§2: Search Problems

CS 325 - Artificial Intelligence

Search Problem

- Rational agents need to perform sequences of actions in order to achieve goals.
- Intelligent behavior can be generated by having a look-up table or reactive policy that tells the agent what to do in every circumstance, but:
  Such a table or policy is difficult to build
  All contingencies must be anticipated
- A more general approach is for the agent to have knowledge of the world and how its actions affect it and be able to simulate execution of actions in an internal model of the world in order to determine a sequence of actions that will accomplish its goals.
- This is the general task of problem solving and is typically performed by searching through an internally modelled space of world states.

Search Problem

A "search problem" in the context of artificial intelligence (AI) refers to a class of problems where an AI agent or system needs to find a solution or a sequence of actions that lead to a goal state in a large, often complex, and sometimes unknown search space.
Search problems are fundamental in AI and can be found in various domains, including robotics, natural language processing, game playing, and more.

Search Problem

Search: Searching is a step-by-step procedure to solve a search-problem in AI domain. A search problem can have various factors:

- Search Space: This is the set of all possible states or configurations that the AI agent can explore. It's often represented as a graph, tree, or some other data structure. For example, in a chess game, the search space represents all possible board positions.
- Initial State: This is the starting point from which the search begins. It's the current state of the problem.
- Goal State: This is the desired state that the AI agent aims to reach. The goal state defines what constitutes a successful solution to the problem.

Search Problem

- Actions: These are the allowable moves or operations that the AI agent can perform to transition from one state to another. In chess, actions correspond to legal moves.
- Transition Model: This defines the result of applying an action to a particular state. It describes how the state changes as a result of an action.
- Cost Function: In many search problems, there's a cost associated with each action. The cost function is used to measure how expensive or desirable a particular sequence of actions is.

Search Problem

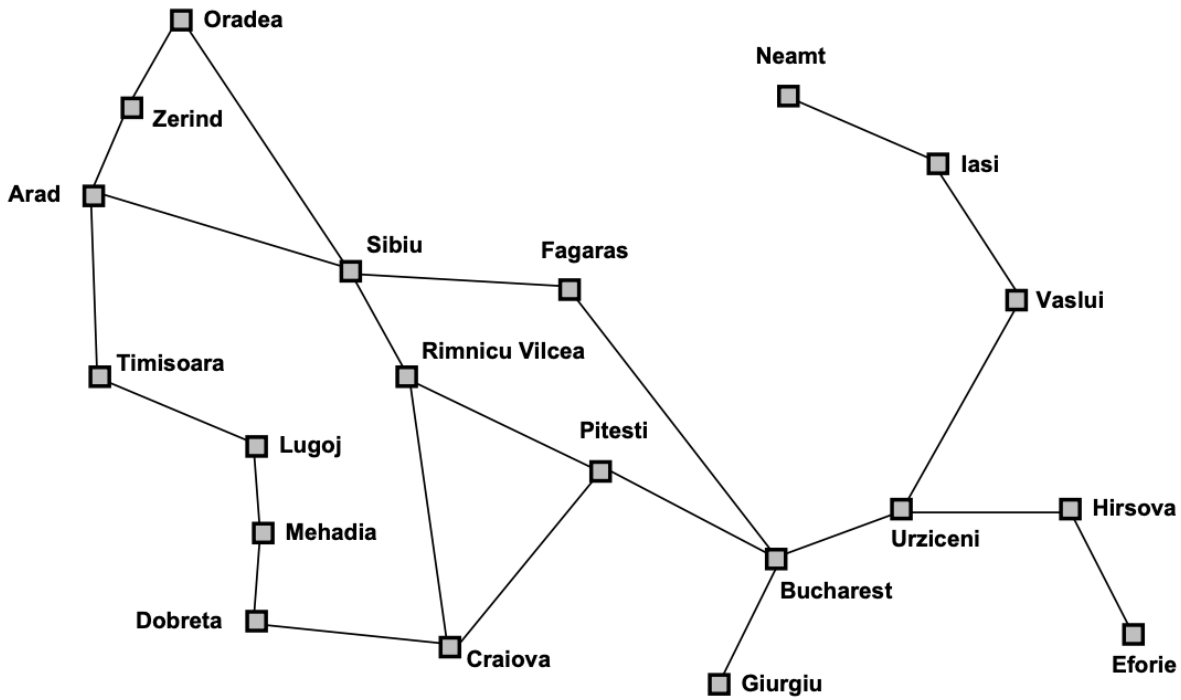
Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
Actions: It gives the description of all the available actions to the agent.
Transition model: A description of what each action do, can be represented as a transition model.
Path Cost: It is a function which assigns a numeric cost to each path.
Solution: It is an action sequence which leads from the start node to the goal node.
Optimal Solution: If a solution has the lowest cost among all solutions.

Search Problem

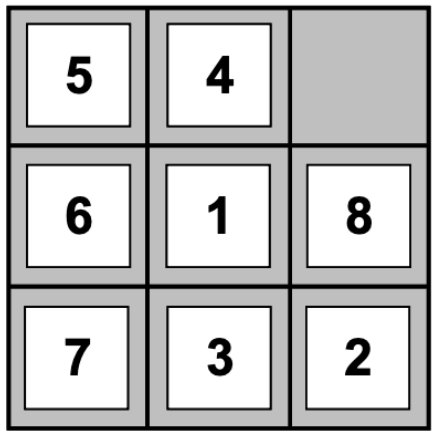
- Given:
  An initial state of the world
  A set of possible possible actions or operators that can be performed.
  A goal test that can be applied to a single state of the world to determine if it is a goal state.
- Find:
  A solution stated as a path of states and operators that shows how to transform the initial state into one that satisfies the goal test.
- The initial state and set of operators implicitly define a state space of states of the world and operator transitions between them. May be infinite.

Sample Route Finding Problem

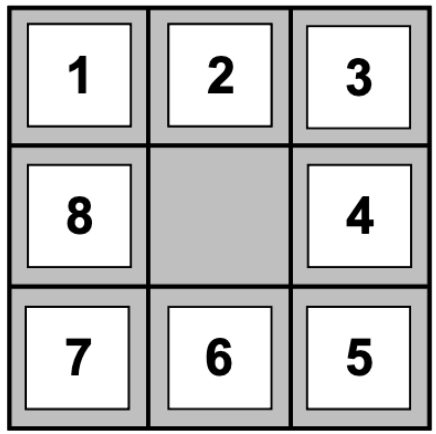
- Initial state: Arad
- Goal state: Bucharest
- Path cost: Number of intermediate cities, distance traveled, expected travel time



Sample Toy Problem

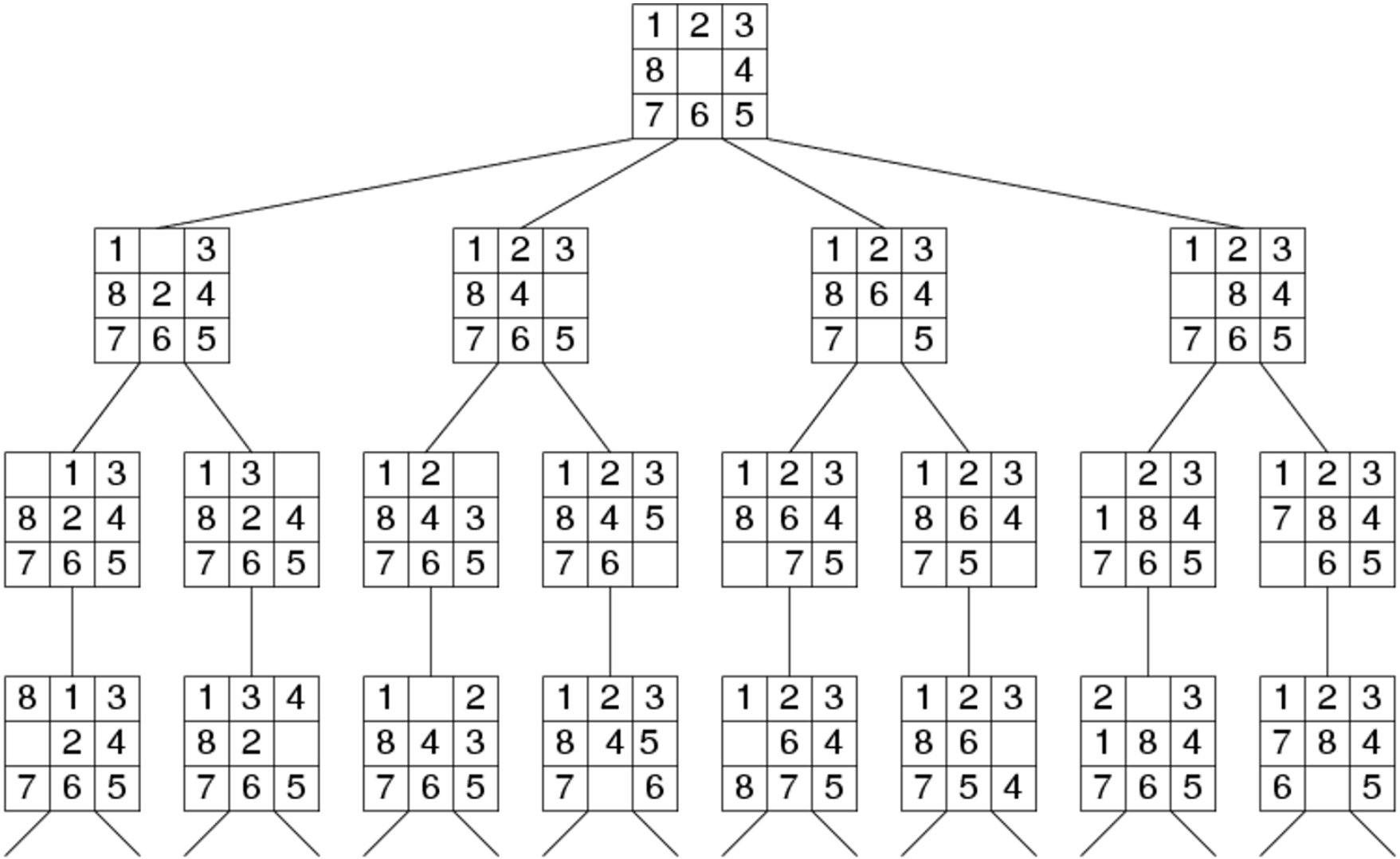


Start State



Goal State

Search Tree Example: 8-Puzzle Problem Space



© Daniel S. Weld

More Problems

- Route finding
- Robot navigation
- Web searching
- Travelling salesman problem
- VLSI layout

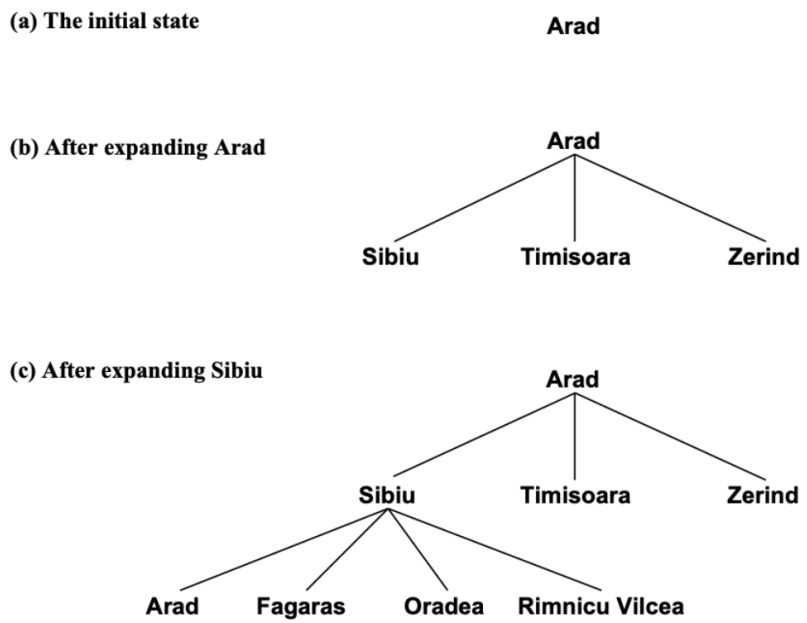
Searching Concepts

- A state can be expanded by generating all states that can be reached by applying a legal operator to the state.
- State space can also be defined by a successor function that returns all states produced by applying a single legal operator.
- A search tree is generated by generating search nodes by successively expanding states starting from the initial state as the root.
- A search node in the tree can contain
  Corresponding state
  Parent node
  Operator applied to reach this node
  Length of path from root to node (depth)
  Path cost of path from initial state to node

State Space Graph

- 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

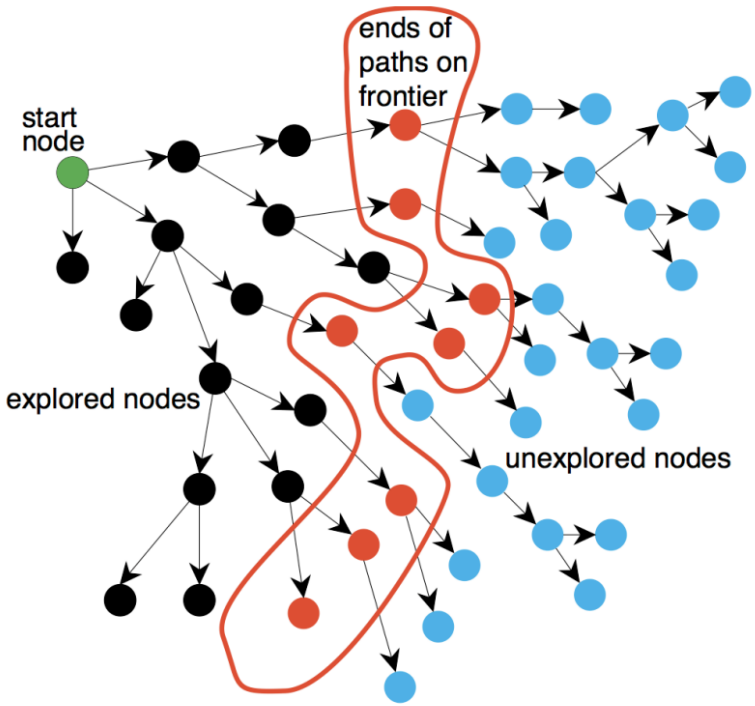
Expanding Nodes and Search




```
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
```

Expanding Nodes and Search

- Important ideas:
  Fringe
  Expansion
  Exploration strategy



Search Algorithm

- Easiest way to implement various search strategies is to maintain a queue of unexpanded search nodes.
- Different strategies result from different methods for inserting new nodes in the queue.

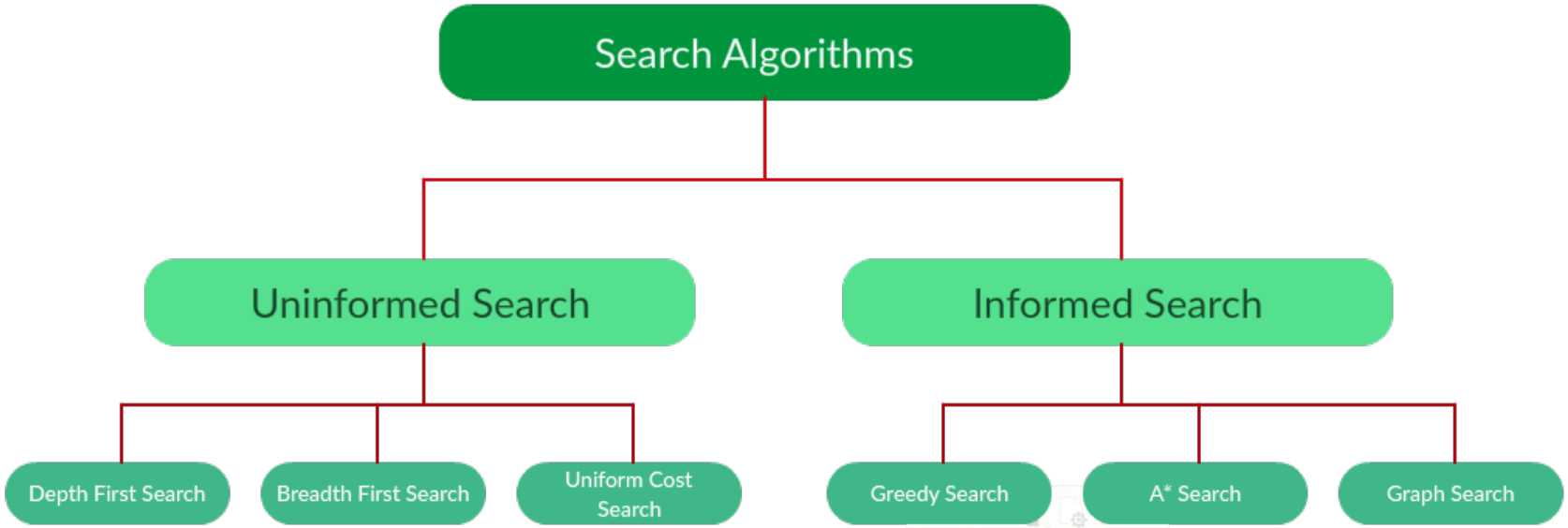
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Search Strategies

- Properties of search strategies
  Completeness
  Time Complexity
  Space Complexity
  Optimality
- **Uniformed search strategies:** (blind, exhaustive, bruteforce) do not guide the search with any additional information about the problem.
- **Informed search strategies:** (heuristic, intelligent) use information about the problem (estimated distance from a state to the goal) to guide the search.

Types of Search Algorithms



Uninformed Search Strategies

- Uninformed search strategies, also known as blind search strategies, are algorithms used in artificial intelligence to explore a search space without any specific information or heuristics guiding their decisions. These strategies are often used when little or no additional knowledge about the problem is available.
- As the name 'Uninformed Search' means the machine/agent blindly follows the algorithm regardless of whether right or wrong, efficient or in-efficient.

Uninformed Search Strategies

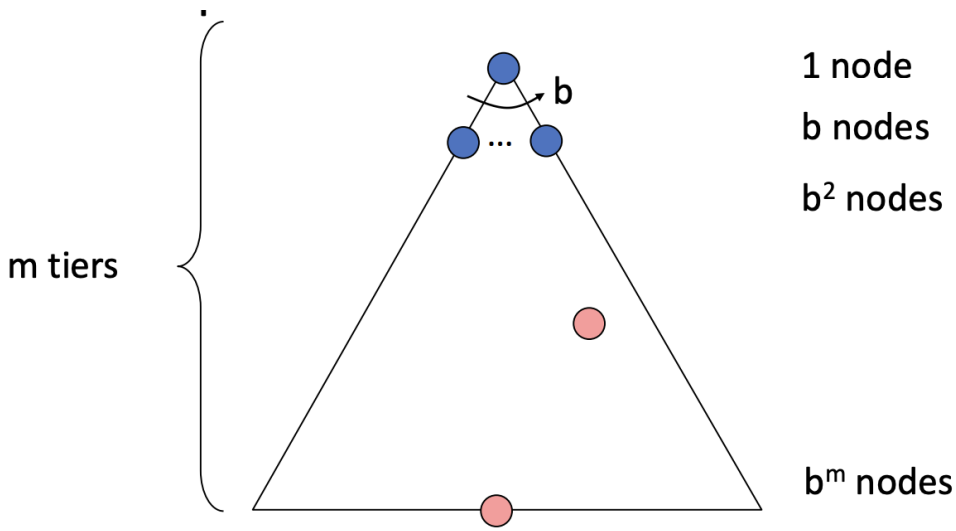
- These algorithms are brute force operations, and they don't have extra information about the search space; the only information they have is on how to traverse or visit the nodes in the tree.
- The search algorithm produces the search tree without using any domain knowledge, which is a brute force in nature. They don't have any background information on how to approach the goal or whatsoever.

Uninformed Search Strategies

- Uninformed search strategies are typically used when you have limited information about the problem domain, or when you want to ensure completeness in finding a solution.
- The choice of the appropriate strategy depends on the specific characteristics of the problem, such as the size of the search space and the availability of memory and computational resources.

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?
- Search tree:
  b is the branching factor
  m is the maximum depth
  solutions at various depths



Breadth-First Search

- Breadth-First Search (BFS) is an uninformed search algorithm used in artificial intelligence and computer science to explore or traverse a graph or tree data structure.
- It systematically explores all the nodes (vertices) in the search space, starting from the initial state and moving level by level, before moving deeper into the search tree.
- BFS is primarily used for finding the shortest path in unweighted graphs and can also be used to solve a wide range of search problems.

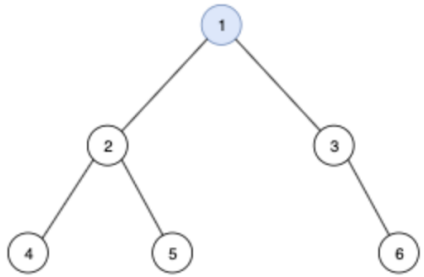
Breadth-First Search

- BFS explores the search space level by level, starting from the initial state.
- It expands all nodes at the current level before moving to the next level.
- BFS guarantees finding the shallowest goal state, making it complete.
- However, it can be memory-intensive, especially in large search spaces.

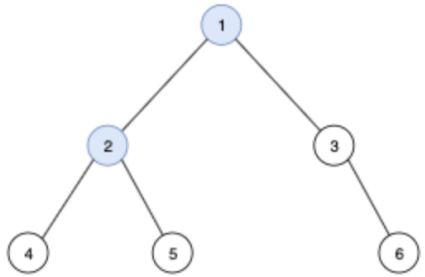
Breadth-First Search Steps

- Initialization:
  Start from the initial state and enqueue it in the queue.
- Exploration:
  - While the queue is not empty: Dequeue the front node from the queue. This node represents the current state being explored.
  - Check if the current node is the goal state. If yes, a solution has been found.
  - If not, expand the current node by generating its child nodes (neighbors) based on available actions.
  - Enqueue the child nodes that have not been visited and mark them as visited.
- Termination:
  If the queue becomes empty and the goal state has not been found, the algorithm terminates without a solution.

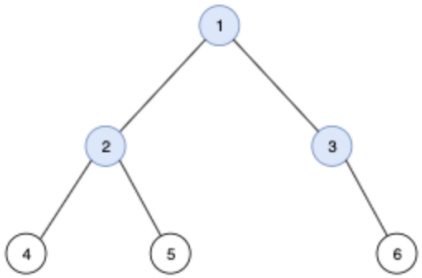
Breadth-First Search



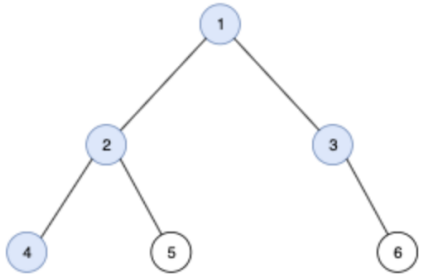
Step-1: Node 1 is marked visited



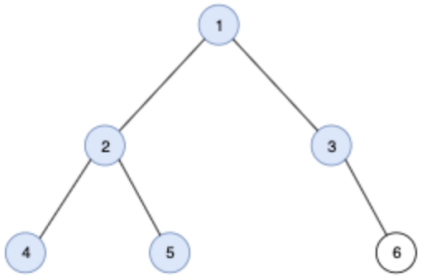
Step-2: Node 2 is marked visited



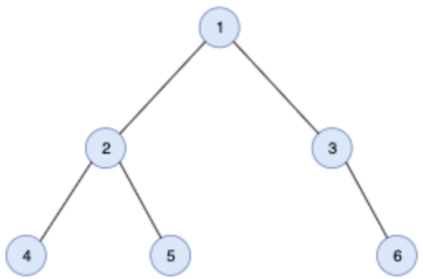
Step-3: Node 3 is marked visited



Step-4: Node 4 is marked visited



Step-5: Node 5 is marked visited



Step-6: Node 6 is marked visited

Breadth-First Search Characteristics

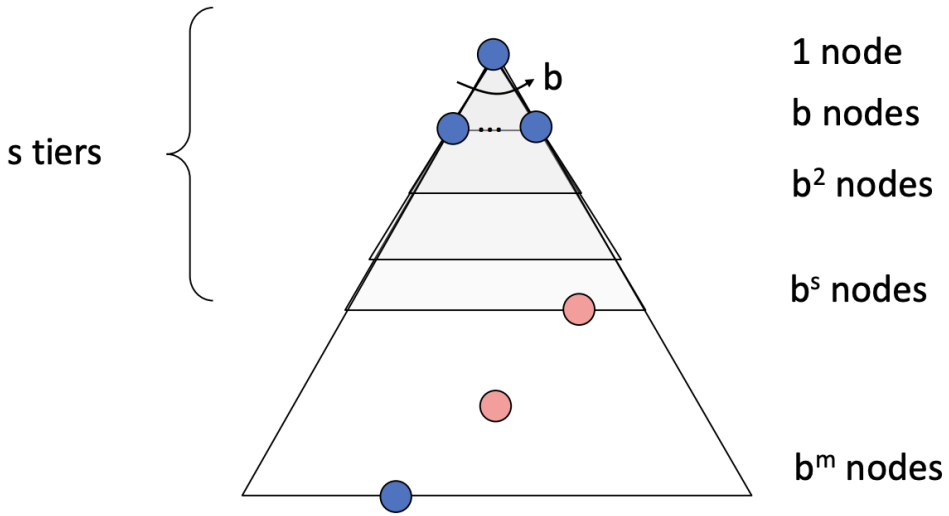
- FIFO Queue: BFS uses a first-in, first-out (FIFO) queue to maintain the order in which nodes are explored. Nodes are enqueued in the order they are discovered and dequeued from the front of the queue.
- Completeness: BFS is guaranteed to find the shallowest goal state if there is one. This means it will find the shortest path in an unweighted graph.
- Optimality: If all step costs are equal (unweighted graph), BFS will find an optimal solution, as it always explores nodes in increasing order of distance from the initial state.
- Memory Usage: BFS can be memory-intensive, especially in graphs with a large number of nodes or when exploring deep search spaces. Each level of the search tree must be stored in memory until it is fully explored.

Breadth-First Search Complexity

- Assume there are an average of b successors to each node, called the branching factor.
- Therefore, to find a solution path of length d must explore $1 + b + b^2 + b^3 + ... + b^d$ nodes
- Plus need b^d nodes in memory to store leaves in queue.

Breadth-First Complexity

- How much space does the fringe take? Has roughly the last tier, so $\mathcal{O}(b^s)$
- Is it complete? if a solution exists
- Is it optimal?



Breadth-First Search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Breadth-First Search Advantages

- Completeness: BFS is a complete search algorithm. It guarantees that it will find a solution if one exists. This makes it a good choice when you want to ensure that all possible solutions are explored.
- Optimality (in unweighted graphs): In unweighted graphs, BFS guarantees that it will find the shortest path from the initial state to the goal state. This is because it explores nodes level by level, ensuring that the shortest path is found before longer paths.
- No Heuristic Required: Unlike some other search algorithms, such as A* search, BFS does not require a heuristic function to guide the search.
- Simplicity: BFS is relatively simple to implement and understand. It uses a basic queue data structure and a straightforward traversal strategy.

Breadth-First Search Disadvantages

- Memory Usage: One of the most significant drawbacks of BFS is its memory usage. It needs to store all nodes at each level in memory until the entire level has been explored.
- Inefficiency in Large or Deep Graphs: BFS may become inefficient or impractical in large or deep search spaces because of its memory requirements. In such cases, it can exhaust available memory resources.
- Doesn't Work Well with High Branching Factors: When the branching factor (the number of child nodes for each node) is high, BFS may need to explore a vast number of nodes before finding a solution.
- Lack of Guidance: BFS does not incorporate any guidance or heuristics to prioritize certain branches of the search space. It explores all branches uniformly, which can be inefficient if there's prior knowledge that some paths are more likely to lead to a solution.

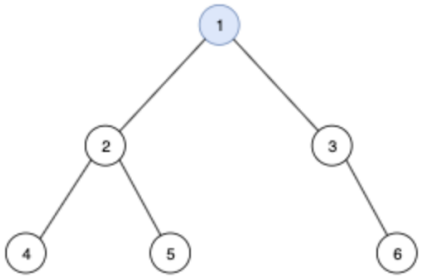
Breadth-First Search

- BFS is a versatile algorithm and has applications in various fields, including pathfinding, network analysis, and web crawling.
- Its main strengths lie in its completeness and optimality in finding the shortest path in unweighted graphs.
- However, it may not be the best choice for memory-intensive or highly branched search spaces.

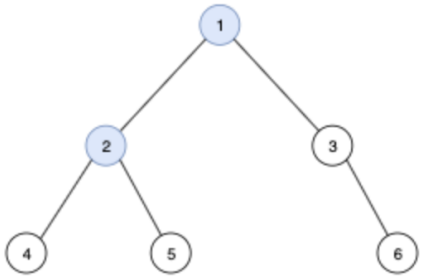
Depth-First Search

- Depth-First Search (DFS) is an uninformed search algorithm.
- Unlike Breadth-First Search (BFS), which explores all nodes at a given depth level before moving to the next level, DFS explores as deeply as possible along one branch before backtracking.
- This means that it goes as far down a path as it can before trying other paths.

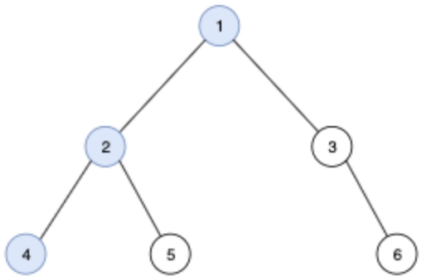
Depth-First Search



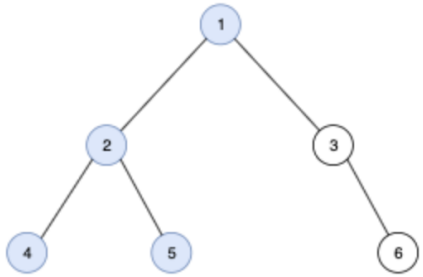
Step-1: 1 is marked visited



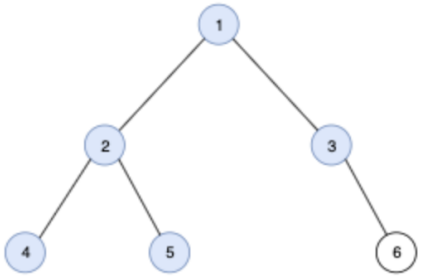
Step-2: 2 is marked visited



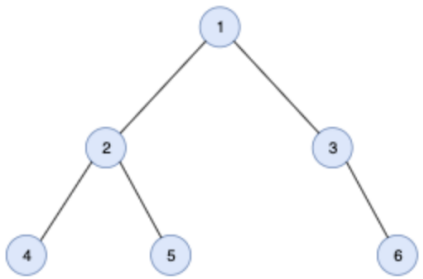
Step-3: 4 is marked visited



Step-4: 5 is marked visited



Step-5: 3 is marked visited



Step-6: 6 is marked visited

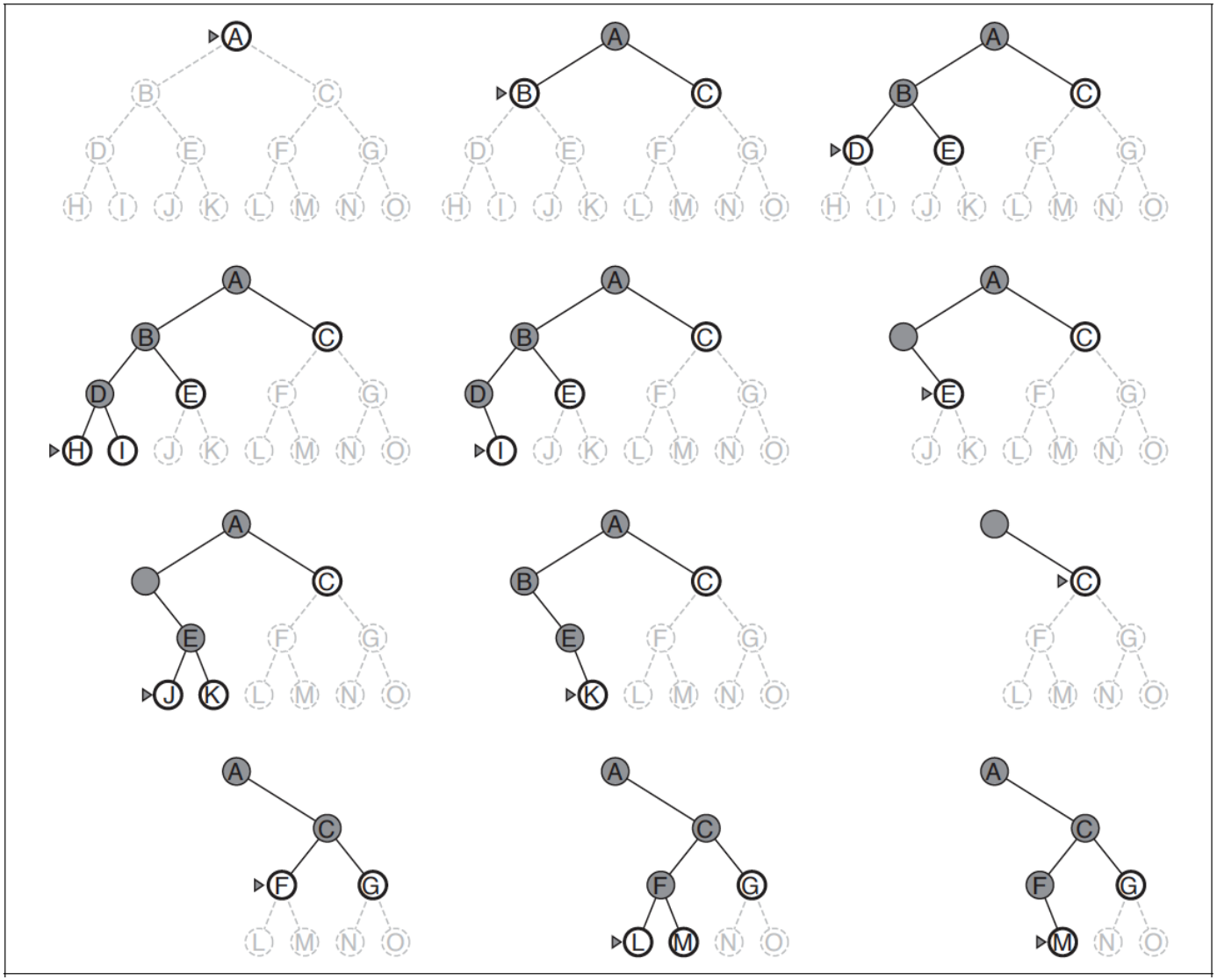
Depth-First Search Steps

- Initialization: Start from the initial state (node) and push it onto the stack.
- Exploration: While the stack is not empty:
  - Pop the top node from the stack. This node represents the current state being explored.
  - Check if the current node is the goal state. If yes, a solution has been found.
  - If not, expand the current node by generating its child nodes (neighbors) based on available actions.
  - Push the child nodes onto the stack that have not been visited and mark them as visited.
- Backtracking: If a node is encountered with no unvisited child nodes, the algorithm backtracks by popping nodes from the stack until it finds a node with unvisited children.
- Termination: If the stack becomes empty and the goal state has not been found, the algorithm terminates without a solution.

Depth-First Search Characteristics

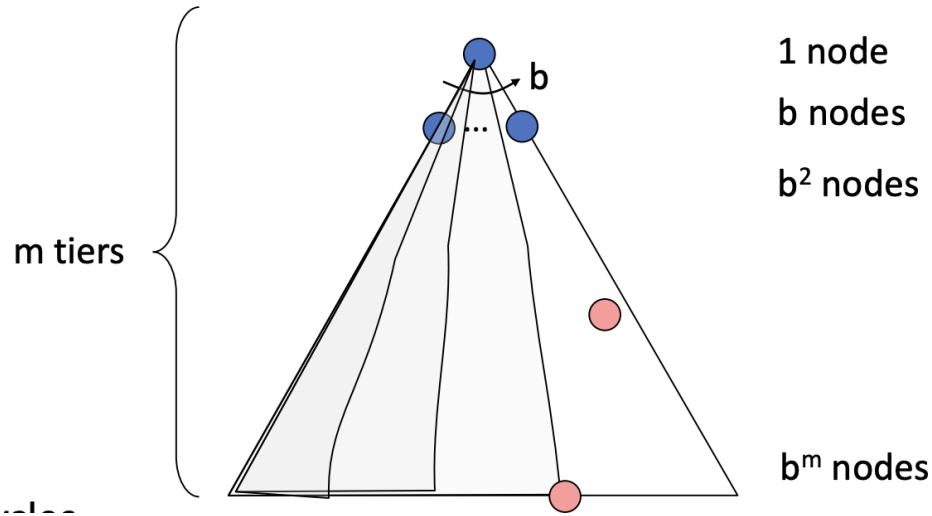
- Stack Data Structure: DFS uses a stack to keep track of nodes to be explored.
- Completeness: DFS is not guaranteed to find the shallowest goal or the shortest path, and it may not find a solution in some cases, especially if the search space is infinite or if the tree/graph is very deep.
- Memory Efficiency: DFS is often more memory-efficient than BFS because it only needs to store a single branch of the search tree at a time, rather than all branches at a given level.

Depth-First Search



Depth-First Search

- How much space does the fringe take? Only has siblings on path to root, so $\mathcal{O}(bm)$
- Is it complete? could be infinite, so only if we prevent cycles.
- Is it optimal? No, it finds the "leftmost" solution, regardless of depth or cost



Depth-First Search

- Not guaranteed to be complete since might get lost following infinite path.
- Not guaranteed optimal since can find deeper solution before shallower ones explored.
- Time complexity in worst case is still $\mathcal{O}(b^d)$ since need to explore entire tree. But if many solutions exist may find one quickly before exploring all of the space.
- Space complexity is only $\mathcal{O}(bm)$ where m is maximum depth of the tree since queue just contains a single path from the root to a leaf node along with remaining sibling nodes for each node along the path.

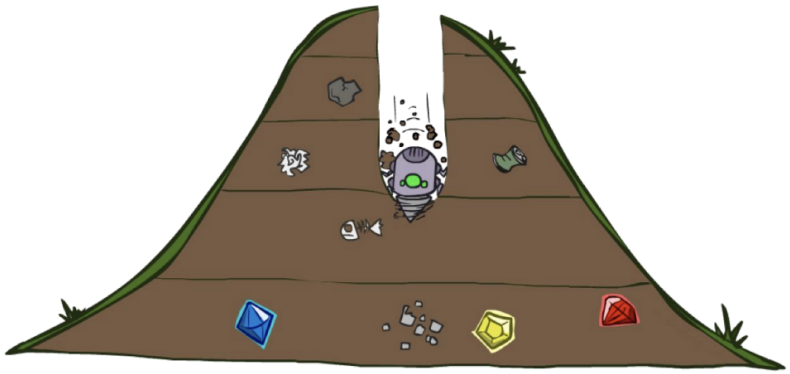
Depth-First Search Advantages

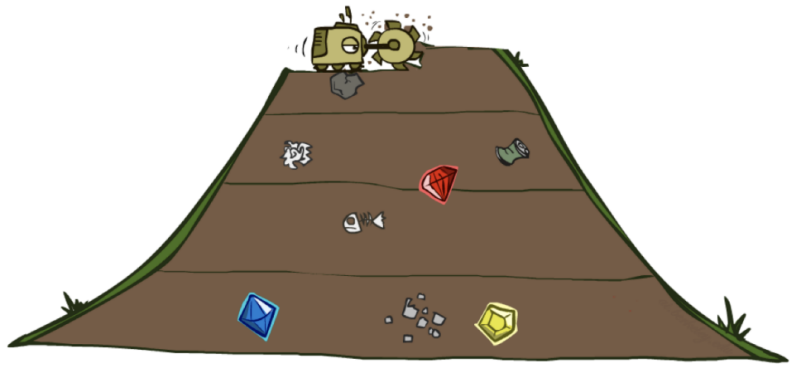
- Memory Efficiency: DFS is generally more memory-efficient than Breadth-First Search (BFS). It only requires memory proportional to the depth of the search tree, as it explores one branch of the tree at a time. This makes DFS suitable for deep or infinite search spaces.
- Simplicity: DFS is conceptually simple to implement and understand. It can be implemented using a straightforward recursive function or an explicit stack.
- Completeness: DFS is complete if the search space is finite. It will eventually find a solution or determine that none exists.

Depth-First Search Disadvantages

- Lack of Optimality: DFS does not guarantee finding the shortest path or the shallowest goal. It can get trapped in deep branches before exploring shallower ones, which can lead to suboptimal solutions.
- Completeness in Infinite Spaces: In infinite search spaces, DFS may not terminate because it can potentially explore indefinitely deep branches. Implementations need to incorporate safeguards to handle infinite spaces.
- Vulnerability to Cycles: Without proper handling, DFS can get stuck in cycles, especially in graphs with cycles. Implementations must include mechanisms to track visited nodes to avoid infinite loops.
- Not Well-Suited for Finding Multiple Solutions: DFS may terminate after finding one solution in problems with multiple valid solutions. To find all solutions, additional modifications are needed.

DFS vs BFS





Uniform Cost Search

- Uniform Cost Search (UCS) is a graph traversal and search algorithm used in artificial intelligence and computer science.
- UCS explores the search space by expanding nodes in increasing order of the cost to reach them from the starting node.
- It is "uniform" in the sense that it prioritizes exploring nodes with lower accumulated costs before nodes with higher costs.

Uniform Cost Search

- Priority Queue: UCS uses a priority queue to maintain the nodes to be explored. The priority of a node is determined by the cost of the path to reach that node.
- Optimality: UCS guarantees to find the optimal path in terms of the total cost.
- Completeness: If the edge costs are non-negative, UCS is complete, meaning it will find a solution if one exists. However, if the edge costs are negative, UCS may not work correctly.
- Memory Usage: UCS can be memory-intensive, especially in graphs with a large number of nodes or when the edge costs vary widely. The priority queue stores nodes based on their cost, which can lead to high memory consumption.

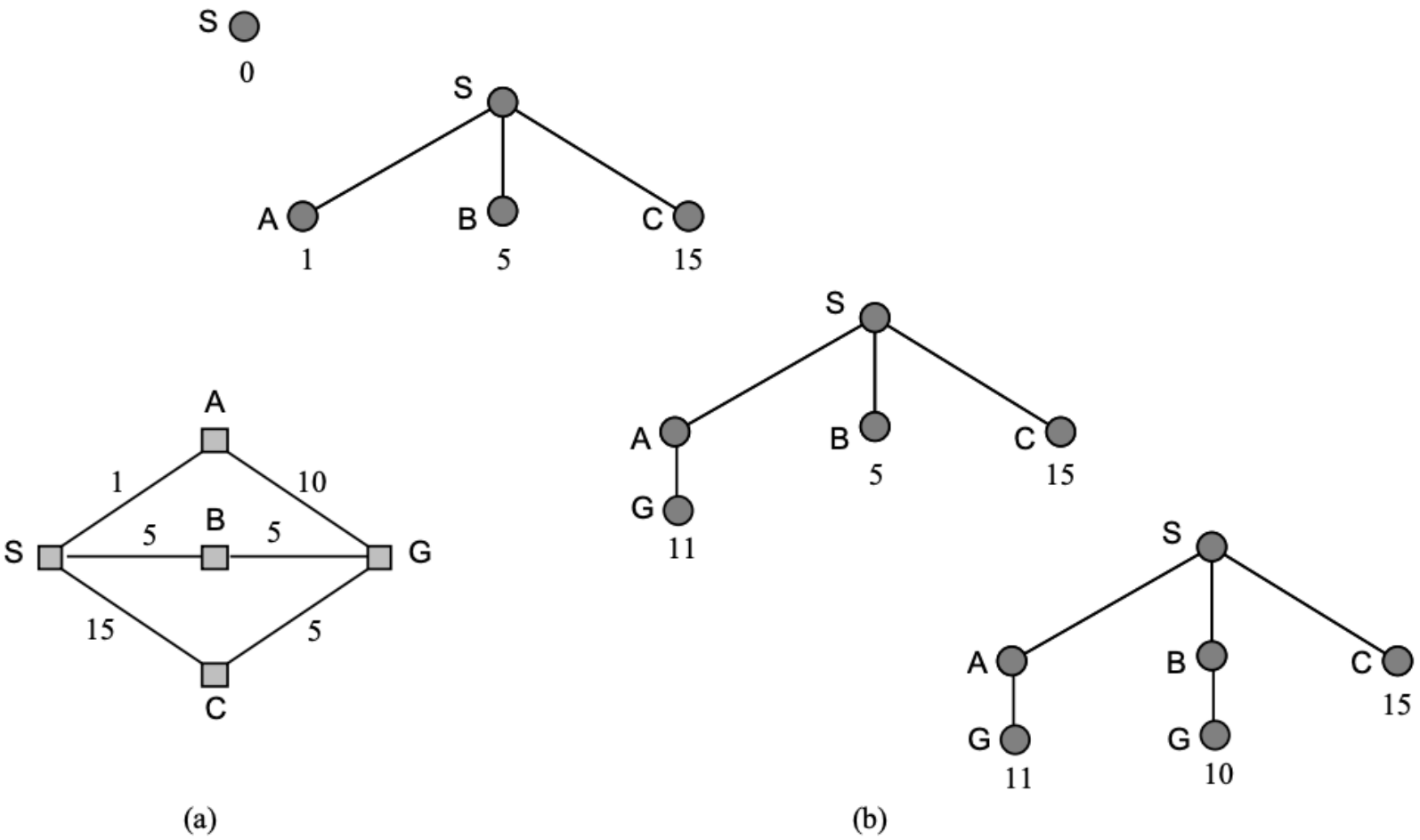
Uniform Cost Search

- Like breadth-first except always expand node of least cost instead of of least depth (i.e. sort new queue by path cost).
- Do not recognize goal until it is the least cost node on the queue and removed for goal testing.
- Therefore, guarantees optimality as long as path cost never decreases as a path increases (non-negative operator costs).
- Fringe is a priority queue(priority: cumulative cost).

Uniform Cost Search

- Initialization: Start from the initial node and initialize a priority queue with the initial node and its cost (which is initially 0).
- Exploration: While the priority queue is not empty:
  - Dequeue the node with the lowest cost from the priority queue.
  - If the dequeued node is the goal node, a solution has been found.
  - Otherwise, expand the dequeued node by generating its neighboring nodes and calculating the cost to reach them.
  - For each unvisited neighboring node, enqueue it into the priority queue with its calculated cost.
- Termination: If the priority queue becomes empty and the goal node has not been found, the algorithm terminates without a solution.

Uniform Cost Search



Uniform Cost Search

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

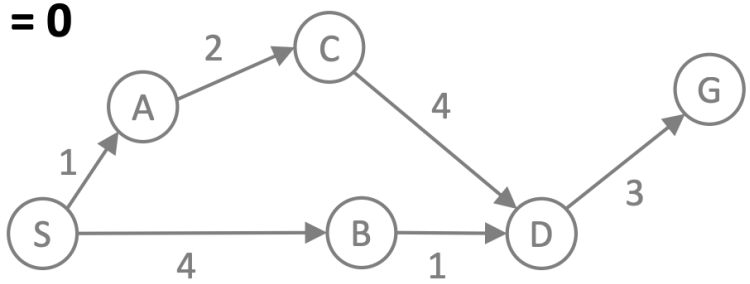
initialize the frontier as a **priority queue using node's path_cost as the priority**

add initial state of problem to frontier **with path_cost = 0**

loop do

if the frontier is empty then

return failure



choose a node (with minimal path_cost) and remove it from the frontier

if the node contains a goal state then

return the corresponding solution

for each resulting child from node

add child to the frontier with **path_cost** = **path_cost**(node) + cost(node, child)