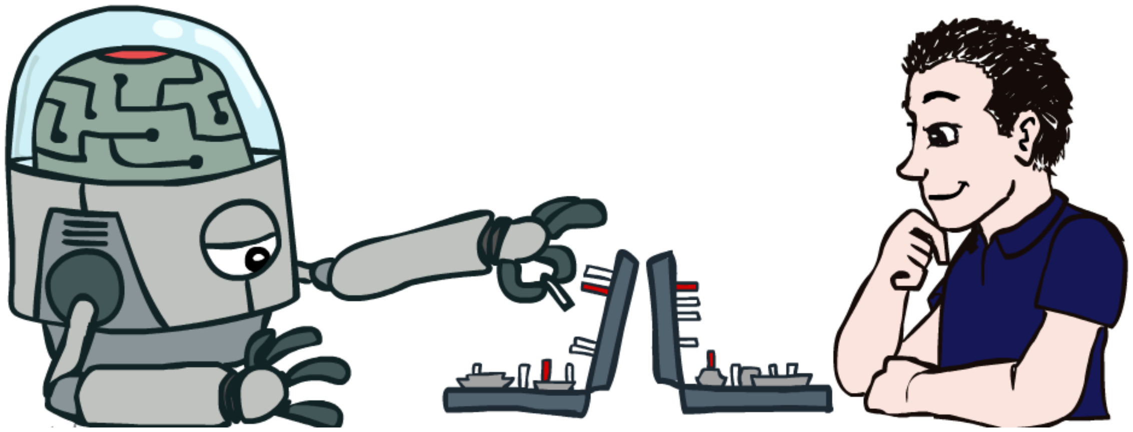
ARTIFICIAL INTELLIGENCE- CS411

Prof. Alaa Sagheer



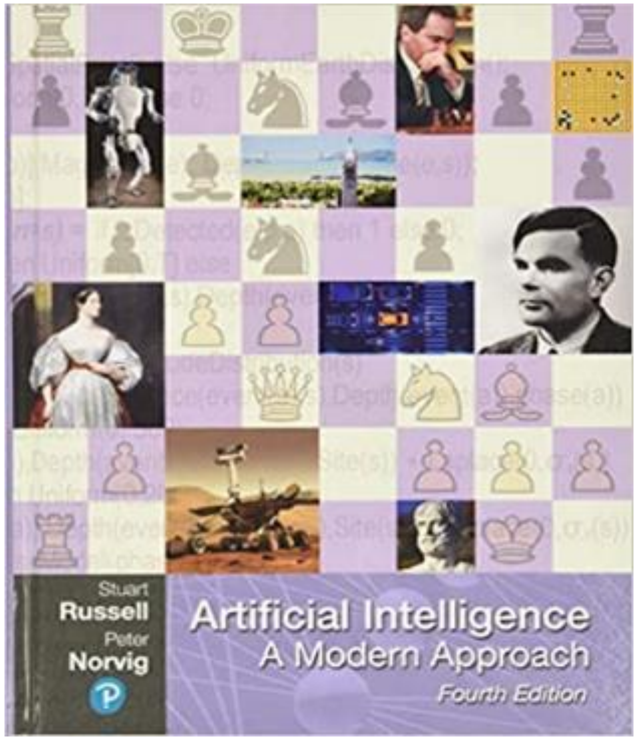
Artificial Intelligence "CS 411"

- **Textbook**:

S. Russell and P. Norvig

Artificial Intelligence: A Modern Approach

Prentice Hall, 2020, *Fourth Edition*



- **Place:** Male and Female Campus
- **Grading:**

  Homework & Activities (5%),

  Project @ Lab (10%),

  Quizzes @ Class (10%),

  Quizzes @ Lab (15 %),

  Mid-term exam (20%),

  Final exam (40%),

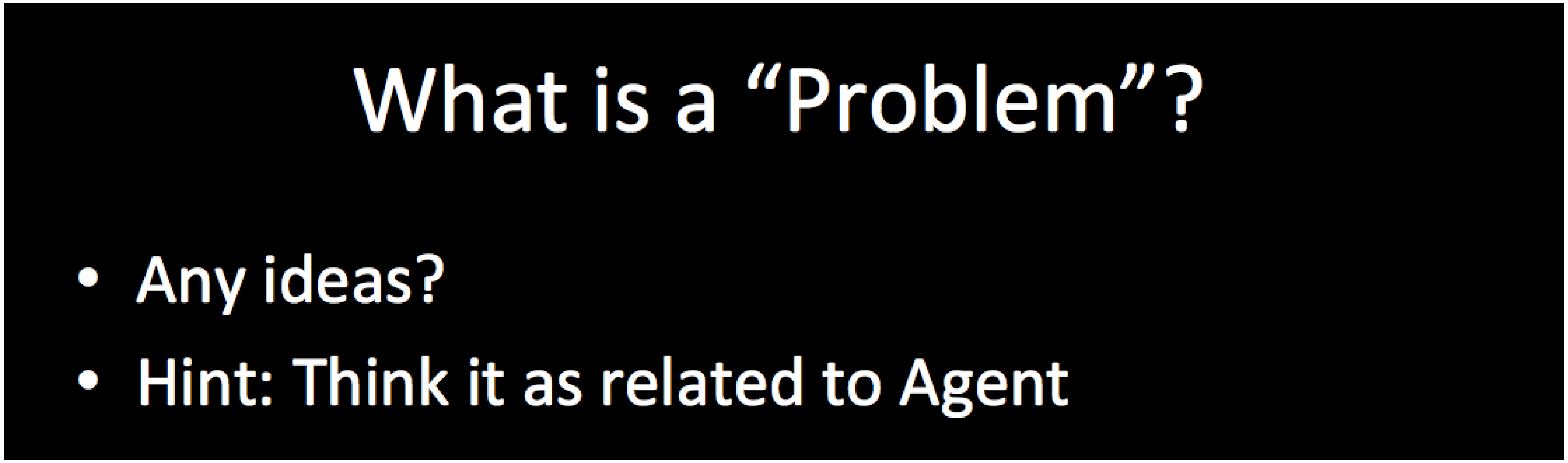
Problem Solving

Introduction

“Graph and Tree”

Lecture I

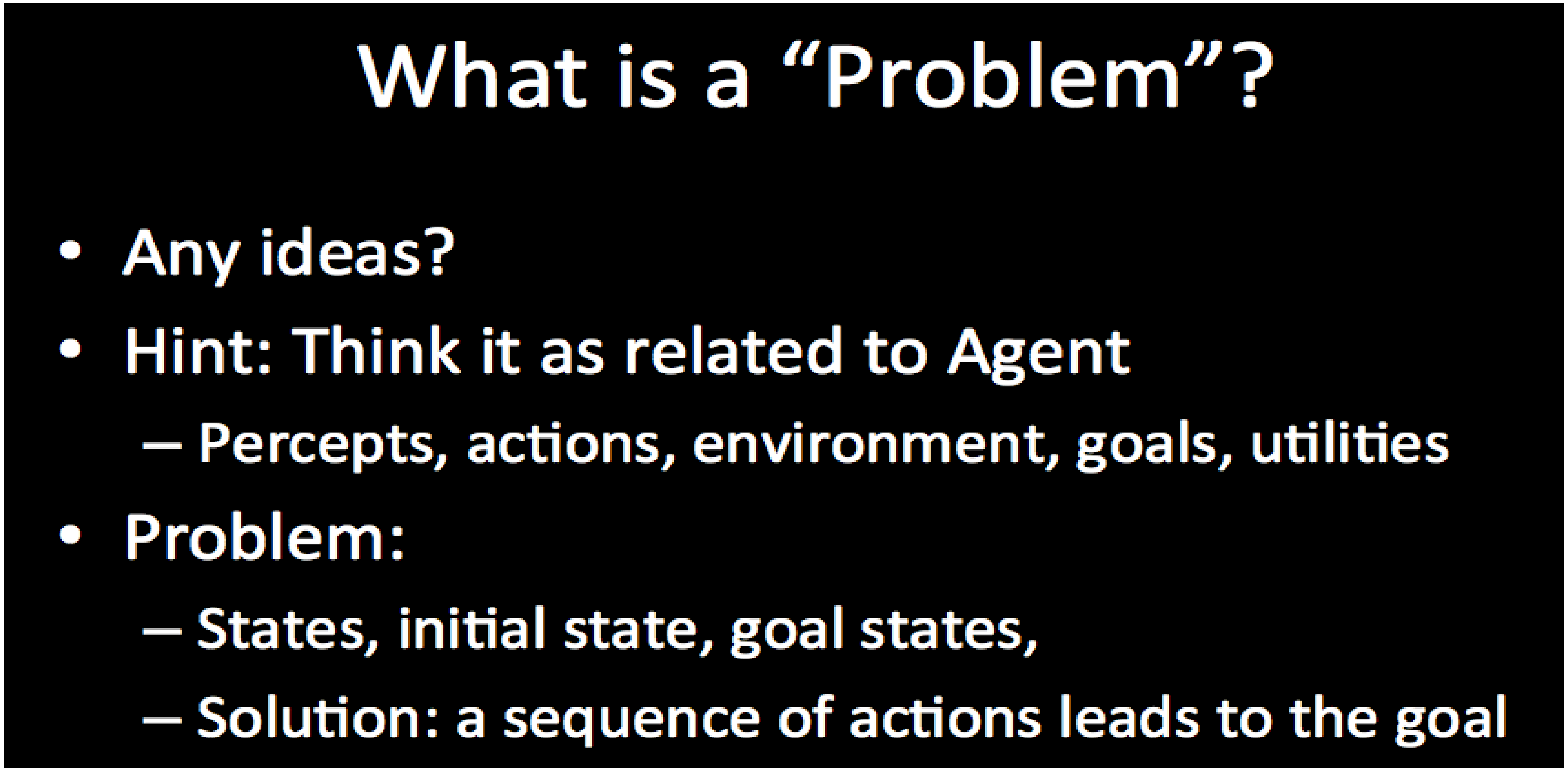
A Problem?



What is a "Problem"?

- Any ideas?
- Hint: Think it as related to Agent

A Problem?

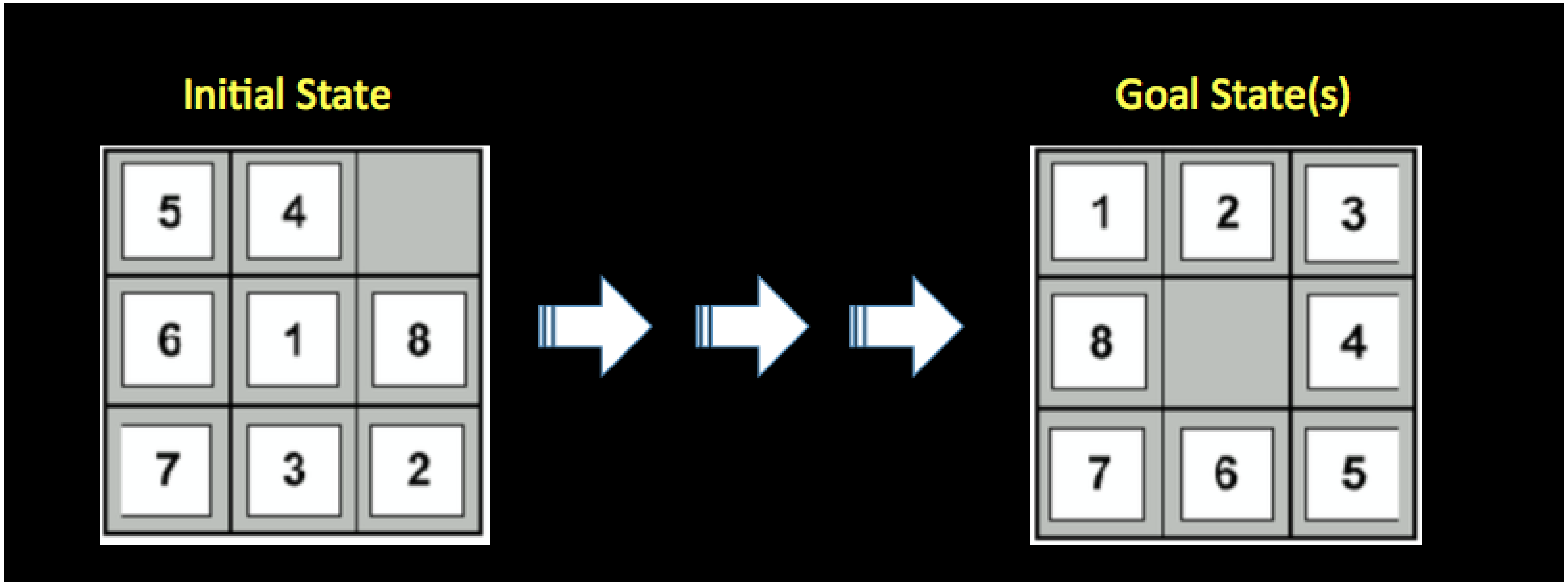


What is a "Problem"?

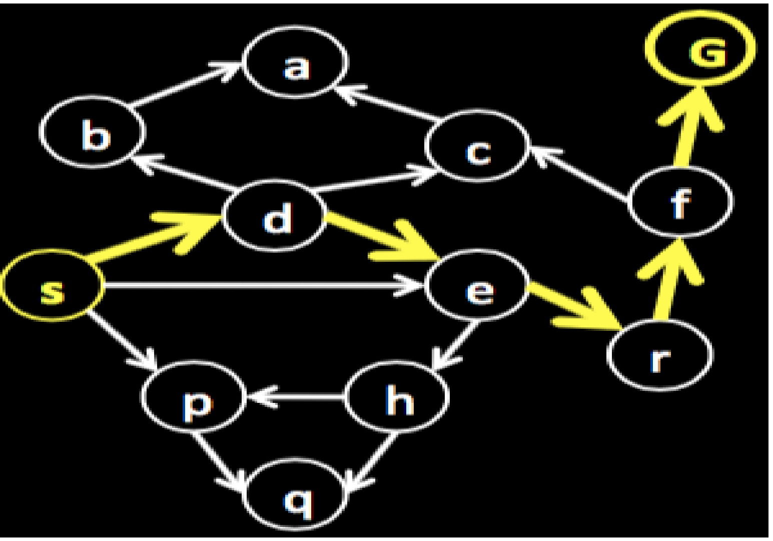
- Any ideas?
- Hint: Think it as related to Agent
  - Percepts, actions, environment, goals, utilities
- Problem:
  - States, initial state, goal states,
  - Solution: a sequence of actions leads to the goal

A Problem?

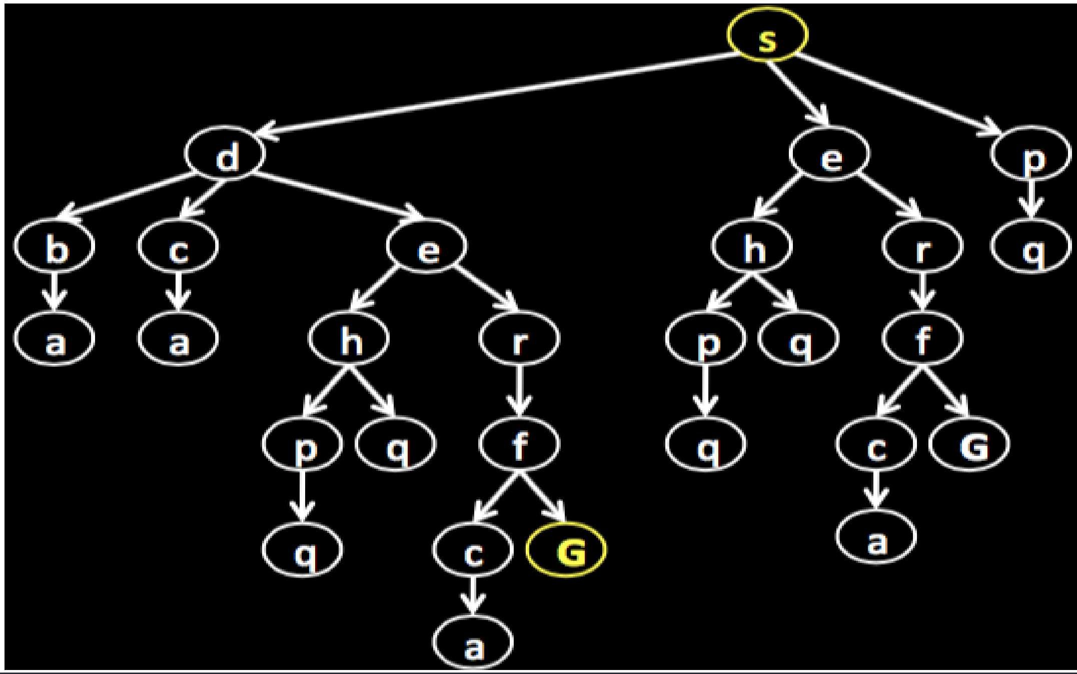
Example: 8 - Puzzles



Graphs VS. Trees

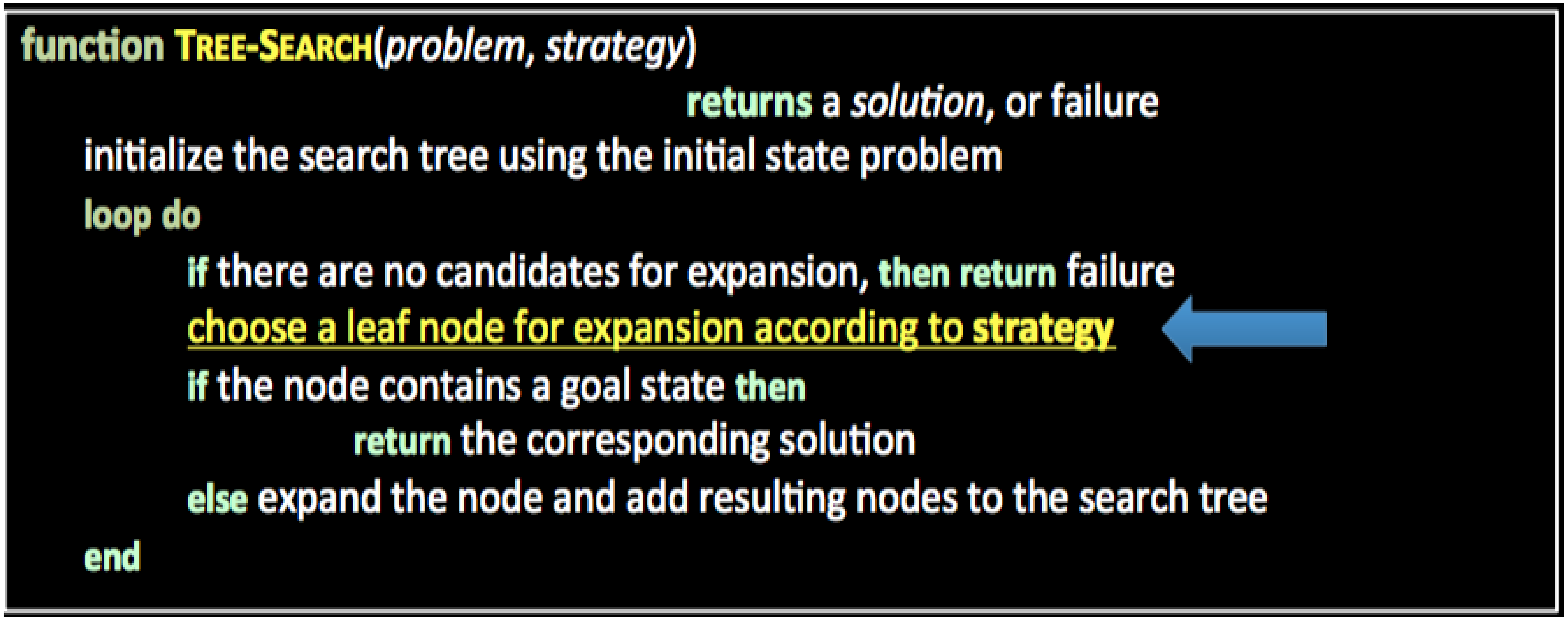


Each NODE in the search tree is an entire PATH in the state graph (how many nodes in the graph appear multiple times in the search tree)



Tree Search Algorithm

Basic idea: systematic exploration of simulated state-space by generating successors of explored states



```
function TREE-SEARCH(problem, strategy)
                                        returns a solution, or failure
    initialize the search tree using the initial state 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 resulting nodes to the search tree
    end
```

3- Problem Solving

Solving Problem by Searching

Lecture I

Course Learning Outcomes

Chapter 3
Problem Solving

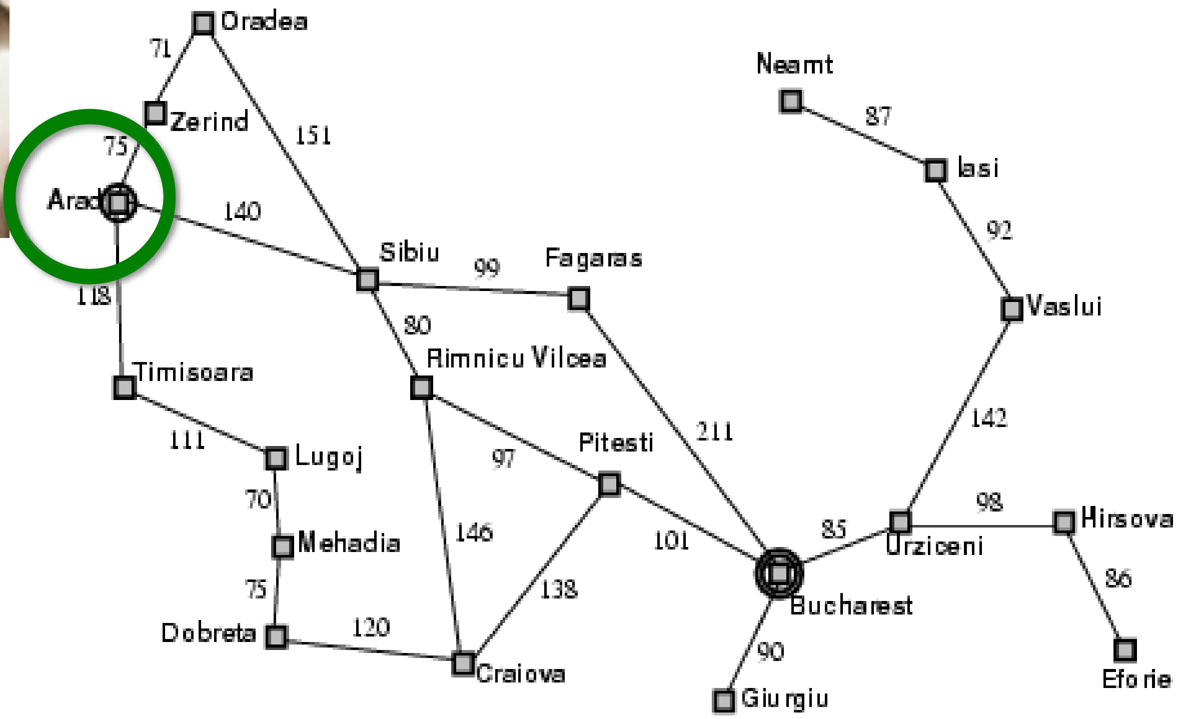
At the end of the course students will be able to

1. Compare different types of learning paradigms.
2. Apply suitable learning algorithm to a classification task.
3. Compare and contrast common models used for knowledge representation.
4. **Design and implement an AI algorithm based solution to a problem.**
5. **Compare and contrast various techniques for goal search.**
6. Explain the distinction between various types of reasoning and inference techniques.
7. Apply rule-based, case-based and model-based reasoning techniques.

- Simplest agents (reflex agents, which base their actions on a direct mapping from states to actions) cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.
- Goal-based agents can succeed by considering future actions and the desirability of their outcomes.

Problem-Solving Agents

We have a tourist would like to travel from Arad to Bucharest. Of course he will seek about which path with achieves his goal and go to required destination.



- Simplest agents (reflex agents, which base their actions on a direct mapping from states to actions) cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.
- Goal-based agents can succeed by considering future actions and the desirability of their outcomes.

Problem-Solving Agents

Are Deciding what to do by finding *sequences of actions* that lead to desirable states
•Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can adopt a goal, (e.g. Taxi)
•Goals help to organize Agent's behavior by limiting the objectives that the agent is trying to achieve.
•Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
•A goal is a set of world states- exactly those states in which the goal is satisfied. The agent's task is to find out which sequence of actions will get it to a goal state *.

Problem-Solving by Search

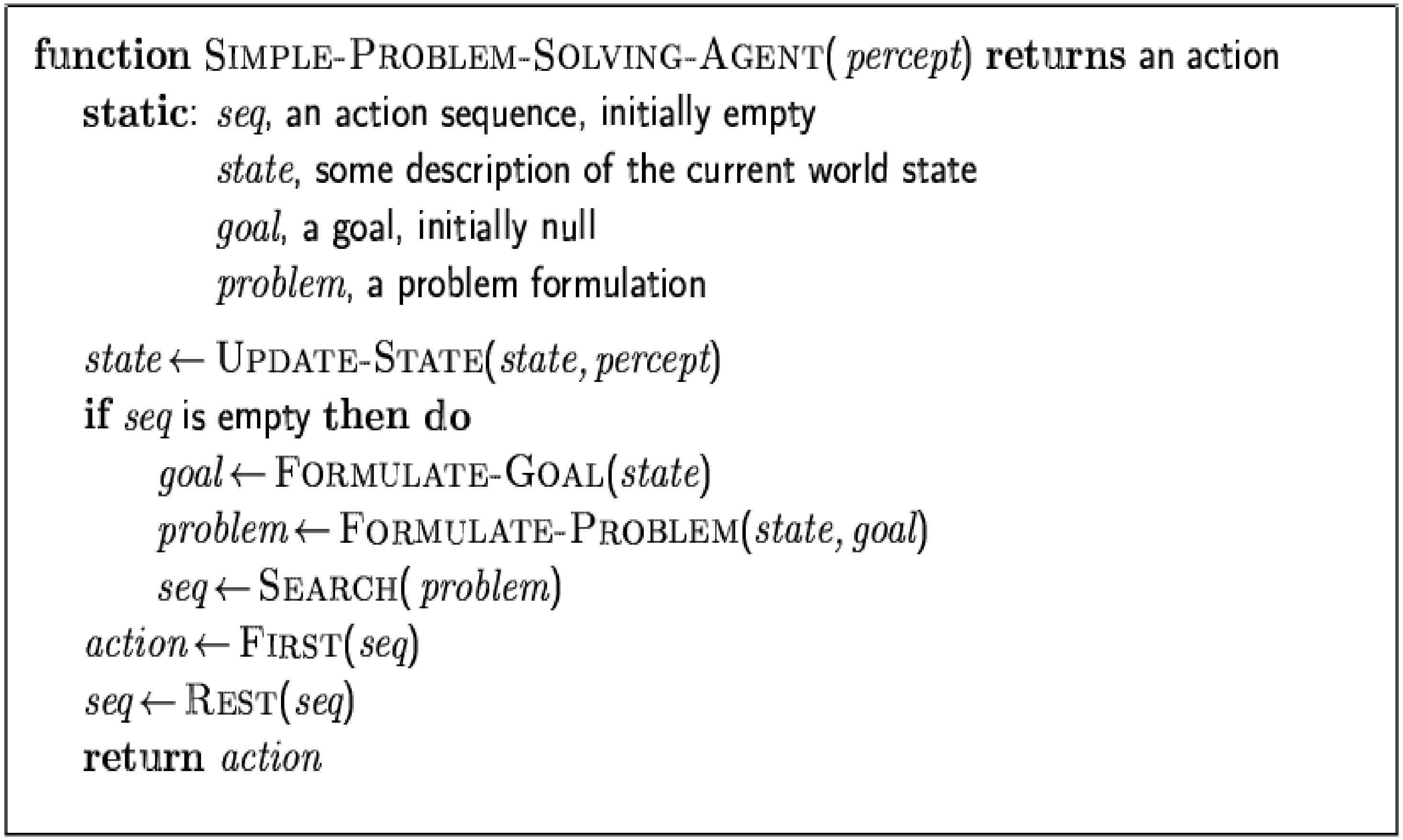
An agent with several immediate options of unknown value can decide what to do by just examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.

This Process of looking for such a sequence is called Search

A search algorithm takes a problem as input and returns a solution in the form of an action sequence, once a solution is found, the actions it recommends can be carried out.

Formulate, Search, Execute

Problem-Solving by Search



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

After formulating a goal, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do- typically, the first action of the sequence- and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

Well-defined problem and solution (1)

A problem can be defined formally by four components:

1- The **initial state** that the agent starts in,

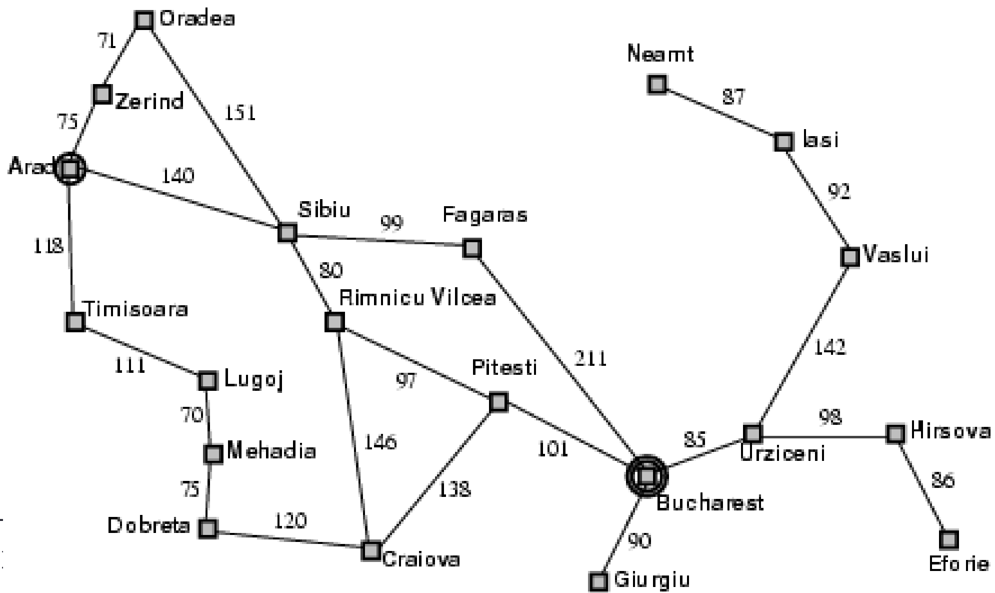
2- A description of the possible **actions** available to the agent

Successor Function

Given a particular state x, SUCCESSOR-FN(x) returns a set of (action, successor) ordered pairs. where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action.

Example: From the state *In* (Arad), the successor function for this problem would return:
{(*Go*(Sibiu), *In* (Sibiu)), (*Go*(Timisoara), *In* (Timisoara)), (*Go*(Zerind), *In* (Zerind))}

Together, the *initial state* and *successor function* implicitly define the **state space** of the problem, which is the set of all states reachable from the initial state. A **path** in the state space is a sequence of states connected by a sequence of actions.



Well-defined problem and solution (2)

3- The **goal test**, which determines whether a given state is a goal state or not. When we have explicit set of possible goal states, the test checks whether the given state is one of them or not.

4- A **path cost** function that assigns a numeric cost to each path. This cost function reflects the performance measure. (Example: Time to Bucharest so the cost of a path might be its length). So the cost of a path might be its length in kilometers.

A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost function. An optimal solution has the lowest path cost among all solutions.

Example of Problems

A **toy problem** is intended to illustrate (or exercise) various problem-solving methods. It can be given a concise, exact description. This means that it can be used easily by different researchers to compare the performance of algorithms.

A **real-world** problem is one whose solutions people actually care about. They tend not to have a single agreed-upon description

Toy Problems

- **The Vacuum World**
- **The Puzzle World**

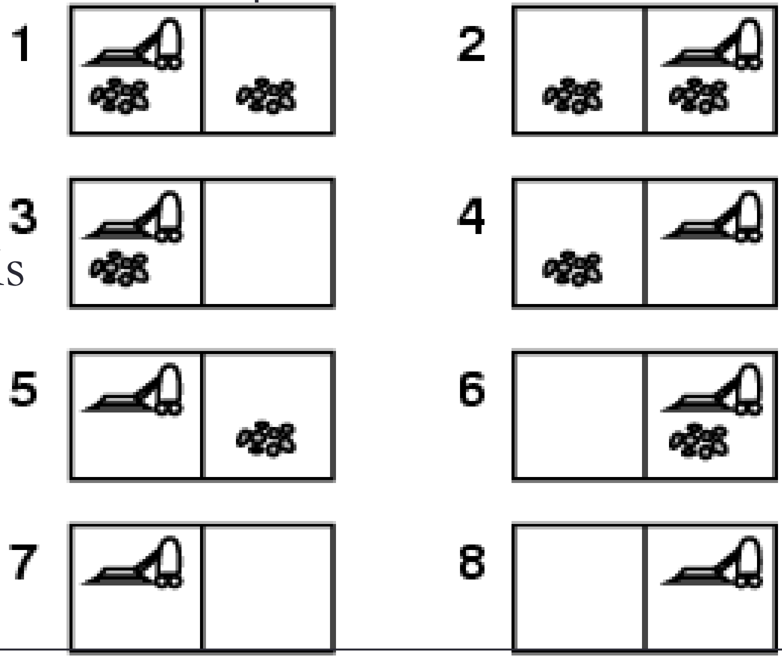
Toy Problems (1)

❖ The Vacuum World

States: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states. **(n rooms??)**

- **Initial state**: Any state can be designated as the initial state,
- **Successor function**: This generates the legal states that result from trying the three actions (Left, Right, and Suck). The complete state space is shown below.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets messed up once cleaned.



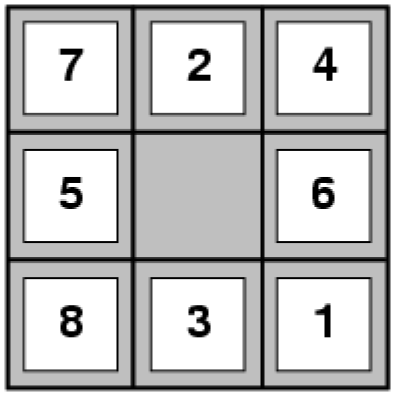
Toy Problems (2)

❖ The 8-Puzzle

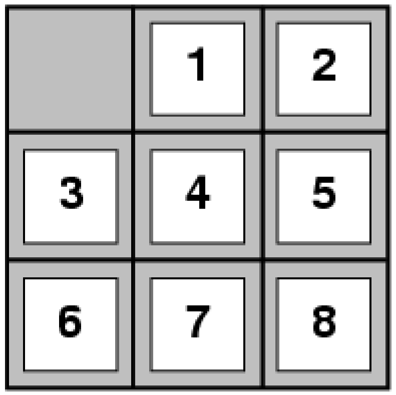
It consists of a 3 x 3 board with eight numbered tiles and a blank space.

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares

- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- **Successor function**: This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down)
- **Goal test**: This checks whether the state matches the goal configuration (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.



Start State



Goal State

The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding.

Real-world Problems (1)

❖ The Airline travel problem

It is example of **Route-finding** problem (e.g. routing, military, TSP, …)

States: Each is represented by a location (e.g., an airport) and the current time.

- **Initial state**: This is specified by the problem.
- **Successor function**: This returns the states resulting from taking any scheduled flight leaving later than the current time plus the within-airport transit time, from the current airport to another
- **Goal test**: Are we at the destination by some prespecified time?
- **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on

Real-world Problems (2)

❖ The Travel Salesman Problem

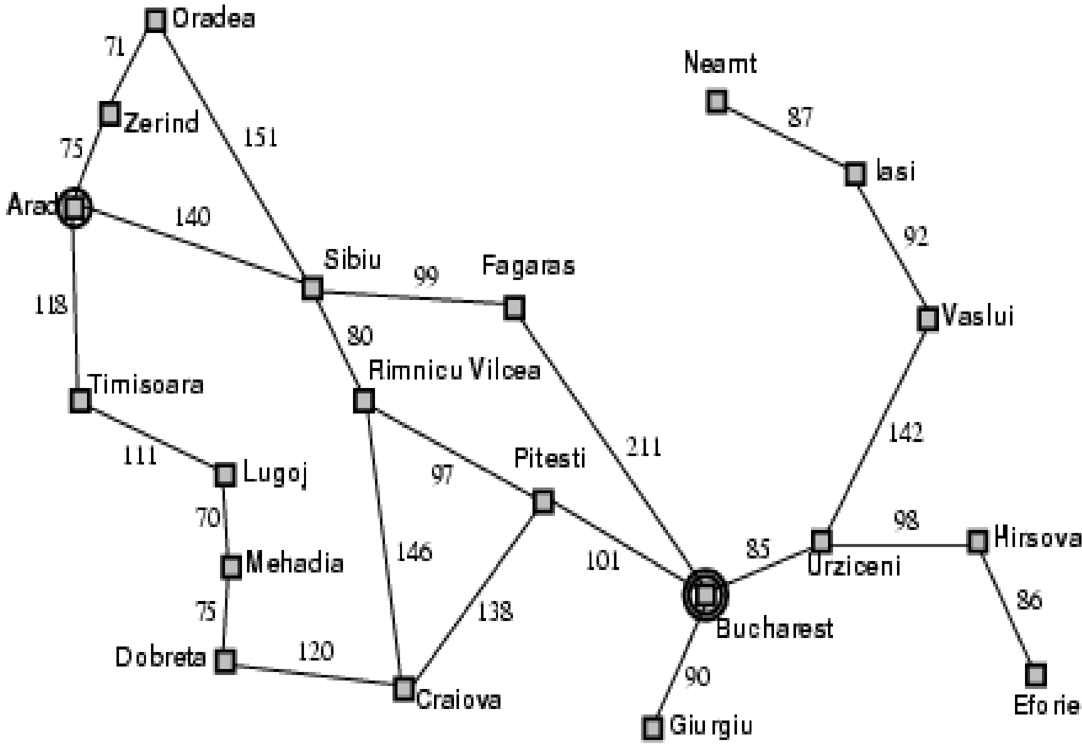
It is example of **Touring** problem, in which each city must be visited exactly once. The aim is to find the shortest tour (or path).

- The state space is quite different. Each state must include not just the current location but also the set of cities the agent has visited.
- So the initial state would be

  "In Bucharest; visited {Bucharest},"
- A typical intermediate state would be:

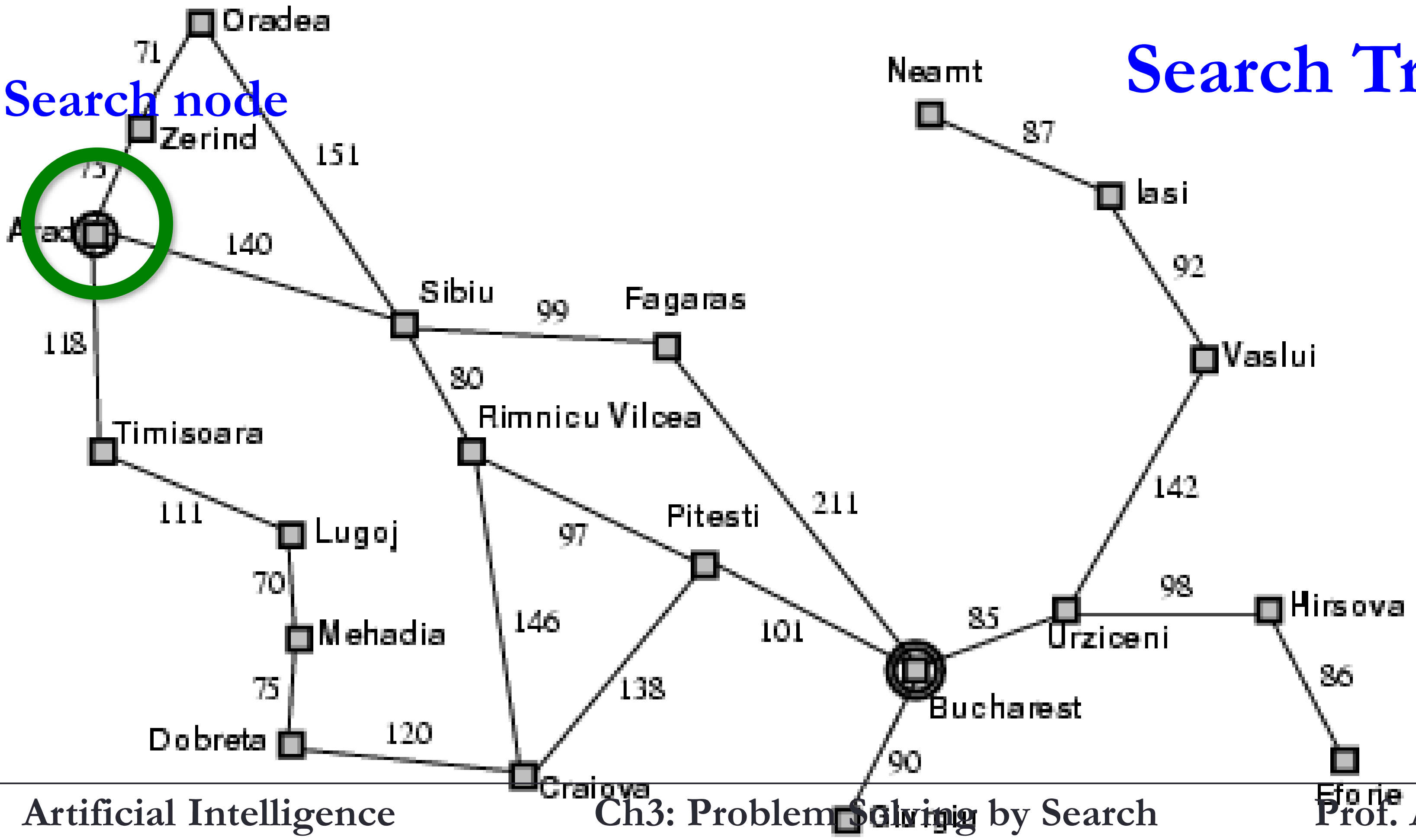
"In Vaslui; visited {Bucharest, Urziceni, Vaslui},"

and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited



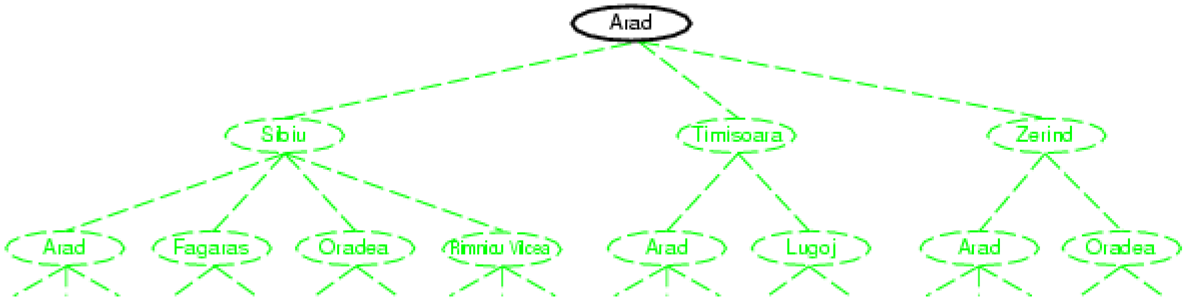
Searching for Solutions (1)

Having formulated some problems, we now need to **solve them**. This is done by a search through the state space. This chapter deals with search techniques that use an explilci **search tree** that is generated by the initial state and the successor function that together define the state space. The root of the search tree is a **search node**

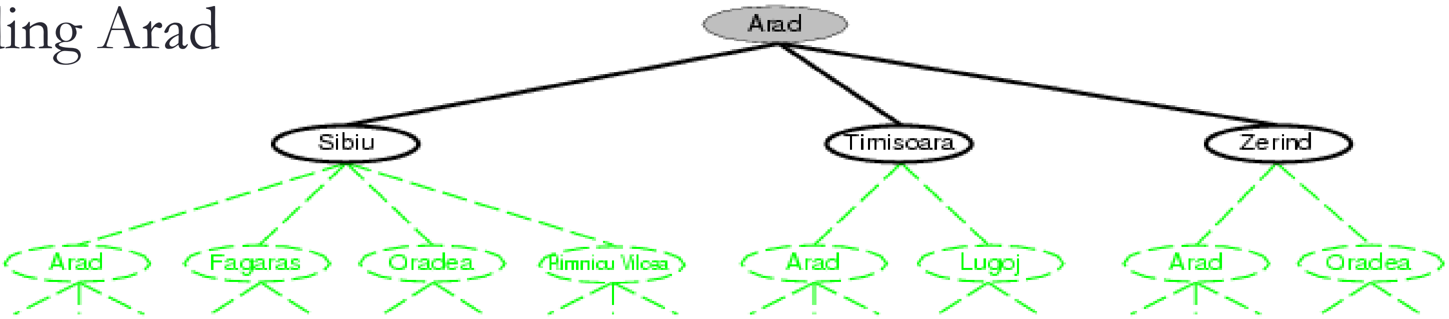


Searching for Solutions (2)

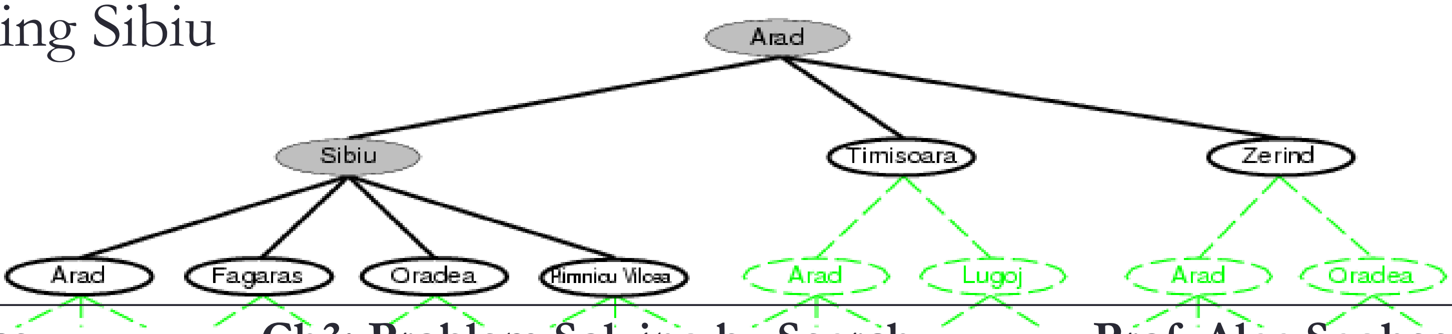
(a) The initial state



(b) After expanding Arad

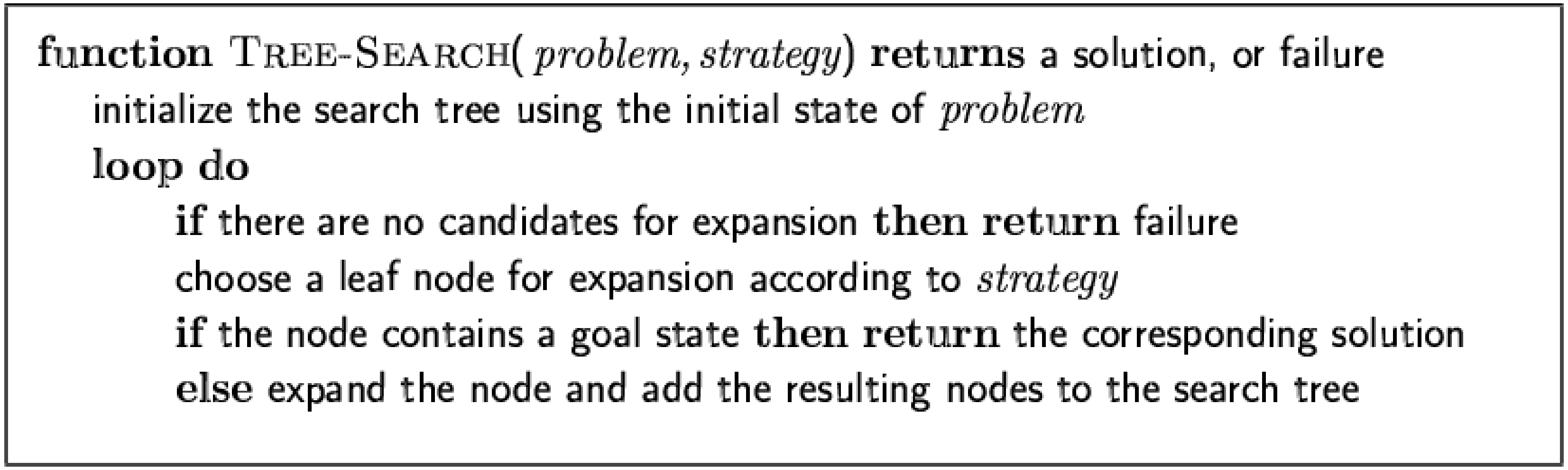


(c) After expanding Sibiu



Searching for Solutions (3)

❖ **Expanding Search Strategy**



```
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
```

State space vs. **Search tree ???**

For the route finding problem, there are only 20 states in the state space. But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes.

For example, "Arad- Sibiu", "Arad- Sibiu- Arad", "Arad-Sibiu-Arad-Sibiu" are the first three of an infinite sequence of paths, and so on.

Searching for Solutions (4)

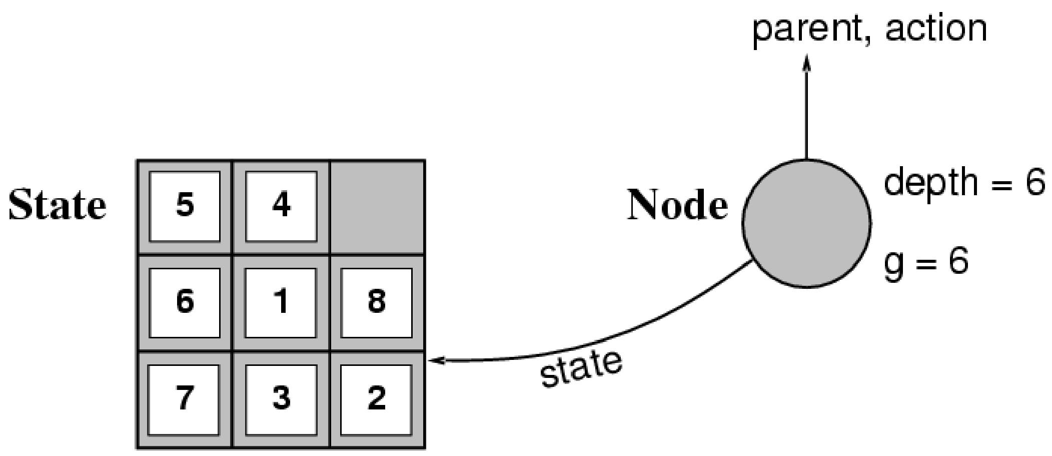
❖ Expanding Search Strategy

```
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
```

Nodes vs. **States ???**

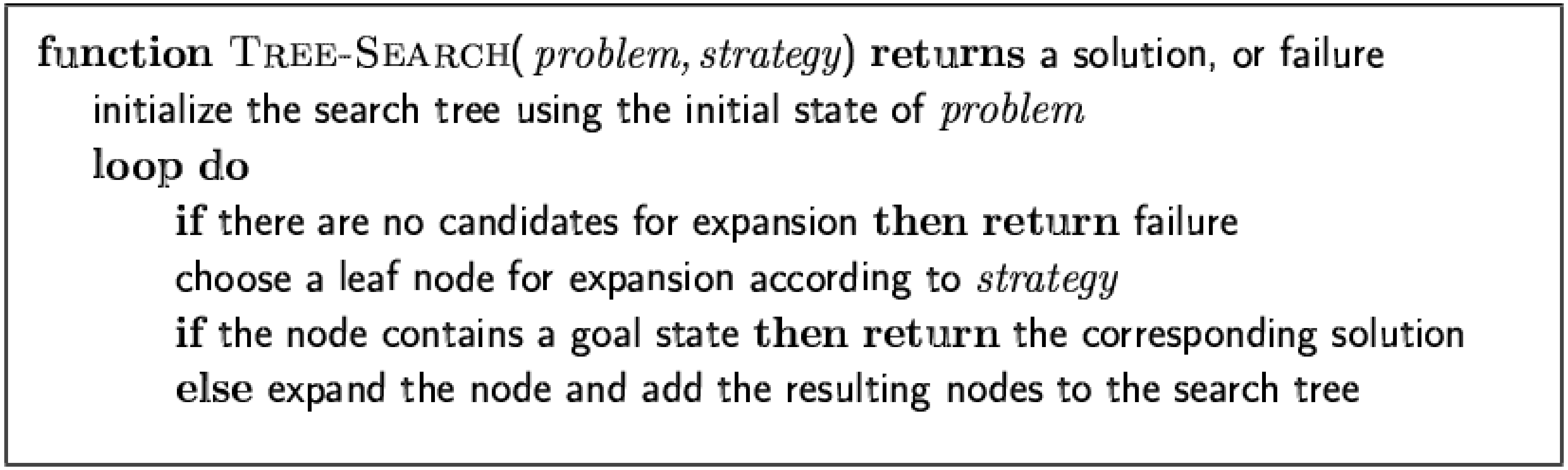
A **state** corresponds to a configuration of the world.

A **node** is a data structure from which the search tree is constructed. Each node has a parent node, a state, action, path cost g(x), depth.



Searching for Solutions (4)

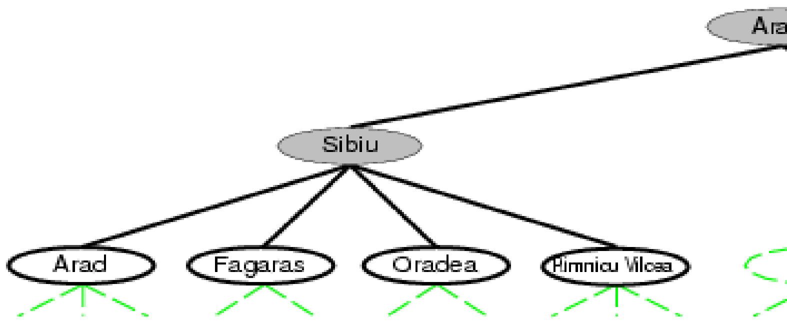
❖ **Expanding Search Strategy**



```
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
```

The collection of nodes that have been generated but not yet expanded is called **Fringe.**

Each element of the fringe is a **leaf node**, that is, a node with no successors in the tree



Measuring problem-solving performance

The output of a problem-solving algorithm is either *failure* or a *solution*. We evaluate an algorithm's performance in four ways:

- **Completeness:** Is the algorithm guaranteed to find a solution?
- **Optimality:** Does the strategy find the optimal solution?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

- Time is often measured in terms of the number of nodes generated during the search, and Space in terms of the maximum number of nodes stored in memory.
- The effectiveness of a search algorithm depends on the **search cost-** which typically depends on the time complexity and space complexity.
- Finally, we can use the **total cost,** which combines the search cost and the path cost of the solution found.

Properties of Search strategies

Completeness

If a solution exists, is it always found?

Properties of Search strategies

Completeness

If a solution exists, is it always found?

Optimality

Is the solution found always the lowest cost?

Properties of Search strategies

Completeness

If a solution exists, is it always found?

Optimality

Is the solution found always the lowest cost?

Time Complexity

How many search nodes will be generated?

Properties of Search strategies

Completeness

If a solution exists, is it always found?

Optimality

Is the solution found always the lowest cost?

Time Complexity

How many search nodes will be generated?

Space Complexity

How many search nodes must stay in main memory?

Properties of Search strategies

- Time and space complexity are measured in terms of
  - *b:* branching factor (i.e. maximum number of successors of any node)
  - *d:* depth of the least-cost solution (or goal)
  - *m*: maximum depth of any path in the state space (may be ∞)
  -

3- Problem Solving

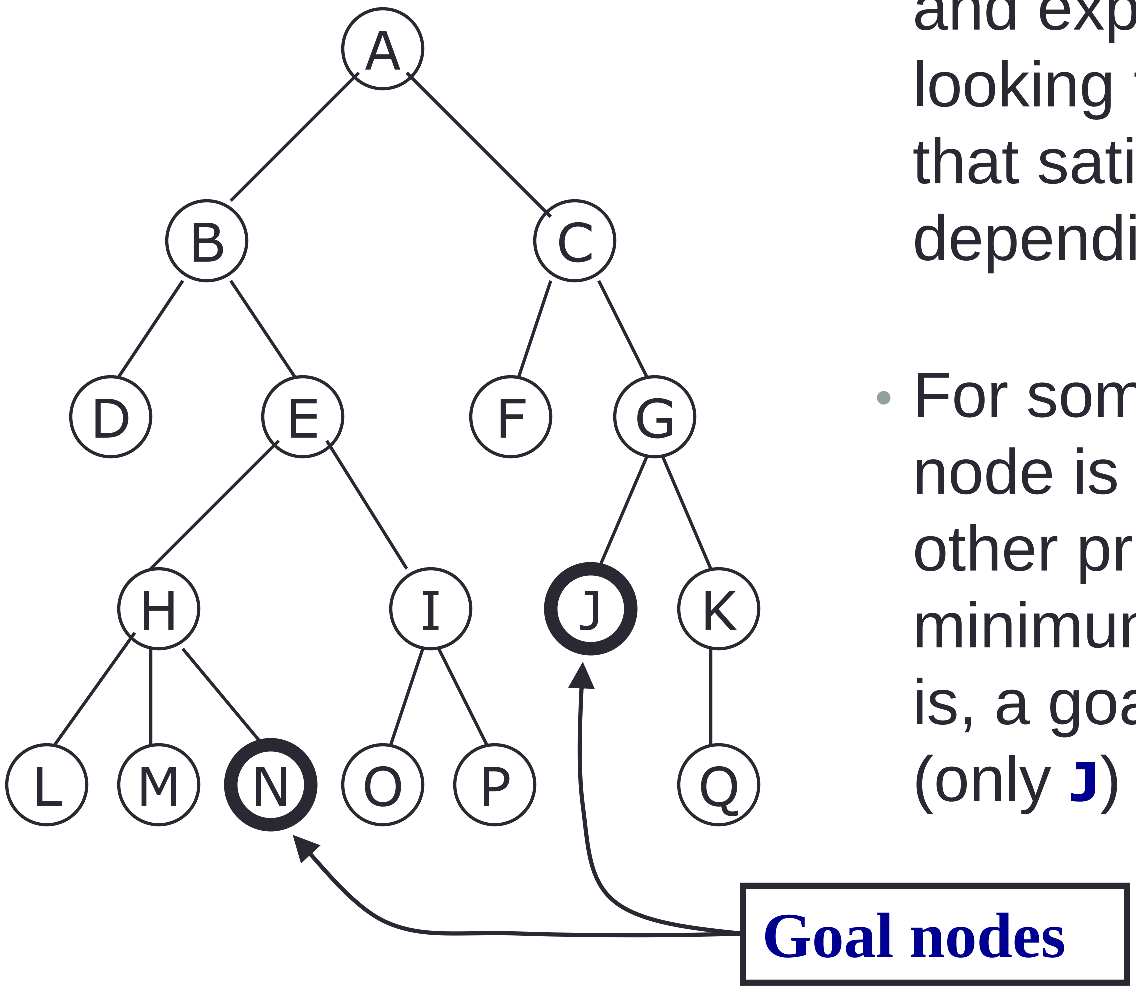
Uninformed Search

Lecture II

Uninformed Search Strategies

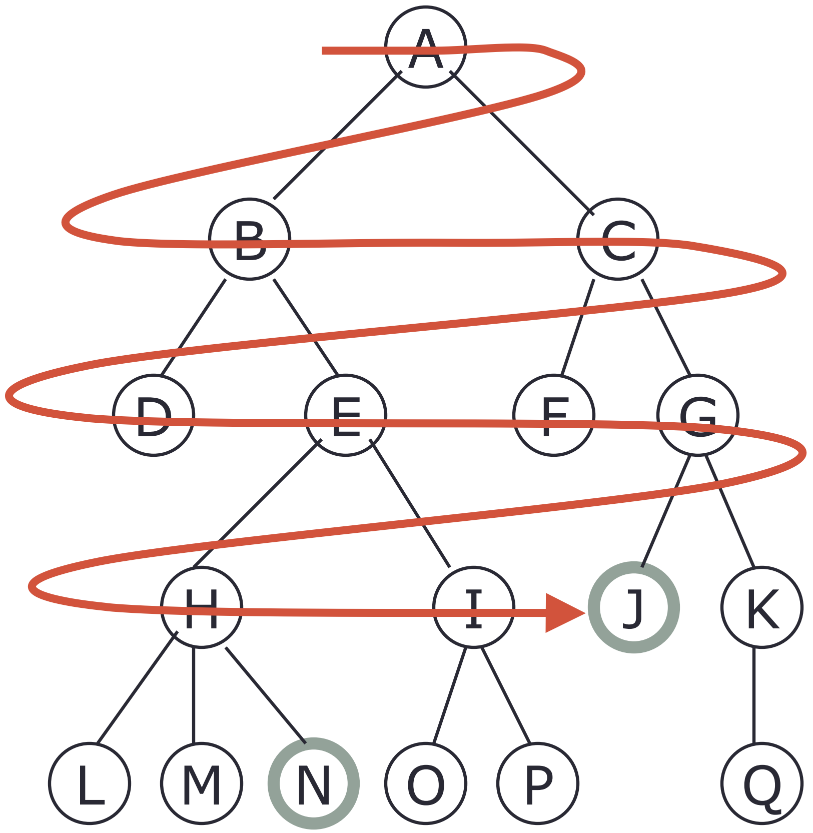
- Uninformed (or **Blind Search**) search strategies use only the information available in the problem definition..There is no additional Information. All they can do is generate successors and distinguish a goal state from a non goal state.
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Tree Search



- A tree search starts at the root and explores nodes from there, looking for a goal node (a node that satisfies certain conditions, depending on the problem)
- For some problems, any goal node is acceptable (**N** or **J**); for other problems, you want a minimum-depth goal node, that is, a goal node nearest the root (only **J**)

Breadth-First Search - Example

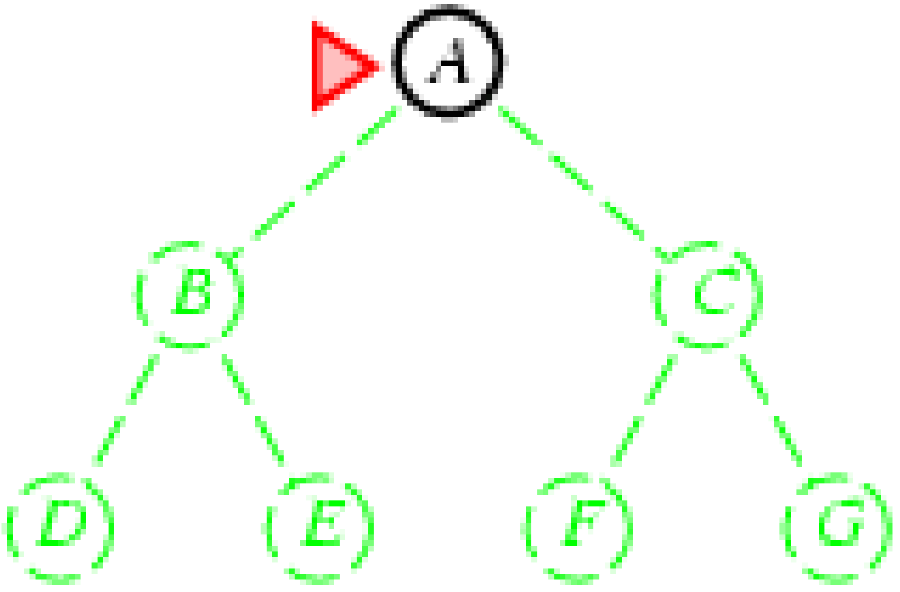


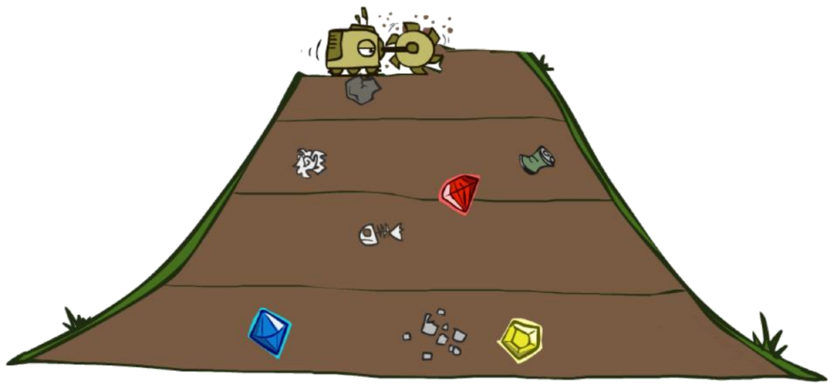
- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order **A B C D E F G H I J K L M N O P Q**
- **J** will be found before N

Breadth-First Search

- The root node is expanded first, then all the successors of the root node are expanded next, then their successor, and so on.
- Implementation:
  - *Fringe* is a FIFO queue, i.e., new successors go at end. i.e. Nodes that are visited first will be expanded first.
  -

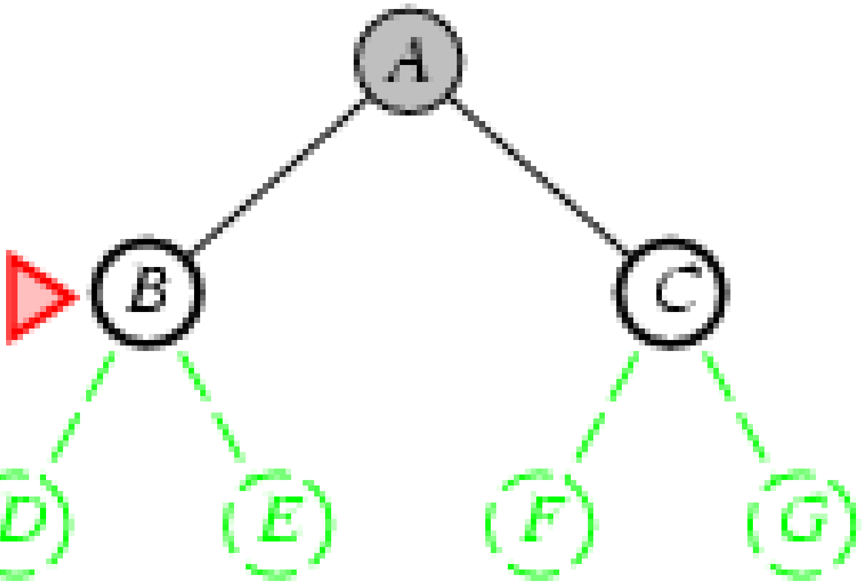
FIFO = First-In-First-Out





Breadth-First Search

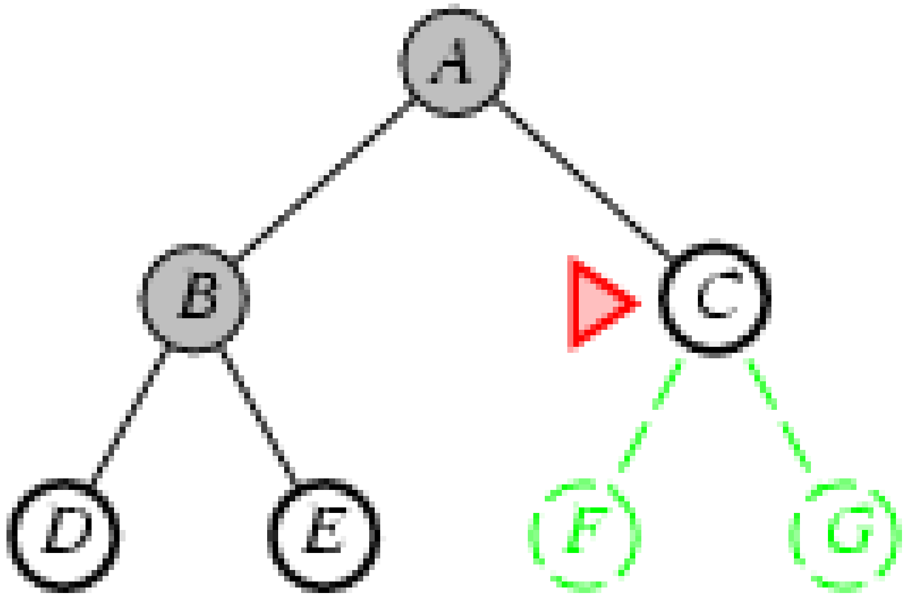
- The root node is expanded first, then all the successors of the root node are expanded next, then their successor, and so on.
- Implementation:
  - *Fringe* is a FIFO queue, i.e., new successors go at end. i.e. Nodes that are visited first will be expanded first.
  -



At each stage, the node to be expanded next is indicated by a red marker

Breadth-First Search

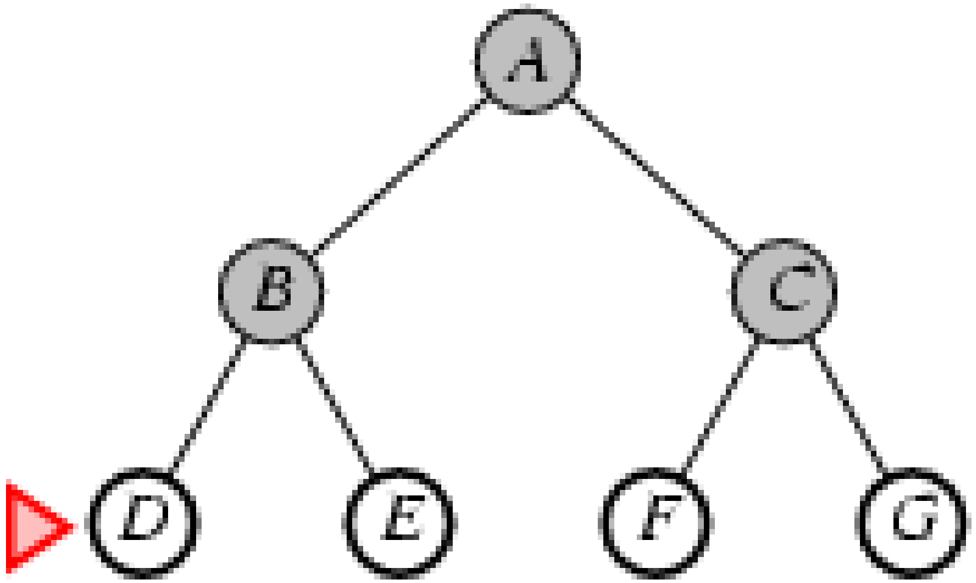
- The root node is expanded first, then all the successors of the root node are expanded next, then their successor, and so on.
- Implementation:
  - *Fringe* is a FIFO queue, i.e., new successors go at end. i.e. Nodes that are visited first will be expanded first.
  -



At each stage, the node to be expanded next is indicated by a red marker

Breadth-First Search

- The root node is expanded first, then all the successors of the root node are expanded next, then their successor, and so on.
- Implementation:
  - *Fringe* is a FIFO queue, i.e., new successors go at end. i.e. Nodes that are visited first will be expanded first.
  -



At each stage, the node to be expanded next is indicated by a red marker

Properties of breadth-first search

- Regarding **Time and Memory**, it is not always the strategy of choice!
- Consider a state space where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.
- Now suppose that the solution is at depth d. In the worst case, we would expand all but the last node at level d (since the goal itself is not expanded), generating $b^{d+1} - b$ nodes at level d+1*.

- Then the maximum total number of nodes generated is:

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Breadth-first Properties

Strategy? ▸ Example

First-in First-Out Queue

Complete?

Yes, if number of branches is finite

Optimal?

Yes, if step costs are all identical

Worst Case Time Complexity?

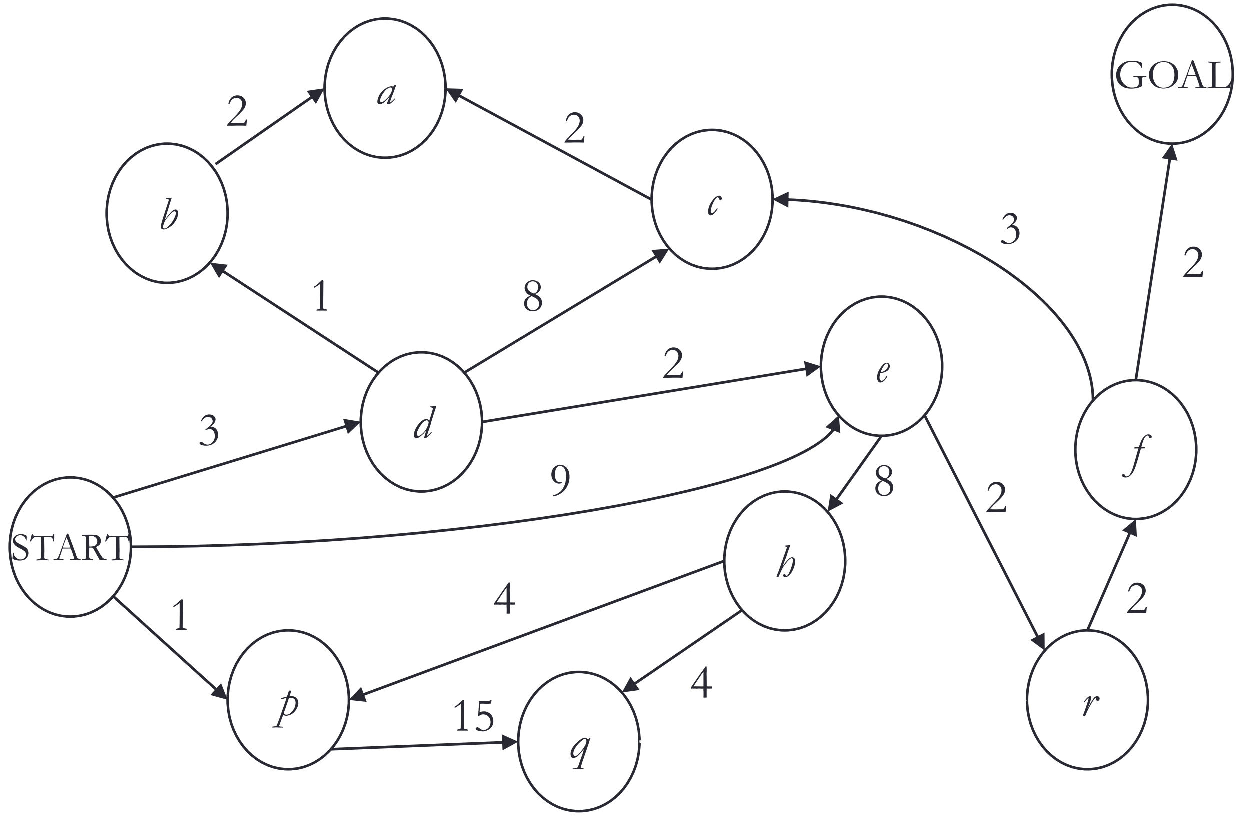
$O(b^{d+1})$, branching factor b, depth of goal state d

Worst Case Space Complexity?

$O(b^{d+1})$, branching factor b, depth of goal state d

Space is the bigger problem (more than time)

Cost-Sensitive Search

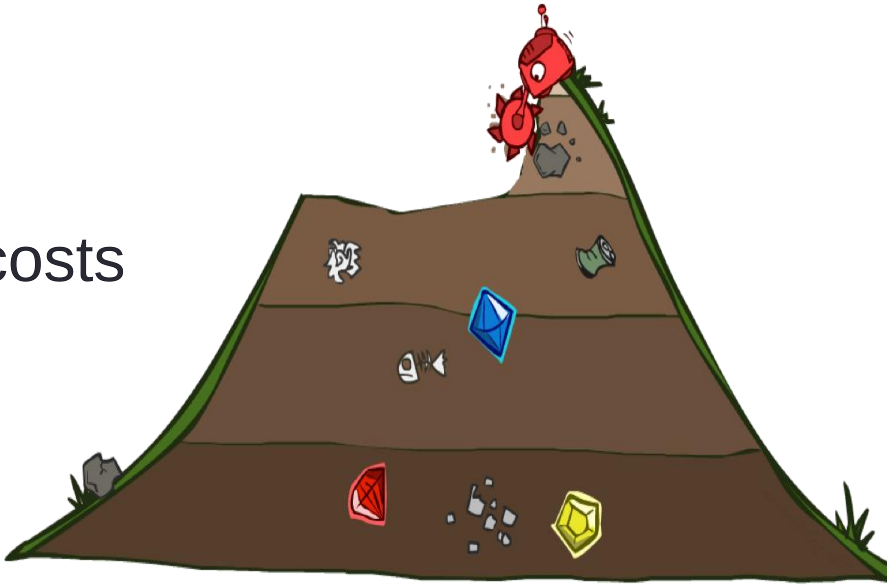


BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

Uniform-cost search

Breadth-first search is optimal when all step costs are equal, because it always expands the *shallowest* unexpanded node.

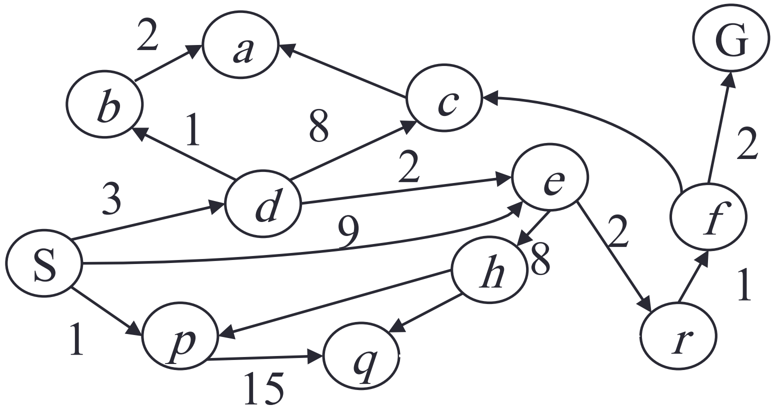
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* in each branch up to the Goal.
- Implementation:
  - *fringe* = queue ordered by path cost
  -
- Equivalent to breadth-first if step costs
- all equal
-

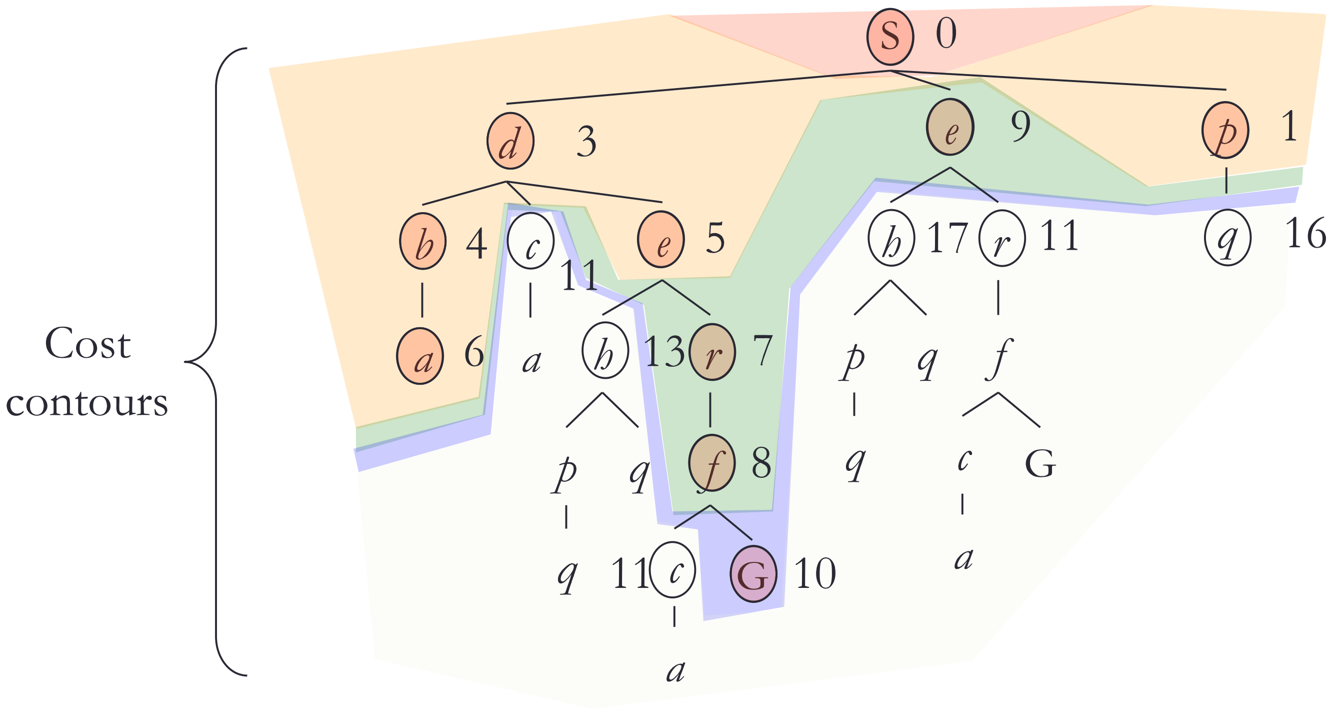


Uniform-cost search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

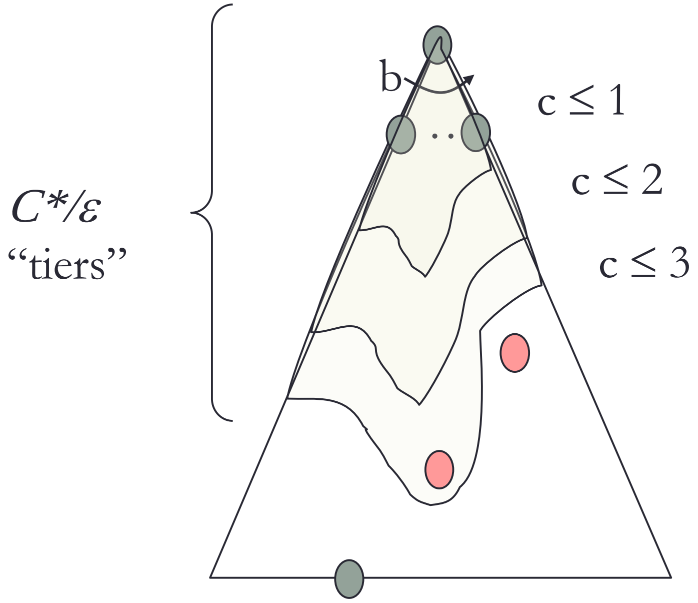




Uniform-cost search Properties

- 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



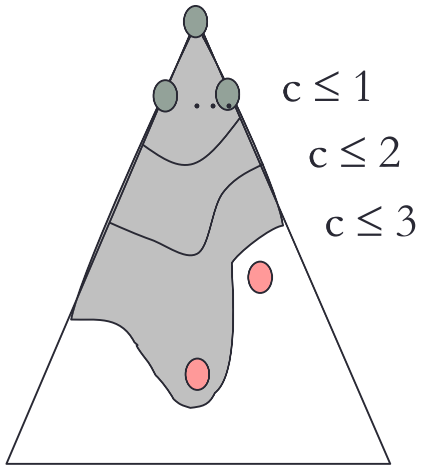
Uniform-cost search Properties

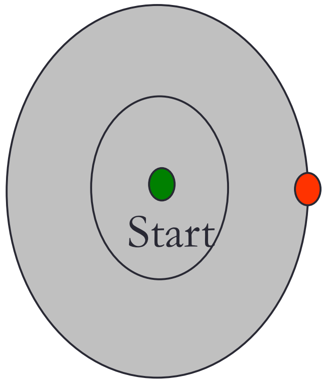
- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

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

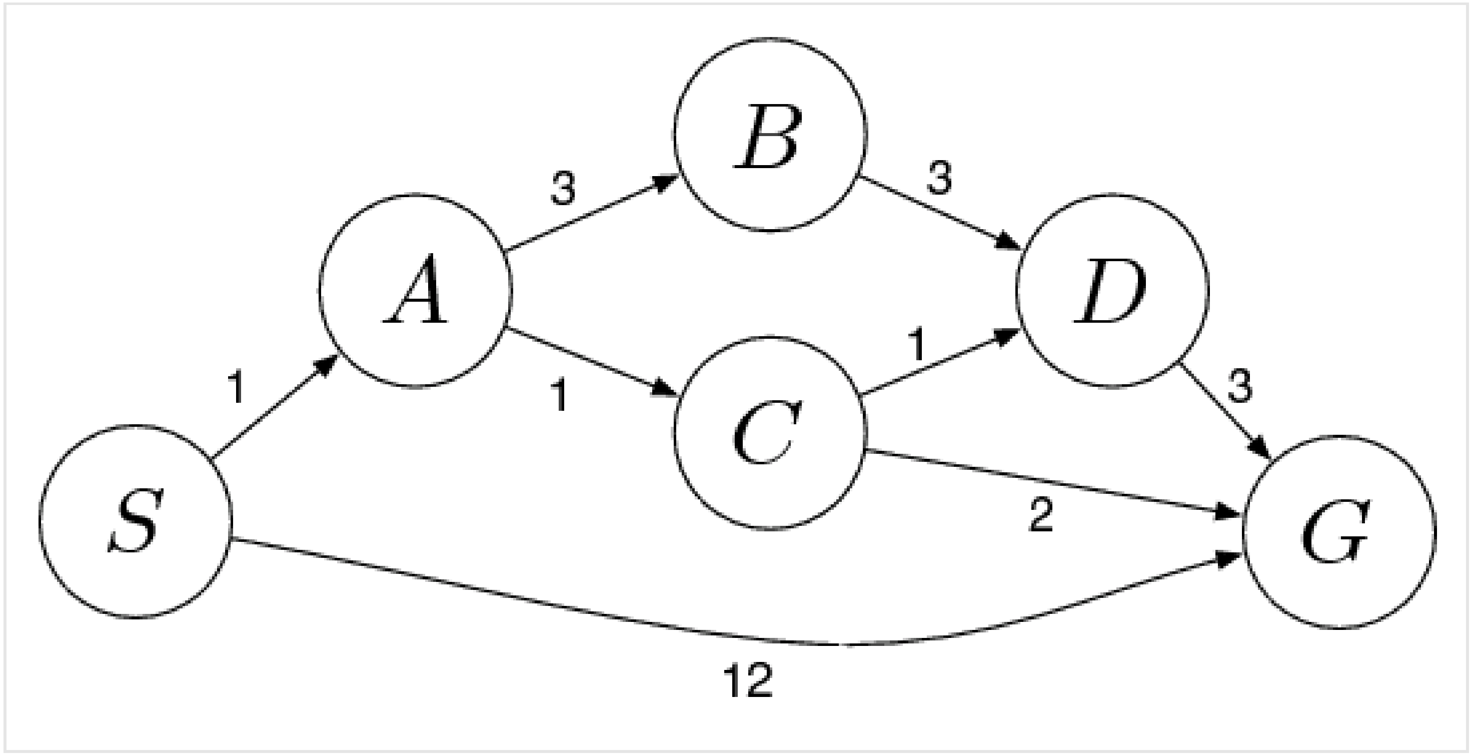
- We’ll fix that soon!



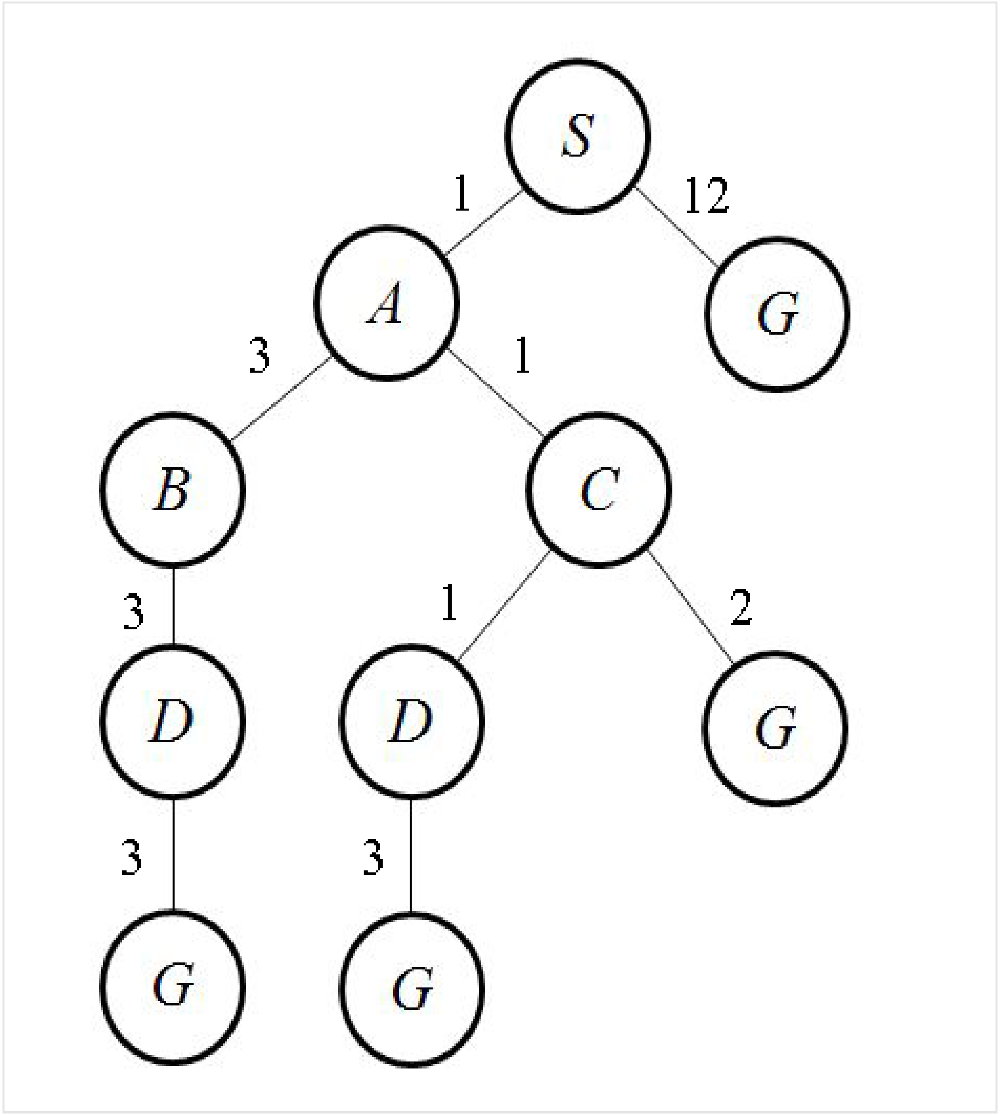


[Demo: empty grid UCS (L2D5)]
[Demo: maze with deep/shallow water DFS/BFS/UCS (L2D7)]

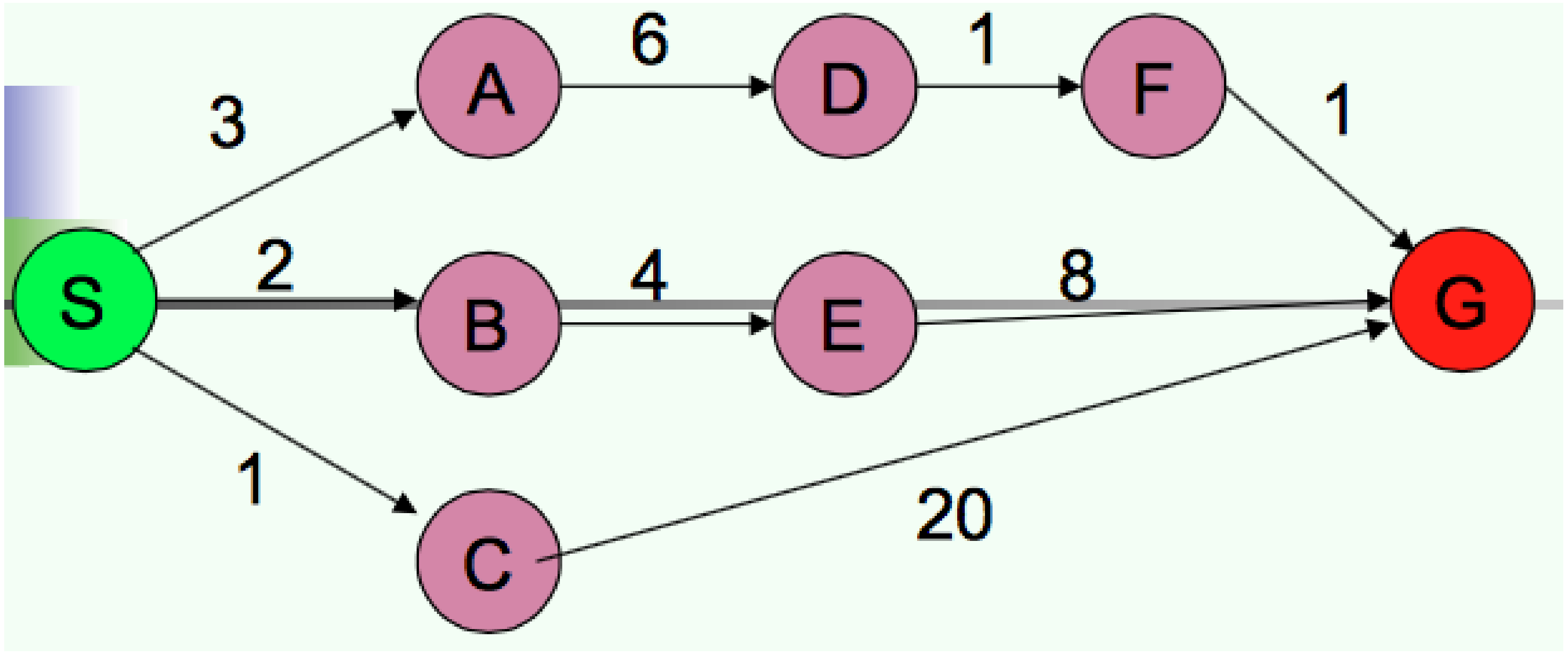
Example



Solution



Excercise



Uniform-cost Search

Strategy? ‣ Example

Lowest Cost First Priority Queue

Complete?

Yes, if number of branches is finite and steps are all positive

Optimal?

Yes

Worst Case Time Complexity?

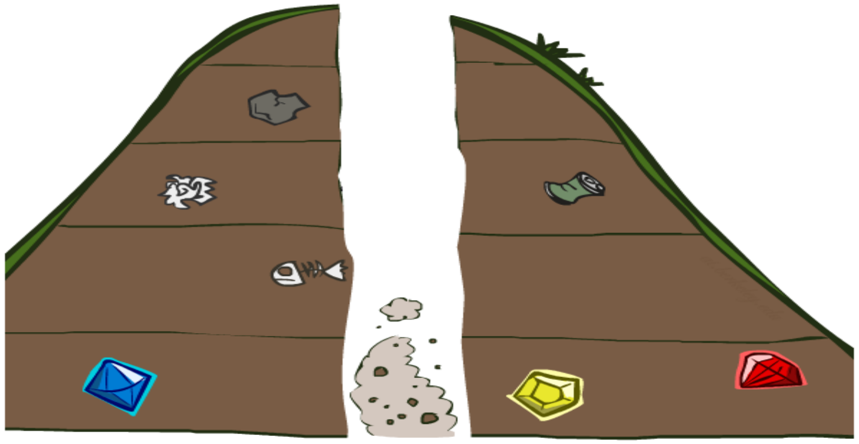
$O(b^{\lfloor C^*/\epsilon \rfloor + 1})$, optimal cost C^*, minimum step cost ϵ

Worst Case Space Complexity?

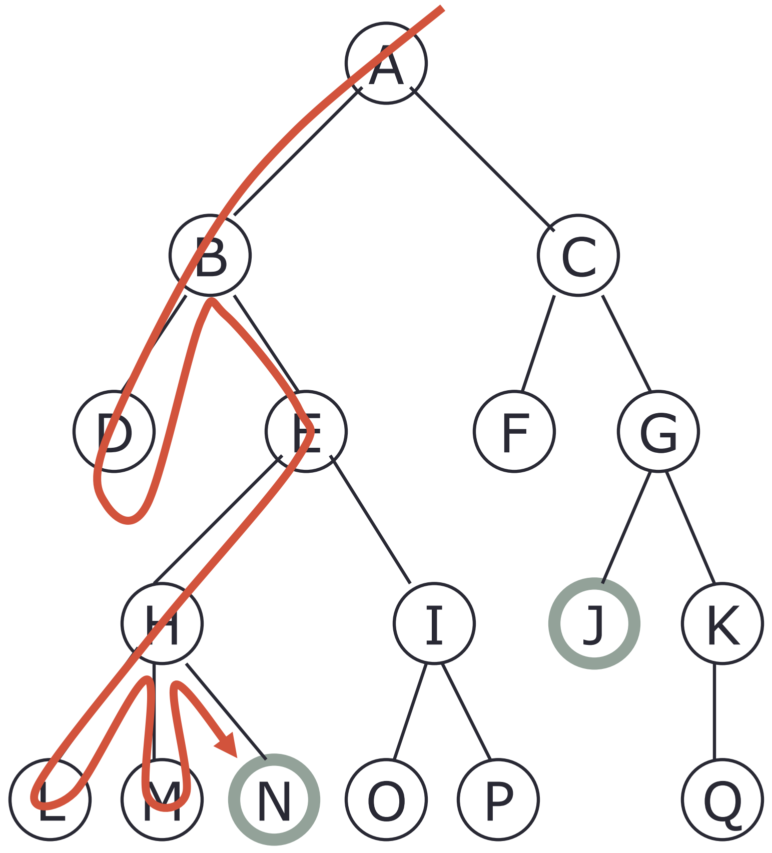
$O(b^{\lfloor C^*/\epsilon \rfloor + 1})$, optimal cost C^*, minimum step cost ϵ

Depth-first search

- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.



Depth-first search - Example



- For example, after searching **A**, then **B**, then **D**, the search backtracks and tries another path from **B**
- Nodes are accessed in the order **A B D E H L M N I O P C F G J K Q**
- **N** will be found before **J**
- Nodes are tested in the order **DLMNHOPIEBFJQKGCA**

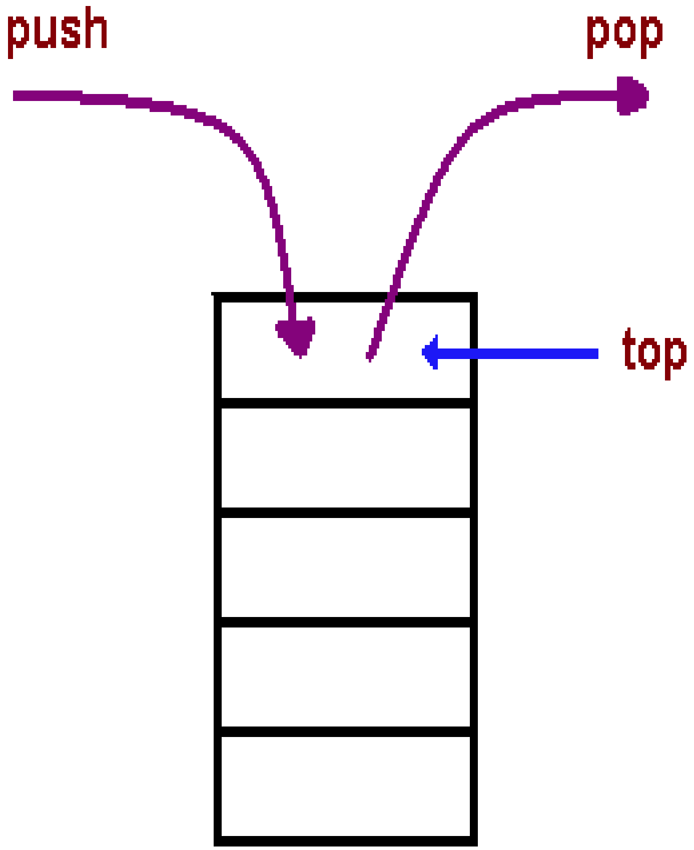
Depth-first search

- **Implementation:**
  - This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) strategy, also known as a stack.
  - As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

Depth-first search has very modest memory requirements. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

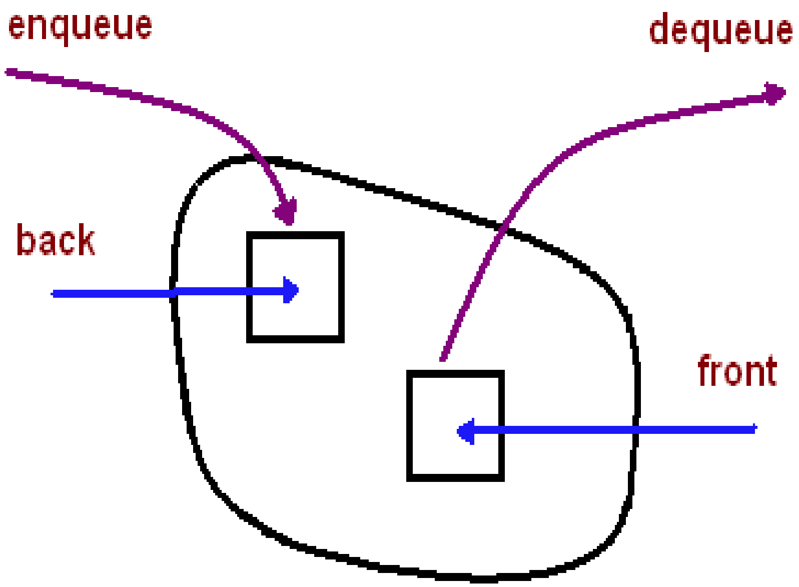
Review: Queue and Stack data structure

- A **stack** is a limited access data structure can be viewed as a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.
- Elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top.
- A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



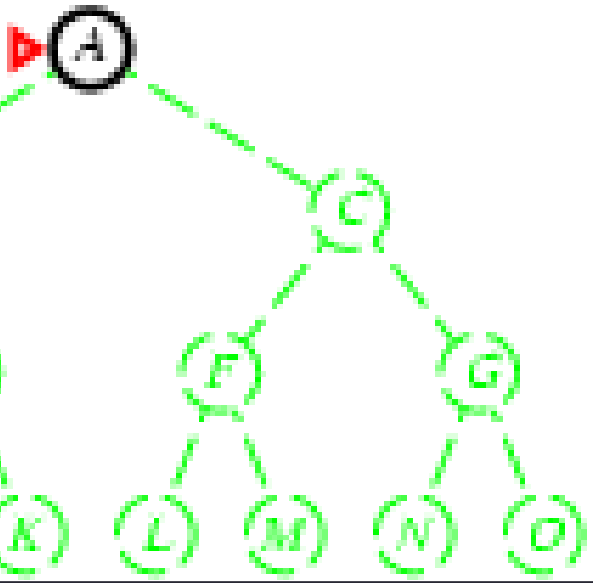
Review: Queue and Stack data structure

- A **queue** is a container of objects (a linear collection) that are inserted and removed according to the **first-in first-out (FIFO)** principle.
- Two operations are allowed: **enqueue** means to insert an item into the back of the queue, **dequeue** means removing the front item. The picture demonstrates the FIFO access.
- An excellent example of a queue is a line of students in the food court. New additions to a line made to the back of the queue, while removal (or serving) happens in the front.



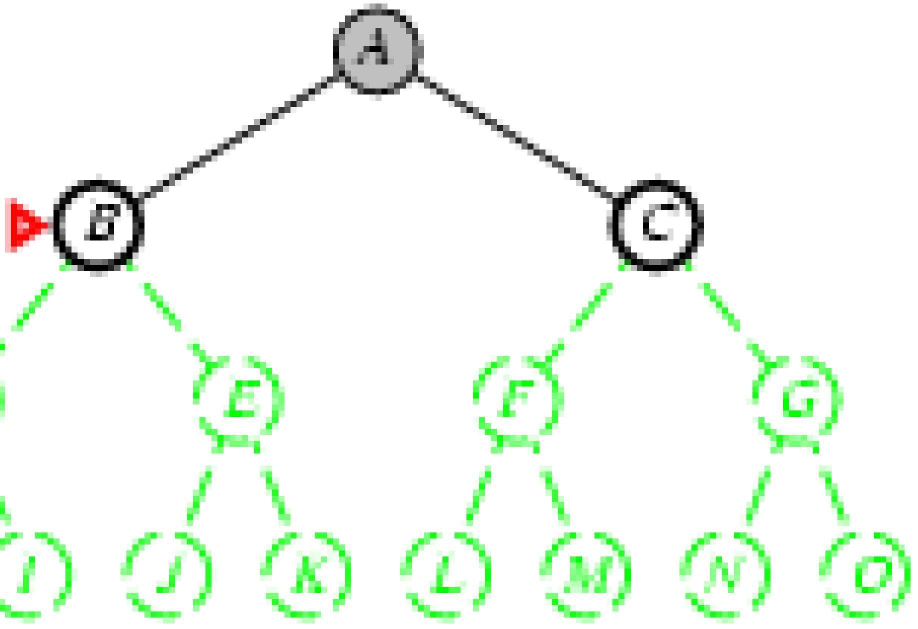
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.



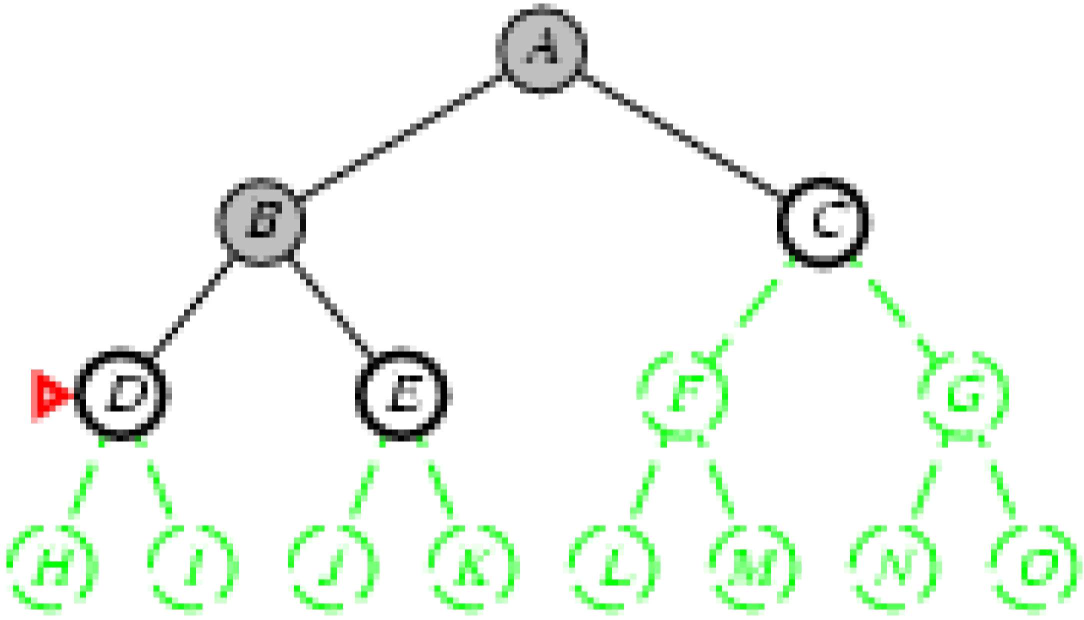
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



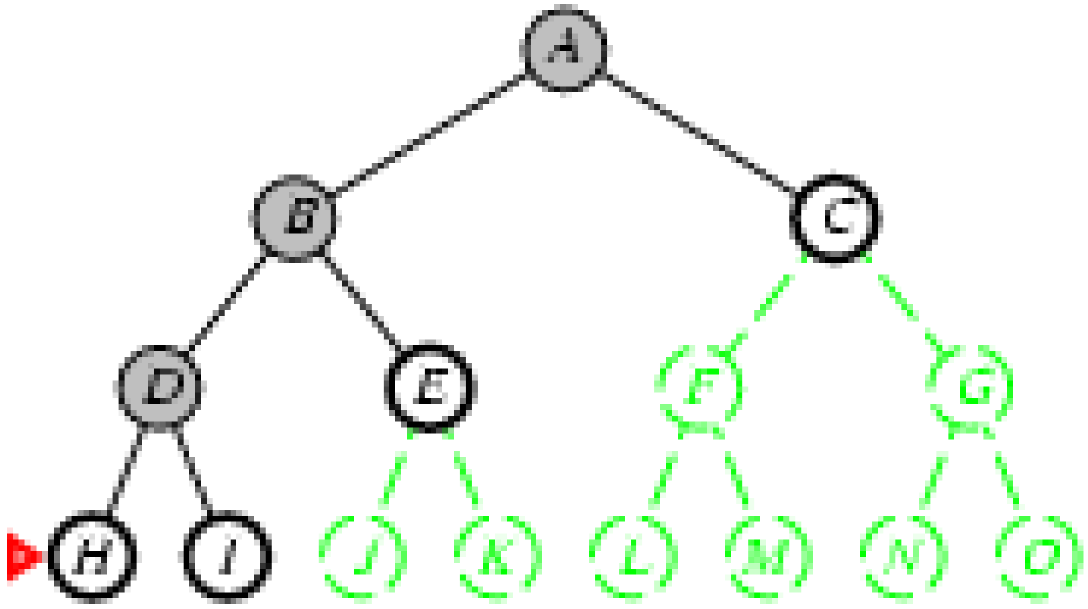
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
    - *fringe* = LIFO queue, i.e., put successors at front
    -



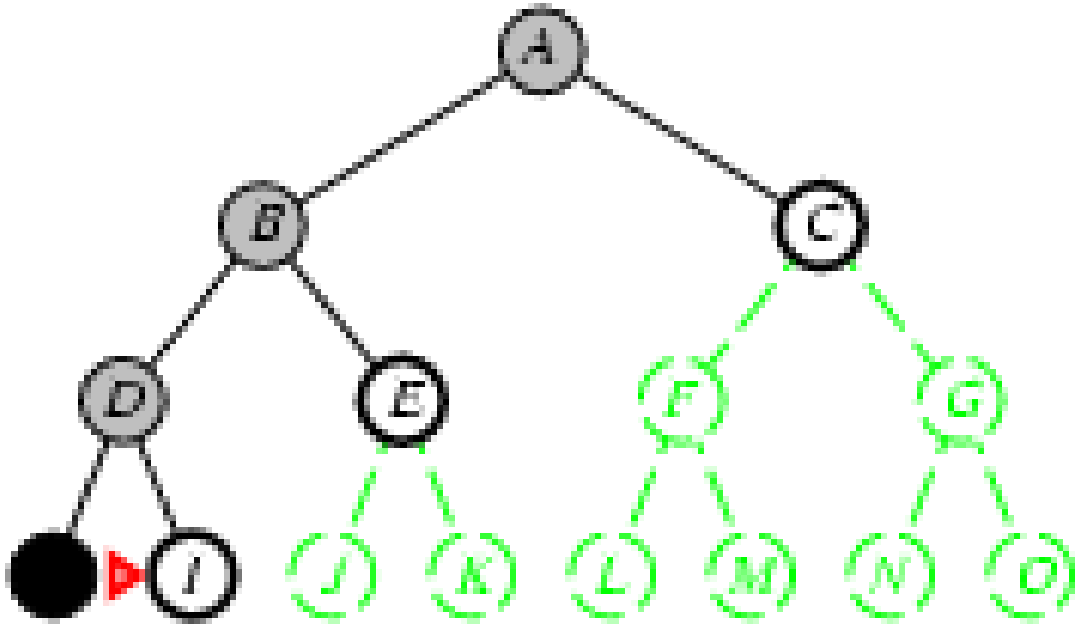
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



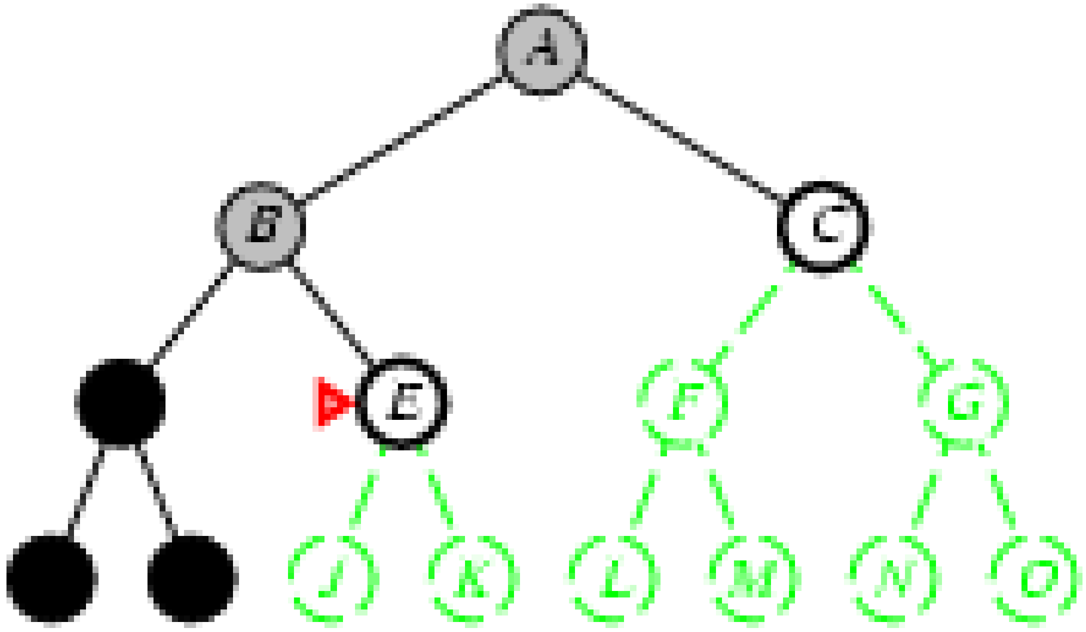
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
    - *fringe* = LIFO queue, i.e., put successors at front
    -



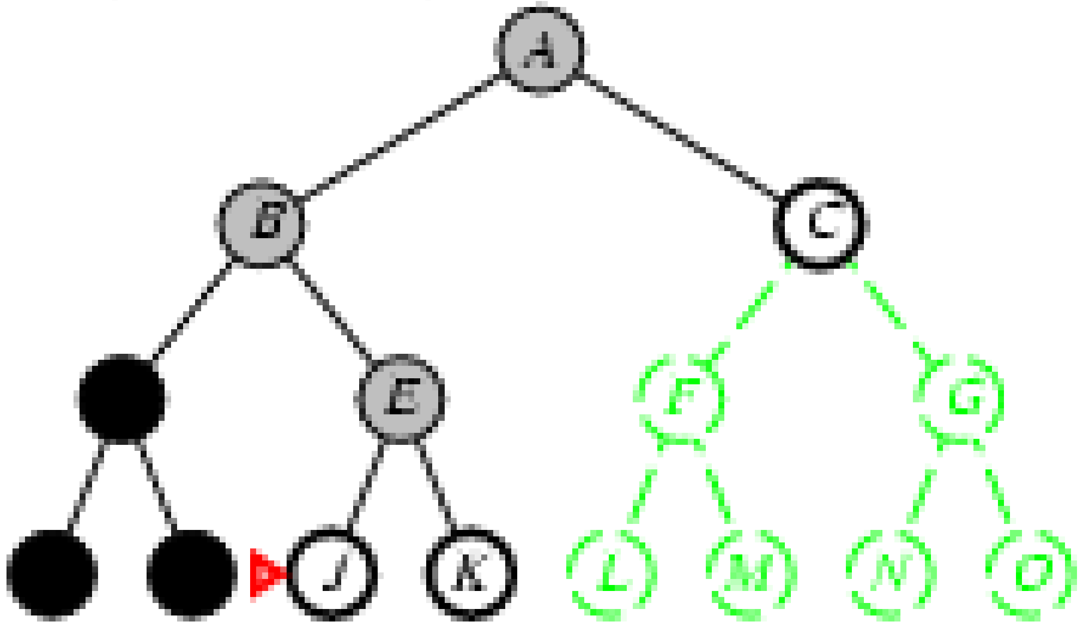
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
    - *fringe* = LIFO queue, i.e., put successors at front
    -



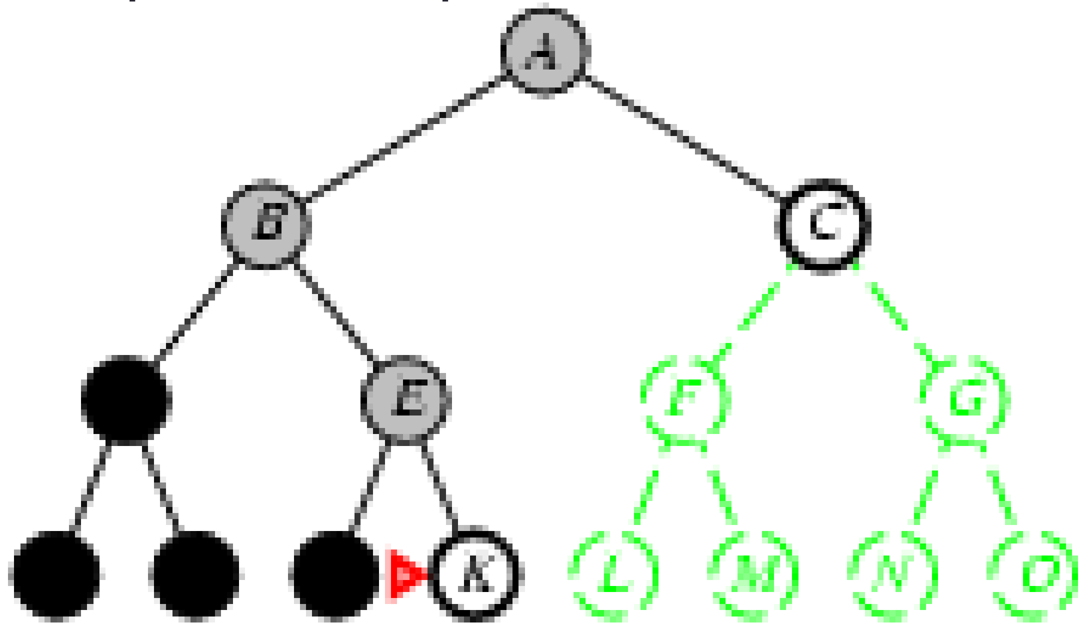
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



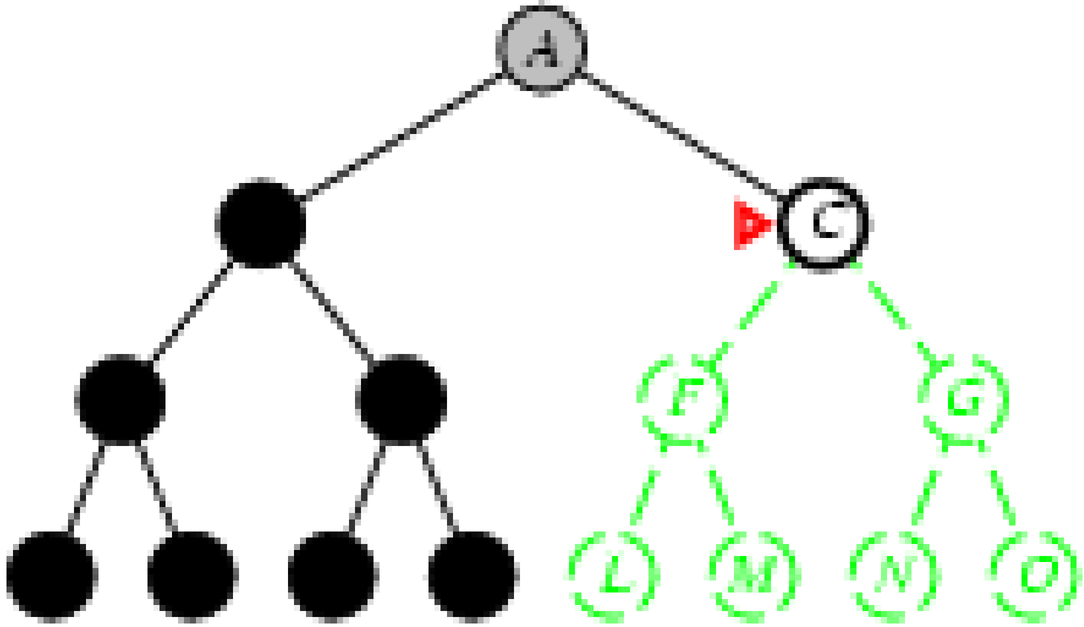
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



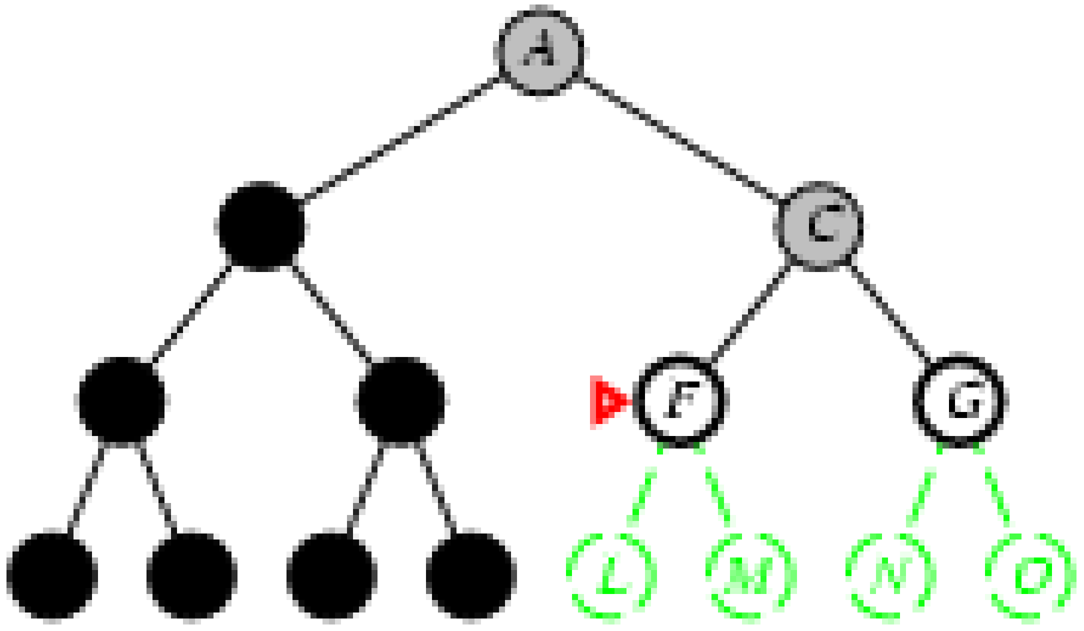
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



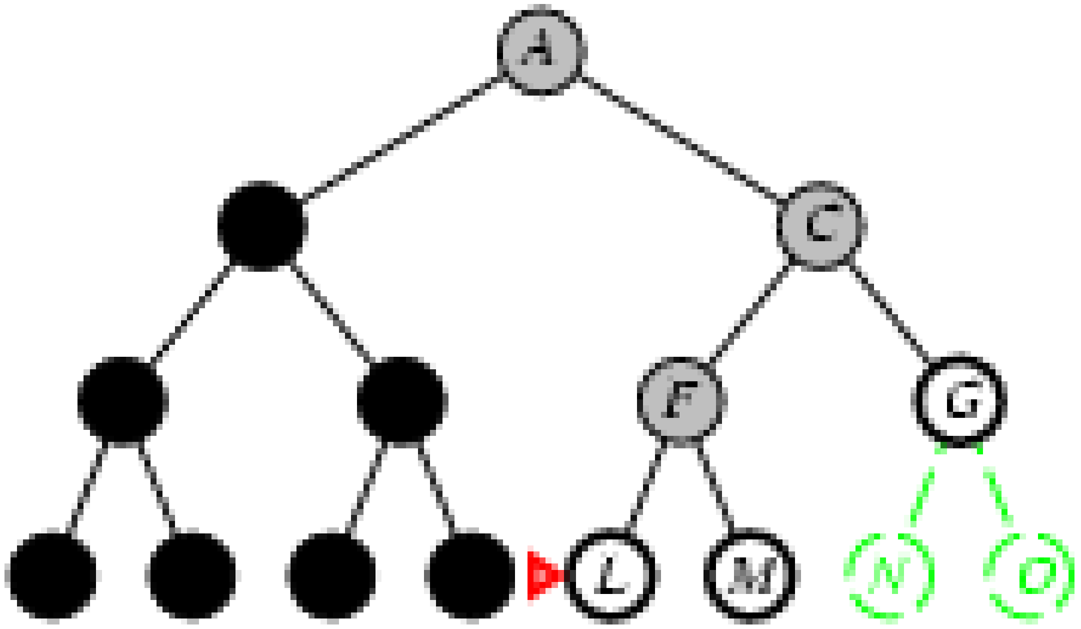
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
    - *fringe* = LIFO queue, i.e., put successors at front
    -



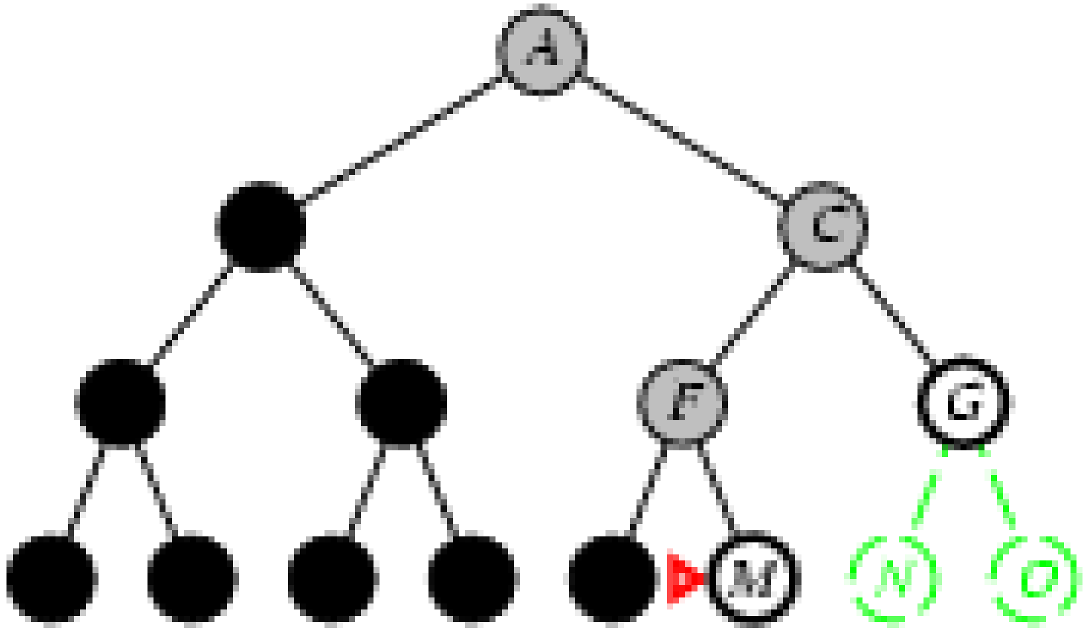
Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



Depth-first search

- Expand the *deepest* unexpanded node in the current fringe.
-
- Implementation:
    - *fringe* = LIFO queue, i.e., put successors at front
    -



Depth-first Properties

Strategy? ▸ Example

Last-in First Out Stack

Complete?

Yes, if finite number of states and no cyclic paths

Optimal?

No, lower goal states may be found first

Worst Case Time Complexity?

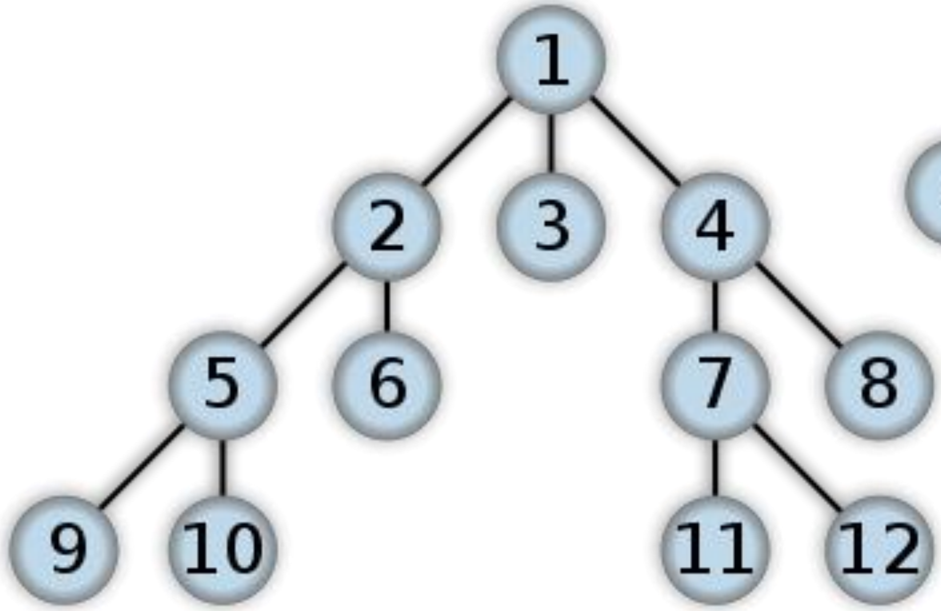
$O(b^m)$, branching factor b, maximum depth m

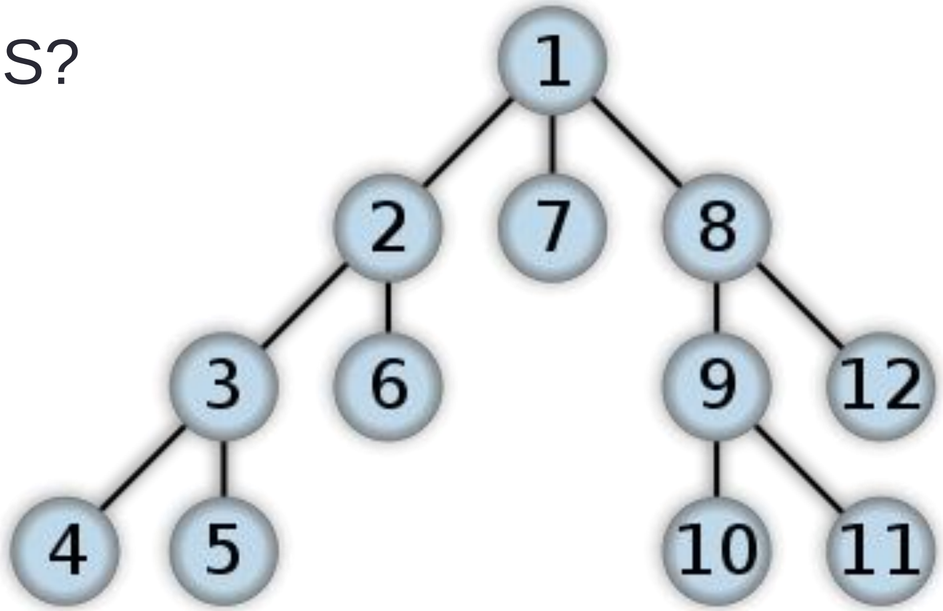
Worst Case Space Complexity?

$O(bm)$, branching factor b, maximum depth m

BFS vs. DFS search

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





Depth-First vs. Breadth-First

- When a breadth-first search succeeds, it finds a minimum-depth (nearest the root) goal node
- When a depth-first search succeeds, the found goal node is not necessarily minimum depth

- For a large tree, breadth-first search memory requirements may be excessive
- For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node

How can we combine the advantages and avoid the disadvantages of these two search techniques?

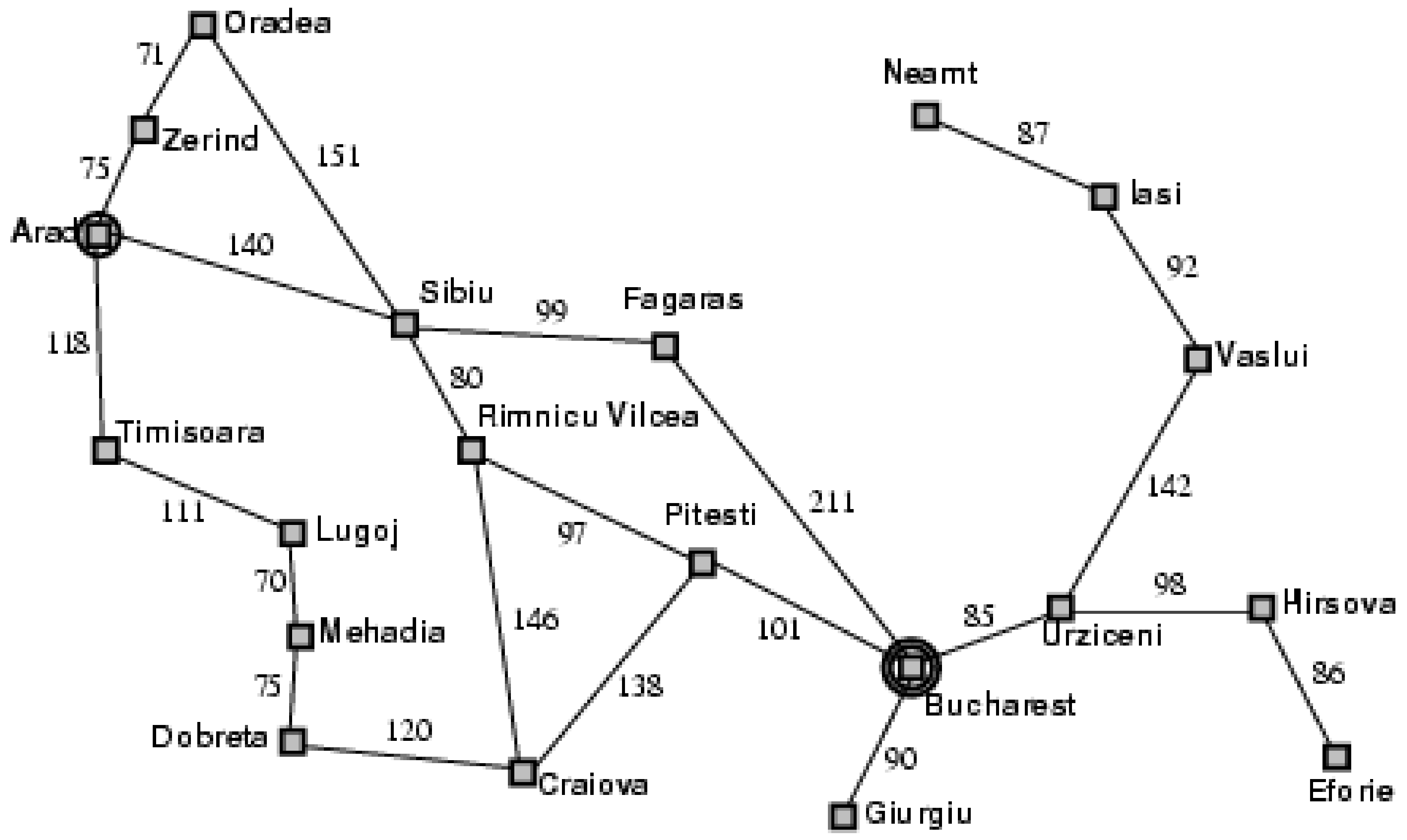
Depth-limited search

- The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors.
- Sometimes, depth limits can be based on knowledge of the problem.

  For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $l = 19$ is a possible choice.

  But in fact if we studied the map carefully, we would discover that any city can be reached from any other city at most 10 steps (the **diameter** of the state space), which leads to a more efficient depth-limited search.
- For most problems, however, we will not know a good depth limit until we have solved the problem.

Depth-limited search



BFS vs. DFS search

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

- **Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it's not necessary to store all of the child pointers at each level.**
- **Depending on the data and what you are looking for, either DFS or BFS could be advantageous.**

<u>**For example**</u>

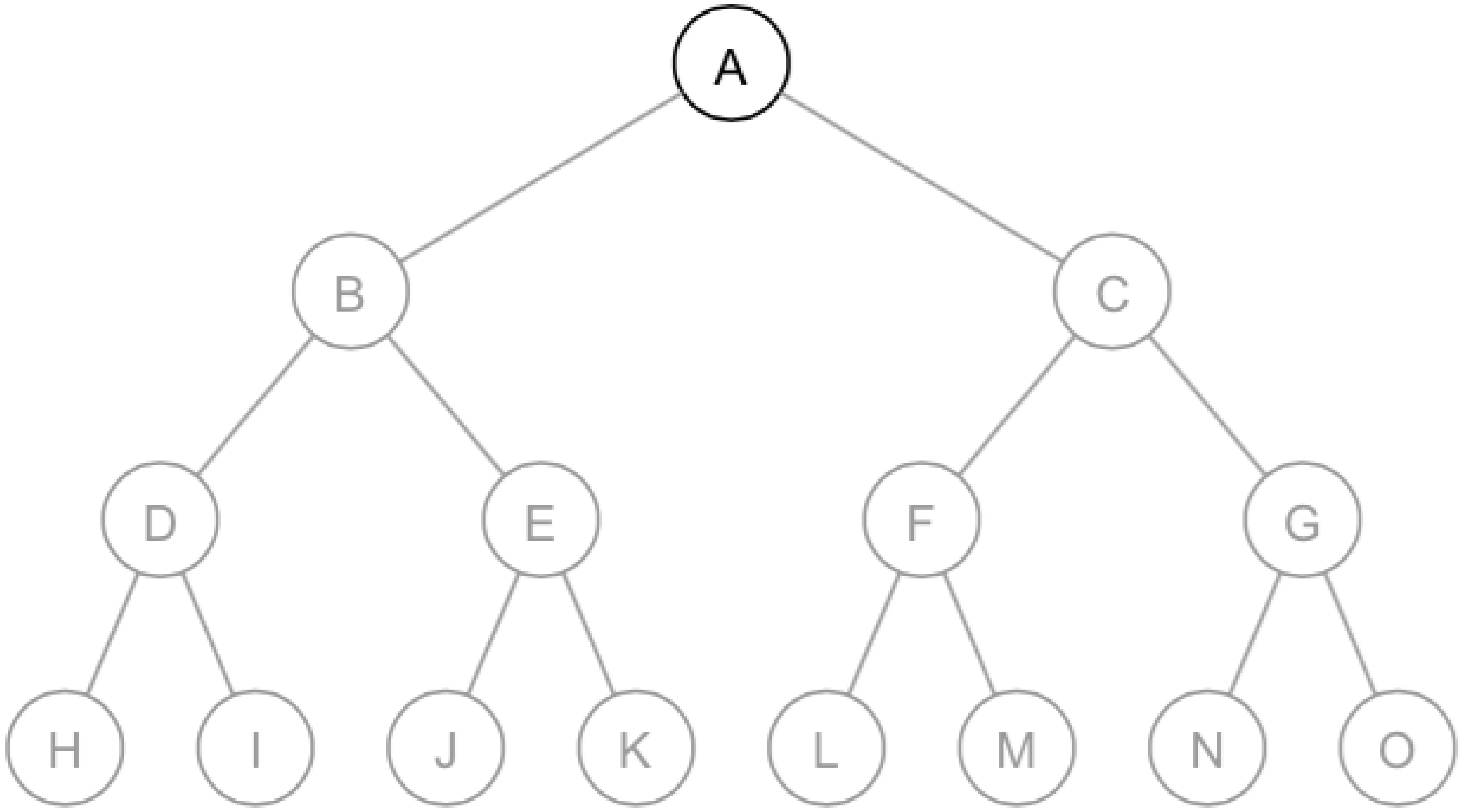
Given a family tree, if one were looking for someone on the tree who's still alive, then it would be safe to assume that person would be on the bottom of the tree. This means that a BFS would take a very long time to reach that last level. A DFS, however, would find the goal faster.

But, if one were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree.

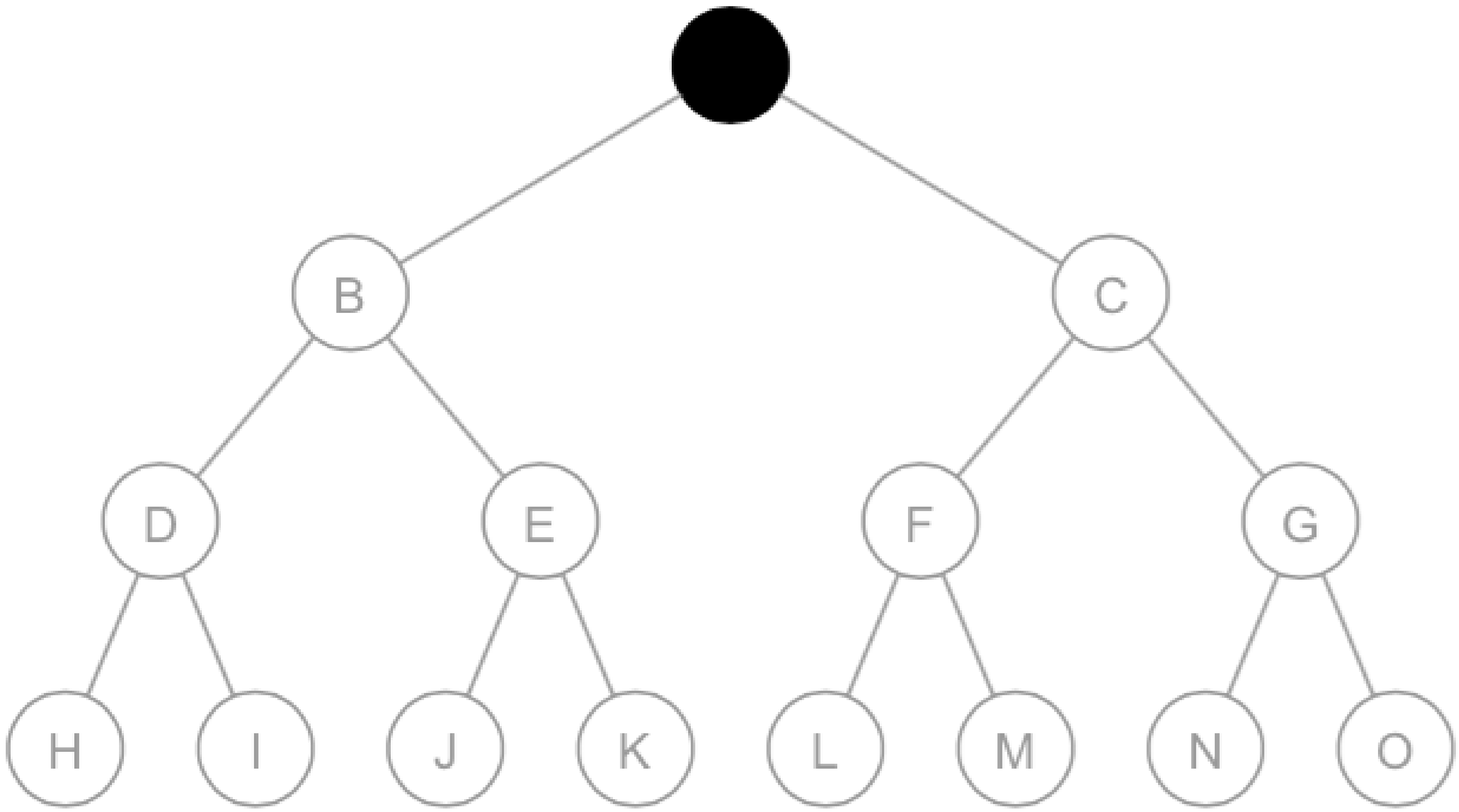
Iterative deepening search

- The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits.
- It does this by increasing the limit gradually, first 0, second 1, third 2, and so on until a goal is found.
- This (i.e. goal is found) will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Algorithm is below, it combines the benefits of depth-first and breadth-first search,
- Like depth-first search, its memory requirements are very modest,
- Like breadth-first search, it is complete when the branching factor is finite and optimal as well.
- It terminates when a solution is found or if the depth- limited search returns failure, meaning that no solution exists.

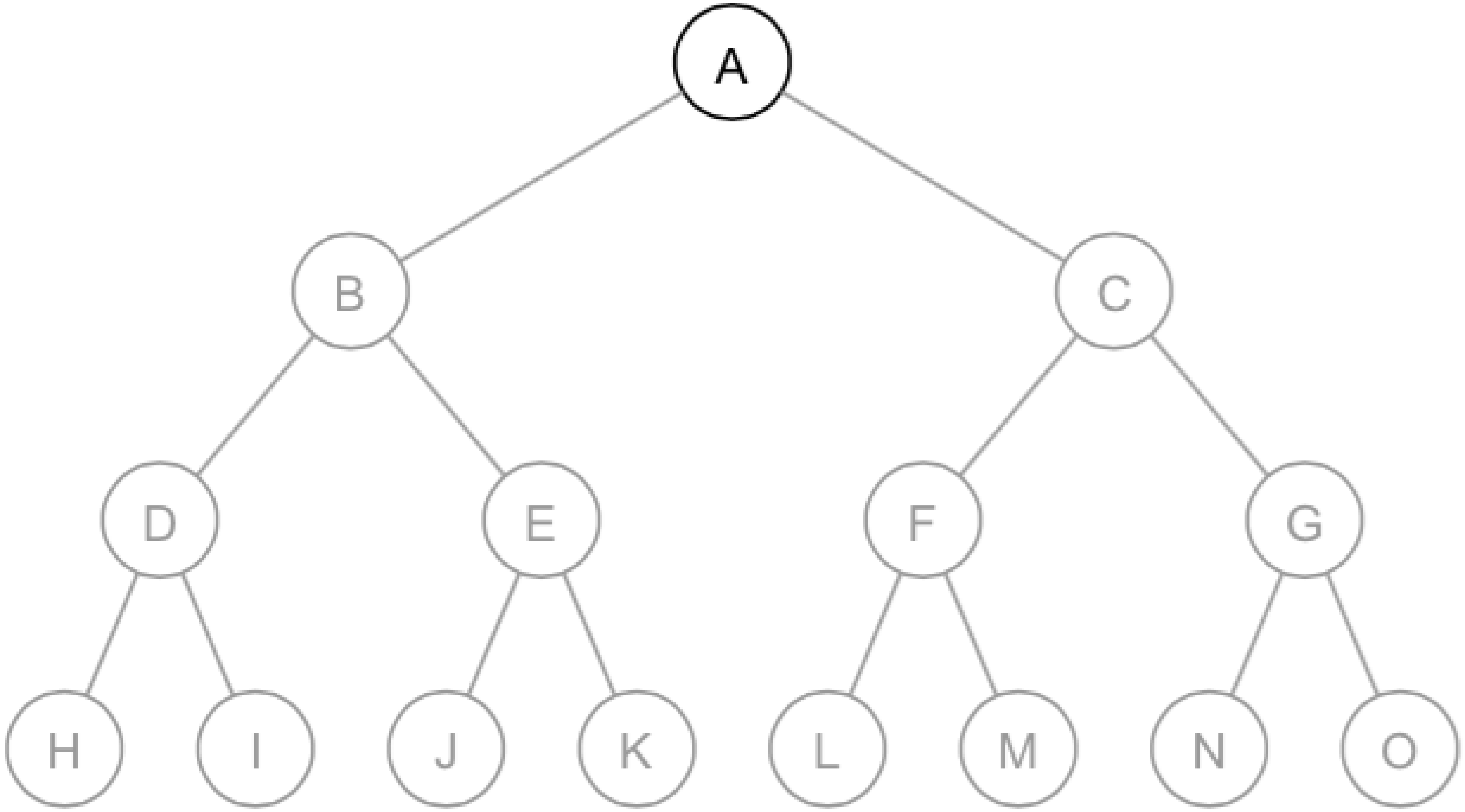
Iterative deepening search / =0



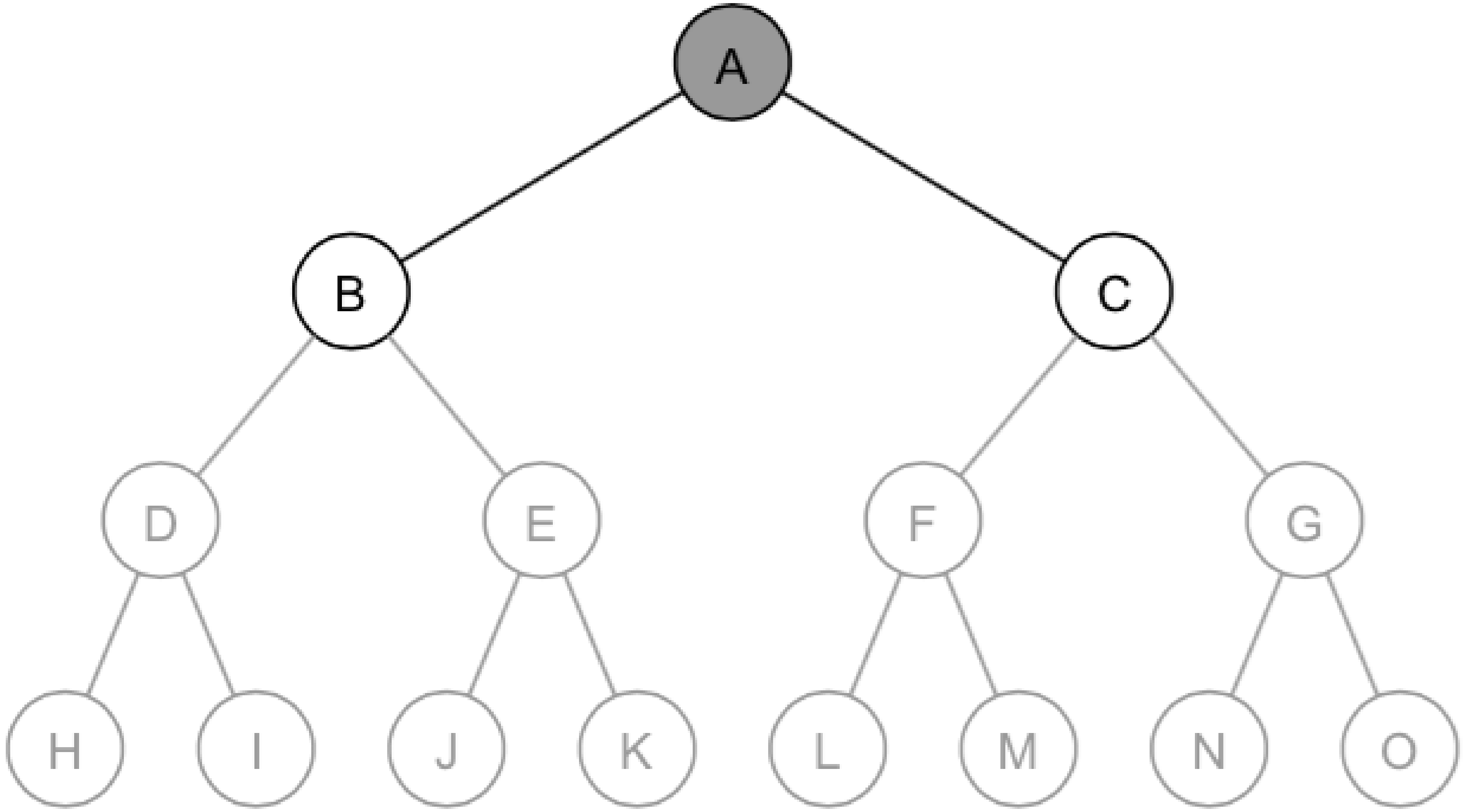
Iterative deepening search / =0



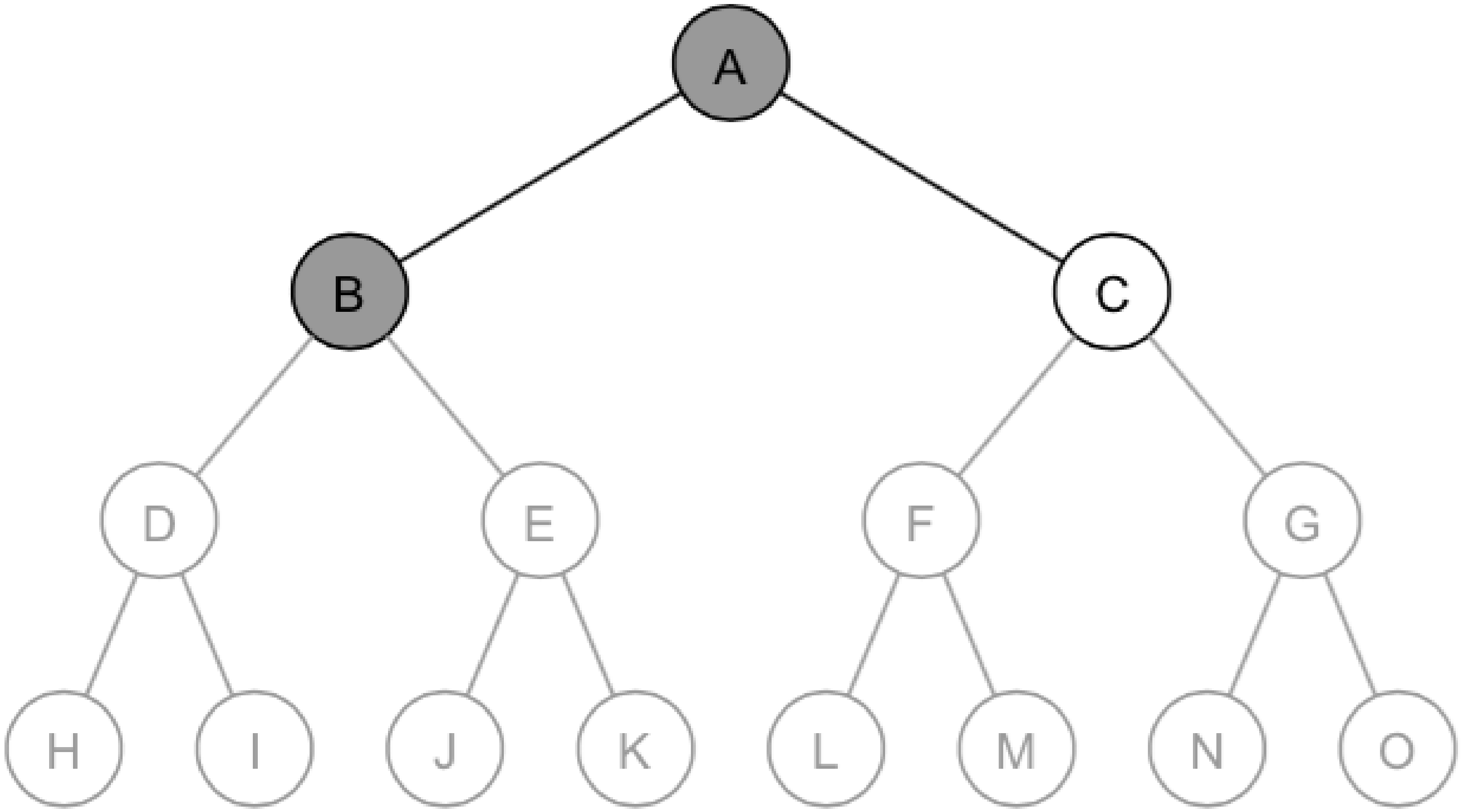
Iterative deepening search / =1



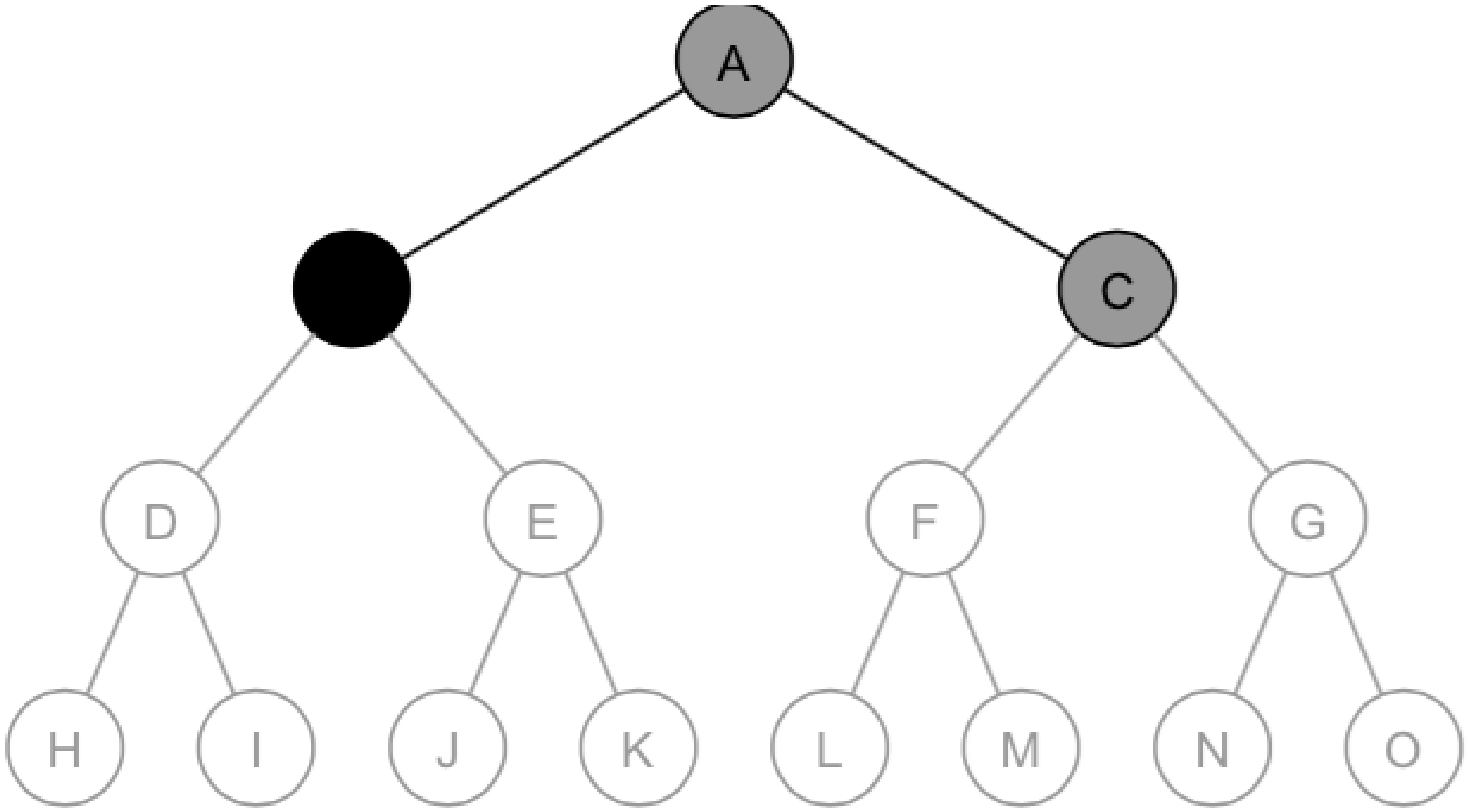
Iterative deepening search / =1



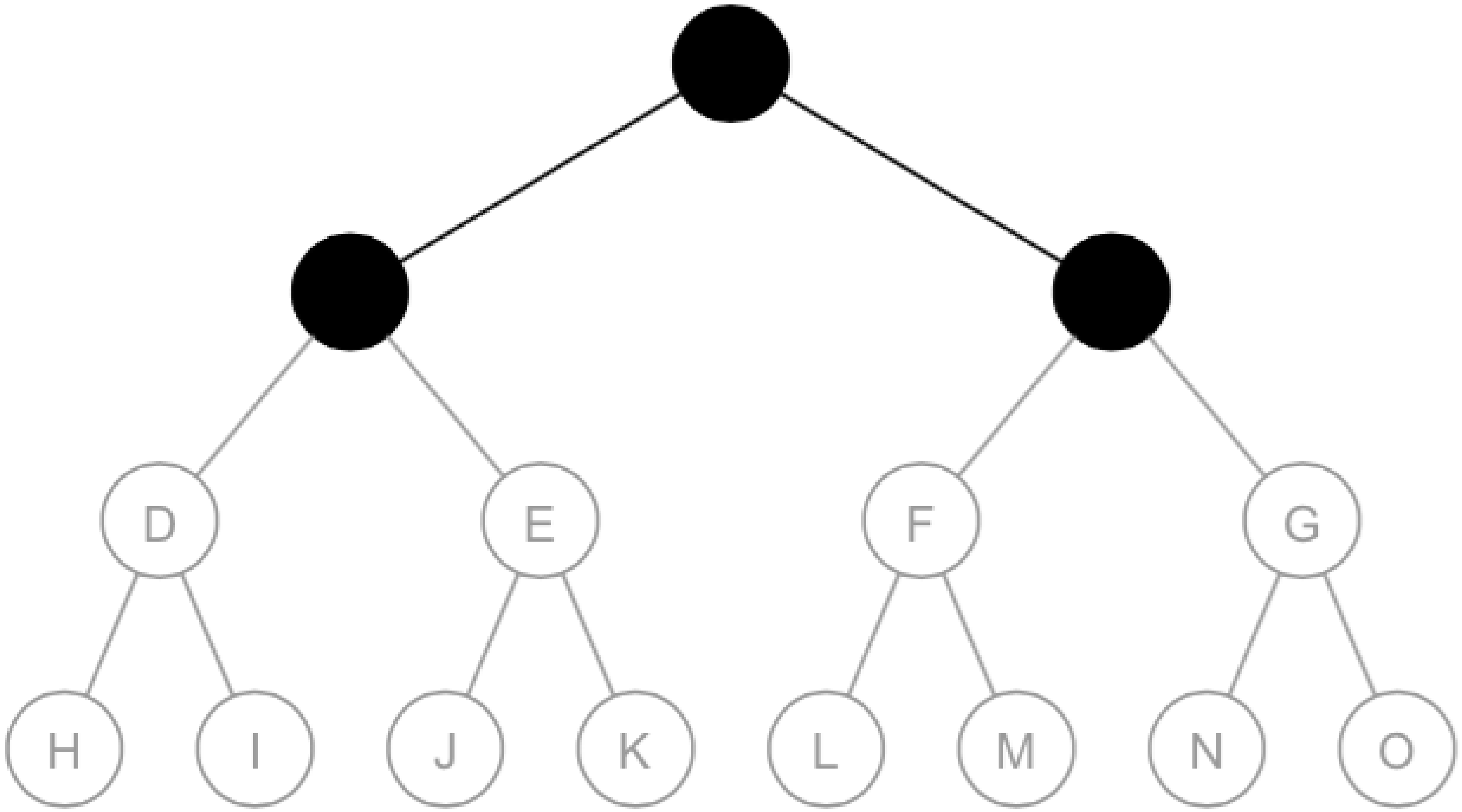
Iterative deepening search / =1



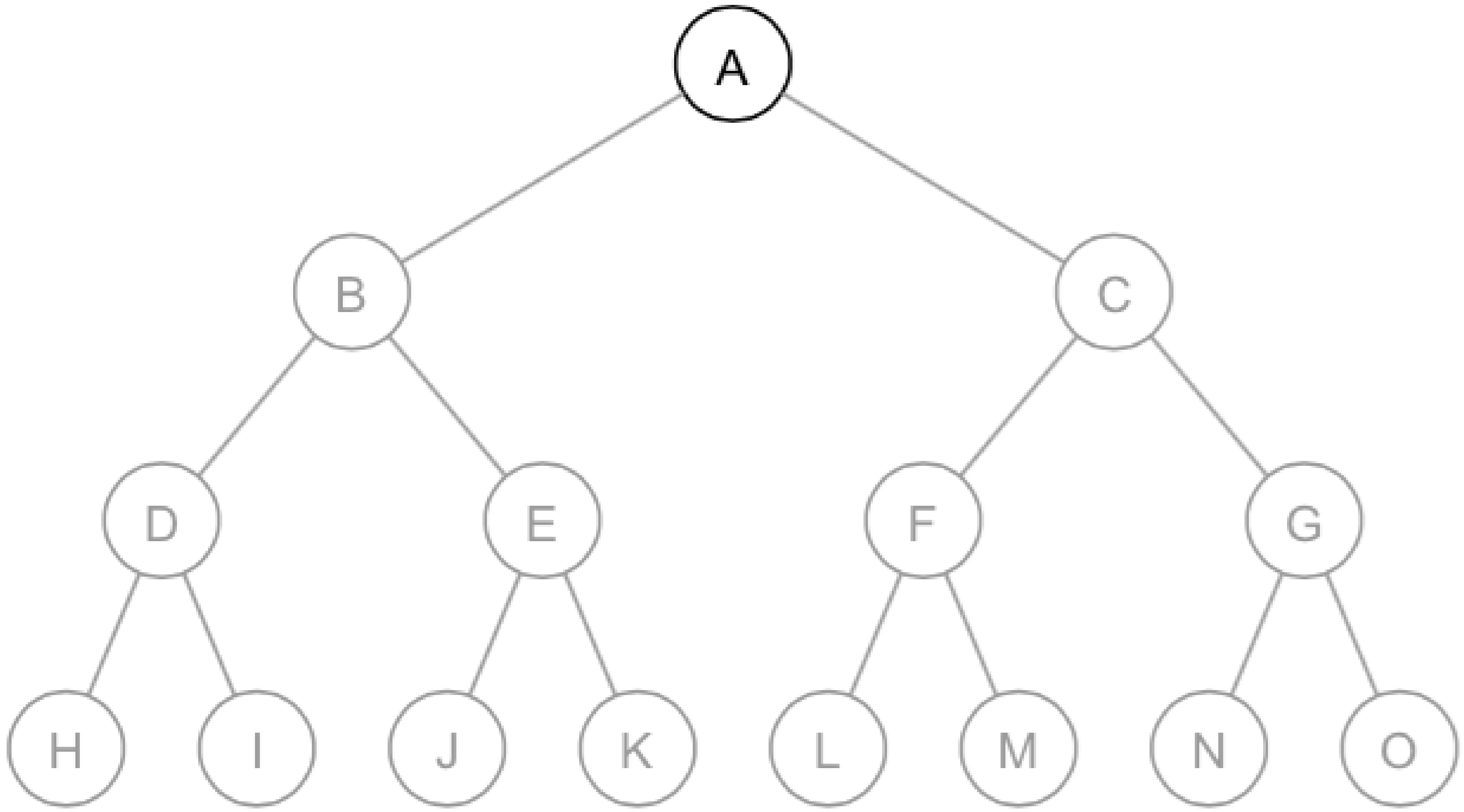
Iterative deepening search / =1



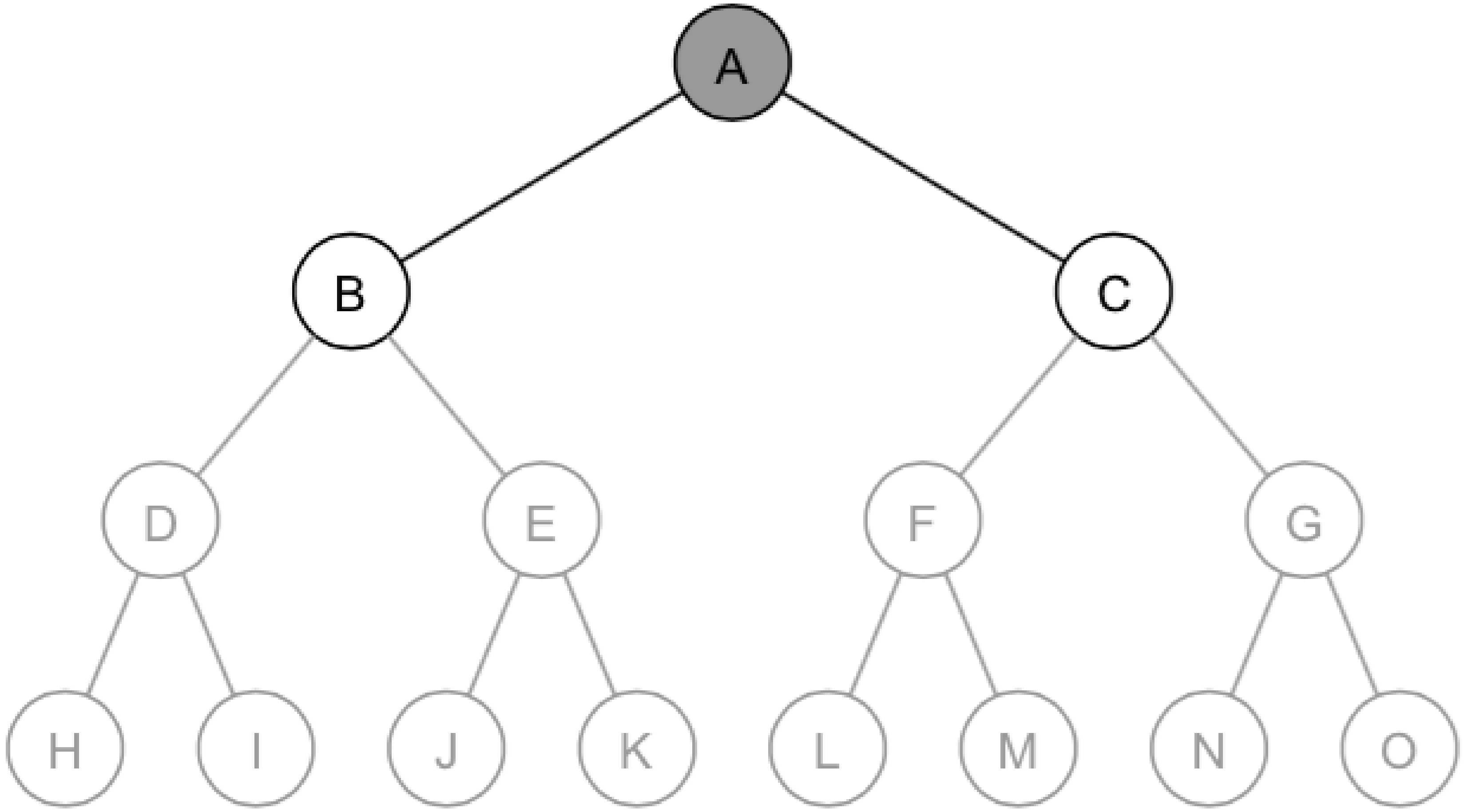
Iterative deepening search / =1



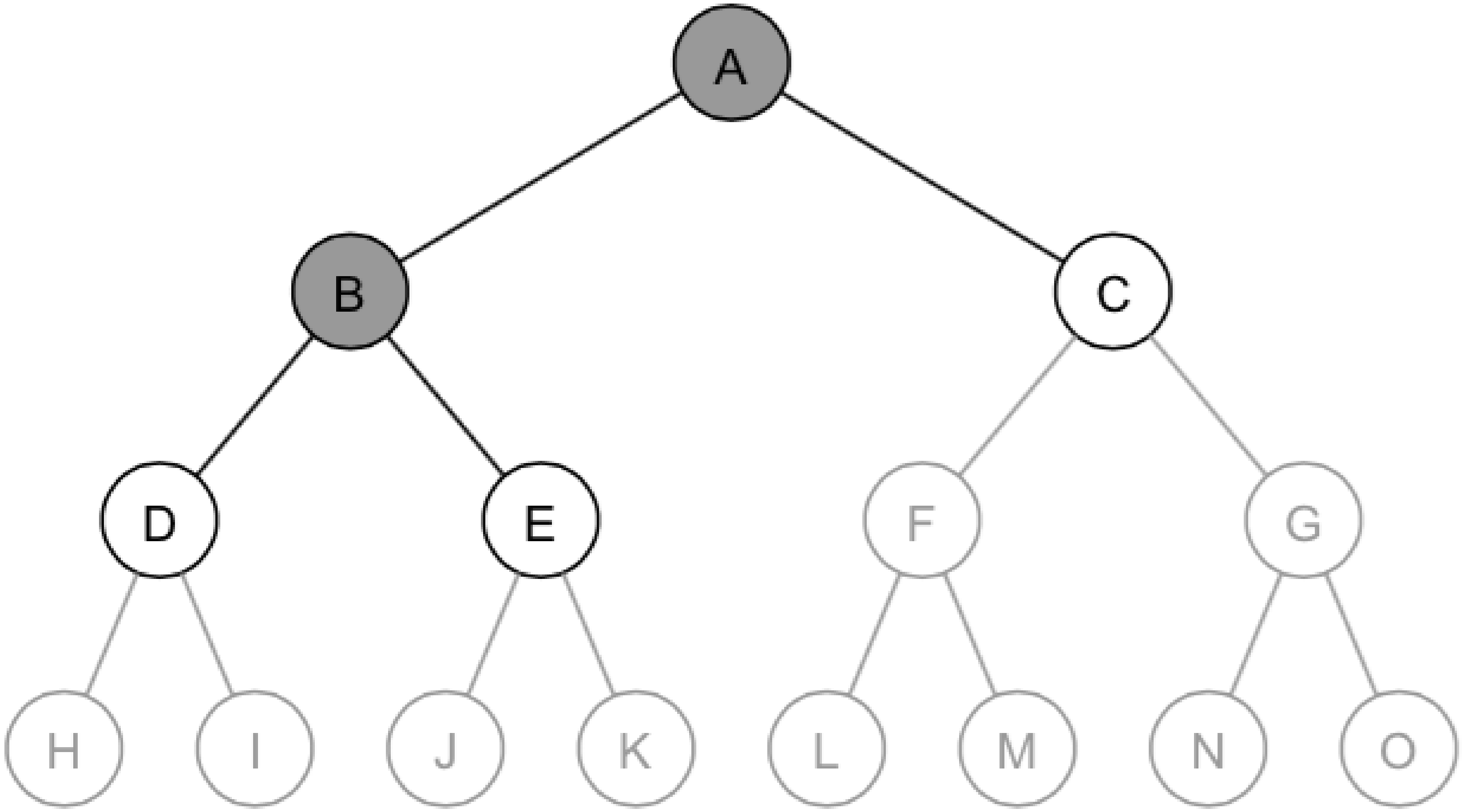
Iterative deepening search / =2



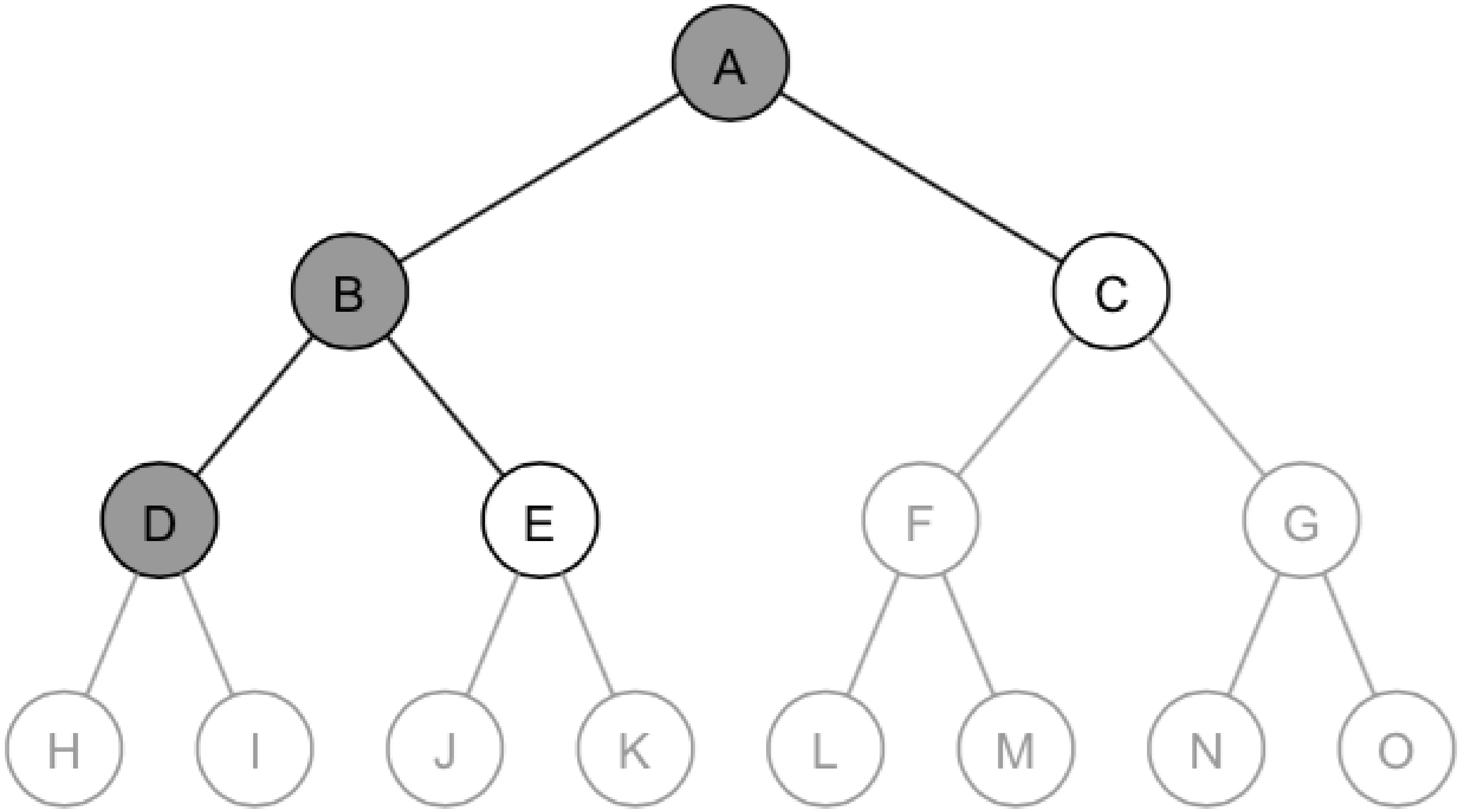
Iterative deepening search / =2



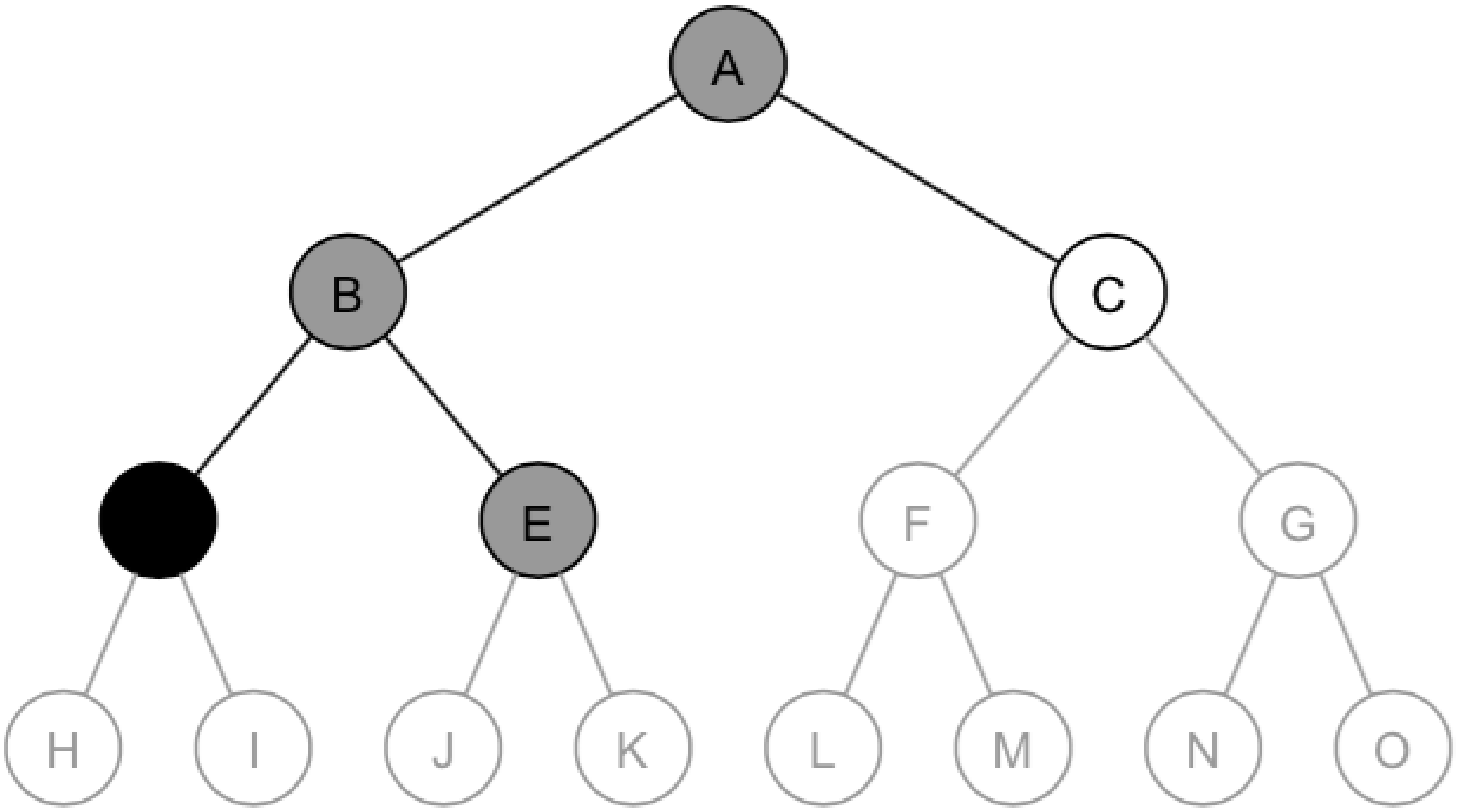
Iterative deepening search / =2



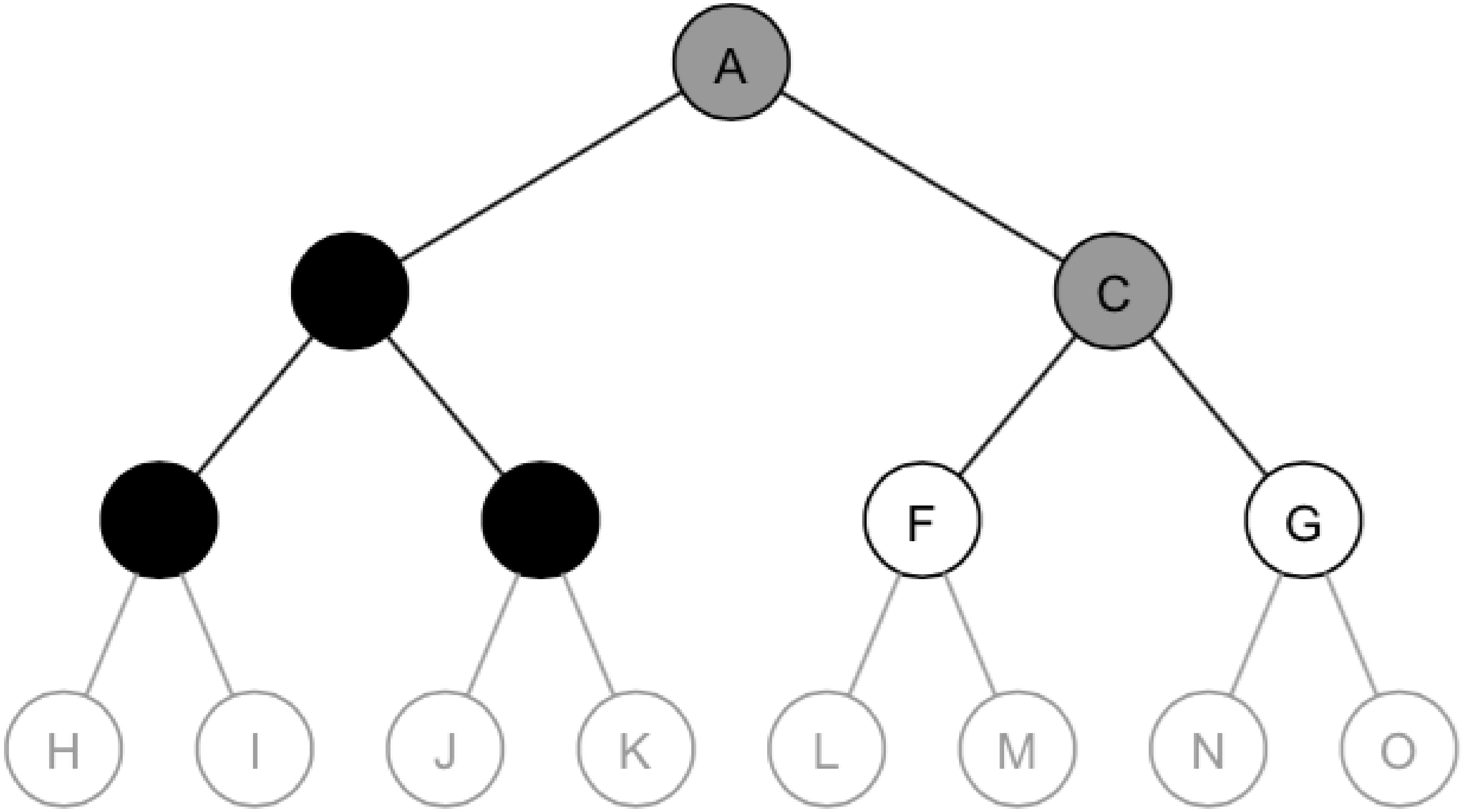
Iterative deepening search / =2



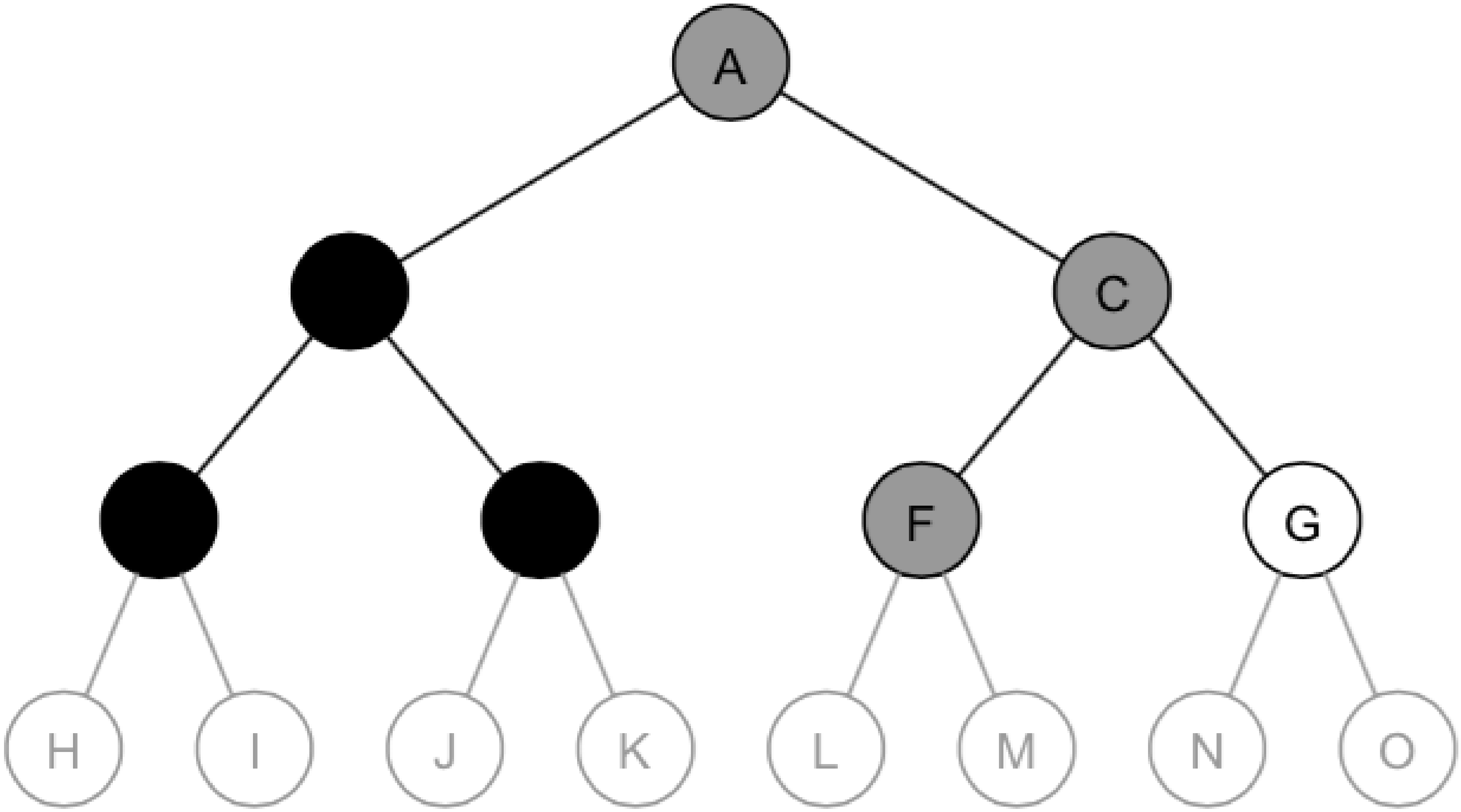
Iterative deepening search / =2



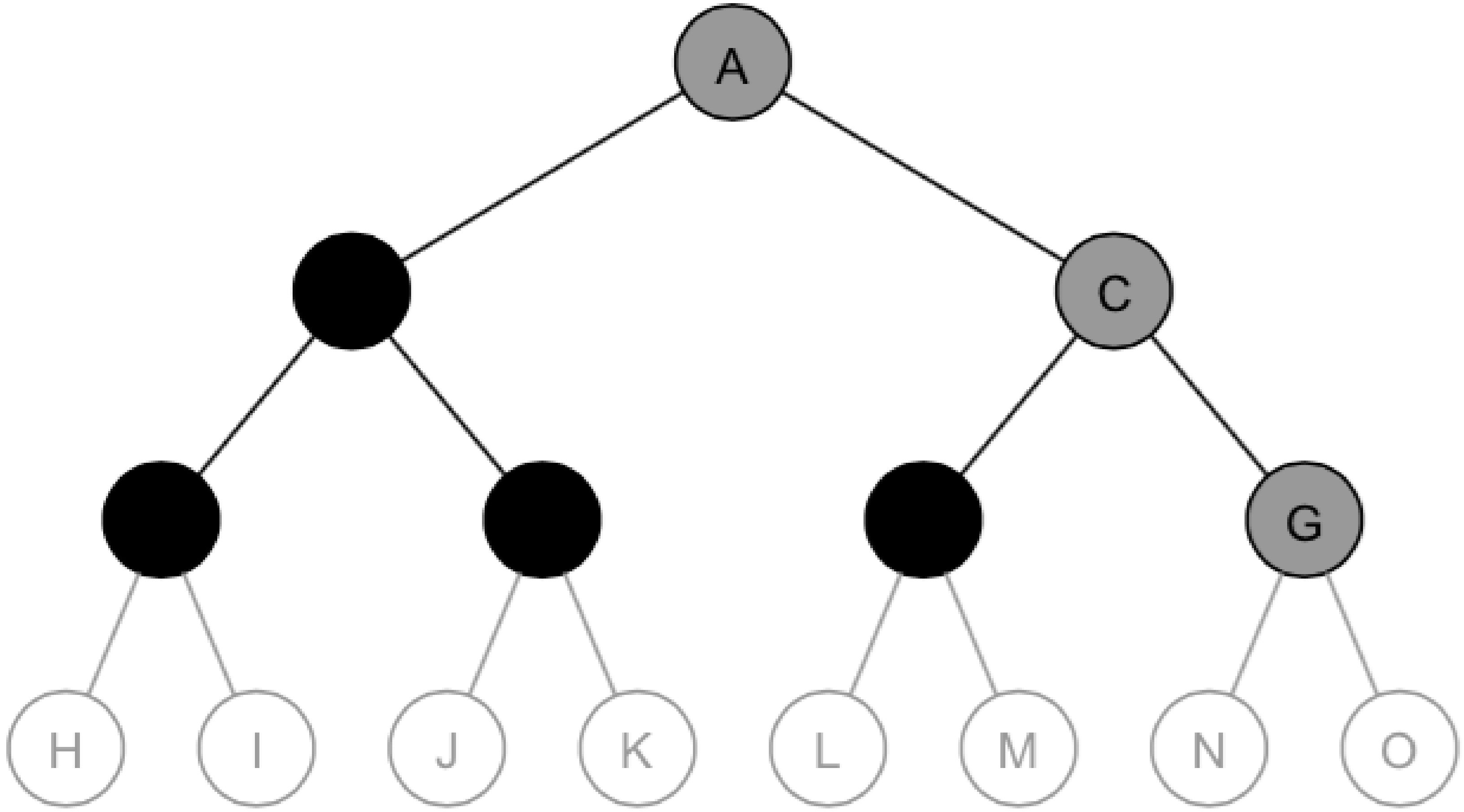
Iterative deepening search / =2



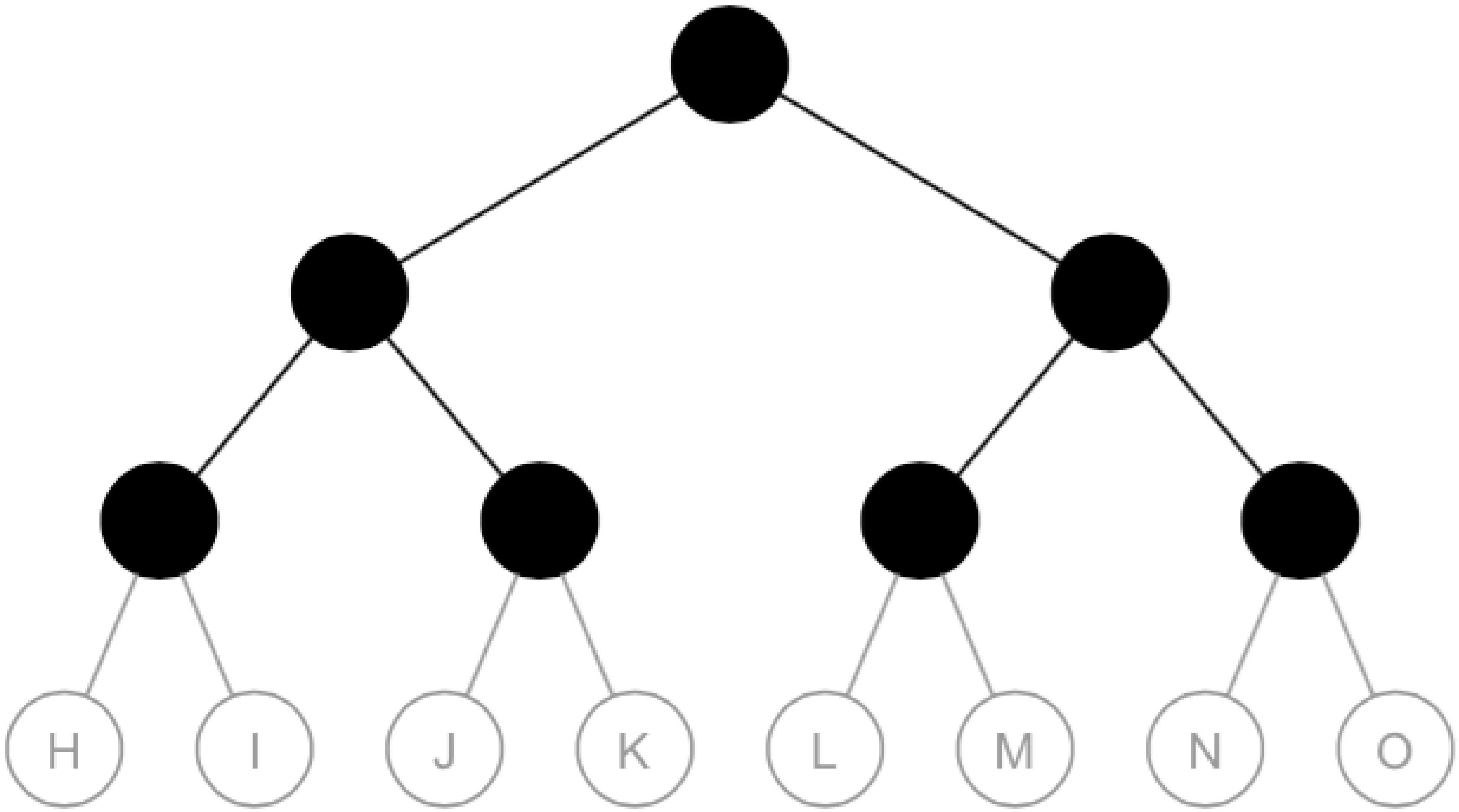
Iterative deepening search / =2



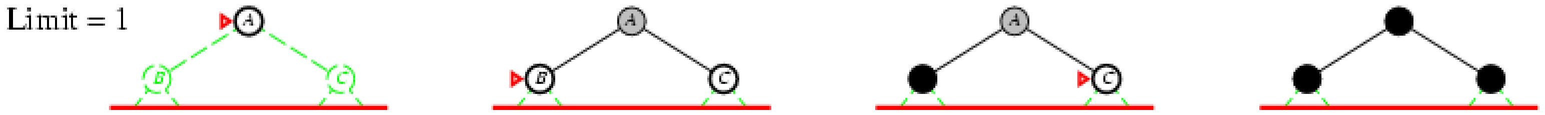
Iterative deepening search *l* =2



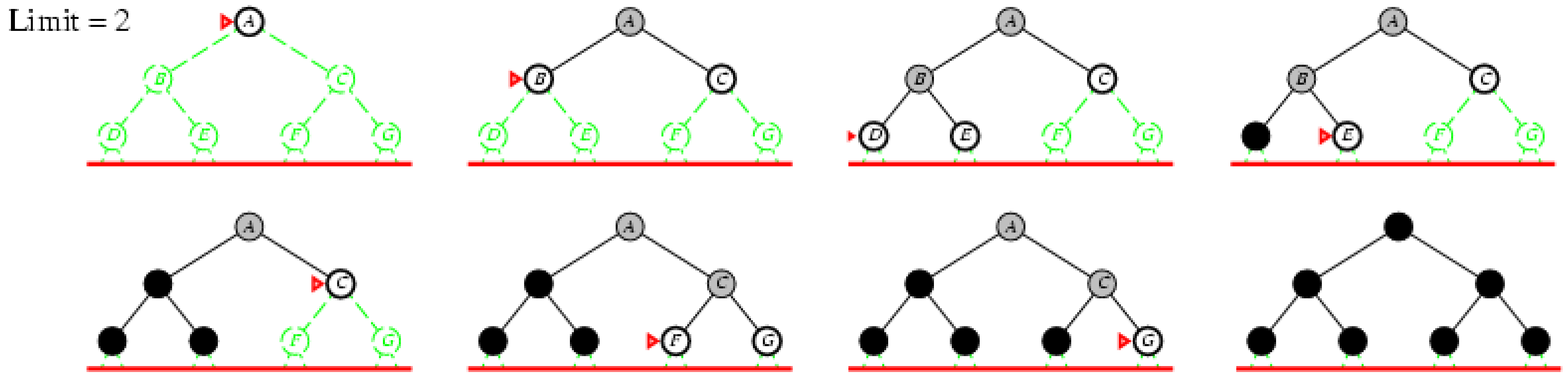
Iterative deepening search / =2



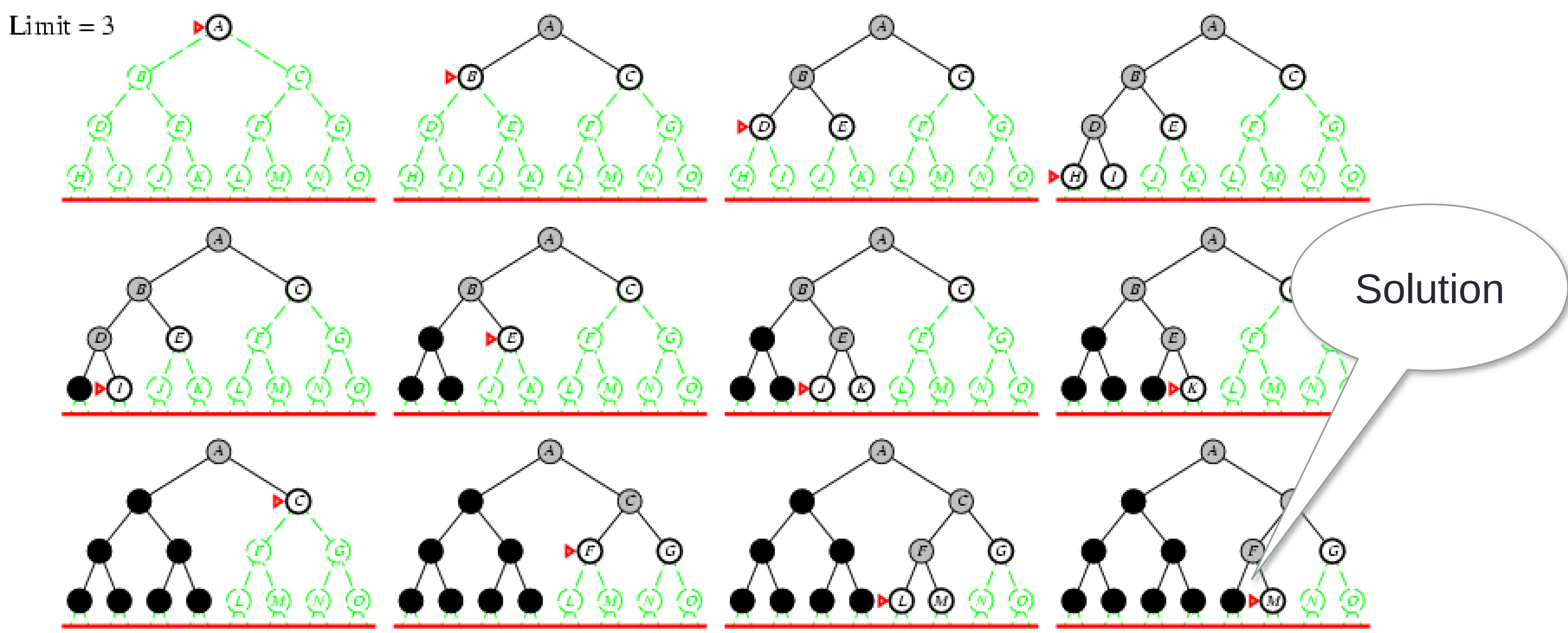
Iterative deepening search *l* =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3

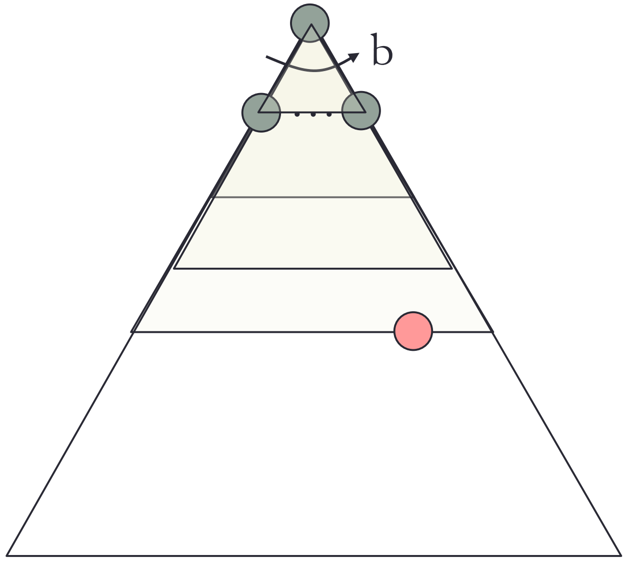


- It may seem wasteful, because states are generated multiple times.
- However, this is not very costly! The reason is that in a search tree with the same branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

Iterative deepening search

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - 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!



Iterative deepening vs. Breadth first

In an iterative deepening search, the nodes on the bottom level (depth *d*) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated *d* times. So the total number of nodes generated is:

$$N(IDS) = (d)\, b + (d-1)\, b^2 + \ldots + (1) b^d,$$

$$N(BFS) = b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b)$$

- The former shows time complexity of $O(b^d)$ where the second is $O(b^{d+1})$,
- Iterative deepening is FASTER than Breadth First search, despite the repeated generation of states. For example, if $b=10$ and $d=5$, then we have:

  N (IDS) = 50+400+3.000+20.000+100.000 = 123.450
  N(BFS) = 10+100+1.000+10.000+100.000+999.990=1,111,100

Iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Iterative deepening vs. Depth-first

❖ **Time requirements on binary tree with 7 levels**

Nodes at each level	**Nodes searched by DFS**	**Nodes searched by iterative IDS**
1	1	1
2	+2 = 3	+3 = 4
4	+4 = 7	+7 = 11
8	+8 = 15	+15 = 26
16	+16 = 31	+31 = 57
32	+32 = 63	+63 = 120
64	+64 = 127	+127 = 247
128	+128 = **255**	+255 = **502**

Iterative deepening vs. Depth-first

❖ Time requirements for depth-first iterative deepening on branching factor 4

Nodes at each level	Nodes searched by DFS	Nodes searched by iterative IDS
1	1	1
4	+4 = 5	+5 = 6
16	+16 = 21	+21 = 27
64	+64 = 85	+85 = 112
256	+256 = 341	+341 = 453
1024	+1024 = 1365	+1365 = 1818
4096	+4096 = 5461	+5461 = 7279
16384	+16384 = **21845**	+21845 = **29124**

Iterative deepening vs. Depth-first

- When searching a binary tree to depth 7:
  - **DFS** requires searching 255 nodes
  - **IDS** requires searching 502 nodes
  - **IDS** takes only about twice as long
- When searching a tree with branching factor of 4 (each node may have four children):
  - **DFS** requires searching 21845 nodes
  - **IDS** requires searching 29124 nodes
  - **IDS** takes about 4/3 = 1.33 times as long

The higher the branching factor, the lower the relative cost of iterative deepening depth first search

Iterative Deepening Properties

Strategy? ‣ Example

Last-in First Out with depth limit

Complete?

Yes, if number of branches is finite

Optimal?

Yes, if step costs are all identical

Worst Case Time Complexity?

$O(b^d)$, branching factor b, depth of goal state d

Worst Case Space Complexity?

$O(bd)$, branching factor b, depth of goal state d

Avoiding Repeated States

- There is a possibility of wasting time by expanding states that have already been encountered and expanded before.
- Failure to detect repeated states can turn a <u>linear</u> problem into an <u>exponential</u>
- If an algorithm <u>remembers every state that it has visited</u>, then it can be viewed as exploring the state-space graph directly.
- We can modify the general TREE-SEARCH algorithm to include a data structure called the <u>**closed list**</u>, which stores every expanded node.
- If the current node matches a node on the closed list, it is discarded instead of being expanded.

SUMMARY

- This chapter has introduced methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the: agent can construct sequences of actions that achieve its goals; this process is called **search.**
- Before an agent can start searching for solutions, it must formulate a **goal** and then use the goal to formulate a **problem.**
- A problem consists of four parts: the **initial state,** a set of **actions,** a **goal test** function, and a **path cost** function. The environment of the problem is represented by a **state space.** A **path** through the state space from the initial state to a goal state is a **solution.**
- Search algorithms are judged on the basis of **completeness, optimality, time complexity,** and **space complexity.** Complexity depends on *b,* the branching factor in the state space, and *d*, the depth of the shallowest solution.
- **Breadth-first search** selects the shallowest unexpanded node in the search tree for expansion. It is complete, optimal for unit step costs, and has time and space complexity of $O(b^{d+1})$. The space complexity makes it impractical in most cases.

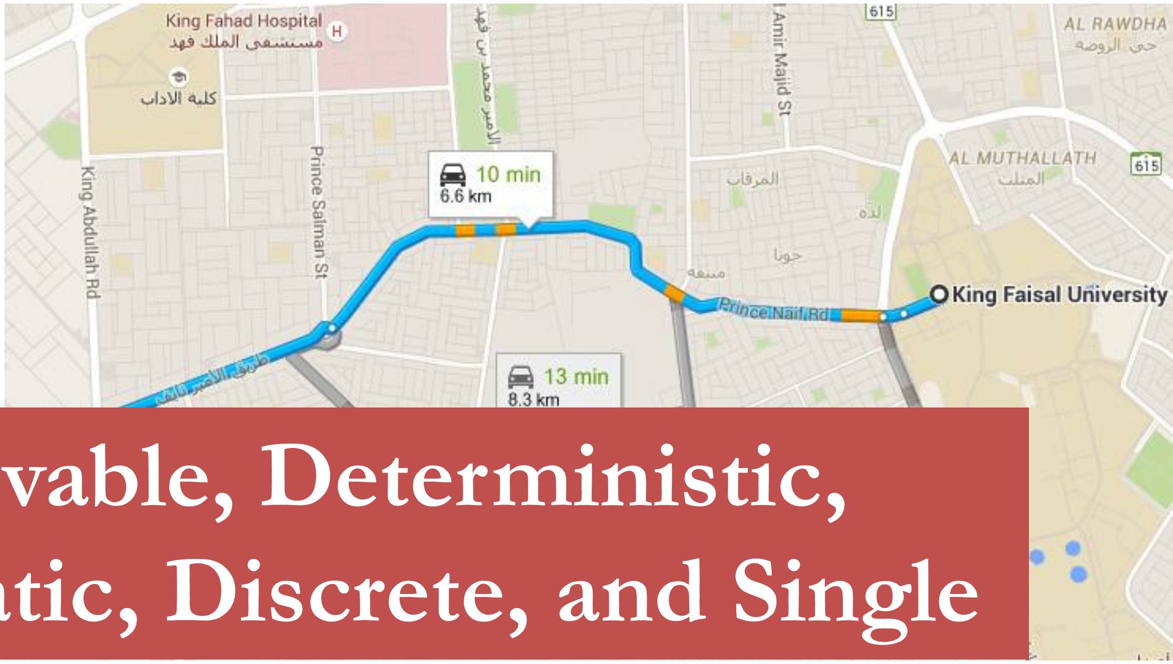
SUMMARY

- **Uniform-cost search** is similar to breadth-first search but expands the node with lowest path cost, *g (n)* . It is complete and optimal if the cost of each step exceeds some positive bound.
- **Depth-first search** selects the deepest unexpanded node in the search tree for expansion. It is neither complete nor optimal, and has time complexity of $O(b^m)$ and space complexity of $O(bm)$, where m is the maximum depth of any path in the state space.
- **Depth-limited search** imposes a fixed depth limit on a depth-first search.
- **Iterative deepening search** calls depth-limited search with increasing limits until a goal is found. It is complete, optimal for unit step costs, and has time complexity of $O(b^d)$ and space complexity of $O(b^d)$.

Exercises

Q1: What are the properties of a search problem?

- Fully or Partially Observable?
- Deterministic or Stochastic?
- Episodic or Sequential?
- Static or Dynamic?
- Discrete or Continuous?
- Single or Multi-Agent?



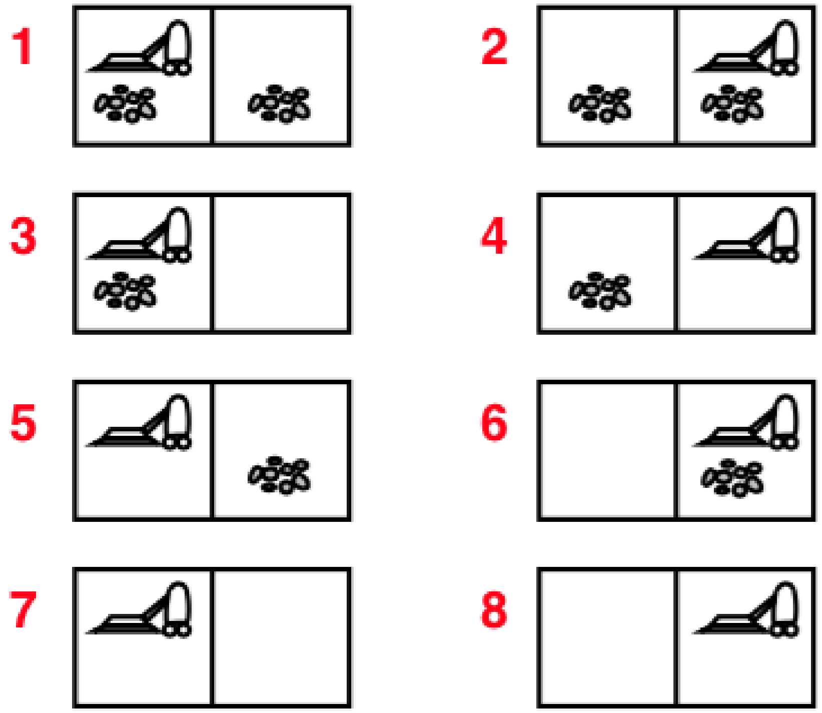
Fully observable, Deterministic, Sequential, Static, Discrete, and Single

Exercises

Q1: What is the solution of the Vacuum problem?

Problem

- Start in 1
- Left square actions: Suck or Right
- Right square actions: Suck or Left
- Success: 7 or 8
- Optimal: fewest actions



Exercises

Please try to solve the following exercises:

3.1

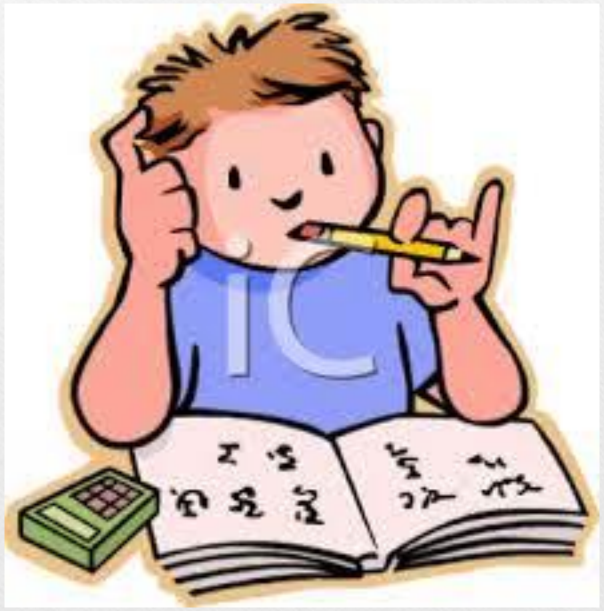
3.2

3.6

3.7-a

3.8- a,b

3.10-3.13



Any questions?