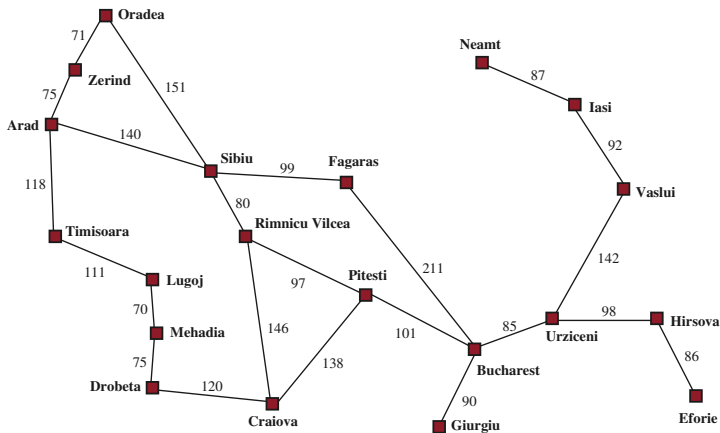


# Artificial Intelligence

## Problem Solving

Christopher Simpkins

# Problem-Solving Agents



- ▶ In this lesson we consider a *state* to be our location in one of these cities.
- ▶ A *goal* is a state in which we are located in a particular city.

This is the essence of problem solving: transforming a current state into a goal state. The first family of algorithms we'll study for problem solving are *search* algorithms.

# Problem Solving Process

To solve a problem, we

- ▶ Formulate a **goal**, e.g., “reach Bucharest”
- ▶ Formulate the **problem** as a set of states and actions that move us from one state to another.
  - ▶ Problem is a **model** – an *abstract* mathematical description.
  - ▶ Abstraction is essence and ignorance.
  - ▶ Key skill in problem formulation is finding the right **level of abstraction**.
- ▶ **Search** the possible sequences of action in our problem model that transforms our state from the current state to the goal state. A sequence of actions that gets us to the goal state is called a *solution*. May be many; pick one.
- ▶ **Execute** the actions in the solution.

# Open-Loop vs. Closed-Loop

- ▶ In an **open-loop** system the agent gets no feedback, i.e., sensor input, after executing an action.
  - ▶ If the agent's model is perfect and actions are deterministic, then the agent can operate in an open-loop fashion, simply executing the actions in the solution one after the other.
- ▶ In a **closed-loop** system gets sensory feedback after every action, so it can check whether the action had the expected effect.
  - ▶ If the environment is partially observable or actions are nondeterministic, closed-loop control is necessary.
  - ▶ Say the agent executes to **ToSibiu** action but ends up in **Zerind**. Closed-loop feedback will alert the agent to this fact so it can re-plan.

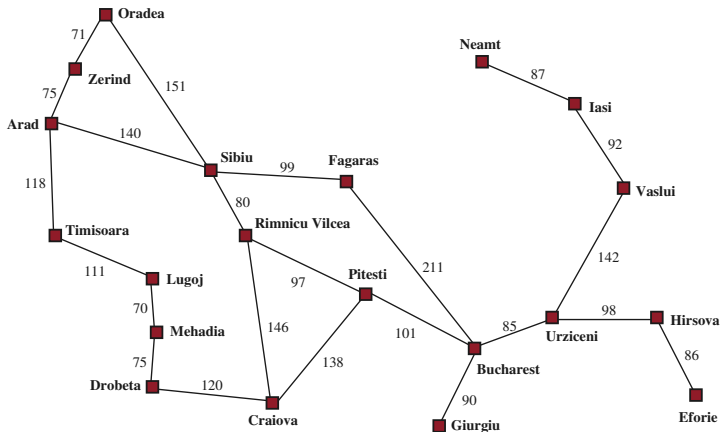
# Search Problems and Solutions

A search problem consists of:

- ▶ A set of **states**, which we call a **state space**.
- ▶ **Initial state**
- ▶ A set of **goal states**.
  - ▶ Typically use an **IS-GOAL**( $s$ ) predicate function to identify goal states.
- ▶ Sets of **actions** available in each state, **ACTION**( $s$ )
  - ▶ **ACTION**(Arad) = {ToSibiu, ToTimisoara, ToZerind}
- ▶ A **transition model**, **RESULT**( $s, a$ )
  - ▶ **RESULT**(Arad, ToZerind) = Zerind
- ▶ An **action cost function**, **ACTION-COST**( $s, a, s'$ ) or  $c(s, a, s')$  which returns the cost of executing action  $a$  in state  $s$  and reaching state  $s'$ .

# Solution

- ▶ A solution is a path from the start state to the a goal state.
- ▶ An optimal solution is a solution with lowest cost among all solutions.

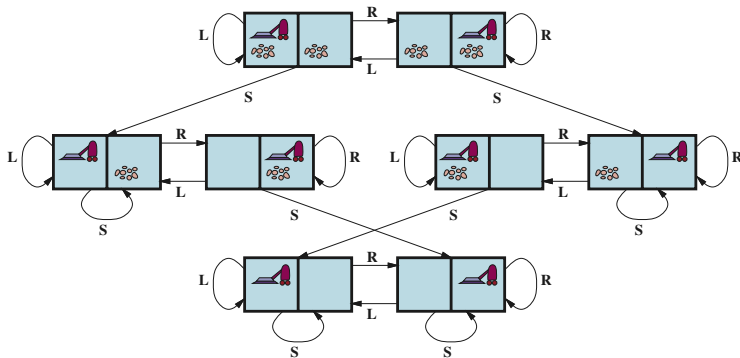


- ▶ How many paths are there from Arad to Bucharest?
- ▶ What is/are the solutions to the Arad-to-Bucharest problem (assume perfect information – fully observable, known dynamics, and deterministic actions)?

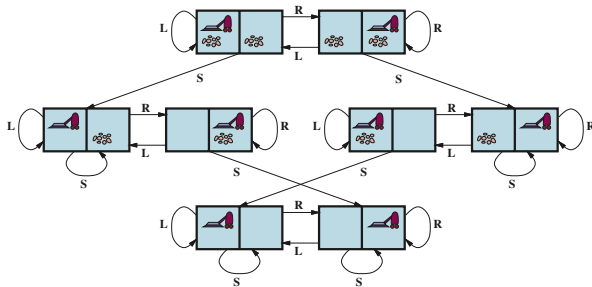
## Example Problems

- ▶ **Standardized problems** use idealized environments designed to illustrate or exercise various problem-solving methods. See, for example, [Gymnasium](#).
  - ▶ A **grid world** is a standardized environment whose states are organized as a grid, and whose actions include moving between adjacent grids.
- ▶ **Real-world problems** are formulated for specific real-world tasks, like the problem specification used for Roombas.

Here's a standardized environment for the vacuum cleaner agent, formulated as a grid world:



# Vacuum Cleaner Grid World



- ▶ **States** include both the agent's location, and characteristics of the environment. For the vacuum world, that's  $2 \cdot 2^2 = 8$  states.
- ▶ **Initial state** is an arbitrary choice of the possible states. Sometimes this choice is important.
- ▶ **Actions** for this vacuum world are **L**, **R**, and **Suck**.
  - ▶ For 2D grids we can choose between
    - ▶ **absolute** movement, like **Up** and **Right**, a.k.a., cardinal directions, or
    - ▶ **egocentric** movement, like **TurnRight**, **MoveForward**. How does this affect the state description?
- ▶ **Goal states** are those in which every location is clean.
- ▶ **Action cost** (path cost) is 1.

# Route Finding

- ▶ **States:** a location (e.g., an airport) and the time.
  - ▶ If action cost (e.g., a flight segment) depends on previous segments, fares, etc., the state must include these details.
- ▶ **Initial state:** The user's home airport.
- ▶ **Actions:** Take any flight from the current location, in any seat class, leaving after the current time, or for connecting flights, after sufficient in-airport transfer time.
- ▶ **Transition model:** The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
  - ▶ Example  $T(s, a, s')$ :  $T(S(ATL, 10:00), A(DL875), S(LGA, 12:00))$  (DL875 has a flight time of 2 hours).
- ▶ **Goal state:** A destination city. Sometimes the goal can be more complex, such as arrive at the destination on a nonstop flight. (Remember, a solution is a path, i.e., sequence of actions.)
- ▶ **Action cost:** A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

# Real-World Problems

- ▶ **Touring problems**
- ▶ **VLSI layout** – minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield
  - ▶ Cell layout – place cells on chip so they don't overlap and have room for connections
  - ▶ Channel routing – find routes for each wire between cells
- ▶ **Robot navigation**
- ▶ **Automatic assembly sequencing** – standard practice in manufacturing since the 1970s.
  - ▶ Solving some automatic assembly problems could earn you a [Nobel Prize!](#)

# Search Algorithms

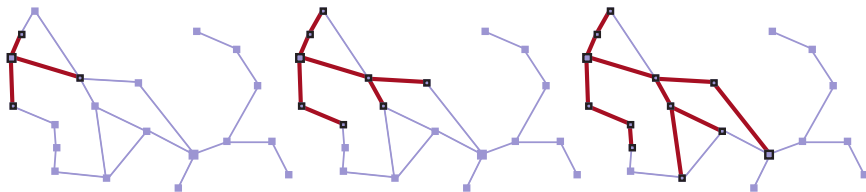
A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.

- ▶ In general, the states and actions of a problem create a state space graph.
- ▶ Here we consider algorithms that superimpose a **search tree** over the state-space graph.
- ▶ **Nodes** correspond to states, **edges** correspond to actions

Don't confuse state space with search tree.

- ▶ State space is set of states, and actions that cause transitions between states.
- ▶ Search tree describes paths between these states, reaching towards the goal(s).
  - ▶ May be many nodes for a given state, but each path from root to node is unique.

Here is a search tree being imposed on the Romania state space graph by a search algorithm.



# Elements of Search Algorithms

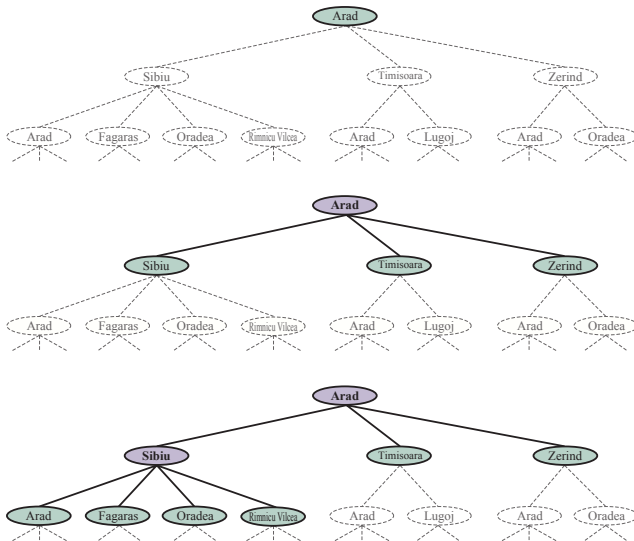
Essence of search:

- ▶ Choose a child node to consider next. “Who’s first?”
- ▶ Put aside other nodes for later. “Who’s next?”

Root node is initial state. At each node we can **expand** the node, which grows the tree, by taking actions (adding edges) that lead to successor states (generate successor/child nodes). Search algorithms must keep track of:

- ▶ *Expanded* nodes. We test expanded nodes before dealing with frontier.
- ▶ *Frontier* nodes, which are generated but not yet expanded.
  - ▶  $Reached = Expanded + Frontier$

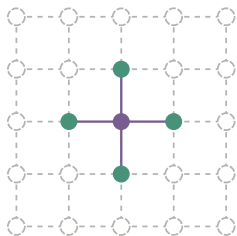
# Search Progression



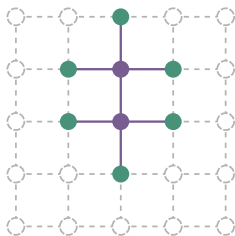
- ▶ *Expanded* nodes are lavender with bold letters.
- ▶ *Frontier* nodes are green with normal weight font.
- ▶ Nodes in dashed-line ovals are candidates for expansion.

# Separation Property of Graph Search

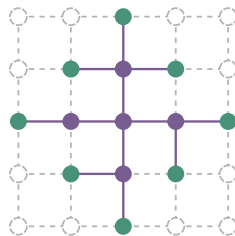
The **frontier** separates the interior region of expanded nodes from the exterior region of unexpanded nodes.



(a)



(b)



(c)

Frontier is in green. Interior is lavender. Exterior is faint dashed.

- ▶ (a) Only root expanded.
- ▶ (b) Top frontier node expanded.
- ▶ (c) Remaining successors of root expanded in clockwise order.

## Implementation Note: The `yield` statement

A function containing a `yield` statement is a **generator**. Use a generator to turn a data generating process into an iterator. Node expansion is a data generating process.

```
1 In [36]: def by_twos(start: int, end: int):
2     ...:     x = start
3     ...:     while x < end:
4     ...:         yield x
5     ...:         x += 2
6     ...:
7
8 In [37]: by_twos(1, 9)
9 Out[37]: <generator object by_twos at 0x109010ee0>
10
11 In [38]: list(Out[37])
12 Out[38]: [1, 3, 5, 7]
13
14 In [39]: for x in by_twos(1, 10):
15     ...:     print(f"{x=}")
16     ...:
17 x=1
18 x=3
19 x=5
20 x=7
21 x=9
```

# Search Data Structures

Node:

- ▶ `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  $g(\text{node})$  as a synonym for PATH-COST.

Frontier is a **queue** with operations:

- ▶ `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.

Queues 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 recently added node; we shall see it is used in depth-first search.

# Best-First Search

Best-first search is an abstract search algorithm. Name can be tricky to understand.

- ▶ *Best* way to pick the *first* node to consider next.
- ▶ We use a generalization of queues, called a *priority queue*, to store the *frontier*.
- ▶ An evaluation function,  $f(node)$ , imposes an ordering on the nodes in the priority queue.

*The evaluation function considers the path to the node, not any property of the node itself. Remember, a solution to a search problem is characterized by the path from the root to the goal, not some characteristic of the goal.*

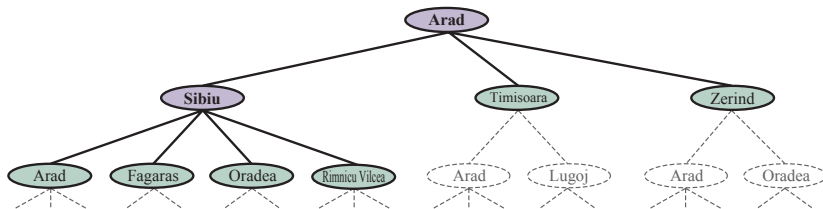
We'll now describe several uninformed search algorithms. I recommend you also look at their [implementations in Python](#), which may be easier to follow.

## Best-First Search Algorithm

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

**function** EXPAND(*problem*, *node*) **yields** nodes  
   $s \leftarrow \text{node.STATE}$   
  **for each** *action* **in** *problem.ACTIONS*(*s*) **do**  
     $s' \leftarrow \text{problem.RESULT}(s, \text{action})$   
     $\text{cost} \leftarrow \text{node.PATH-COST} + \text{problem.ACTION-COST}(s, \text{action}, s')$   
    **yield** NODE(*STATE*=*s'*, *PARENT*=*node*, *ACTION*=*action*, *PATH-COST*=*cost*)

# Redundant Paths



In the path from **Arad** to **Sibiu** to **Arad**,

- ▶ **Arad** is a **repeated state** and
- ▶ the path is a **cycle**, or **loopy path**.

Cycle special case of **redundant path**: multiple paths to the same state. Three approaches:

1. Remember reaches states, like best-first search. Best when reached states fits in memory.
2. Don't worry about repeated states. Works when repeated states rare or impossible.
  - ▶ **Graph search** checks for redundant paths, which occur in graphs in general.
  - ▶ **Tree-like search** does not check for redundant paths, since trees are acyclic graphs.
3. Only check for cycles, not other kinds of redundant paths.
  - ▶ E.g., search path in reverse

# Measuring Problem-Solving Performance

- ▶ **Completeness:** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
  - ▶ Complete search algorithms must be **systematic**.
  - ▶ Easier to achieve for finite state spaces.
  - ▶ In an infinite state space with no solution, search won't terminate.
- ▶ **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?
- ▶ **Time complexity:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
- ▶ **Space complexity:** How much memory is needed to perform the search?

For *explicit* graphs, like Romania, time and space complexity typically expressed in terms of number of vertices (state nodes),  $|V|$ , and number of edges,  $|E|$  (state-action pairs, which generate  $((s, a, s')$  triples).

For *implicit* state space graphs we characterize time and space complexity in terms of depth,  $d$  (number of actions in an optimal solution), and branching factor,  $b$  (number of successor nodes per node). For most of our discussions, we'll use this characterization.

# Uninformed Search Strategies

Uninformed search strategies have no information about which actions are better for reaching a goal. In these cases we can only do systematic searches of the state space. We'll discuss

- ▶ Breadth-first search
- ▶ Uniform-Cost search (Dijkstra's algorithm)
- ▶ Depth-first search
- ▶ Depth-limited searchand
- ▶ Iterative deepening search.
- ▶ Bidirectional search

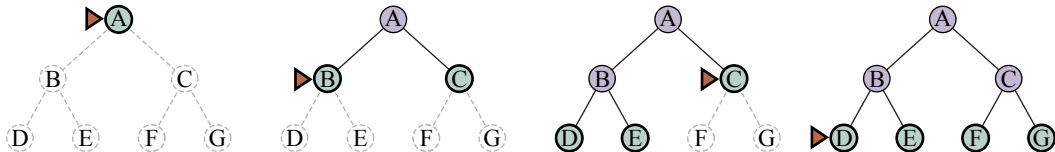
# Breadth-First Search

- ▶ Good when path costs are uniform.
- ▶ Equivalent to best-first search where  $f(\text{node})$  is the depth of the node
- ▶ Guaranteed to find minimal number of actions because it evaluates depth  $d$  before generating depth  $d + 1$ .

But three optimizations afforded by the BFS algorithm and uniform path costs:

- ▶ FIFO queue instead of priority queue
- ▶ *Reached* is a set instead of a mapping  $S \rightarrow \text{Node}$ 
  - ▶ With uniform path costs, as soon as BFS finds a node, it's the fastest way to it.
- ▶ **Early goal test** – as soon as we expand a node, we can test it.

# BFS Algorithm



**function** BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

*node*  $\leftarrow$  NODE(*problem*.INITIAL)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

*frontier*  $\leftarrow$  a FIFO queue, with *node* as an element

*reached*  $\leftarrow$  {*problem*.INITIAL}

**while not** IS-EMPTY(*frontier*) **do**

*node*  $\leftarrow$  POP(*frontier*)

**for each** *child* **in** EXPAND(*problem*, *node*) **do**

*s*  $\leftarrow$  *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*

# Analysis of BFS

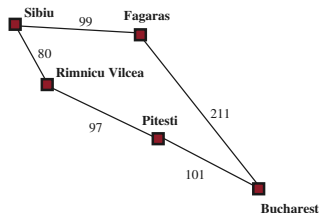
- ▶ Complete, because it generates all nodes at each depth.
- ▶ Time complexity: at each level,  $b$  nodes for each  $b$  predecessors, so
  - ▶  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$
- ▶ Space complexity:  $O(b^d)$  because all nodes are stored while the search proceeds.

Uninformed search is not appropriate for exponential complexity problems except for smallest instances. Assuming your computer can process 1 million nodes per second and store each node in 1 Kb,

- ▶ For a problem with  $b = 10$  and  $d = 10$ , how long will it take search and how much space will be required?
- ▶ Same problem, but with  $d = 14$ ?

# Uniform-Cost Search (Dijkstra's Algorithm)

BFS where the best-first  
 $f(\text{node})$  is the path cost to  
the current node.



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

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

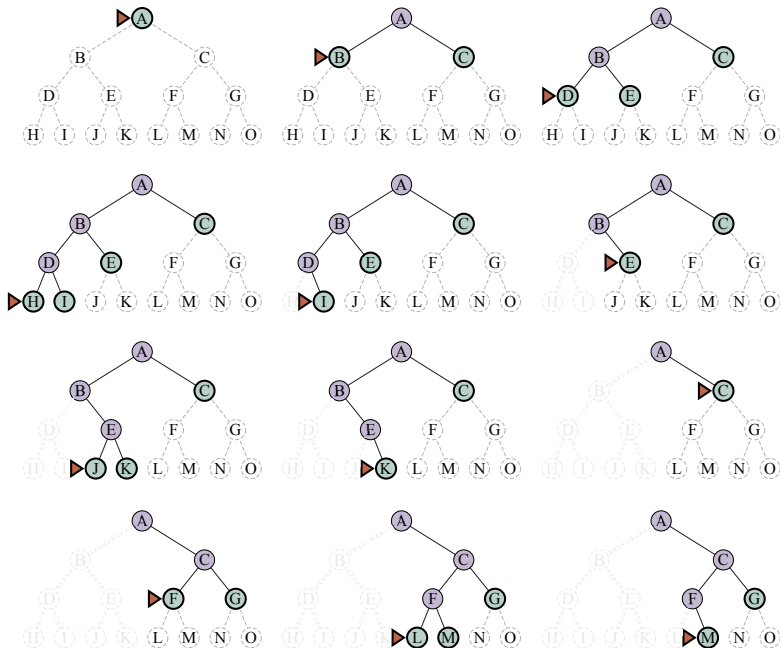
```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure  
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

# Analysis of Uniform-Cost Search

Let  $C^*$  be the cost of the optimal solution and  $\epsilon > 0$  be a lower bound on the cost of each action.

- ▶ Complete, like BFS
- ▶ Cost-optimal, because a solution will be at least as low cost as any other in the frontier.
- ▶ Time and space complexity are  $O(b^{1+\lceil \frac{C^*}{\epsilon} \rceil})$ .
  - ▶ Since lower cost paths are always explored first, even when a higher cost path might be the one to lead to an optimal solution, can be worse than BFS.
  - ▶ If all action costs equal, then it's like BFS,  $O(b^{1+d})$ .

# Depth-First Search - FIFO Frontier



# Analysis of DFS

- ▶ Not cost-optimal – returns first solution it finds
- ▶ For state space that are finite trees:
  - ▶ Complete
  - ▶ Time  $O(n)$  where  $n$  is number of states
  - ▶ Space:  $O(bm)$ , where  $b$  is branching factor and  $m$  is max depth of tree.
- ▶ For (acyclic) graph state spaces, may expand same state via multiple paths.
  - ▶ For cyclic graph state spaces, need to check for cycles to prevent infinite loops.
- ▶ For infinite state spaces, not complete – may get stuck in an infinite subtree.

Why bother with DFS at all? **Memory efficiency**

# Depth-Limited Search

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

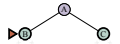
- ▶ If limit,  $\ell$ , too small, won't find goal.
- ▶ To guarantee completeness, choose  $\ell \geq \text{diameter}$ 
  - ▶ Diameter of a state space graph: maximum number of actions necessary to transition from any state to any other state.

# Iterative Deepening Search

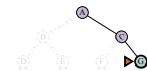
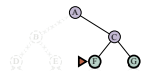
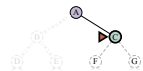
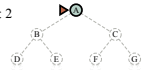
limit: 0



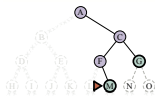
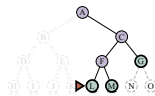
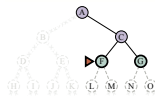
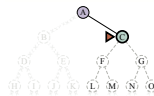
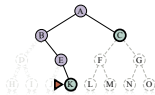
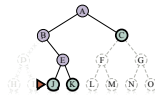
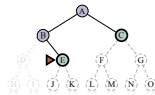
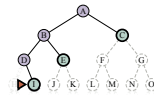
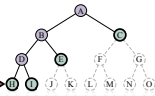
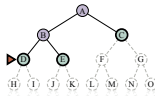
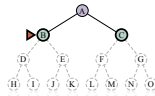
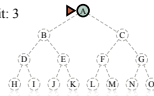
limit: 1



limit: 2



limit: 3



# Analysis of Iterative Deepening Search

- ▶ Cost-optimal for state spaces where all actions have same cost.
- ▶ For state space that are finite trees, where  $b$  is branching factor and  $m$  is max depth of tree:
  - ▶ Complete for finite acyclic spaces, or finite cyclic spaces with cycle checking
  - ▶ Space:  $O(bd)$  if there is a solution,  $O(bm)$  if no solution,
  - ▶ Time  $O(b^d)$  if there is a solution,  $O(b^m)$  if no solution.
    - ▶  $N(IDS) = (d)b^1 + (d-1)b^2 + \dots + b^d$
- ▶ For (acyclic) graph state spaces, may expand same state via multiple paths.
  - ▶ For cyclic graph state spaces, need to check for cycles to prevent infinite loops.
- ▶ For infinite state spaces, not complete – may get stuck in an infinite subtree.

*In general, iterative deepening search is the preferred uninformed search method when the search state space is larger than can fit in memory and The depth of the solution is not known.*

# Bidirectional Best-First Search

```
function BiBF-SEARCH( $problem_F, f_F, problem_B, f_B$ ) returns a solution node, or failure
   $node_F \leftarrow \text{NODE}(problem_F.INITIAL)$  // Node for a start state
   $node_B \leftarrow \text{NODE}(problem_B.INITIAL)$  // Node for a goal state
   $frontier_F \leftarrow$  a priority queue ordered by  $f_F$ , with  $node_F$  as an element
   $frontier_B \leftarrow$  a priority queue ordered by  $f_B$ , with  $node_B$  as an element
   $reached_F \leftarrow$  a lookup table, with one key  $node_F.STATE$  and value  $node_F$ 
   $reached_B \leftarrow$  a lookup table, with one key  $node_B.STATE$  and value  $node_B$ 
   $solution \leftarrow failure$ 
  while not TERMINATED( $solution, frontier_F, frontier_B$ ) do
    if  $f_F(\text{TOP}(frontier_F)) < f_B(\text{TOP}(frontier_B))$  then
       $solution \leftarrow \text{PROCEED}(F, problem_F, frontier_F, reached_F, reached_B, solution)$ 
    else  $solution \leftarrow \text{PROCEED}(B, problem_B, frontier_B, reached_B, reached_F, solution)$ 
  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.
   $node \leftarrow \text{POP}(frontier)$ 
  for each child in EXPAND( $problem, node$ ) do
     $s \leftarrow child.STATE$ 
    if  $s$  not in  $reached$  or  $\text{PATH-COST}(child) < \text{PATH-COST}(reached[s])$  then
       $reached[s] \leftarrow child$ 
      add  $child$  to  $frontier$ 
    if  $s$  is in  $reached_2$  then
       $solution_2 \leftarrow \text{JOIN-NODES}(dir, child, reached_2[s])$ 
      if  $\text{PATH-COST}(solution_2) < \text{PATH-COST}(solution)$  then
         $solution \leftarrow solution_2$ 
  return  $solution$ 
```

Motivation:  $b^{\frac{d}{2}} + b^{\frac{d}{2}} \ll b^d$ .

# Comparing Uninformed Search Algorithms

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

Notes:

- ▶ 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