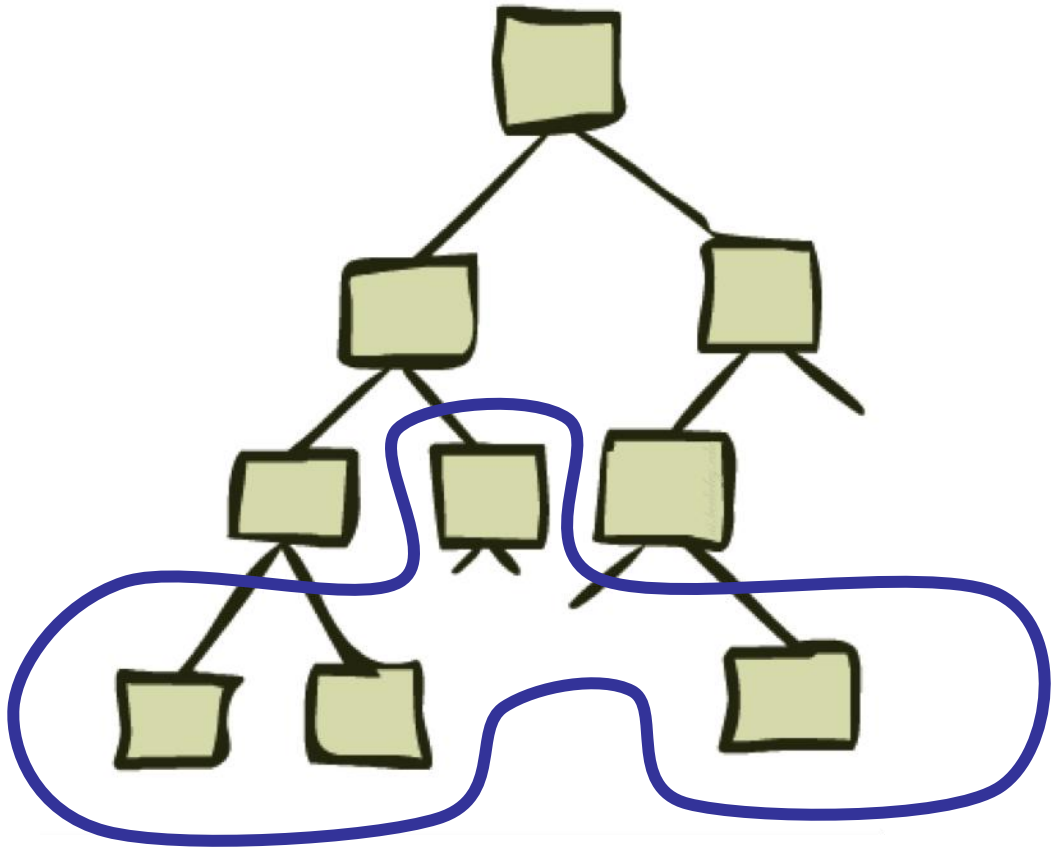


How big is its search tree (from S)?

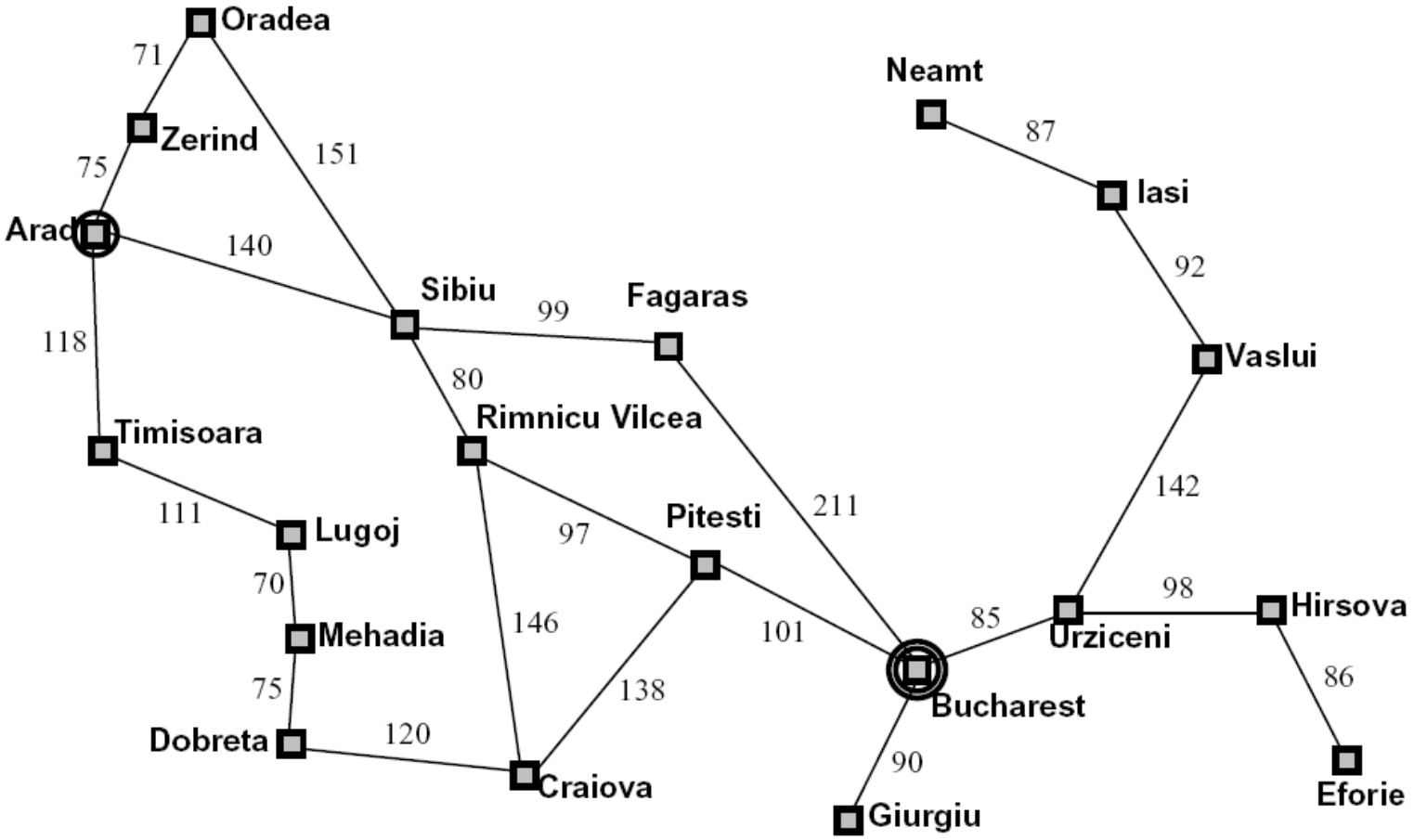


Important: Lots of repeated structure in the search tree!

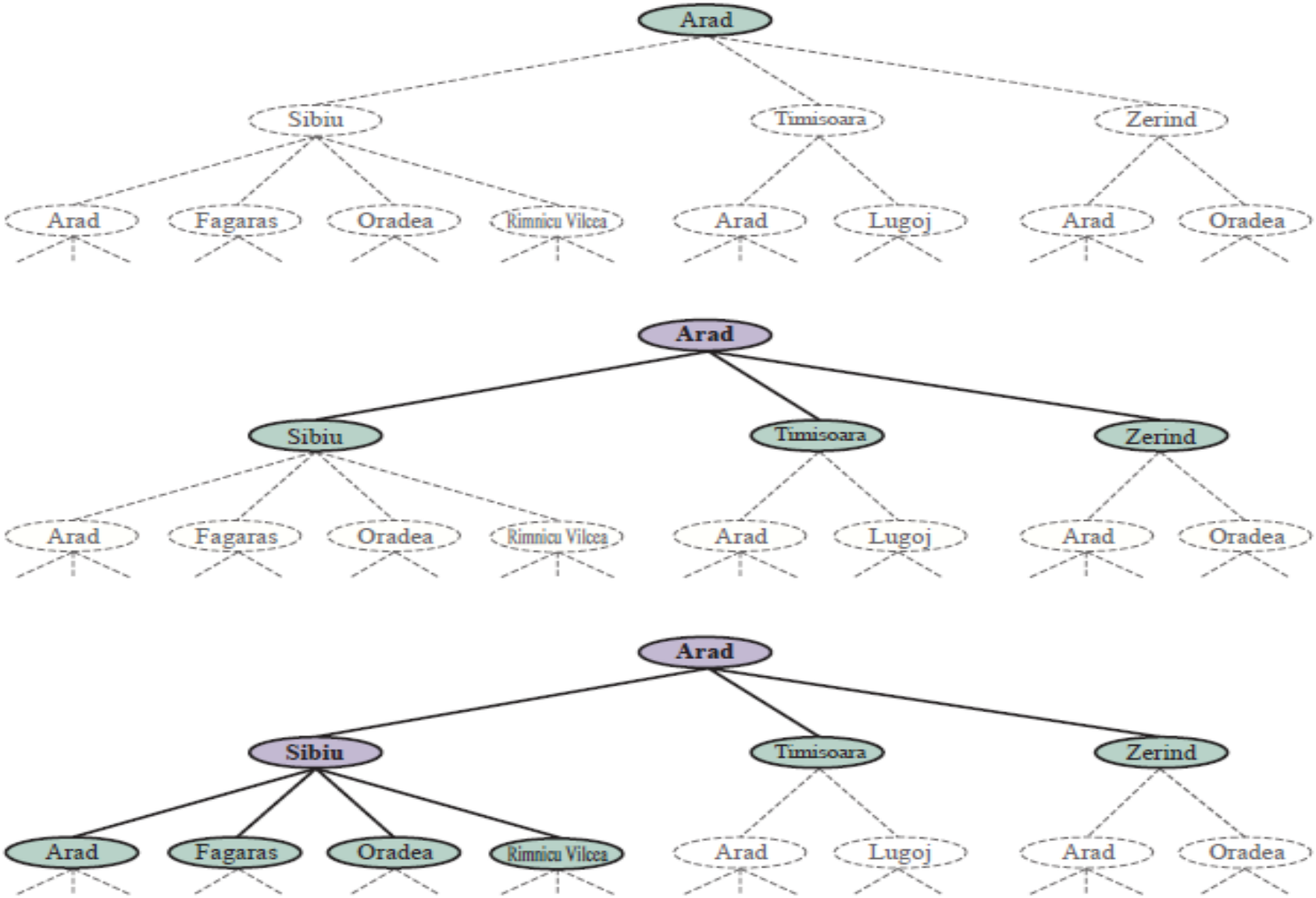
Tree Search



Search Example: Romania

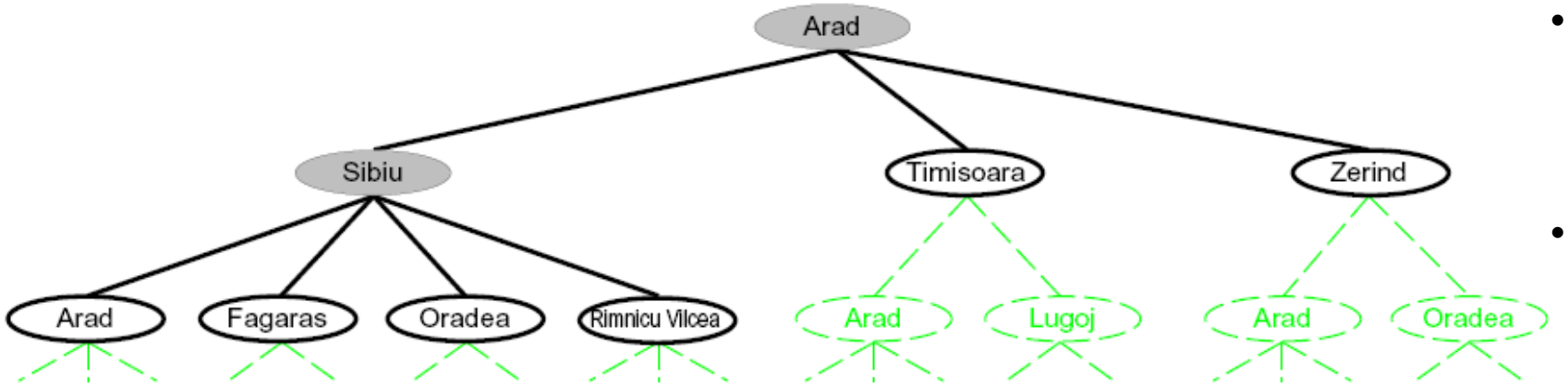


Three partial search trees for finding a route from Arad to Bucharest



- Expand out potential plans (tree nodes)
- Nodes that have been expanded are lavender with bold letters;
- nodes on the frontier that have been generated but not yet expanded are in green;
- the set of states corresponding to these two types of nodes are said to have been reached. Nodes that could be generated next are shown in faint dashed lines.
- Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; That can't be an optimal path, so search should not continue from there.

Searching with a Search Tree



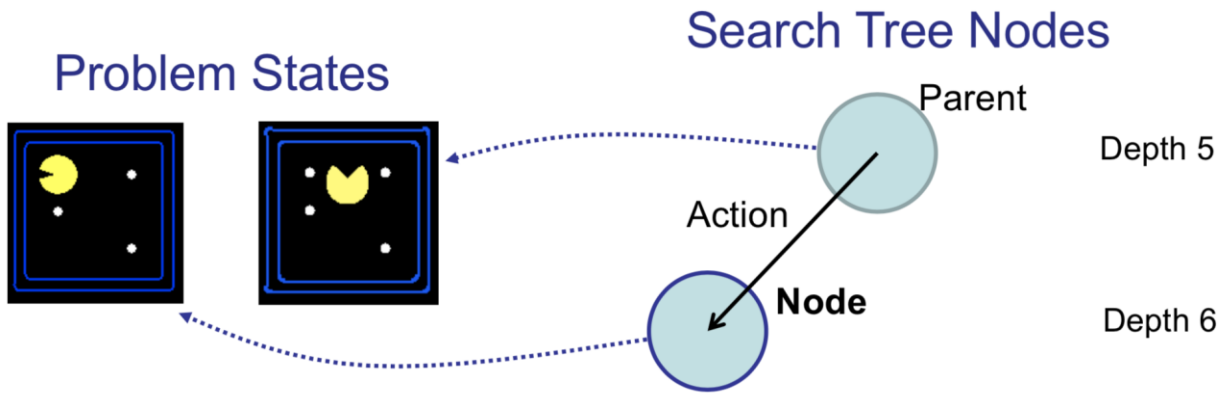
- We can expand the node, by considering the available
  - ACTIONS for that state,
  - using the RESULT function
- to see where those actions lead to, and generating a new node (called a child node or successor node) for each of the resulting states.
- Each child node has Arad as its parent node.

- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe (frontier) of partial plans under consideration
  - Try to expand as few tree nodes as possible

- Remember functions:
  - ACTIONS(Arad) = {ToSibiu,ToTimisoara,ToZerind}
  - RESULT(Arad, ToZerind) = Zerind

States vs. Nodes

- Vertices in state space graphs are problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors
- Vertices in search trees ("Nodes") are plans
  - Contain a **problem state** and one parent, a path length, a depth, and a cost
  - Represent a plan (sequence of actions) which results in the node's state
- **The same problem state may be achieved by multiple search tree nodes**



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 (data structure: queue, stack or priority queue)
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

Search Strategies

- A strategy is defined by picking **the order of node expansion**
- Strategies are evaluated along the following dimensions:
  - **completeness:** does it always find a solution if one exists?
  - **time complexity:** number of nodes generated/expanded
  - **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 ∞)

Best-first search

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- How do we decide which node from the frontier to expand next?
- A very general approach is called best-first search, in which we choose a node, n, with minimum value of some evaluation function, f(n).
- On each iteration we choose a node on the frontier with minimum f (n) value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes.
- Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path.

Search data structures

- Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with four components:
  - node.STATE: the state to which the node corresponds;
  - node.PARENT: the node in the tree that generated this node;
  - node.ACTION: the action that was applied to the parent's state to generate this node;
  - node.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, we use g(node) as a synonym for PATH-COST.
- We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are:
  - IS-EMPTY(frontier) returns true only if there are no nodes in the frontier.
  - POP(frontier) removes the top node from the frontier and returns it.
  - TOP(frontier) returns (but does not remove) the top node of the frontier.
  - ADD(node, frontier) inserts node into its proper place in the queue.

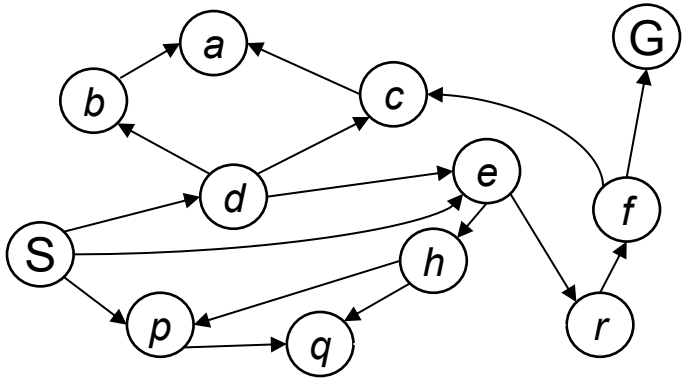
Search data structures

- Three kinds of queues are used in search algorithms:
  - A priority queue first pops the node with the minimum cost according to some evaluation function, f . It is used in best-first search.
  - A FIFO queue or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
  - A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most Stack recently added node; we shall see it is used in depth-first search.
- The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

Uninformed Search Strategies

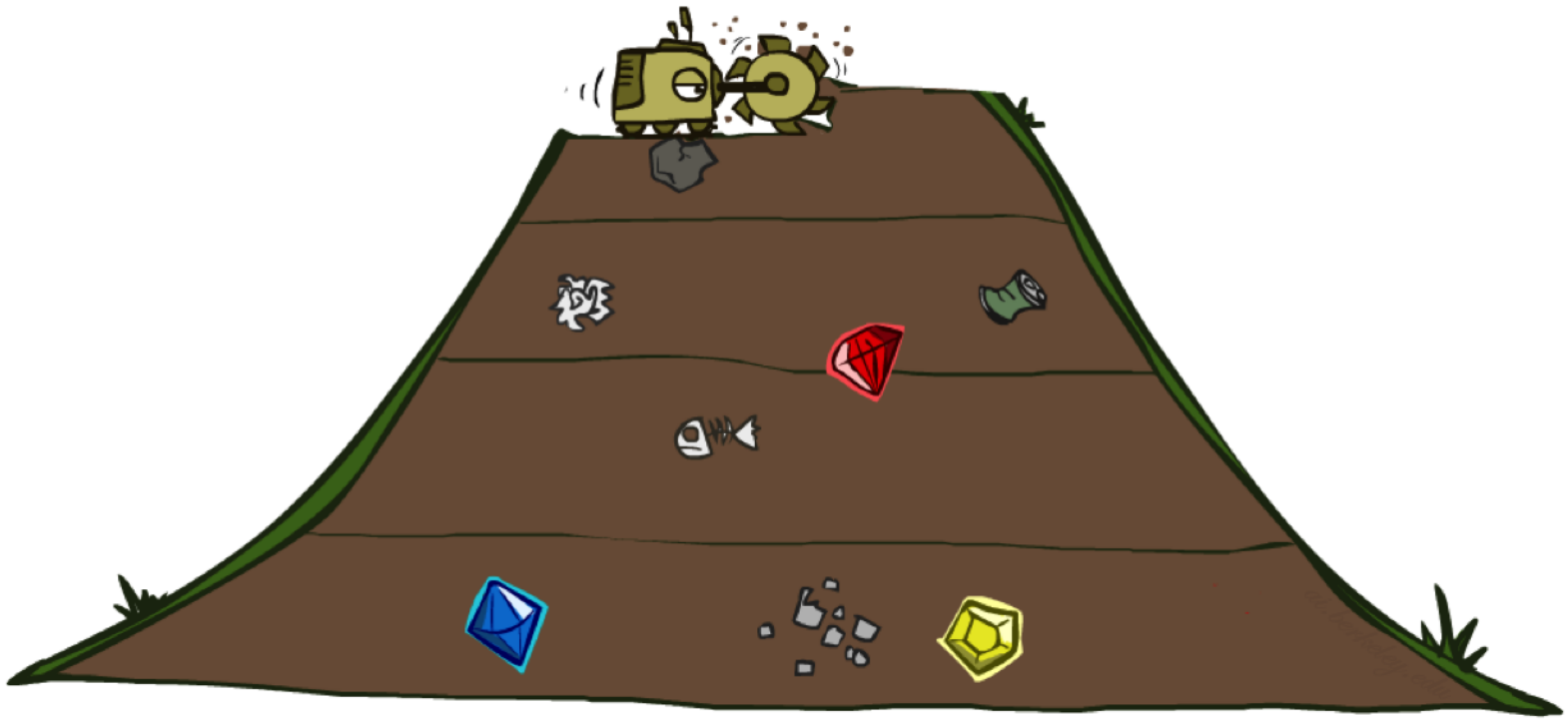
- **Uninformed** strategies use only the information available in the problem definition
- Breadth-first search
- Depth-first search
- Uniform-cost search
- Depth-limited search
- Iterative deepening search

Example: Tree Search



Breadth-First Search

- When all actions have the same cost, an appropriate strategy is breadth-first search,
- in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.



Breadth-first search on a simple binary tree

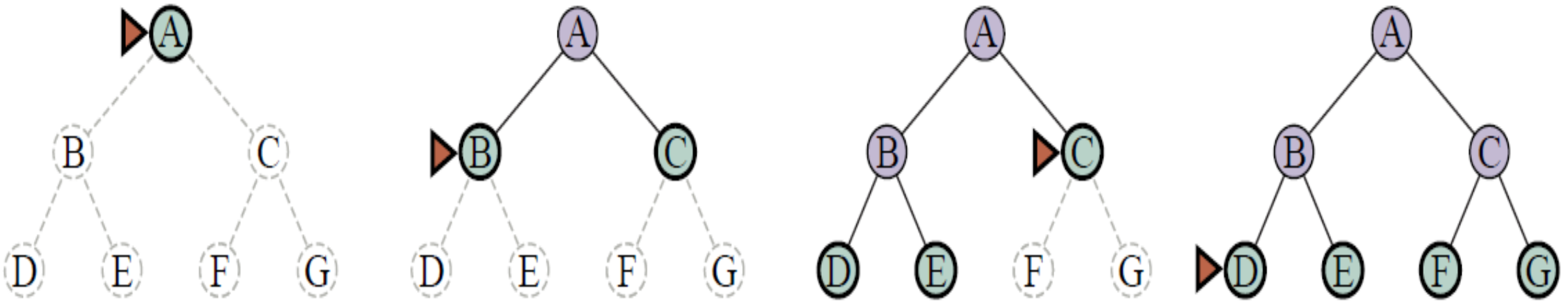
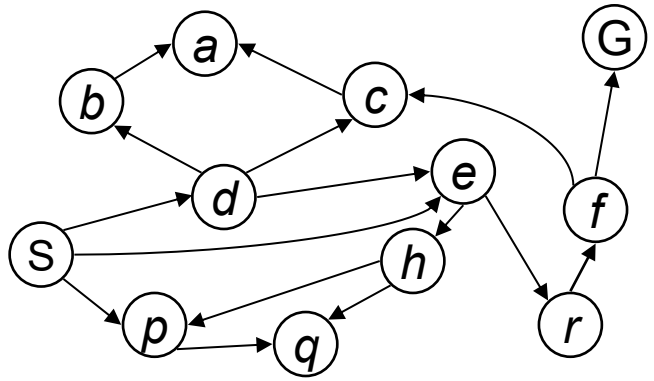


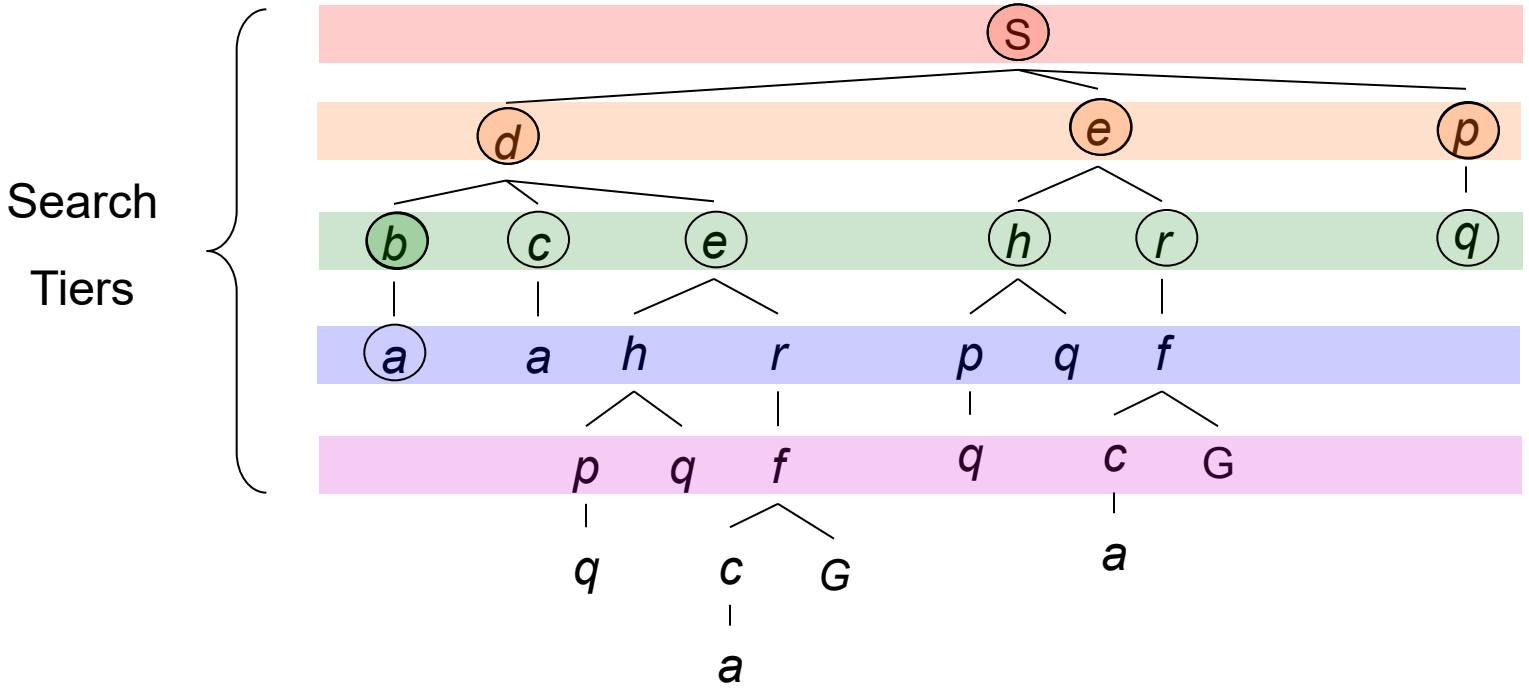
Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Breadth-First Search

Strategy: expand a shallowest node first

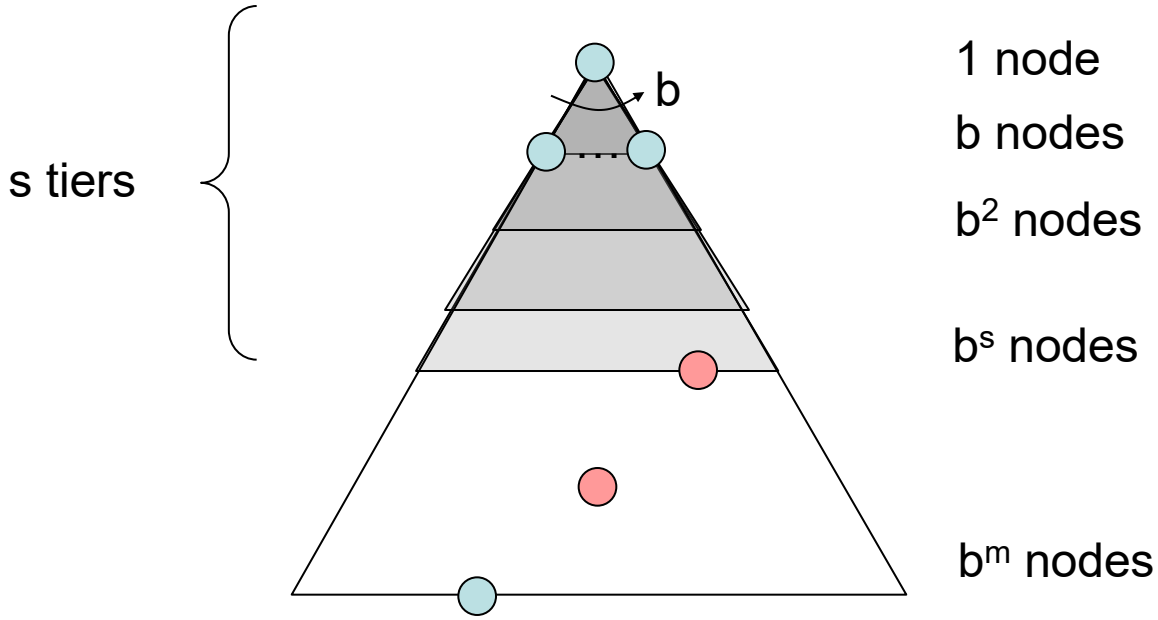
Implementation: Fringe is a FIFO queue





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 $O(b^s)$
  - keeps every node in memory
- 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)



Exponential time and space

Exponential Space (and time) Is Not Good…

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- (*Memory* requirements are a bigger problem than *execution* time.)

DEPTH	NODES	TIME	MEMORY
2	*110*	*0.11 milliseconds*	*107 kilobytes*
4	*11110*	*11 milliseconds*	*10.6 megabytes*
6	10^6	*1.1 seconds*	*1 gigabytes*
8	10^8	*2 minutes*	*103 gigabytes*
10	10^{10}	*3 hours*	*10 terabytes*
12	10^{12}	*13 days*	*1 petabytes*
14	10^{14}	*3.5 years*	*99 petabytles*

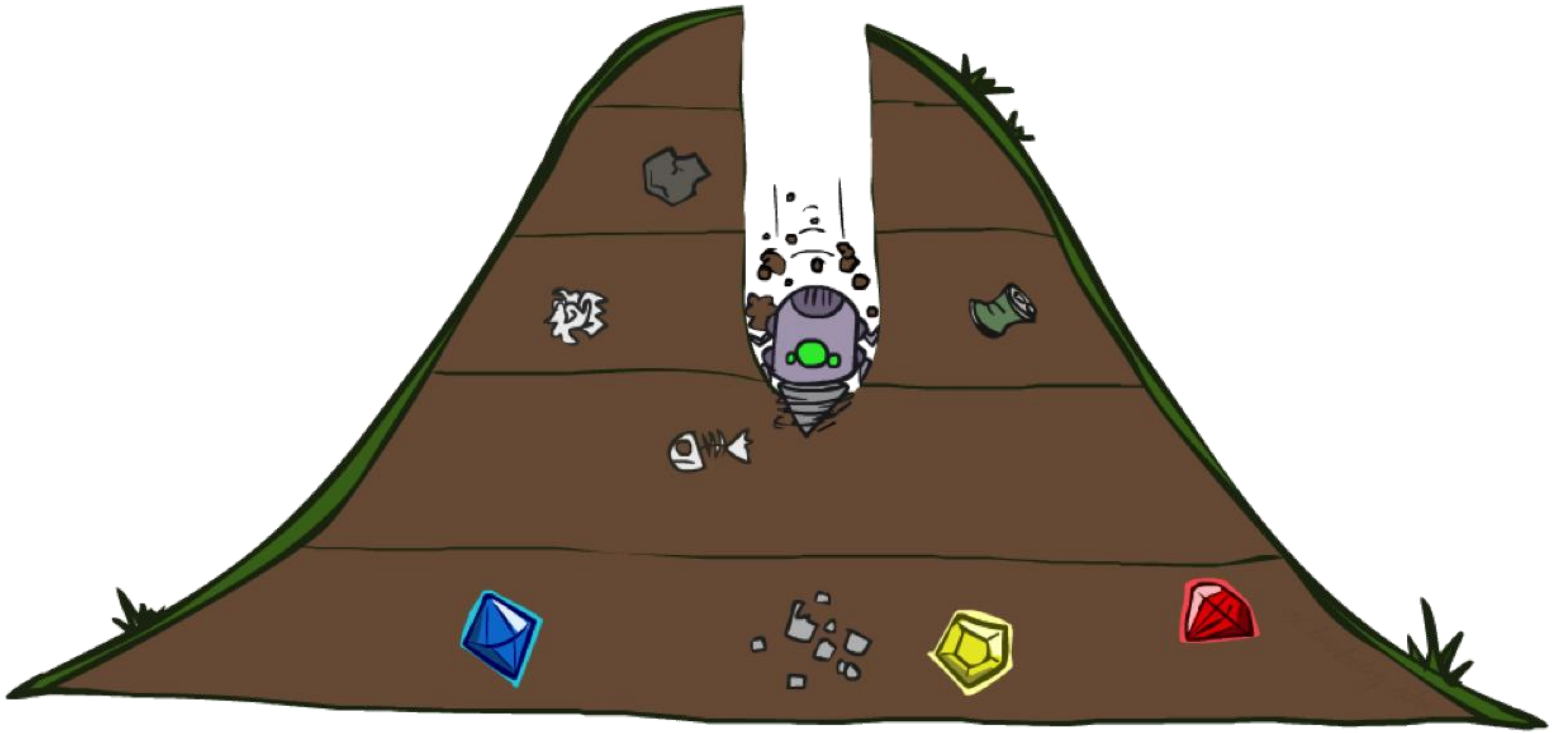
Assumes b=10, 1M nodes/sec, 1000 bytes/node

Breadth-first search algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
     while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure
```

Depth-First Search

- Depth-first search always expands the deepest node in the frontier first.
- The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it.



Depth-first search on a binary tree

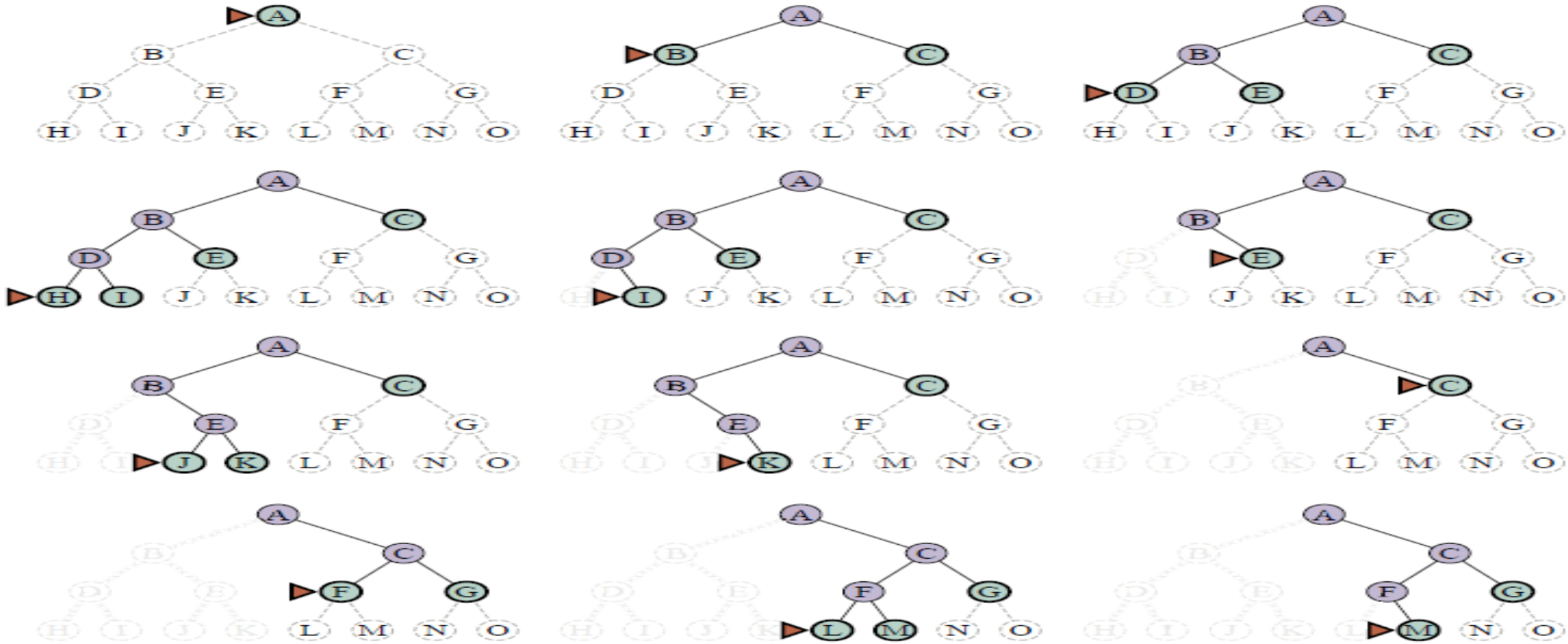
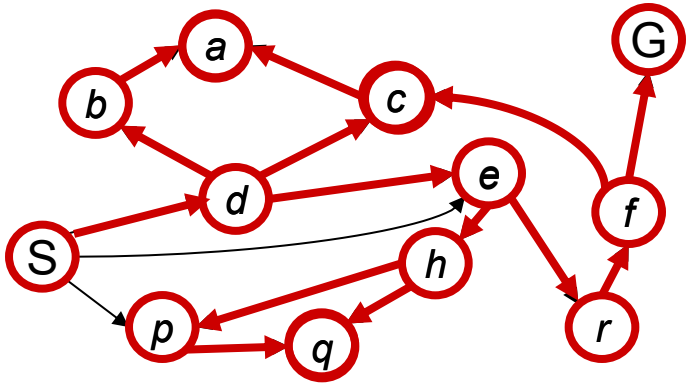


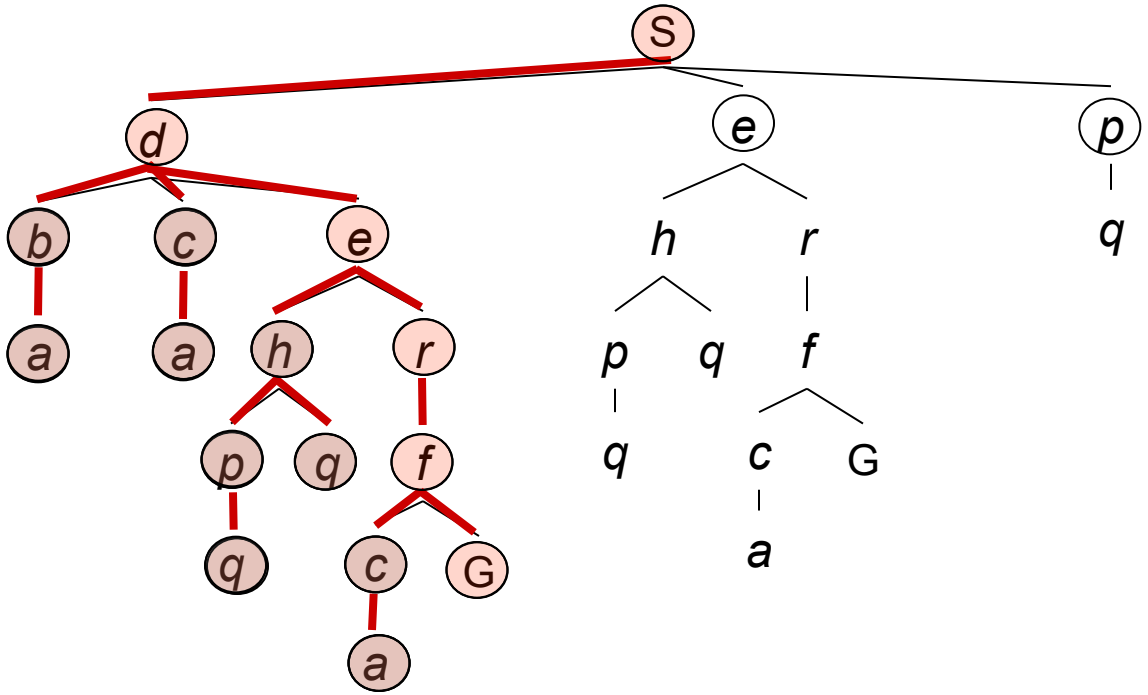
Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack





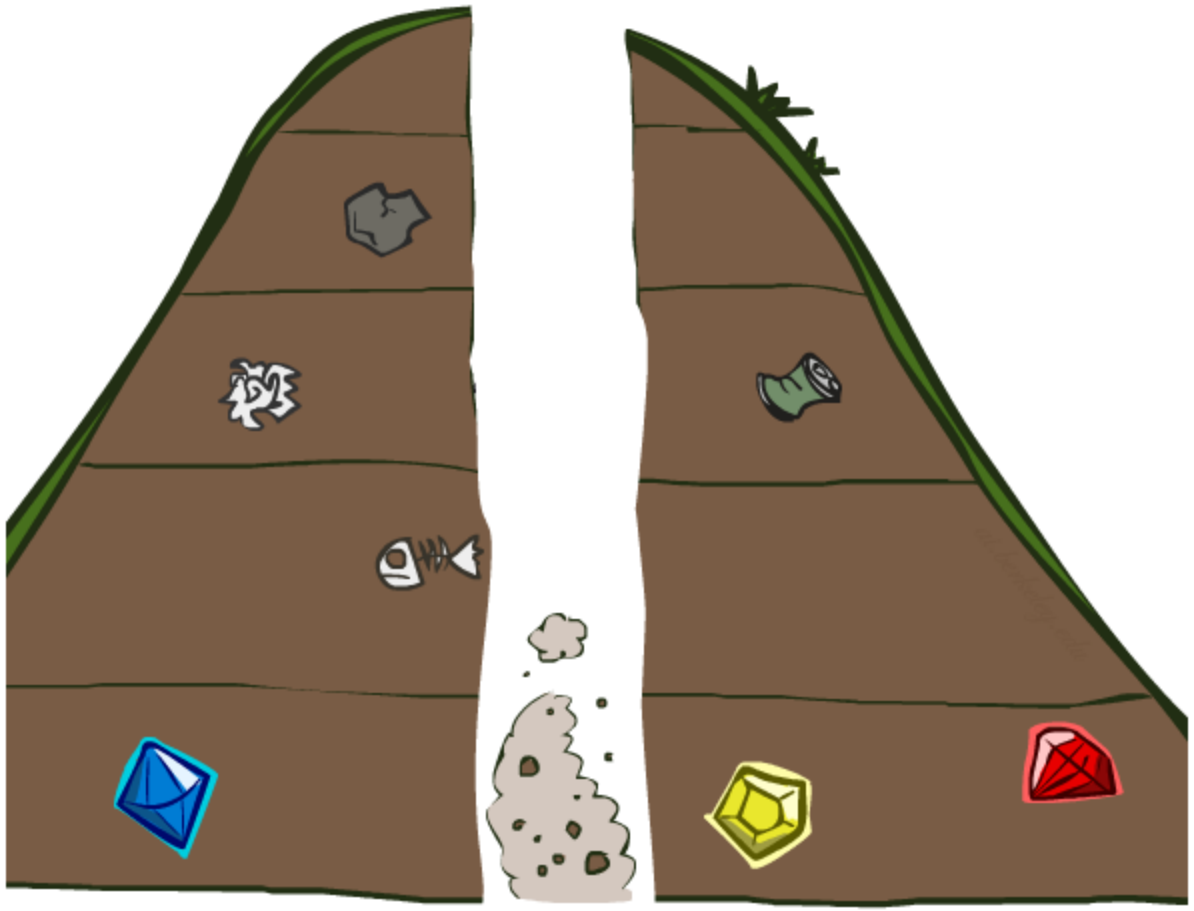
Depth-first search implementation

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- Depth-first search could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f is the negative of the depth.

Search Algorithm Properties

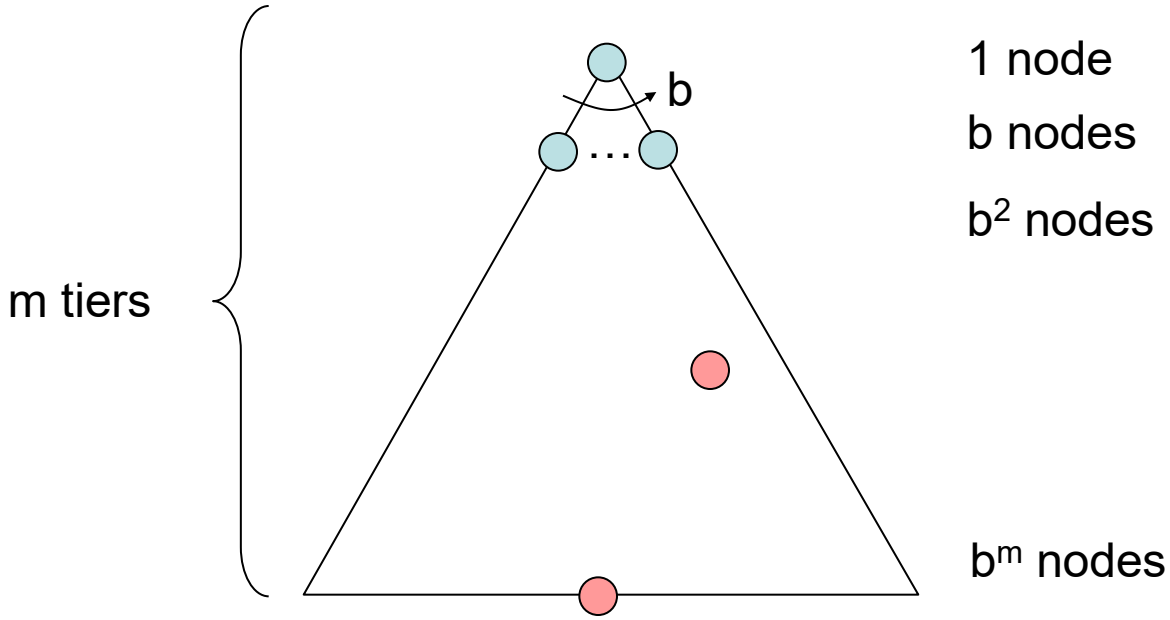


Search Algorithm Properties

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

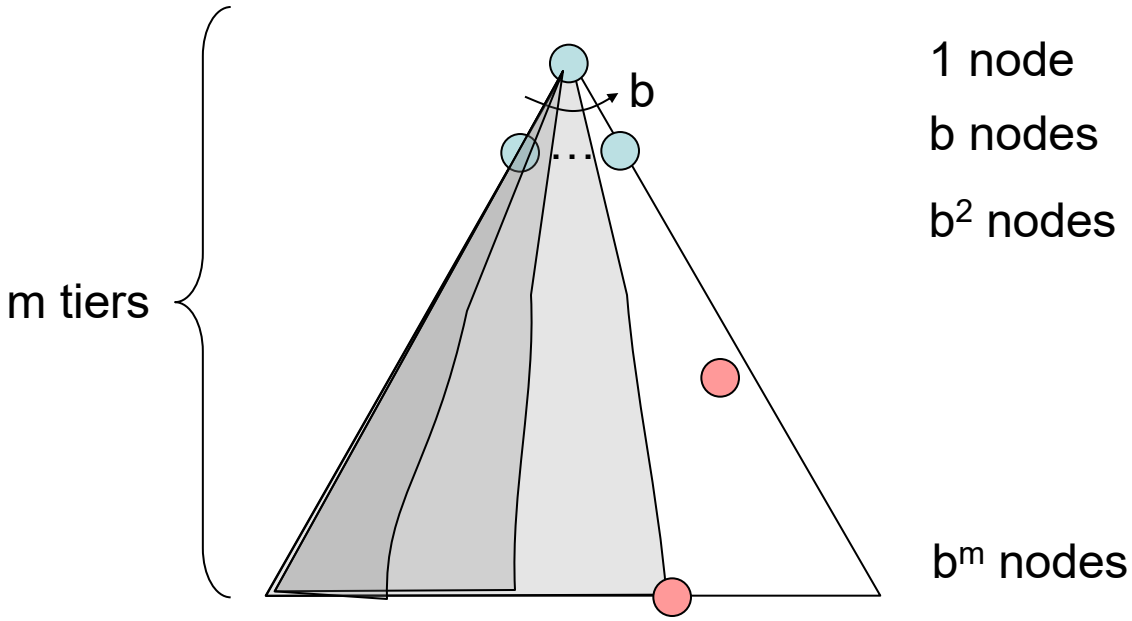
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

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

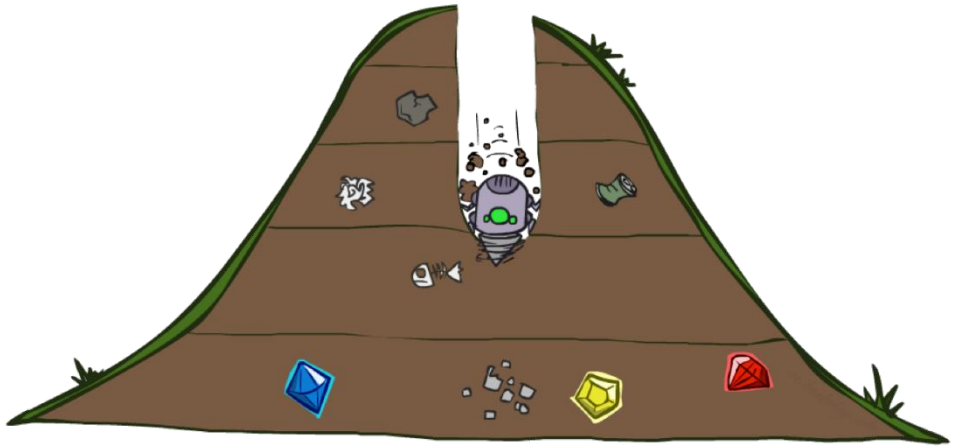


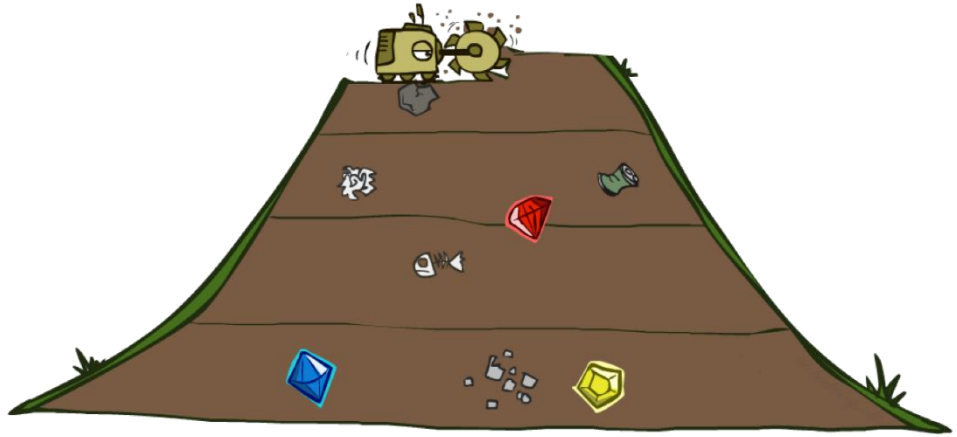
Depth-First Search (DFS) Properties

- 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?
  - No.
  - m could be infinite, so only if we prevent cycles (more later). fails in infinite-depth spaces, spaces with loops
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



Quiz: DFS vs BFS





Quiz: DFS vs BFS

- When will BFS outperform DFS?
- When will DFS outperform BFS?

- Use depth-first if
  - *Space is restricted*
  - There are many possible solutions with long paths and wrong paths are usually terminated quickly
  - Search can be fine-tuned quickly
- Use breadth-first if
  - *Possible infinite paths*
  - Some solutions have short paths
  - Can quickly discard unlikely paths

Breadth-first

- ☑ Complete,
- ☑ Optimal
- ☒ *but* uses $O(b^d)$ space

Depth-first

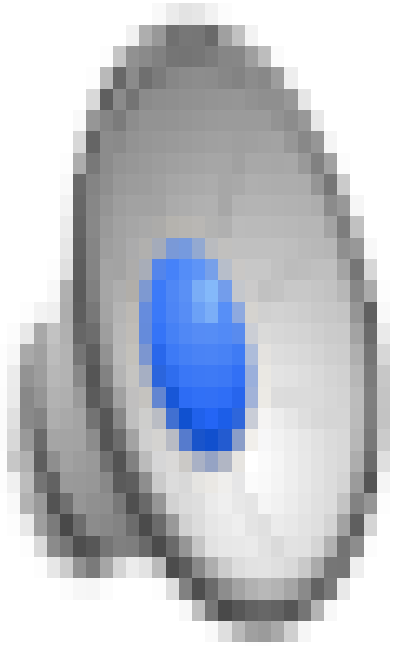
- ☒ Not complete *unless m is bounded*
- ☒ Not optimal
- ☒ Uses $O(b^m)$ time; terrible if m >> d
- ☑ *but* only uses ***O(b*m) space***

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 ∞)

[Demo: dfs/bfs maze water (L2D6)]

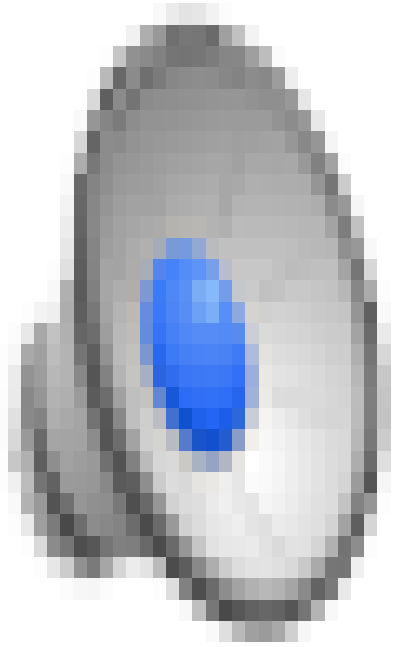
Video of Demo Maze Water DFS/BFS (part 1)

BFS



Video of Demo Maze Water DFS/BFS (part 2)

DFS



Combining BFS and DFS?

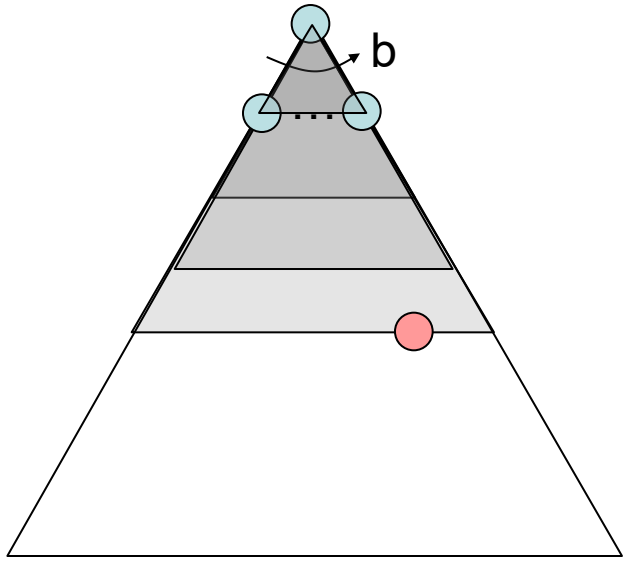
- DFS is efficient in space complexity
- BFS is better in time complexity
- How can we combine strength of both in a method?

Depth-limited search

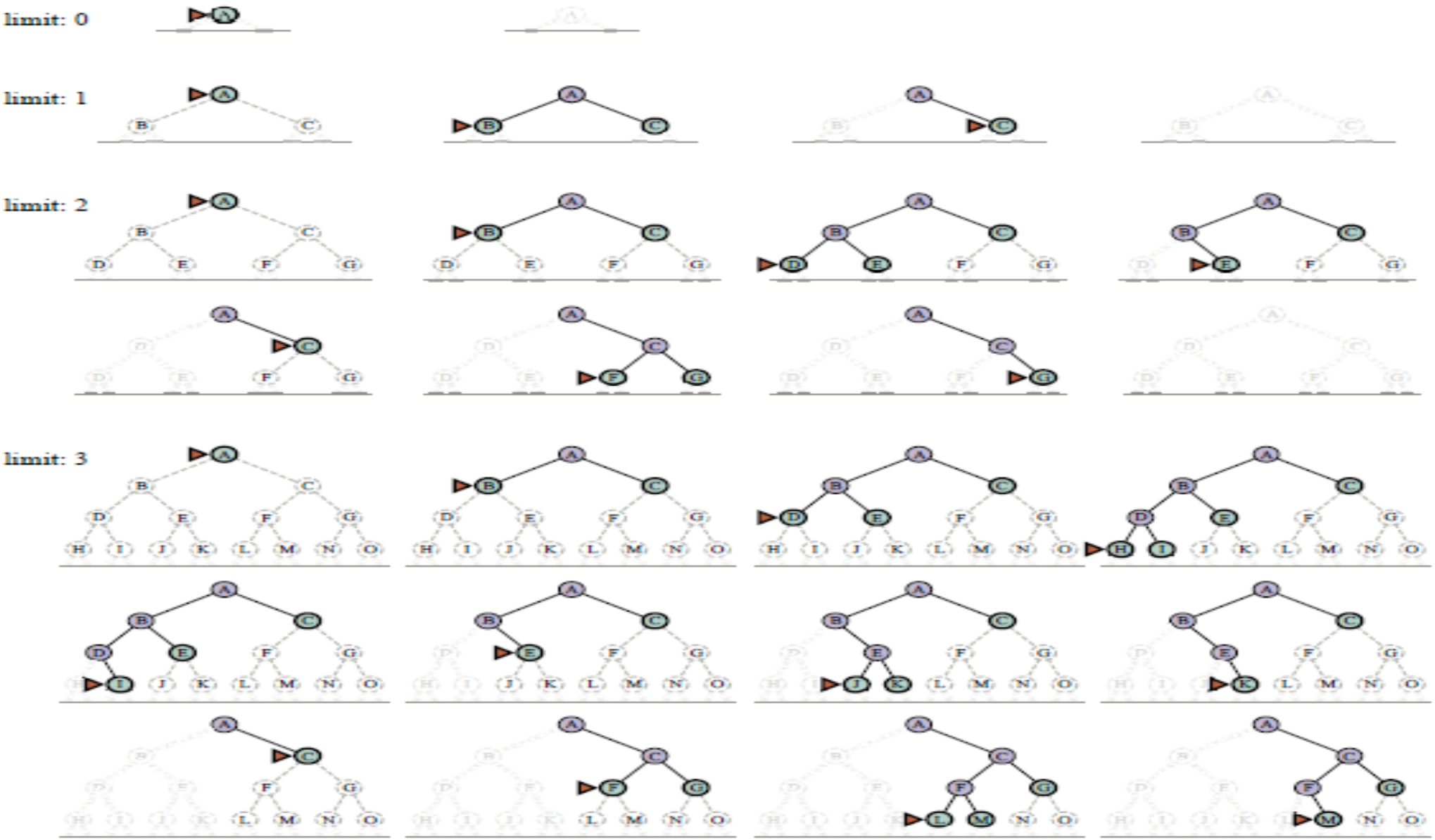
- To keep depth-first search from wandering down an infinite path, we can use depth-limited search, a version of depth-first search in which we supply a depth limit, l, and treat all nodes at depth l as if they had no successors.
- The time complexity is $O(b^l)$ and the space complexity is $O(bl)$.
- Unfortunately, if we make a poor choice for l the algorithm will fail to reach the solution, making it incomplete again.
- Sometimes a good depth limit can be chosen based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, l =19 is a valid limit. But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions.
- This number, known as the diameter of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search.

Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
- Iterative deepening search solves the problem of picking a good value for depth limit, l, by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the failure value rather than the cutoff value.
  - 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!



Four iterations of iterative deepening search for goal M on a binary tree



- Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3.
- Note the interior nodes form a single path.
- The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

Iterative deepening and depth-limited tree-like search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Properties of iterative deepening search

- **Complete**:
  - Yes
- **Time**:
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
  - or more precisely $O(b^d(1 - 1/b)^{-2})$
- **Space**:
  - O(bd)
- **Optimal**:
  - Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree

Properties of iterative deepening search (cont.)

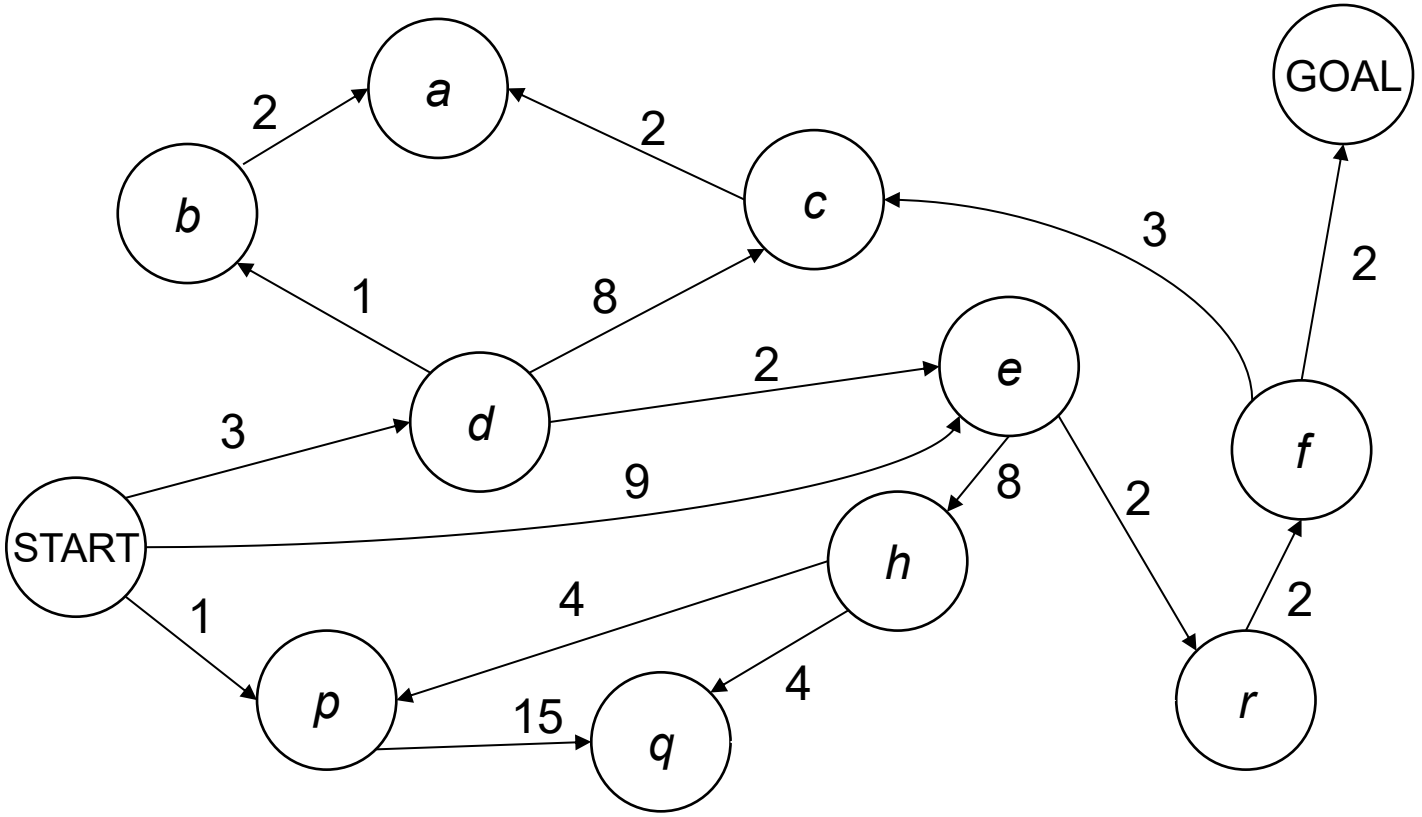
- Numerical comparison for b = 10 and d = 5, solution at far right leaf:

  N(IDS) = 6+50+400+3,000+20,000+100,000=123,456

  N(BFS) = 10+100+1,000+10,000+100,000+999,990=1,111,100

- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is generated

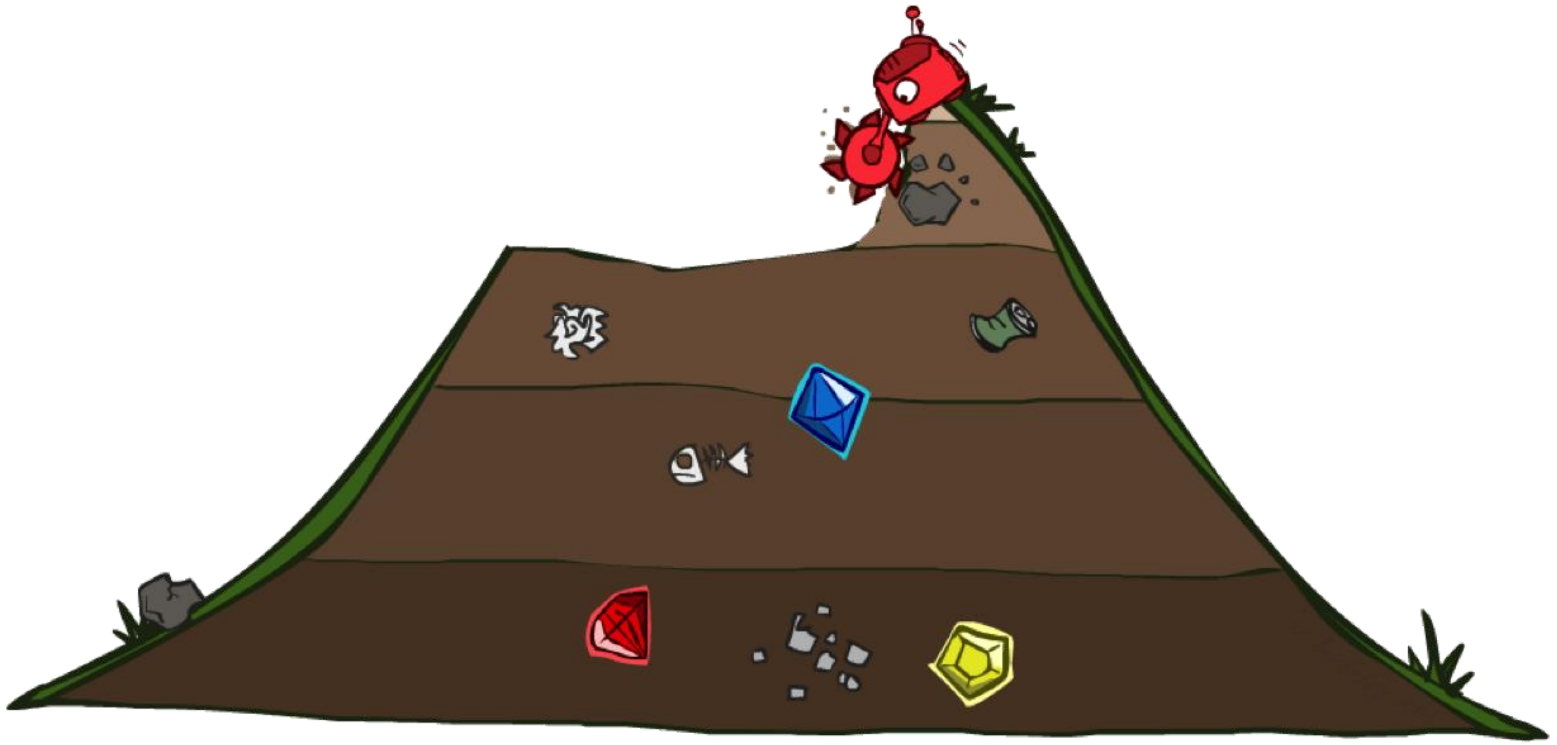
Cost-Sensitive Search



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.

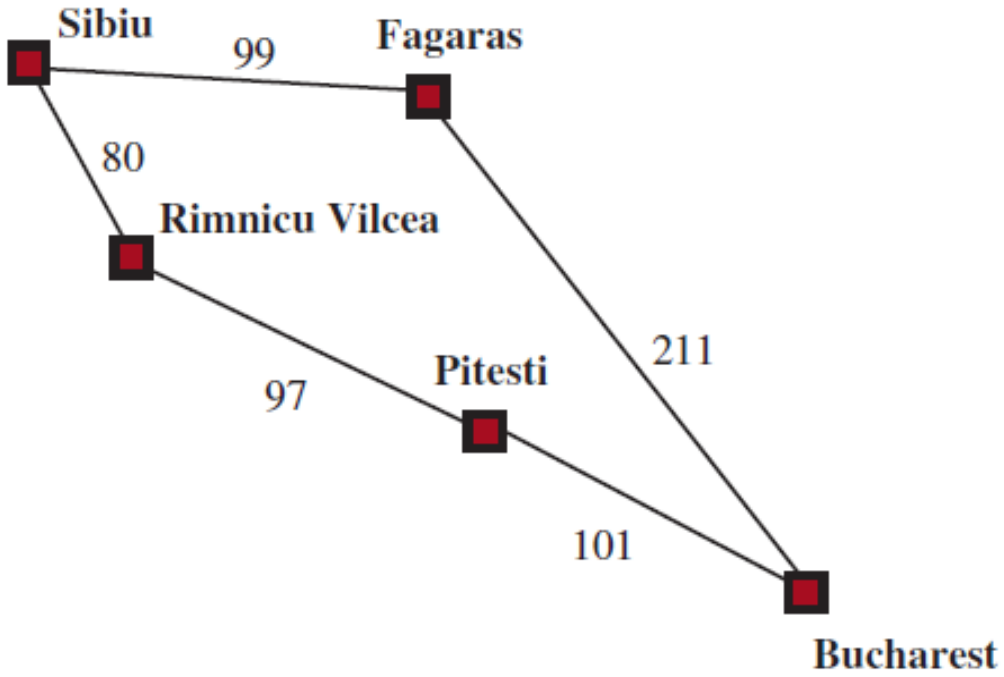
Uniform Cost Search (Dijkstra's algorithm)

- When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node.
- This is called
  - Dijkstra's algorithm by the theoretical computer science community, and
  - uniform-cost search by the AI community.



Part of the Romania state space, selected to illustrate uniform-cost search.

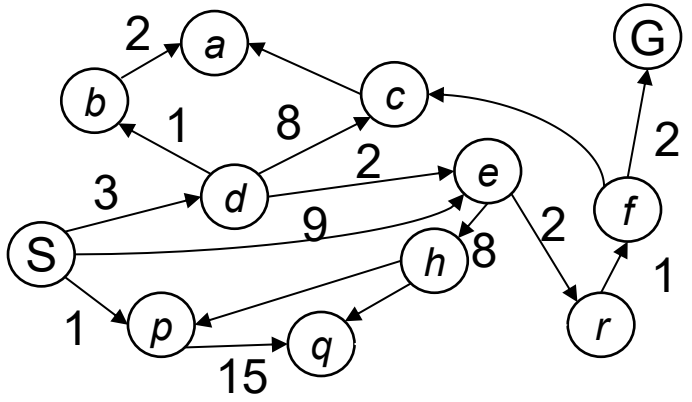
- The problem is to get from Sibiu to Bucharest.
- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.
- The least cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80+97=177.
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310.
- Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal.
- The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost 80+97+101=278.
- It has a lower cost, so it replaces the previous path in reached and is added to the frontier.
- Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path (the one through Fagaras).



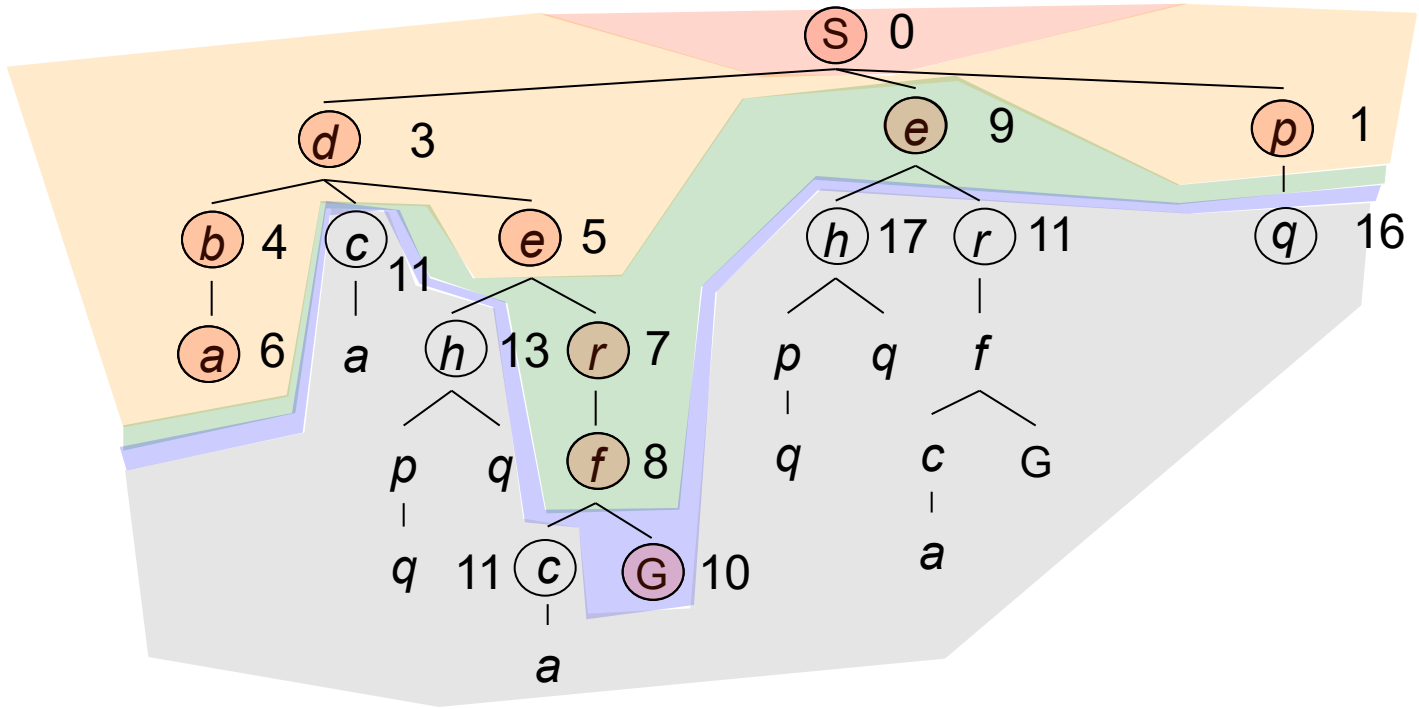
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



Cost contours



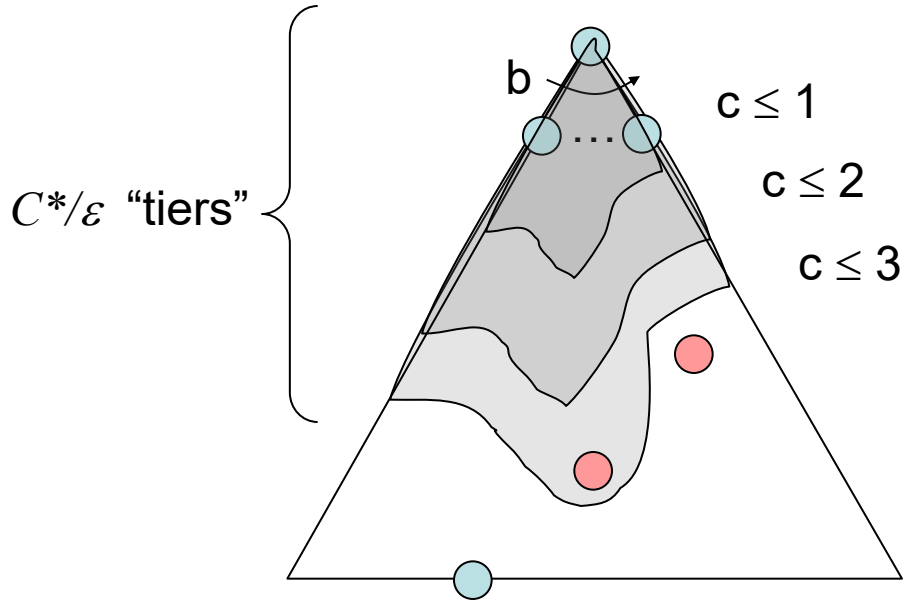
Uniform-cost search implementation

- The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on
- uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST-FIRST-SEARCH with PATH-COST as the evaluation function,

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-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 ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



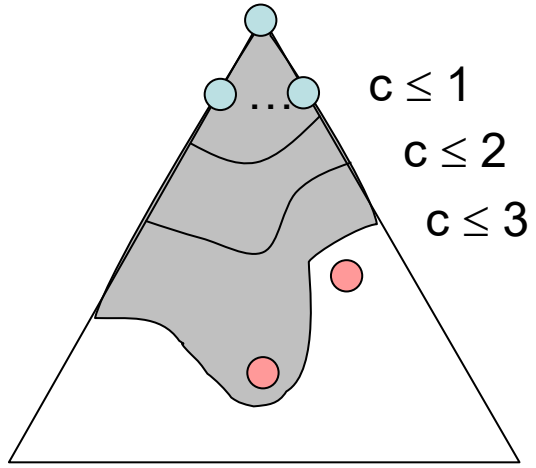
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)

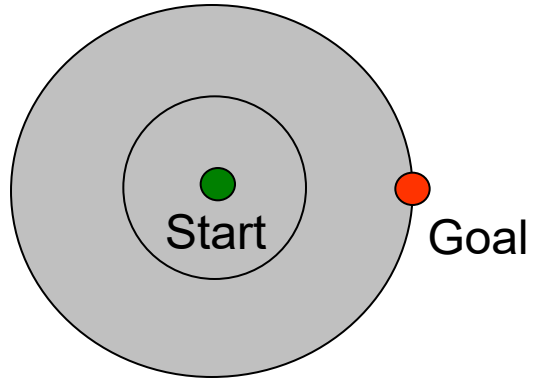
Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal! because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier.
- Uniformcost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path.

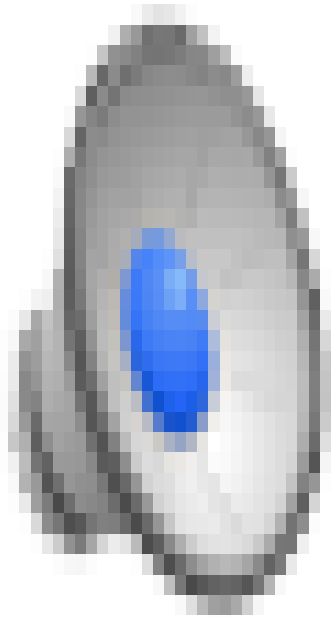
- The bad:
  - Explores options in every “direction”
  - No information about goal location





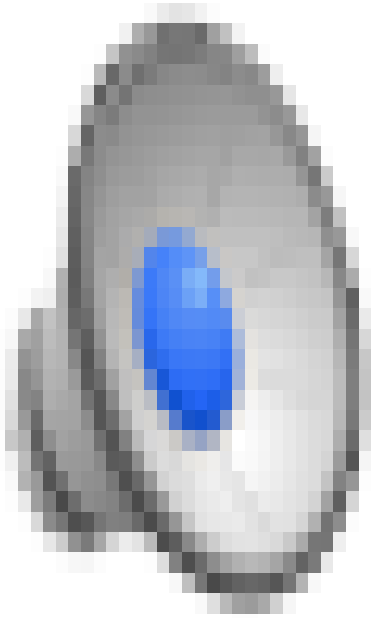
[Demo: empty grid UCS (L2D5)]
[Demo: maze with deep/shallow water DFS/BFS/UCS (L2D7)]

Video of Demo Empty UCS



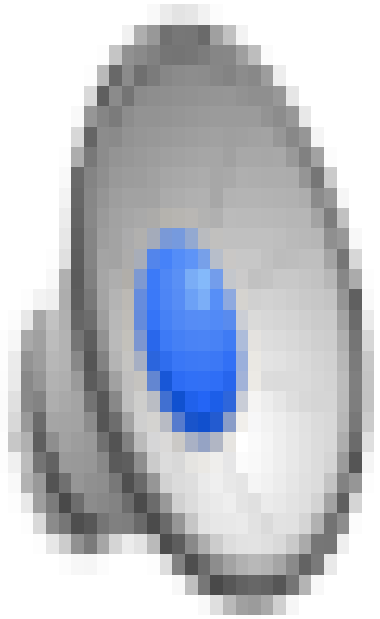
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 1)

(This is BFS.)



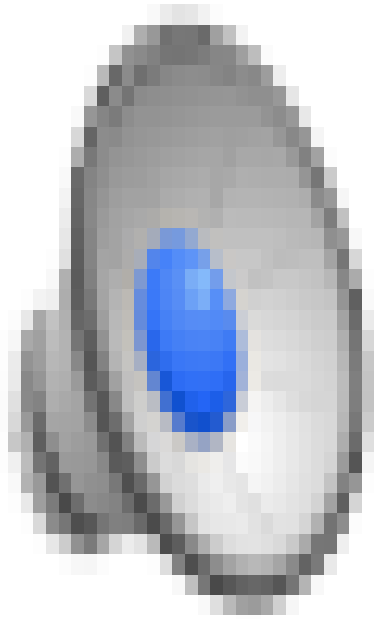
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 2)

(This is UCS.)



Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 3)

This is DFS



Bidirectional search

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

- The algorithms we have covered so far start at an initial state and can reach any one of multiple possible goal states.
- An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.

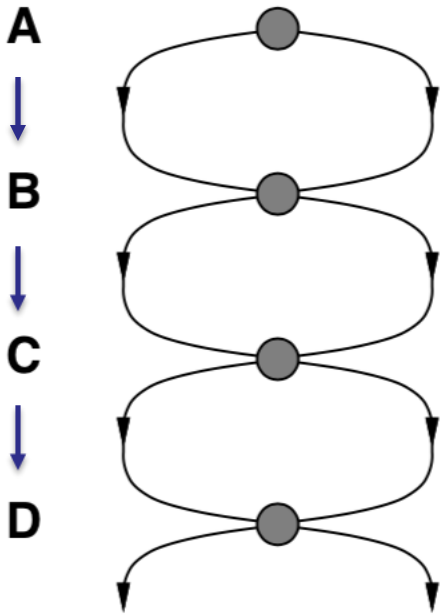
Comparing uninformed search algorithms

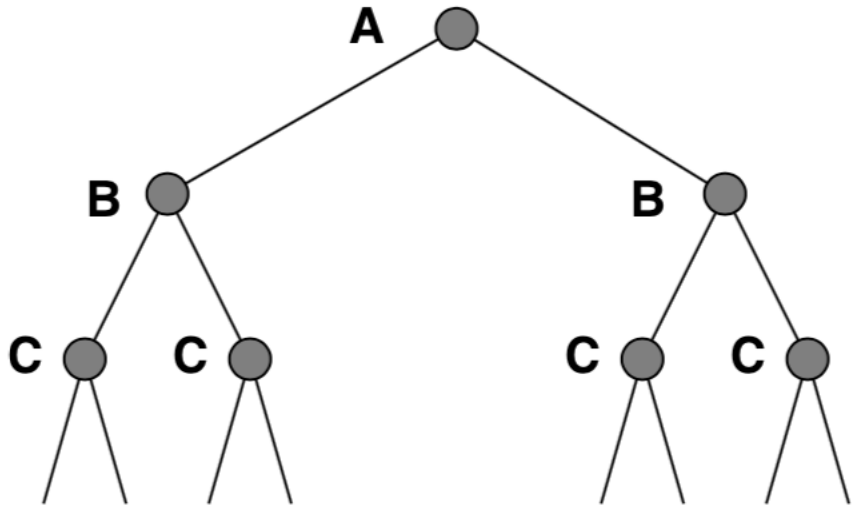
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Repeated states

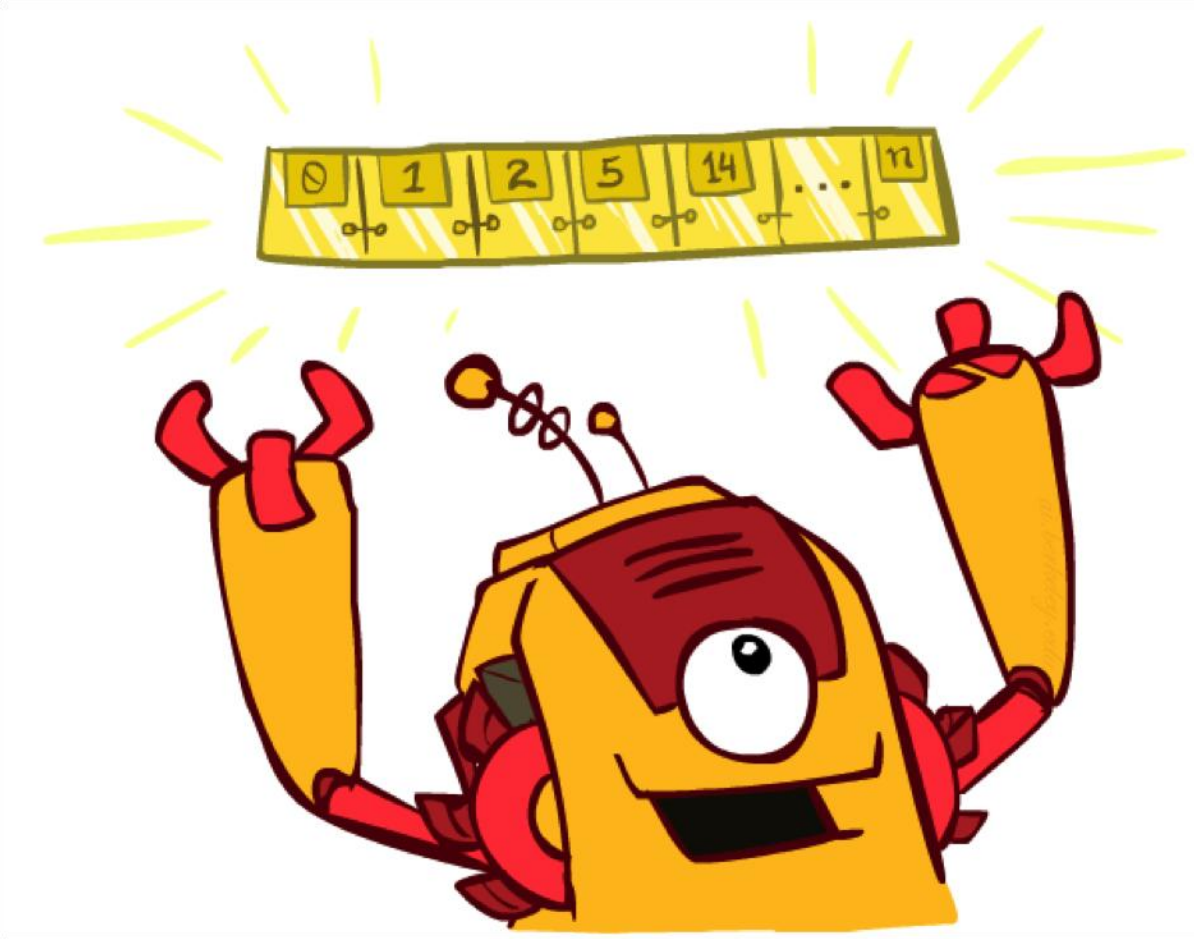
- Failure to detect repeated states can turn a linear problem into an exponential one!





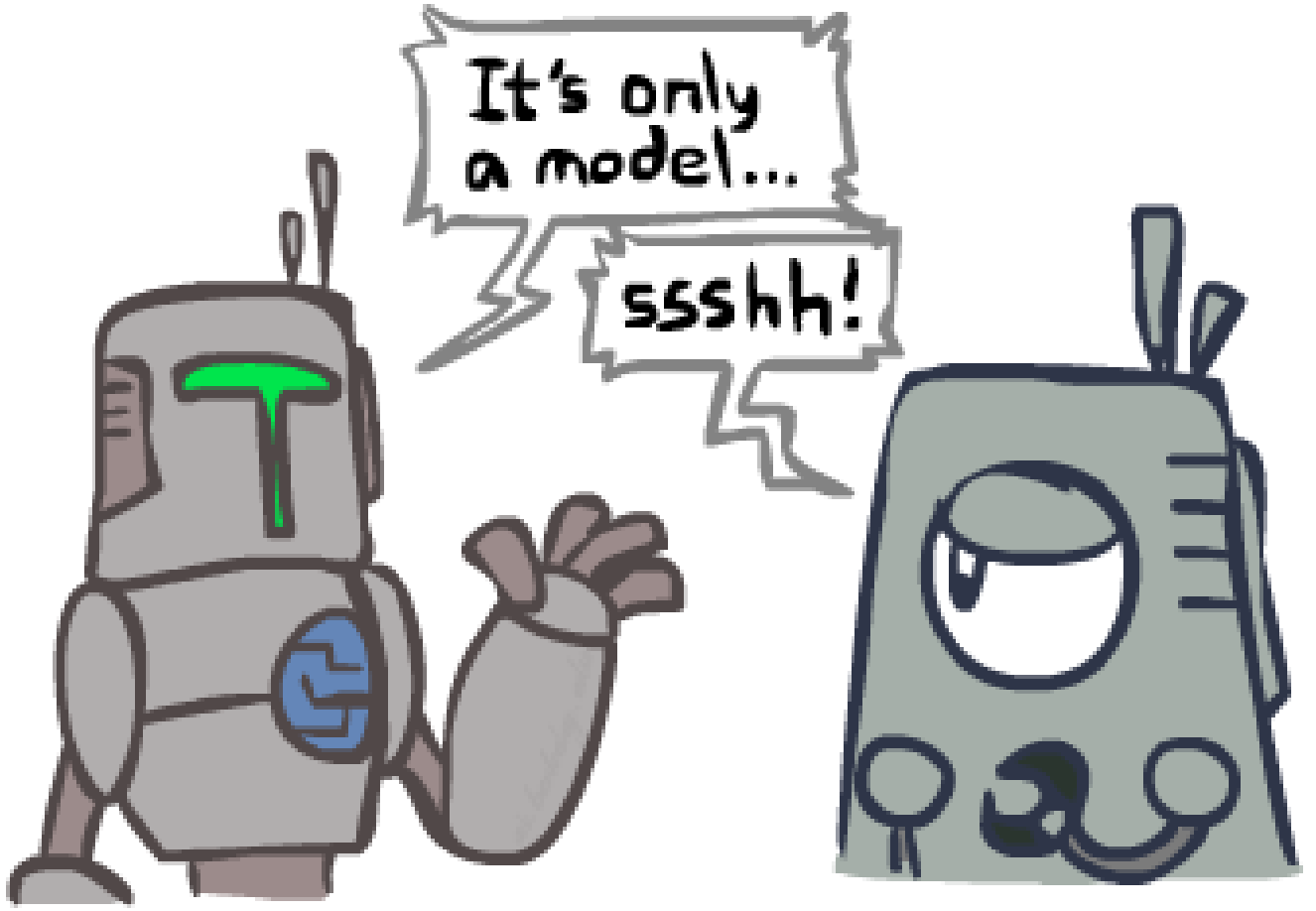
The One Queue

- 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 stacks and queues
  - Can even code one implementation that takes a variable queuing object



Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models…



Search Gone Wrong?