Notion of State

States are used to represent different possible situations or steps in the journey toward finding a solution to a problem.

Key Concepts:

1. **State Representation:**
   - A state is a snapshot of all relevant information at a particular point in the problem-solving process. For example, in a chess game, a state would include the positions of all the pieces on the board.
2. **Initial State:**
   - This is the starting point of the search. It's the state from which the search begins. In a maze-solving problem, the initial state would be the starting position of the agent in the maze.
3. **Goal State:**
   - The goal state is the condition or configuration that the problem-solving process aims to reach. For instance, in a puzzle, the goal state is the final solved configuration.

State Space

Definition of State Space:

- The state space of a problem is the complete set of all possible states that can be generated from the initial state by applying a sequence of operations or actions. Each state in the state space represents a unique configuration of the system at a specific point in the problem-solving process.
- For example, in a chess game, each possible arrangement of pieces on the board corresponds to a different state. The state space would thus include every conceivable board configuration that could arise during the game.

2. State Space as a Graph:

- The state space can be visualized as a graph where:
  - **Nodes (Vertices):** Represent individual states.
  - **Edges:** Represent transitions between states, which occur as a result of applying an action or operator.
- This graph may be finite or infinite depending on the problem. For example, in a game with a limited number of moves, the state space is finite. However, in problems like robot navigation in an unbounded environment, the state space could potentially be infinite.

Path in the State Space:

- A path in the state space is a sequence of states connected by transitions (edges). The search process is essentially about finding a path from the initial state to one of the goal states.
- Example: In a maze-solving problem, a path is the sequence of steps taken to move from the entrance to the exit of the maze.

State Space Size:

- The size of the state space can vary significantly depending on the problem:
  - **Finite State Space:** Problems like the 8-puzzle or chess have a finite number of states. In the 8-puzzle, there are 9! = 362,880 possible states.
  - **Infinite State Space:** Problems like mathematical optimization or robot path planning in an open field might have an infinite state space.

State Space Complexity:

- The complexity of a search problem is often measured by the size of its state space and the branching factor (the average number of children per node). Problems with large state spaces or high branching factors are generally more difficult to solve.

Example: The 8 Puzzle

- States? Locations of tiles
- Actions? Move blank left, right, up, down
- Goal test? = goal state (given)
- Path cost? 1 per move

- [Note: optimal solution of n-Puzzle family is NP-hard]

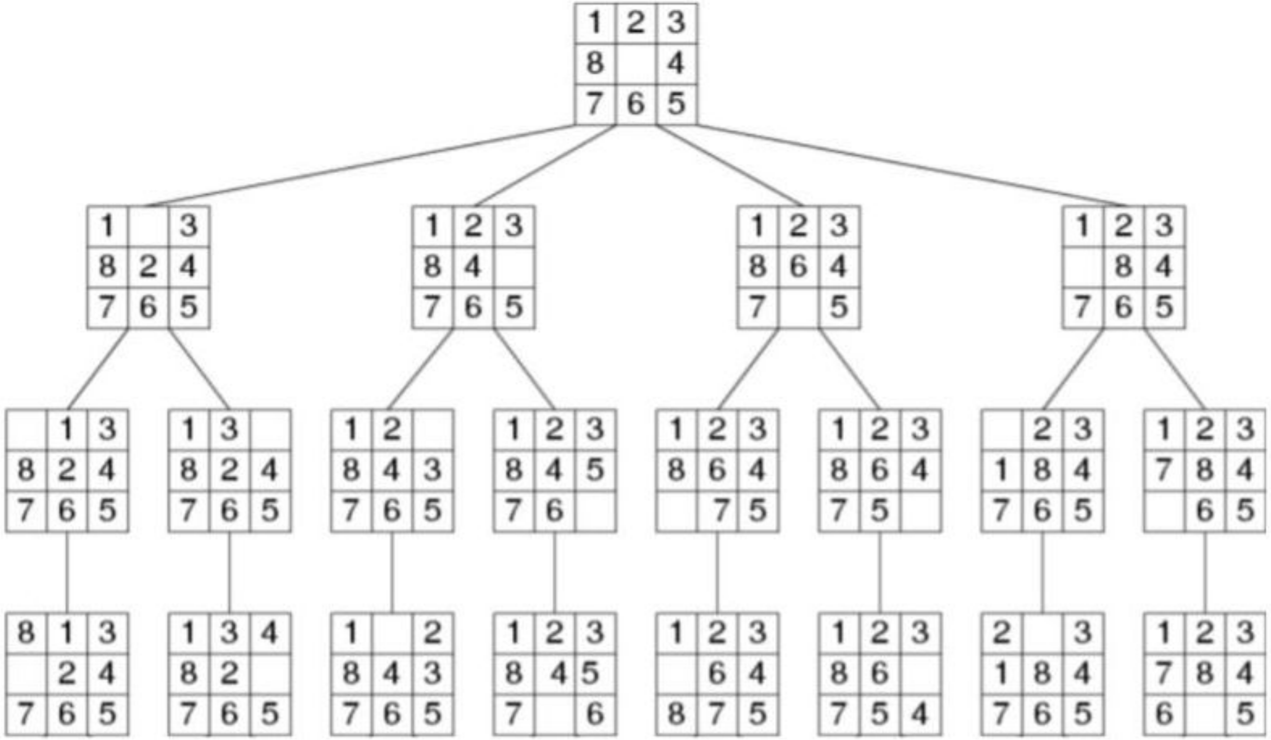
1	2	3
8		4
7	6	5

Start State

	1	2
3	4	5
6	7	8

Goal State

State Space of 8 Puzzle Problem



Search Problem

A search problem can be defined formally as follows:

- A set of possible states that the environment can be in. We call this the state space.
- The initial state that the agent starts in.
- A set of one or more goal states. Sometimes there is one goal state (e.g., Bucharest), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).
    - We can account for all three of these possibilities by specifying an *Is-GOAL* method for a problem.
- The actions available to the agent. Given a state S, Action(S) returns a finite set of actions that can be executed in S. We say that each of these actions is applicable in S.
    - For example, Action(Arad) = {ToSibiu, ToTimisoara, ToZerind}

Search Problem

- A transition model, which describes what each action does. Result(S,A) returns the state that results from doing action A in state S.
  - For example, Result(Arad, ToZerind) = Zerind
- An action cost function, denote by Action-Cost(S,A,S’) that gives the numeric cost of applying action A in state S to reach state S’.
- A sequence of actions forms a path, and a solution is a path from the initial state to a goal state.
  - We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- An optimal solution has the lowest path cost among all solutions.

Abstraction in Problem Solving

Abstraction is a fundamental concept in computer science and artificial intelligence. It involves simplifying a problem by reducing the amount of detail in the representation, focusing on the essential aspects that are necessary to solve the problem. This process allows us to manage complexity more effectively and can make problem-solving more efficient.

Key Aspects of Abstraction:

1. **Removing Detail:**
   - **Definition:** Abstraction involves stripping away unnecessary details from a problem to create a simplified model that retains the core components relevant to solving the problem.
   - **Example:** In a navigation problem, instead of considering every street and building, we might abstract the city as a grid where intersections are nodes and streets are edges.
2. **Validity of Abstraction:**
   - **Definition:** An abstraction is valid if any solution derived in the abstract model can be translated back into a valid solution in the original, more detailed problem space.
   - **Example:** If we solve the navigation problem using our simplified grid model, we must be able to convert the path found in this model back into actual streets and turns in the real city.

Abstraction in Problem Solving

- **Significance:** Validity ensures that the abstraction does not omit crucial details that would render the abstract solution unusable in the real world.

2. **Usefulness of Abstraction:**
   - **Definition:** An abstraction is useful if solving the problem in the abstract model is easier or more efficient than solving it in the original detailed model.
   - **Example:** Solving a navigation problem on a simplified grid is typically faster and computationally less expensive than accounting for every street, traffic light, and pedestrian in the real city.
   - **Significance:** Usefulness ensures that the abstraction provides practical benefits, such as reduced computational resources or time savings, making it a valuable tool in problem-solving.

Example Problems

A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.

- **Vacuum world**
- **Sokoban puzzle**
- **Sliding-tile puzzle**

Example Problems

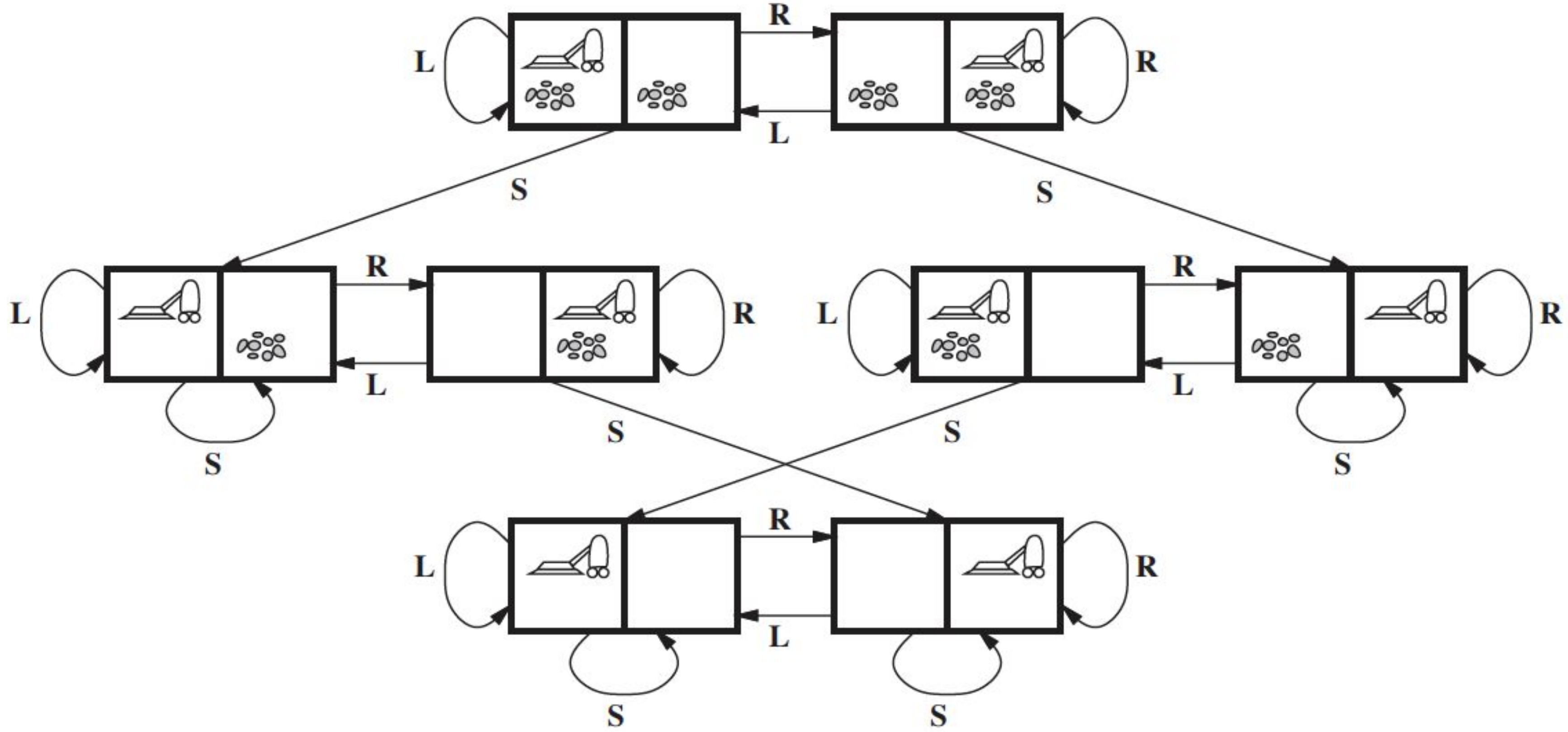
A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

- **Route-finding problem**
- **Touring problems**
- **Traveling salesperson problem (TSP)**
- **VLSI layout problem**
- **Robot navigation**
- **Automatic assembly sequencing**

Vacuum World

States:

- **Definition:** The state is determined by the agent's (vacuum cleaner's) location and the status of the dirt in each location.
- **Representation:** In a simple environment with 2 locations (left and right), the agent can be in either location, and each location can be clean or dirty.
- **Possible States Calculation:** Since the agent can be in one of 2 locations and each location can be either clean or dirty, we have $2 \times 2^2 = 8$ possible states.
  - Example States:
    1. Agent in the left location, both locations clean.
    2. Agent in the left location, left location dirty, right location clean.
    3. Agent in the left location, left location clean, right location dirty.
    4. Agent in the left location, both locations dirty.
    5. Agent in the right location, both locations clean.
    6. Agent in the right location, left location dirty, right location clean.
    7. Agent in the right location, left location clean, right location dirty.
    8. Agent in the right location, both locations dirty.


R
L
R
L
S
S
R
L
R
L
S
L
R
R
L
S
S
S
R
L
L
R
S
S

Vacuum World

Initial State:

- **Definition:** The initial state is the starting configuration of the environment.
- **Flexibility:** Any of the possible states can be designated as the initial state.
    - Example: The agent starts in the left location with both locations dirty.

Actions:

- **Available Actions:** In this simple environment, the agent has three possible actions:
    - **Left:** Move to the adjacent left location.
    - **Right:** Move to the adjacent right location.
    - **Suck:** Clean the current location.

Vacuum World

Transition Model:

- **Expected Effects:** Actions generally have the anticipated outcomes, but there are specific exceptions:
  - **Boundaries:**
    - Moving Left in the leftmost location has no effect.
    - Moving Right in the rightmost location has no effect.
  - **Cleaning:**
    - Sucking in a clean location has no effect.
- **State Transitions:** The transition model describes how each action changes the state of the environment.

Goal State

- The states in which every cell is clean.

Path Cost:

- **Definition:** The path cost is a measure of the total cost to reach the goal state from the initial state.
- **Metric:** In this problem, each action has a cost of 1.
  - **Calculation:** The path cost is the number of actions taken (steps) to reach the goal state.
  - **Example:** If it takes 4 actions to clean all locations, the path cost is 4.

Search Algorithms

- A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.
- We consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state.
- Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.
- The **state space** describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.
- The **search tree** describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).
- The **frontier** separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

Search data structures

Three kinds of queues are used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, ***f***. It is used in best-first search.

- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.

- A **LIFO queue** or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.

Redundant Paths

Redundant Paths:

- **Definition:** Redundant paths refer to different sequences of actions that lead to the same state multiple times within a search tree. Essentially, the same state is reached through different paths, but the state itself has already been explored earlier.
- **Impact:** Redundant paths do not provide new information and can unnecessarily increase the size of the search tree, making the search process less efficient.

Repeated States:

- **Definition:** A repeated state occurs when the same state is encountered multiple times during the search process, but through different paths. This can lead to the exploration of the same state more than once, resulting in inefficiency.
- **Impact:** Repeated states can slow down the search process, especially in large state spaces, as the algorithm may end up exploring the same state multiple times.

Cycles:

- **Definition:** A cycle occurs when a path leads back to a state that has already been visited earlier in the current path, forming a loop. Cycles can cause the search to go in circles, potentially leading to infinite loops if not handled properly. A cycle is a special case of a redundant path.
- **Impact:** Cycles can significantly hamper the efficiency of a search algorithm, and in the worst case, can cause the algorithm to never reach a solution if it gets stuck in an infinite loop.

We call a search algorithm a graph search if it checks for redundant paths and a tree search if it does not check.

Measuring problem-solving performance

1. Completeness

- **Definition**: Completeness refers to whether the algorithm is guaranteed to find a solution if one exists and to correctly report failure if no solution is available.
- **Importance**: A complete algorithm ensures that all possible paths are explored (or sufficient paths, depending on the method), making it reliable for finding solutions in search spaces.

2. Optimality

- **Definition**: Optimality measures whether the algorithm can find the solution with the lowest path cost among all possible solutions.
- **Importance**: This is crucial in applications where cost minimization is essential, such as route finding in transportation networks. An optimal algorithm guarantees that the best solution is found, which can have significant implications for resource utilization.

3. Time Complexity

- **Definition**: Time complexity assesses how the execution time of the algorithm grows with the size of the input or the search space.
- **Importance**: Understanding time complexity helps predict performance and efficiency. Algorithms with lower time complexity are preferred, especially for large search spaces, as they can provide solutions more quickly.

4. Space Complexity

- **Definition**: Space complexity measures the amount of memory required by the algorithm to perform the search, including both the space needed to store the search space and any auxiliary data structures.
- **Importance**: Space complexity is crucial for determining how well an algorithm can operate within memory constraints. Algorithms with high space complexity might struggle with larger inputs, making them less practical in memory-limited environments.

Uninformed Search

Uninformed Search Strategies

Uninformed search strategies are search algorithms that operate without additional information about the goal or the nature of the problem beyond the basic definition of the state space.

Key Characteristics of Uninformed Search Strategies:

- **No Heuristic Information**: Uninformed search strategies do not employ any domain-specific knowledge or heuristics to prioritize or evaluate paths in the search space.
- **Exhaustive Exploration**: They often explore the entire search space, which can lead to inefficiencies, especially in large or complex problems.
- **Optimality and Completeness**: Some uninformed search strategies guarantee finding an optimal solution or ensuring completeness (finding a solution if one exists), while others may not.

Types of Uninformed Search Strategies

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Iterative Deepening Search
4. Uniform Cost Search

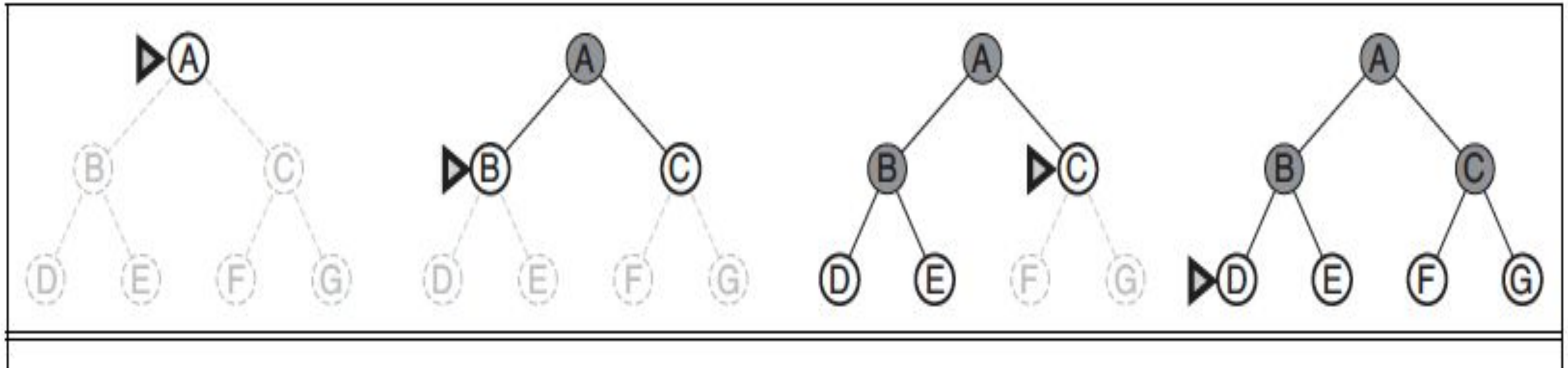
Breadth-First Search (BFS)

Breadth-First Search (BFS) is an uninformed search algorithm used for traversing or searching tree or graph data structures. It explores all the neighbor nodes at the present depth level before moving on to nodes at the next depth level. This strategy is particularly useful for finding the shortest path in unweighted graphs.

Key Characteristics of BFS:

1. **Level Order Exploration**: BFS explores nodes level by level. It starts from the root (or starting node) and explores all adjacent nodes before moving deeper into the tree or graph.
2. **Queue Data Structure**: BFS uses a queue to keep track of nodes that need to be explored. The first node added to the queue will be the first one to be processed (FIFO - First In, First Out).

Example



BFS Algorithm Steps

1. **Initialize**:
    - Create a queue and enqueue the starting node.
    - Maintain a set to keep track of visited nodes to prevent cycles.
2. **Process Nodes**:
    - While the queue is not empty:
        - Dequeue the front node from the queue.
        - If the dequeued node is the goal node, return it as the solution.
        - Otherwise, enqueue all unvisited neighboring nodes of the current node, marking them as visited.
3. **Terminate**:
    - If the queue becomes empty without finding the goal node, report that no solution exists.

Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

BFS Performance Measure

Completeness: BFS is complete, meaning that if there is a solution, it will find it (assuming the search space is finite).

Optimality: BFS is optimal for unweighted graphs, as it guarantees finding the shortest path (in terms of the number of edges) to the goal node.

Time Complexity: The time complexity of BFS is $O(b^d)$, where b is the branching factor (the average number of children per node) and d is the depth of the shallowest solution. This means that BFS can become inefficient for large search spaces.

Space Complexity: The space complexity is also $O(b^d)$ because all nodes at the current depth level need to be stored in memory before moving to the next level.

Formal Definition of Branching Factor (*b*) and Depth (*d*)

Branching Factor (b):

- The branching factor ***b*** of a graph/tree is defined as the average number of children (or adjacent nodes) each node has. In other words, for a node ***v***, the branching factor ***b*** represents the number of immediate neighbors (children) that can be reached from ***v***.
- Mathematically, if ***v*** is a node and ***N(v)*** represents the set of neighbors (children) of ***v***, then the branching factor ***b*** is the average of **|*N(v)*|** across all nodes:

$$b = \frac{1}{|V|} \sum_{v \in V} |N(v)|$$

where **|*V* |** is the total number of nodes in the graph.

Depth (d):

- The depth ***d*** of a solution in a search tree or graph is defined as the number of edges in the shortest path from the root node (or starting node) to the goal node. It represents the "level" at which the goal node is located in the search tree.
- For a search problem, ***d*** is the depth of the shallowest goal node. It is the minimum number of steps required to reach the goal from the starting point.

Derivation of Time Complexity O(b^d) for BFS

Number of Nodes Explored:

- At depth 0 (root level), BFS explores 1 node (the root node).
- At depth 1, BFS explores up to b nodes (the children of the root).
- At depth 2, BFS explores up to b^2 nodes (the children of the nodes at depth 1).
- Continuing this pattern, at depth d, BFS explores up to b^d nodes.

The total number of nodes explored by BFS up to depth **d** can be approximated by summing the nodes at each level:

$$\text{Total nodes explored} = 1 + b + b^2 + \cdots + b^d$$

- The sum of this geometric series can be approximated by its largest term when b>1 and d is sufficiently large:

$$1 + b + b^2 + \cdots + b^d \approx b^d$$

Therefore, the time complexity of BFS, which is proportional to the number of nodes explored, is: **O(b^d)**

This represents the worst-case time complexity, where BFS has to explore all nodes up to the depth d before finding the goal node.

Proof of Completeness

To prove the completeness of BFS, we argue that BFS explores all possible nodes at each depth level before moving on to the next level. Here's how it works:

- **Step 1:** BFS starts at the root node (or starting node) and explores all its immediate neighbors (nodes at depth 1).
- **Step 2:** BFS then moves on to explore all neighbors of the nodes at depth 1 (nodes at depth 2).
- **Step 3:** This process continues level by level until BFS either finds the goal node or exhausts all possible nodes in the graph.
- **Assumption:** Suppose there is a solution, i.e., a path from the starting node ***s*** to the goal node ***g***.
- **Observation:** Since BFS explores nodes level by level, it will eventually reach the depth where the goal node ***g*** resides. BFS will examine all nodes at this depth, including ***g***, assuming the graph is finite and the goal node exists.

Conclusion: BFS is guaranteed to find the goal node if one exists. Therefore, BFS is complete for finite graphs.

Proof of Optimality

Proof by Contradiction:

- **Assumption:** Assume BFS is not optimal. This implies that there exists a path ***P*** from the start node ***s*** to the goal node ***g*** found by BFS that is not the shortest path. Let the actual shortest path be P_{short}, with a length (number of edges) of ***d***.
- **Observation:** BFS explores all nodes at depth 1 before moving to depth 2, all nodes at depth 2 before moving to depth 3, and so on. Therefore, BFS will explore all paths of length ***d*** before it explores any paths of length greater than ***d***.
- **Contradiction Setup:** Since P_{short} is the shortest path, its length is ***d***. According to the BFS algorithm, all paths of length ***d*** will be explored before any longer paths, including ***P***. Hence, BFS should have found P_{short} before considering ***P***, which contradicts the assumption that BFS found a longer path first.
- **Contradiction Conclusion:** The assumption that BFS is not optimal leads to a contradiction. Therefore, BFS must be optimal, finding the shortest path in an unweighted graph.

BFS Applications

Finding the Shortest Path in Unweighted Graphs: BFS is commonly used to find the shortest path in unweighted graphs, such as in social networks or maps, where all edges are considered equal.

Level-Order Traversal in Trees: BFS performs level-order traversal, which is useful for printing tree nodes level by level, and is essential in various tree algorithms.

Finding Connected Components in Graphs: BFS can identify connected components in undirected graphs, helping to analyze the structure and connectivity of networks.

Web Crawlers: BFS is utilized in web crawlers to explore web pages systematically, ensuring that all linked pages are visited in a structured manner.

Pathfinding in AI: BFS is employed in AI for pathfinding in game environments or mazes, helping characters find the shortest route from a starting point to a target.

Limitations of BFS

- **Exponential Growth:** The most significant limitation of BFS is its exponential growth in time and space complexity due to the $O(b^d)$ complexity. As the depth d and branching factor b increase, the number of nodes BFS must explore becomes infeasible.

- **Memory Consumption:** BFS requires storing all nodes at the current depth level in memory, leading to high memory consumption. This can be a bottleneck in large graphs.

- **Irrelevance in Weighted Graphs:** BFS is optimal for unweighted graphs, but in weighted graphs, algorithms like Dijkstra's algorithm or A* search are preferred as they account for varying edge weights.

Scenarios Where BFS Might Be Inefficient

1. Graphs with Large Branching Factors

- **Inefficiency:** In graphs where each node has a large number of children (high branching factor), BFS can quickly generate an overwhelming number of nodes to explore at each level.
- **Example:** Consider a social network graph where each person (node) has hundreds of connections (edges). At each level, BFS needs to explore a vast number of connections, leading to high memory consumption and slower performance.
- **Example Analysis:** If b=10 and d=5, BFS would need to explore and store up to 10^5=100,000 nodes, which can be computationally expensive and memory-intensive.
- **Impact:** The exponential growth in the number of nodes explored at each level can lead to excessive memory usage and longer computation times.

2. Deep Graphs

- **Inefficiency:** In very deep graphs, where the shortest path to the goal node lies at a significant depth, BFS might take a long time to reach the goal since it explores all nodes at each depth level before moving deeper.
- **Example:** In a game tree where the solution is several moves deep, BFS will systematically explore all possible moves at each level, which can be inefficient compared to depth-first strategies that dive deeper more quickly.
- **Impact:** For deep solutions, BFS may explore a large number of irrelevant nodes at shallower levels, leading to unnecessary computation.

3. Memory-Intensive Searches

- **Inefficiency:** BFS stores all nodes at the current level in memory before proceeding to the next level. In large graphs, this can lead to high memory usage, which can be a significant limitation on systems with limited memory resources.
- **Example:** In a large maze represented as a graph, BFS needs to store all possible paths at each level, which can quickly consume large amounts of memory.
- **Impact:** The requirement to store all nodes at each level can make BFS infeasible for very large graphs or on systems with constrained memory.

4. Uniformly Unweighted Graphs

- **Inefficiency:** While BFS is optimal for finding the shortest path in unweighted graphs, it can be inefficient in scenarios where the graph is large and uniformly unweighted, meaning there are many equally valid paths. BFS will explore all paths level by level without any prioritization, leading to excessive exploration.
- **Example:** In a large, uniform network where all connections are equally viable, BFS will explore all paths indiscriminately, leading to redundant exploration.
- **Impact:** In uniformly unweighted graphs, BFS may spend unnecessary time exploring multiple paths that all lead to the same result.

5. BFS in Infinite Graphs

- **Inefficiency:** In infinite or very large graphs, BFS is impractical because it must explore all nodes at each depth level, which is impossible in an infinite structure.
- **Example:** Consider an infinite grid where the goal node is far away. BFS will attempt to explore all nodes at each distance from the starting point, which is not feasible.
- **Impact:** BFS is not suitable for infinite graphs, as it will never terminate unless the goal is extremely close to the starting point.

6. Graphs with Cycles

- **Inefficiency:** In graphs with cycles, BFS must keep track of visited nodes to avoid re-exploration, which adds overhead. If the graph contains many cycles, this tracking can become complex and resource-intensive.
- **Example:** In a network with redundant connections, BFS must ensure it doesn't revisit the same nodes through different paths, which can slow down the algorithm.
- **Impact:** The need to manage cycles increases both memory and computational overhead, making BFS less efficient.

7. Finding Multiple Paths

- **Inefficiency:** BFS is designed to find the shortest path but is less efficient when the goal is to find multiple paths or all paths to a goal. BFS will redundantly explore all possibilities, leading to unnecessary computation.
- **Example:** In a network routing scenario where multiple paths are needed, BFS will explore all nodes at each level, even after finding a valid path.
- **Impact:** For scenarios requiring multiple solutions, BFS's level-by-level exploration can lead to excessive computation compared to more targeted search algorithms.

8. Sparse Graphs

- **Inefficiency:** In sparse graphs, where the number of edges is much lower than the maximum possible number of edges, BFS can become inefficient due to the large distances between connected nodes. BFS explores all nodes level by level, and in a sparse graph, this can mean traversing many levels where only a few nodes are connected, leading to wasted exploration.
- **Example:** Consider a sparse social network where most people have very few connections. BFS will explore each level, but since few connections exist, it will spend a lot of time processing nodes that do not lead to the goal.
- **Impact:** The level-by-level exploration in sparse graphs can lead to BFS traversing many irrelevant or empty levels, increasing the time and computational resources required to find the goal.

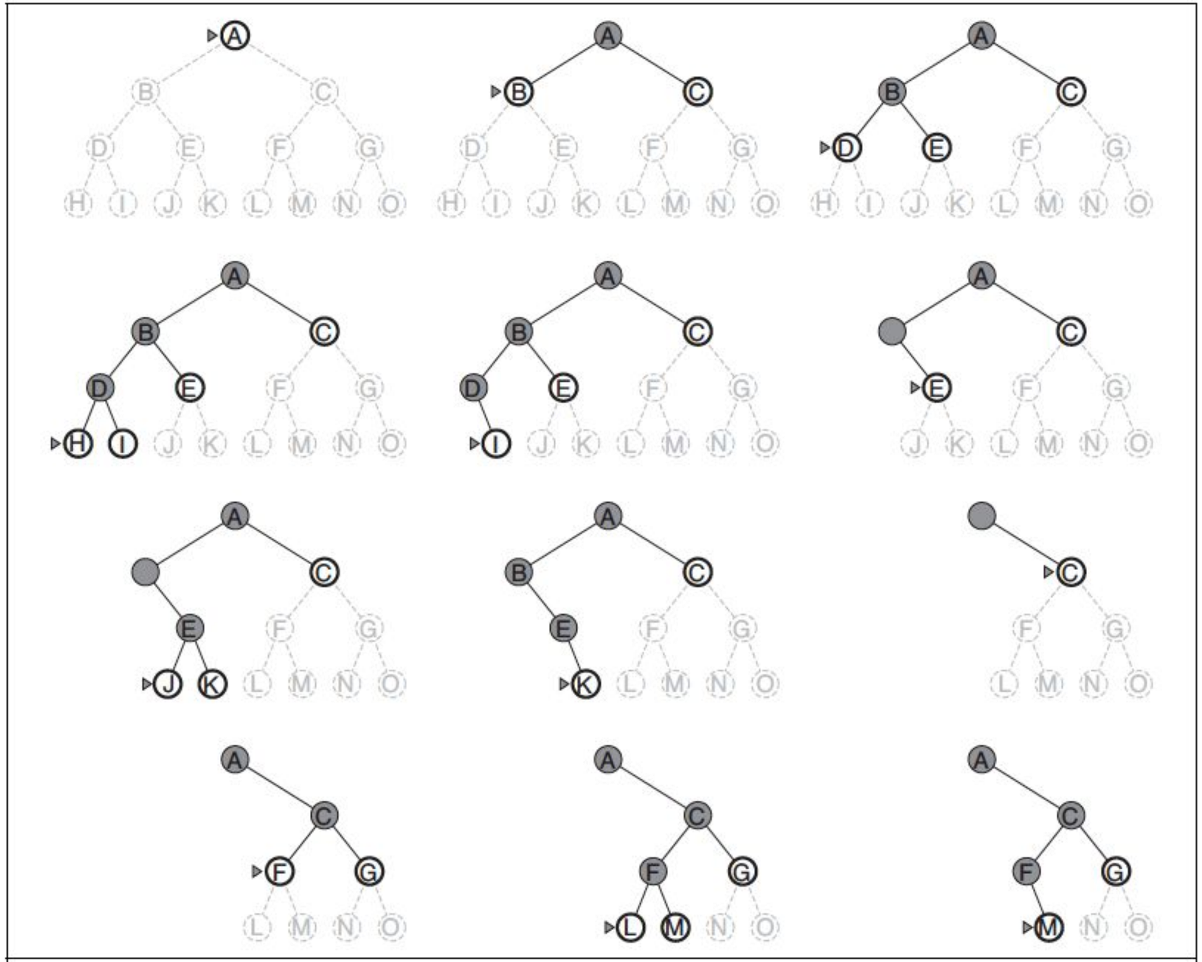
Depth-First Search (DFS)

Depth-First Search (DFS) is an uninformed search algorithm used for traversing or searching tree or graph data structures. It explores as far down a branch as possible before backtracking, making it distinct from other search strategies like Breadth-First Search (BFS).

Key Characteristics of DFS:

1. **Stack Data Structure**: DFS uses a stack to keep track of the nodes to be explored. This can be implemented using a recursive approach (which utilizes the call stack) or explicitly with an iterative approach using a manual stack.
2. **Backtracking**: When DFS reaches a node with no unvisited neighbors, it backtracks to the most recent node with unexplored neighbors. This characteristic makes it suitable for exploring all possible paths in a search space.

Example



DFS Algorithm Steps

1. **Initialize**:
    - Create a stack to keep track of nodes to explore.
    - Push the starting node onto the stack.
    - Maintain a set or list to keep track of visited nodes to avoid cycles.
2. **Process Nodes**:
    - While the stack is not empty:
        - Pop a node from the top of the stack.
        - If the popped node is the goal node, return it as the solution.
        - Otherwise, mark the node as visited.
        - Push all unvisited neighbors of the current node onto the stack.
3. **Terminate**:
    - If the stack becomes empty without finding the goal node, report that no solution exists.

NOTE: Algorithm based on the steps to be done by YOU!

DFS Performance Measure

Completeness:

- **Finite Graphs:** DFS is complete in finite graphs, meaning that if there is a solution, DFS will find it eventually. This is because DFS explores as far as possible along each branch before backtracking, ensuring that all nodes will eventually be explored.
- **Infinite Graphs:** DFS is not complete in infinite graphs, as it may get stuck in an infinitely deep branch without ever finding the solution. For example, if a graph contains an infinite path and the solution lies elsewhere, DFS might never reach the solution.

Optimality:

- DFS is **not optimal** in general. Since DFS explores each branch to its fullest depth before backtracking, it does not necessarily find the shortest path to the goal. The first solution found by DFS might not be the optimal one. For instance, if there is a shorter path that lies in a different branch, DFS may miss it and find a longer path first.

Time Complexity:

- The time complexity of DFS depends on the number of vertices (V) and edges (E) in the graph.
- In an **adjacency list** representation, the time complexity of DFS is: O(V+E)
- This is because DFS visits every vertex and every edge in the graph exactly once during the search process.

Space Complexity:

- The space complexity of DFS is mainly determined by the depth of the recursion stack (in the recursive implementation) or the size of the stack data structure (in the iterative implementation).
- In the **worst case**, the space complexity is O(d), where d is the maximum depth of the search tree. In the worst-case scenario, DFS might explore a path that is as deep as the maximum depth of the graph.
- If the graph is very deep or has many branches, the space complexity can be significant, potentially leading to stack overflow in the recursive implementation.

Step-by-Step Analysis of Time Complexity

Vertex Exploration

- Each vertex ***v*** in the graph is visited exactly once by DFS. When ***v*** is visited, it is marked as visited, and all its neighbors are explored.
- **Total Time for Vertex Exploration:** Since each vertex is visited once, the total time for vertex exploration is ***O(V)***.

2. Edge Exploration

- For each vertex ***v***, DFS iterates through all its neighbors (i.e., all vertices connected to ***v*** by an edge).
- The key point is that each edge ***(u,v)*** is explored exactly once during the entire DFS process:
  - When DFS visits vertex ***u***, it explores edge ***(u,v)*** to check if ***v*** is visited.
  - Similarly, when DFS visits vertex ***v***, it explores edge ***(v,u)*** to check if ***u*** is visited.
- **Total Time for Edge Exploration:** Since each edge is checked once, the total time for edge exploration is ***O(E)***.

3. Overall Time Complexity

- The time complexity of DFS is the sum of the time required for vertex exploration and edge exploration.
- **Total Time Complexity:**

O(Vertex Exploration) + O(Edge Exploration) = O(V)+O(E)

Time Complexity of DFS = O(V+E)

- This time complexity applies to both connected and disconnected graphs. For disconnected graphs, DFS may be initiated multiple times, once for each disconnected component, but the overall complexity remains ***O(V+E)***.

Proof of Optimality

Proof by Contradiction

To mathematically prove that DFS is not optimal, we'll use a proof by contradiction.

Assumptions:

1. Let ***G=(V,E)*** be an unweighted graph where ***V*** is the set of vertices and ***E*** is the set of edges.
2. Assume that DFS is optimal, meaning that it always finds the shortest path from the start node ***s*** to the goal node ***g***.
3. Let the depth of a node be defined as the number of edges from the start node ***s*** to that node.

Setup:

Consider the following situation:

- Let there be two paths from the start node ***s*** to the goal node ***g***.
  1. **Path 1** has a length of d_1 edges.
  2. **Path 2** has a length of d_2 edges.
- Assume without loss of generality that $d_1 < d_2$, meaning Path 1 is shorter than Path 2.

DFS Execution:

- DFS explores paths by diving deep into the graph, meaning it will explore nodes by moving along the edges until it can no longer proceed or it finds the goal node $\boldsymbol{g}$.
- Let's assume that DFS explores Path 2 first and finds the goal node $\boldsymbol{g}$ at depth $\boldsymbol{d_2}$.

Contradiction Argument:

1. **By our assumption**, DFS is optimal and should find the shortest path, which is Path 1 with length $\boldsymbol{d_1}$.
2. **However**, since DFS explores Path 2 first and reaches the goal node $\boldsymbol{g}$ at depth $\boldsymbol{d_2}$, DFS will return Path 2 as the solution.
3. This contradicts the assumption that DFS is optimal because it found a path of length $\boldsymbol{d_2}$ instead of the shorter path of length $\boldsymbol{d_1}$.

Conclusion:

- Since our assumption that DFS is optimal led to a contradiction, DFS cannot be optimal in general.
- **Thus, DFS is not guaranteed to find the shortest path** from $\boldsymbol{s}$ to $\boldsymbol{g}$ in an unweighted graph.

Example

```
   A
  / \
 B   C
  \ / \
   D   E
    \ /
     F (Goal)
```

Edges:

- A -> B
- A -> C
- B -> D
- C -> D
- C -> E
- D -> F
- E -> F

Paths from A to F:

1. **Path 1:** A -> B -> D -> F (Length = 3)
2. **Path 2:** A -> C -> D -> F (Length = 3)
3. **Path 3:** A -> C -> E -> F (Length = 3)

The path DFS finds is A→B→D→F

The path A→B→D→F is of length 3

```
     A
    / \
   B   C
  /     \
 G       E
 |       |
 D       F (Goal)
 |
 F
```

Edges:

- A -> B
- A -> C
- B -> G
- G -> D
- D -> F
- C -> E
- E -> F

Paths from A to F:

1. **Path 1:** A -> B -> G -> D -> F (Length = 4)
2. **Path 2:** A -> C -> E -> F (Length = 3)

If DFS starts at node A and chooses to explore the left branch first (i.e., goes from A→B), it will follow the longer path A→B→G→D→F, which has a length of 4.

The shorter path A→C→E→F with length 3 might be ignored if the left branch is fully explored first.

Proof of Completeness

Proof by Contradiction

To mathematically prove the completeness of DFS, we will use a proof by contradiction.

Assumptions:

1. Let ***G=(V,E)*** be a finite graph, where ***V*** is the set of vertices (nodes) and ***E*** is the set of edges.
2. Assume that there exists a path ***P*** from the start node ***s*** to the goal node ***g***.
3. Assume that DFS is not complete, meaning that it fails to find the solution (i.e., it does not find ***g***).

DFS Algorithm Characteristics:

- DFS explores nodes by traversing as deep as possible along each branch before backtracking.
- DFS uses a stack (either explicitly or via recursion) to keep track of the nodes to be explored.
- DFS visits each node at most once.

Contradiction Setup:

1. **Finite Graph:** Since ***G*** is finite, there are a finite number of vertices **|*V*|** and edges **|*E*|** in ***G***.
2. **Path Existence:** Let ***P*** **=** $(s, v_1, v_2, \ldots, g)$ be a valid path from ***s*** to ***g*** in ***G***. The path ***P*** has a finite length ***k***, where ***k*** is the number of edges in ***P***.

DFS Behavior:

- DFS will start at node ***s*** and explore one of the paths.
- If DFS does not immediately find ***g***, it will backtrack and explore alternative paths.
- Since ***G*** is finite, DFS will eventually exhaust all possible paths from ***s*** to any other reachable vertex.

Contradiction Argument:

1. **Assume DFS does not find g.**
   - This implies that DFS fails to explore the path ***P*** from ***s*** to ***g***, despite ***P*** being a valid path in the graph.
2. **Since G is finite, DFS will eventually visit every vertex reachable from s.**
   - Let $V' \subseteq V$ be the set of all vertices reachable from ***s***. Since ***P*** is a valid path from ***s*** to ***g***, ***g*** must be in V'.
   - By the nature of DFS, every vertex in V' will eventually be visited because DFS continues until all paths have been explored or the goal is found.
3. **DFS must eventually explore the path P entirely.**
   - As DFS explores all vertices in V', it will visit each vertex in the path $P=(s,v_1,v_2,\ldots,g)$ sequentially.
   - Since ***g*** is the last vertex in ***P***, DFS must eventually visit ***g***.
4. **Contradiction:**
   - The assumption that DFS does not find ***g*** contradicts the fact that DFS visits every vertex in V', which includes ***g***.
   - Therefore, our assumption that DFS is not complete must be false.

DFS Applications

Topological Sorting: Used to order tasks in directed acyclic graphs (DAGs) based on dependencies.

Cycle Detection: Employed to detect cycles in both directed and undirected graphs.

Finding Connected Components: Identifies connected components in undirected graphs, helping analyze graph structure and connectivity.

Solving Puzzles: Useful for solving various puzzles, such as mazes and the N-Queens problem, by exploring all possible configurations.

Generating Mazes: DFS can be used to generate mazes by starting at a point and randomly carving paths until all cells are visited, resulting in a maze-like structure.

Limitations of Depth-First Search (DFS)

1. Non-Optimality

- **Limitation:** DFS does not guarantee finding the shortest path in an unweighted graph. It might return a longer path even when a shorter one exists.
- **Impact:** This makes DFS less suitable for applications where the optimal (shortest) solution is required, such as routing and navigation systems.

2. Incompleteness in Infinite Graphs

- **Limitation:** DFS is not complete in infinite graphs, meaning it may fail to find a solution even if one exists. This occurs because DFS can get stuck in an infinitely deep branch without ever exploring other potentially valid paths.
- **Impact:** In problems with infinite or very large state spaces, such as certain types of puzzle-solving or game AI, DFS may not reliably find a solution.

3. Memory Usage in Deep Recursion

- **Limitation:** In cases where the graph or tree is very deep, the recursive implementation of DFS can lead to a stack overflow due to excessive memory usage. This is especially problematic in environments with limited stack space.
- **Impact:** In deeply nested structures or highly recursive problems, DFS may cause program crashes or excessive memory consumption.

4. Sensitivity to Graph Structure

- **Limitation:** DFS's performance can vary significantly depending on the structure of the graph. In unbalanced trees or graphs with long paths, DFS may waste time exploring deep, irrelevant branches before finding the goal.
- **Impact:** This sensitivity makes DFS less predictable and less efficient in cases where the graph structure is not well understood or controlled.

5. Difficulty in Handling Cycles

- **Limitation:** In graphs with cycles, DFS requires careful handling to avoid revisiting nodes, which can lead to infinite loops and wasted computational effort. Although this can be mitigated by marking visited nodes, it adds complexity and increases space requirements.
- **Impact:** In complex graphs with many cycles, managing these challenges can reduce the efficiency and simplicity of using DFS.

6. Limited Applicability in Weighted Graphs

- **Limitation:** DFS does not take edge weights into account, making it unsuitable for problems where paths have different costs. In such scenarios, algorithms like Dijkstra's or A* are more appropriate.
- **Impact:** For problems involving cost optimization or weighted paths, DFS cannot provide the best solution.

Scenarios Where DFS Might Be Inefficient

1. Graphs with Long Paths and Deep Recursion

- **Inefficiency:** DFS explores as deep as possible before backtracking. In graphs with very long paths (or deep trees), DFS may traverse a long, unproductive path before finding the goal. This deep recursion can lead to inefficiency in both time and space.
- **Example:** In a deep tree where the goal node is located near the root but in a different branch, DFS might explore a very long branch that doesn't lead to the goal, resulting in wasted computation.
- **Technical Issue:** If the graph has a deep structure, the recursive DFS implementation may lead to a stack overflow, causing the program to crash or run inefficiently. This is especially problematic in languages with limited stack sizes.

2. Graphs with Cycles

- **Inefficiency:** In graphs that contain cycles, DFS can become inefficient if it revisits nodes, potentially getting stuck in an infinite loop. This is particularly problematic if the graph is large and contains many cycles.
- **Example:** Consider a graph representing a maze with loops. If DFS does not properly mark visited nodes, it might keep revisiting the same nodes, wasting time and computational resources.
- **Mitigation:** While this can be mitigated by marking visited nodes, the need for additional memory to track visited nodes increases the algorithm's space complexity.

3. Graphs with High Branching Factors

- **Inefficiency:** In graphs with high branching factors (i.e., each node has many neighbors), DFS can become inefficient because it might explore a large number of nodes in a deep branch that does not contain the goal.
- **Example:** In a graph where each node has hundreds of neighbors, DFS might explore a long and irrelevant path, causing the algorithm to perform unnecessary computations.
- **Comparison:** In such cases, BFS might be more efficient as it explores all nodes at the current depth level before moving to the next, ensuring that the shortest path is found.

4. Unbalanced Trees

- **Inefficiency:** In unbalanced trees, where some branches are much deeper than others, DFS can become inefficient because it might explore the deeper branches unnecessarily while the goal node might be located in a shallower branch.
- **Example:** In an unbalanced binary tree, if the left branch is very deep and the right branch is shallow (with the goal node), DFS might waste time exploring the deep left branch first.
- **Impact:** This results in longer execution times and increased memory usage, as DFS will explore many nodes that do not contribute to finding the goal.

5. Sparse Graphs with Distant Goals

- **Inefficiency:** In sparse graphs where the goal node is far from the start node, DFS might explore many unnecessary paths that do not lead to the goal. This is because DFS does not have any heuristic to guide it towards the goal, leading to inefficient exploration.
- **Example:** In a graph representing a road network, if the goal is on the opposite side of the graph, DFS might explore many distant and irrelevant paths before finding the correct one.
- **Performance:** This inefficiency becomes more pronounced in large graphs, where the number of potential paths can be enormous.

6. Finding the Shortest Path in Unweighted Graphs

- **Inefficiency:** DFS is not guaranteed to find the shortest path in an unweighted graph, as it may explore a long path first, even when a shorter path exists. This makes DFS inefficient in scenarios where finding the shortest path is crucial.
- **Example:** In an unweighted graph representing social connections, DFS might find a long path between two people, even when a much shorter path exists.
- **Alternative:** Breadth-First Search (BFS) would be more efficient in such cases, as it guarantees finding the shortest path in unweighted graphs.

7. Search in Infinite Graphs

- **Inefficiency:** DFS is not complete in infinite graphs, meaning that it might not find a solution even if one exists. This is because DFS can get stuck exploring an infinitely deep branch without ever finding the goal.
- **Example:** Consider a graph representing possible configurations in a puzzle. If the graph is infinite, DFS might explore an infinitely long sequence of moves without ever backtracking to find the correct solution.
- **Risk:** This makes DFS inefficient and potentially ineffective in applications involving infinite or very large state spaces.

Iterative Deepening Search

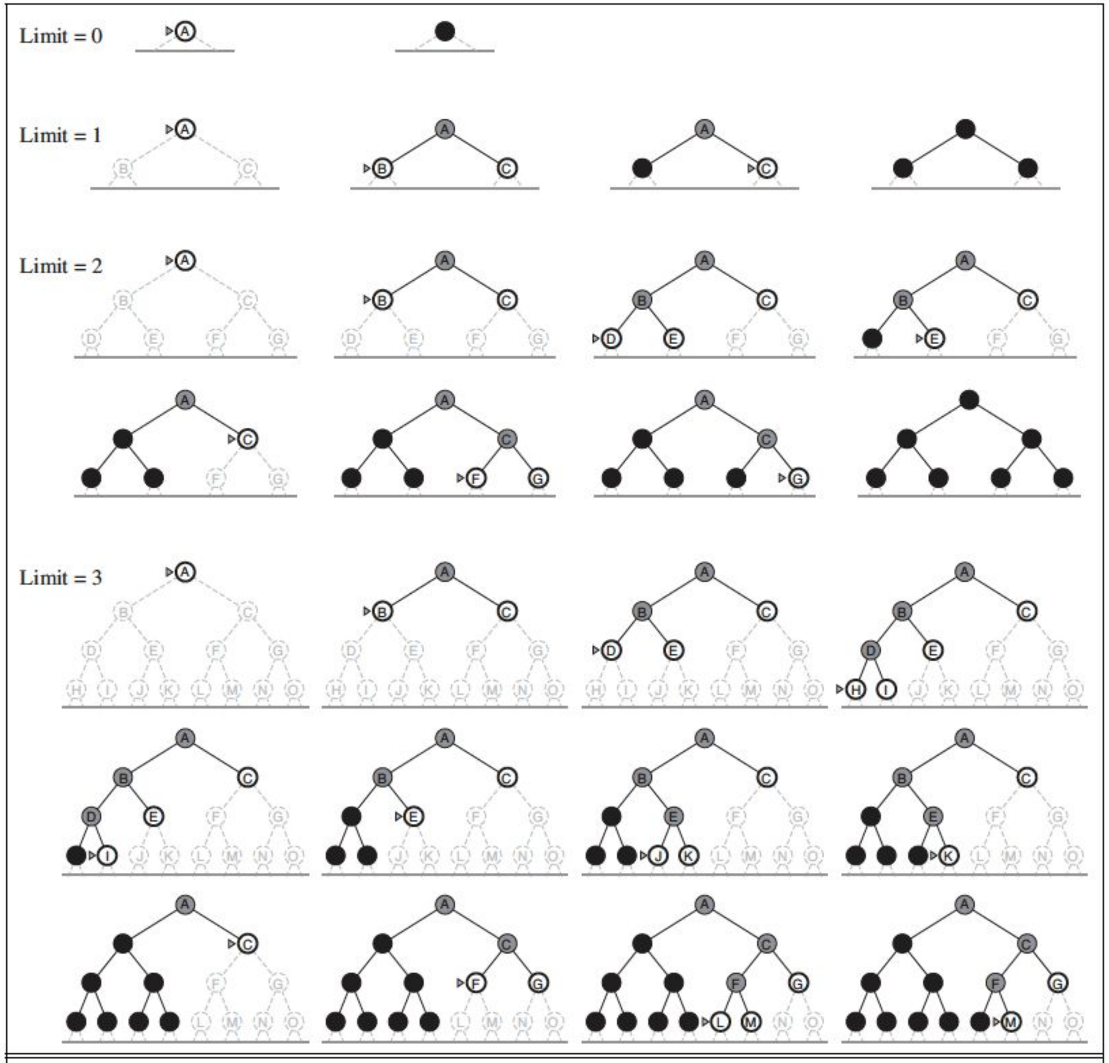
Iterative Deepening Search (IDS) is a search algorithm that combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS). It performs a series of depth-limited searches, incrementally increasing the depth limit with each iteration until a solution is found. This approach is particularly useful for searching large or infinite state spaces while ensuring optimality and completeness.

Key Characteristics of Iterative Deepening Search:

1. **Combination of DFS and BFS**: IDS utilizes the depth-limited search methodology, effectively exploring the search space like DFS but doing so iteratively with increasing depth limits.
2. **Memory Efficiency**: IDS uses much less memory than BFS because it only needs to store the nodes along the current path and the depth limit, making it suitable for large state spaces.

Example

Four iterations of iterative deepening search on a binary tree.



IDS Algorithm Steps

Initialize:

- Set the initial depth limit (usually starting at 0).
- Create a loop that will run until a solution is found.

Depth-Limited Search:

- Perform a depth-limited search for the current depth limit:
  - Start from the initial node and explore its children.
  - If the depth of the current node equals the limit, stop exploring further down that branch.
  - If the current node is the goal node, return it as the solution.
  - Otherwise, push unvisited neighbors onto the stack with their depth incremented by one.

Increment Depth Limit:

- If the depth-limited search fails to find a solution, increment the depth limit by one and repeat the process.

Terminate:

- Continue this process until a solution is found or the search space is exhausted.

Proof of Optimality

Argument for Optimality in Unweighted Graphs:

- **Shallowest Goal First:** In an unweighted graph, the cost of reaching a node is proportional to the depth of the node. Therefore, the optimal solution is the one found at the shallowest depth.
- **Mechanism:** IDS performs a complete search at each depth level before moving to the next level. This means that IDS will explore all nodes at depth ***d*** before exploring any nodes at depth ***d+1***.
- **Guarantee:** Since IDS systematically searches at increasing depths and finds the goal at the shallowest possible depth, it is guaranteed to find the optimal solution (the shallowest goal node) if multiple goals exist.

Mathematical Reasoning:

- Let $g_1, g_2, \ldots, g_n$ be goal nodes at depths $d_1, d_2, \ldots, d_n$, where $d_1 < d_2 < \cdots < d_n$.
- IDS will find g_1 during the iteration with depth limit $l = d_1$.
- Since IDS does not explore deeper levels until all nodes at the current depth have been explored, g_1 will be found before any deeper goals like $g_2, g_3, \ldots$
- **Conclusion:** IDS is optimal in unweighted graphs because it finds the shallowest goal first, ensuring that the least-cost solution is found.

Optimality in Weighted Graphs:

- **Important Note:** IDS is not optimal in weighted graphs where the cost is not solely determined by depth. To guarantee optimality in such scenarios, a different algorithm like Uniform-Cost Search or A* would be necessary.

Proof of Completeness

Argument for Completeness:

- **Mechanism:** IDS performs a series of Depth-Limited Searches (DLS), starting from a depth limit of 0 and incrementally increasing the depth limit by 1 in each iteration.
- **Process:** In each iteration, IDS explores all nodes at the current depth before increasing the limit. This means that, for a finite state space, IDS will eventually explore all possible depths in the search tree.
- **Guarantee:** Since IDS increments the depth limit with each iteration, it will eventually reach the depth at which the goal node is located, provided that the goal exists within the state space.
- **Handling Infinite State Spaces:** In the case of an infinite state space, if a solution exists at some finite depth ***d***, IDS will find it after ***d*** iterations.

Mathematical Reasoning:

- Let ***d*** be the depth of the shallowest goal node in the search tree.
- IDS will perform a DLS with depth limits l = 0,1,2,…,d
- When the depth limit reaches ***l = d***, IDS will find the goal node because it explores all nodes at that depth.

Conclusion: IDS is complete because it systematically explores deeper levels until the goal is found. If the goal exists at any finite depth, IDS will find it.

IDS Performance Measure

Optimality: IDS is optimal for unweighted graphs, meaning it will always find the shortest path (in terms of the number of edges) to the goal node.

Completeness: IDS is complete, ensuring that if a solution exists, it will be found eventually, even in infinite search spaces.

Time Complexity: The time complexity is $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution. Although this is similar to BFS, IDS performs better in terms of memory usage.

Space Complexity: The space complexity is $O(d)$, where ddd is the maximum depth. This is much more efficient than BFS, which has a space complexity of $O(b^d)$.

IDS Applications

Distributed Systems: In distributed systems, IDS can be used to explore state spaces across multiple nodes or processors while minimizing communication overhead. Each node can handle its own depth-limited search, incrementally sharing results to reach a global solution.

AI Planning: IDS can be applied in automated planning systems where actions can lead to exponentially growing state spaces. By incrementally deepening the search, it can effectively manage the exploration of possible plans and their associated consequences.

Exploration in Robotics: IDS can be utilized in robotic exploration tasks, where robots systematically explore unknown environments. By controlling the depth of exploration, robots can cover areas thoroughly without exceeding memory constraints.

Software Model Checking: In software verification and model checking, IDS can be used to exhaustively explore state spaces of finite-state systems. By incrementally increasing depth, it can verify properties of systems without running out of memory.

Genetic Algorithms: In hybrid approaches that combine genetic algorithms with search techniques, IDS can be used to guide the search process through a population of solutions. This allows for focused exploration of the most promising areas of the solution space while managing computational resources effectively.

Limitations of Iterative Deepening Search (IDS)

1. Redundant Exploration of Nodes

Limitation:

- **Redundant Searches:** IDS repeatedly explores the same nodes at shallower depths during each iteration. For example, nodes at depth 1 are explored in every iteration until the depth limit exceeds 1. This redundancy can lead to inefficiencies, particularly when the search space is large.

Scenarios Where This is Inefficient:

- **Large Search Spaces:** In search problems with a large number of nodes, such as complex combinatorial problems, the repeated exploration of nodes can result in significant overhead, making IDS slower than other algorithms that avoid such redundancy.
- **Problems with Shallow Goals:** In scenarios where the goal is located at a shallow depth, IDS's repeated exploration of shallower nodes adds unnecessary computation, making it less efficient compared to a single BFS.

2. High Computational Overhead in Deep Searches

Limitation:

- **Exponential Time Complexity:** The time complexity of IDS is exponential in the depth of the solution because it revisits nodes multiple times across different depth limits. As the depth of the search increases, the number of nodes revisited grows exponentially.

Scenarios Where This is Inefficient:

- **Deep Solutions:** In search problems where the goal is located at a significant depth, such as deep game trees or long-path puzzles, IDS may become inefficient due to the large number of nodes that need to be revisited across multiple iterations.
- **Complex Planning Problems:** In complex AI planning problems where solutions require exploring deep decision trees, IDS's computational overhead can make it less suitable, especially when compared to more direct depth-first approaches.

3. Memory Usage and Resource Constraints

Limitation:

- **Space Complexity vs. Efficiency:** While IDS is designed to be memory-efficient (similar to DFS), it still requires enough memory to handle repeated depth-limited searches. The memory efficiency can be a trade-off against the time complexity, which may not be ideal in all situations.

Scenarios Where This is Inefficient:

- **Resource-Constrained Environments:** In environments with strict memory or computational constraints, such as embedded systems or mobile devices, the overhead of repeated searches may not be justifiable, leading to inefficiency.
- **Real-Time Applications:** In real-time systems, where quick decision-making is critical (e.g., robotics or online pathfinding), the time required by IDS to revisit nodes might introduce delays, making it less suitable for time-sensitive tasks.

Uniform Cost Search

Uniform Cost Search (UCS) is a search algorithm that is used to find the least-cost path from a given starting node to a goal node in a weighted graph. It is a variant of Dijkstra's algorithm and is a form of best-first search that expands the node with the lowest path cost first. UCS is particularly useful for finding optimal solutions in scenarios where path costs vary.

Key Characteristics of Uniform Cost Search:

1. **Cost-Based Expansion**: UCS expands nodes based on the cumulative cost to reach them from the start node. It always chooses the node with the lowest total path cost for expansion.
2. **Data Structures**: UCS uses a priority queue (often implemented as a min-heap) to manage the nodes based on their cumulative costs. This allows for efficient retrieval of the next node to expand.

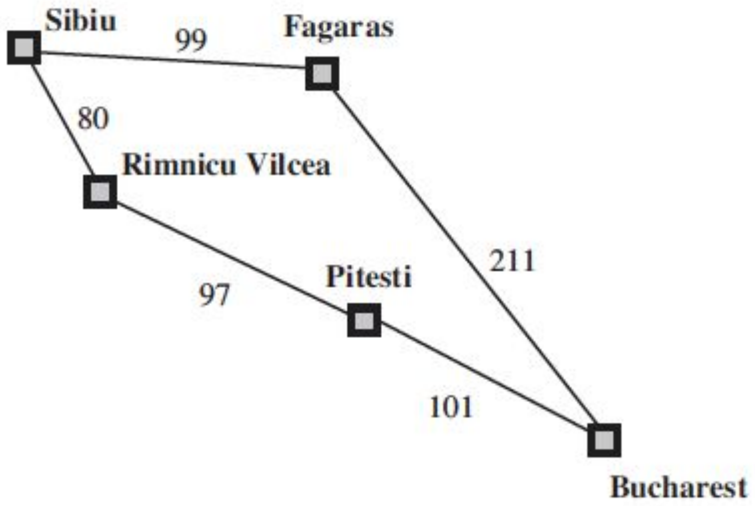
Contrast with BFS and DFS:

- **BFS:** Explores level by level and finds the shortest path in terms of the number of edges (in unweighted graphs).
- **DFS:** Explores deeply before backtracking but does not guarantee finding the shortest or least-cost path.
- **UCS:** Explores paths based on their cumulative cost, ensuring the discovery of the least-cost path in weighted graphs.

Real-world analogy: Imagine you’re navigating a city using public transport. You want to reach your destination using the least expensive route, regardless of how many stops it takes. UCS helps you find this optimal route.

Example

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



Dijkstra’s Algorithm: A specific case of UCS where all edge costs are non-negative, commonly used in graph-based pathfinding.

UCS Algorithm Steps

Initialize the Priority Queue:

- Create a priority queue (often implemented as a min-heap) to store the frontier. The frontier is the set of all leaf nodes available for expansion at any given point.
- Start by inserting the initial node (start state) into the priority queue with a cumulative cost of 0.

Initialize the Explored Set:

- Create an empty set to keep track of the nodes (states) that have already been expanded (explored) to avoid redundant processing.

Begin the Search Loop:

- **While the priority queue is not empty**, repeat the following steps:

Pop the Node with the Lowest Cost:

- Extract the node with the lowest cumulative cost from the priority queue. This node represents the current path with the smallest cost.
- Let this node be called `current_node`, and its associated cost be `current_cost`.

UCS Algorithm Steps

Goal Test:

- Check if `current_node` is the goal state.
- If `current_node` is the goal state, the algorithm terminates and returns the solution, which includes the path and the total cost to reach the goal.

Add the Current Node to the Explored Set:

- If `current_node` is not the goal state, add it to the explored set to ensure that it is not expanded again.

Expand the Current Node:

- For each child (or successor) of `current_node`:
  - **Calculate the Cumulative Cost:** Determine the cumulative cost to reach the child by adding the edge cost from `current_node` to the child's cumulative cost.
  - **Check if the Child is in the Explored Set:**
    - If the child is not in the explored set or the priority queue, add it to the priority queue with its cumulative cost.
    - If the child is already in the priority queue but with a higher cumulative cost, update the priority queue with the new lower cost.
  - **Note:** UCS prioritizes paths with lower costs, so the priority queue ensures that the node with the lowest cumulative cost is expanded next.

Termination:

- If the priority queue becomes empty and the goal has not been found, this indicates that no solution exists within the given state space, and the algorithm returns failure.

UCS Algorithm

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Proof of Optimality

Step 1: Node Expansion and Path Costs

- Consider the moment UCS expands the goal node ***g***. Let ***C(g)*** be the cumulative cost associated with reaching ***g*** along the path UCS followed.
- Assume, for the sake of contradiction, that there exists another path to ***g*** with a lower cost, denoted as ***C'(g) < C(g)***.

Step 2: Contradiction by Assumption

- Since UCS always expands the node with the lowest cumulative cost next, it must have expanded all nodes with cumulative costs less than ***C(g)*** before expanding ***g***.
- If there were a path to ***g*** with a cumulative cost ***C'(g) < C(g)***, UCS would have expanded the nodes along this cheaper path to ***g*** first.
- Therefore, UCS would have reached ***g*** via the cheaper path before considering the path with cost ***C(g)***.

Step 3: Conclusion

- The assumption that there is a cheaper path ***C'(g) < C(g)*** contradicts the way UCS operates, as UCS would have found and expanded this cheaper path first.
- Since UCS expands nodes in order of their cumulative path cost, when UCS expands the goal node ***g***, the path ***C(g)*** must be the minimum possible cost to reach ***g***.
- Therefore, no other path to ***g*** with a cost less than ***C(g)*** exists.

Step 4: Generalization

- This reasoning holds for all nodes in the graph, not just the goal node. Thus, UCS guarantees that when it expands any node, it has found the least-cost path to that node.
- When UCS expands the goal node, it has found the optimal solution to the problem.

Proof of Completeness

Step 1: Initialization and Node Expansion

- UCS starts by initializing the priority queue with the start node ***s*** and a path cost of 0.
- Nodes are expanded in order of increasing cumulative cost, meaning that the node with the lowest cost is expanded first.

Step 2: Handling Non-Negative Costs

- Each edge in the graph has a non-negative cost, meaning that the cumulative cost of any path increases as nodes are expanded.
- Because UCS always expands the node with the smallest cumulative cost, it systematically explores paths in increasing order of their total cost.

Step 3: Exploring All Paths

- Suppose a solution (a path from ***s*** to ***g***) exists.
- UCS will continue expanding nodes and exploring new paths until the goal node ***g*** is reached.
- Because all edge costs are non-negative, the cumulative cost associated with paths cannot decrease, ensuring that UCS doesn't revisit nodes with lower costs unnecessarily.

Step 4: No Infinite Loop or Incomplete Exploration

- Since UCS expands nodes in order of increasing cost, it avoids getting stuck in loops or revisiting nodes excessively.
- If the graph is finite, UCS will eventually exhaust all possible paths leading from sss and will reach ***g*** because it explores every possible path with increasing cost until the goal is found.
- If the graph is infinite but has finite cumulative costs, UCS will still find the goal ***g*** by exploring paths with increasingly higher costs. If a path exists, UCS will eventually expand the goal node.

Step 5: Termination

- UCS terminates as soon as it expands the goal node ***g***. Since UCS prioritizes paths with lower costs, the first time UCS expands ***g***, it does so using the least-cost path.
- Since UCS expands nodes based on cumulative path costs, it ensures that all possible paths to ***g*** are considered, and ***g*** will be reached if reachable.

Conclusion

- **Completeness:** UCS is complete because it systematically explores all possible paths from the start node ***s*** in increasing order of cost. If a path to the goal node ***g*** exists, UCS will eventually find and expand ***g***, ensuring that the goal is reached.

Time Complexity

Assumptions:

- Let ***b*** be the branching factor, which is the average number of children each node has.
- Let ***C**** be the cost of the optimal (least-cost) solution.
- Let ***ϵ*** be the smallest positive edge cost in the graph.
- The state space is assumed to be finite, and all edge costs are non-negative.

Basic Operation of UCS

- UCS explores nodes in increasing order of their cumulative path cost.
- Each time a node is expanded, UCS adds its children to the priority queue with their cumulative path costs.
- The priority queue is used to ensure that the node with the lowest cumulative path cost is expanded next.

Number of Nodes Expanded by UCS

- The key to understanding the time complexity is estimating how many nodes UCS might need to expand to find the optimal solution.

Cost Boundaries and Path Costs:

- Let ***C(n)*** be the cumulative cost of the path from the start node ***s*** to a node ***n***.
- For UCS to explore a node ***n*** before finding the goal node ***g***, the cost ***C(n)*** must be less than or equal to ***C****, where ***C**** is the cost of the optimal solution.
- Since UCS only expands nodes with costs ***C(n) ≤ C****, the number of nodes expanded depends on how many nodes have path costs within this range.

Depth and Branching Factor Consideration:

- The number of nodes that UCS might consider expands exponentially with the depth of the tree. However, in UCS, "depth" is more appropriately measured in terms of path cost rather than the number of edges.
- For each depth level in terms of cost, UCS could potentially explore all nodes whose cumulative costs fall within that level.
- Each node ***n*** can generate up to ***b*** children. If UCS explores nodes up to a cumulative cost ***C****, the number of nodes expanded is related to the total number of nodes within that cost bound.

Estimating the Total Number of Nodes Expanded

- The total number of nodes UCS can expand is determined by how many nodes have a cumulative path cost less than or equal to ***C****.
- Consider the worst-case scenario where UCS might expand nodes even with slightly larger costs ***C(n) > C**** due to ties or small variations in costs. However, in the limit, the number of nodes with path costs within ***C**** is proportional to $b^{C^*/\epsilon}$.
- Therefore, the time complexity of UCS is dominated by the number of nodes with cumulative costs within ***C****, which is approximately:

$$\textbf{Time Complexity} = O(b^{C^*/\epsilon})$$

Conclusion

- The time complexity of UCS is $O(b^{C^*/\epsilon})$, where:
  - ***b*** is the branching factor,
  - ***C****is the cost of the optimal solution, and
  - ϵ is the smallest positive edge cost in the graph.
- This complexity reflects the exponential growth in the number of nodes UCS must expand as the cost C* increases, relative to the smallest cost increment ϵ.

UCS Performance Measure: Summary

Optimality: UCS is optimal because it expands nodes in order of their cumulative cost, ensuring that the first time it reaches the goal, it does so via the least-cost path.

Completeness: UCS is complete as long as all edge costs are non-negative. It will always find a solution if there is one because it explores all possible paths in order of increasing cost.

Time Complexity: UCS has a time complexity of $O(b^{C^*/\epsilon})$, where b is the branching factor, C^* is the cost of the optimal solution, and ϵ is the smallest positive edge cost. The time complexity grows exponentially with the cost of the optimal path.

Space Complexity: UCS has a space complexity of $O(b^{C^*/\epsilon})$, as it must store all paths in the priority queue. This is similar to its time complexity, meaning UCS can be memory-intensive, especially in large search spaces.

UCS Applications

Logistics and Supply Chain Management: UCS can optimize transportation routes, determining the least-cost path for moving goods from warehouses to distribution centers while considering fluctuating transportation costs.

Telecommunication Network Design: UCS is used to find optimal routing paths for data packets in networks, minimizing latency and costs while maximizing bandwidth and reliability.

AI in Robotics: UCS assists in robotic path planning by navigating complex environments with varying movement costs, allowing robots to conserve energy and maximize efficiency in tasks.

Medical Treatment Pathways: UCS optimizes treatment plans by evaluating different options based on cost-effectiveness and resource allocation, helping identify the most efficient pathway to achieve desired health outcomes.

Urban Planning: UCS helps evaluate construction routes for infrastructure development, allowing urban planners to minimize expenses and environmental impact by prioritizing paths with lower costs.

Limitations of UCS

1. High Memory Usage

Limitation:

- **Space Complexity:** UCS requires significant memory to store all the nodes in the priority queue until the search reaches the goal. Since UCS must store every explored path in the queue to ensure that the least-cost path is expanded first, the space complexity can become problematic, particularly in large or dense graphs.

Scenarios Where This is Inefficient:

- **Large Graphs:** In applications such as mapping large cities or vast networks, where the number of nodes and edges is very high, UCS can consume large amounts of memory, potentially exhausting available resources.
- **Dense Graphs:** In graphs where most nodes are connected by edges, the number of potential paths that UCS needs to store increases rapidly, leading to excessive memory usage.

2. Slow Performance in Graphs with Uniform Costs

Limitation:

- **Time Complexity:** When edge costs are relatively uniform (or identical), UCS behaves similarly to Breadth-First Search (BFS), exploring many unnecessary nodes at the same level of cost. This can lead to slow performance because UCS does not take advantage of cost variations to prune the search space effectively.

Scenarios Where This is Inefficient:

- **Uniformly Weighted Graphs:** In scenarios like grid-based pathfinding where all moves have the same cost, UCS will explore a broad set of nodes, making it slower compared to algorithms that exploit heuristics.
- **Flat Terrain Navigation:** In robotics, if a robot is navigating on a flat surface where all paths are equally costly, UCS might explore many unnecessary paths before finding the goal, leading to inefficiency.

3. Sensitivity to Small Variations in Edge Costs

Limitation:

- **Minimal Edge Cost ϵ\epsilonϵ:** The performance of UCS is sensitive to the smallest edge cost in the graph. If the smallest edge cost ϵ\epsilonϵ is very small, UCS may have to explore a large number of paths before distinguishing between slightly different cumulative costs, resulting in high computational overhead.

Scenarios Where This is Inefficient:

- **Graphs with Minor Cost Differences:** In applications such as financial modeling or logistics planning where small cost differences between paths matter, UCS might expand many similar-cost paths, increasing the computational burden.
- **Network Routing with Variable Costs:** In network routing where data packet delivery costs vary slightly due to congestion or traffic, UCS might inefficiently explore many near-optimal paths, increasing processing time.

Informed Search

Three functions

- f(n) = is the evaluation function that best-first search uses to figure out which frontier node to expand
- g(n) = cost from the start node to the current node
- h(n) = guess of the distance/cost from current node to the goal node

f(n) - best node overall cumulatively

g(n) - closer node from the start

h(n) - how far is the goal from me

Informed Search Strategies

Informed search strategies in artificial intelligence (AI) refer to search algorithms that use additional information beyond the basic problem definition to guide the search process towards a solution more efficiently. This additional information typically comes in the form of a heuristic, which is a function that estimates the cost or distance from a given state to the goal state. The purpose of using a heuristic is to make the search process more directed and, ideally, faster by focusing on the most promising paths in the search space.

Key Characteristics of Informed Search Strategies:

1. **Heuristics**: The defining feature of informed search strategies is the use of heuristics. A heuristic is a rule of thumb or an educated guess that helps to estimate how close a current state is to the goal state. For example, in a pathfinding problem, the straight-line distance (Euclidean distance) to the goal can be used as a heuristic.
2. **Efficiency**: By using heuristics, informed search strategies aim to explore fewer nodes than uninformed strategies (like Breadth-First Search or Depth-First Search), making them more efficient in terms of time and memory.

3. **Goal-Directed Search**: Informed search strategies are more goal-directed because they use the heuristic to prioritize which paths to explore first, typically focusing on those that seem most likely to lead to a solution.

4. **Optimality and Completeness**: Some informed search algorithms, are both optimal (they find the best possible solution) and complete (they will find a solution if one exists), provided the heuristic used is admissible (never overestimates the cost to reach the goal).

Heuristics

A **heuristic** is a rule of thumb, a practical method, or a shortcut that helps in decision-making, problem-solving, or discovery. It is not guaranteed to be perfect or optimal, but it is often effective for reaching a satisfactory solution in a reasonable amount of time. In AI, heuristics are used to speed up the process of finding a good solution, especially when the problem space is vast and exhaustive search would be computationally expensive or infeasible.

Characteristics of Heuristics:

1. **Simplification**: Heuristics simplify complex problems by focusing on the most relevant aspects, ignoring less critical details.
2. **Efficiency**: They provide quick, approximate solutions that are "good enough" for the task at hand, even if they are not optimal.
3. **Domain-Specific**: Heuristics are often tailored to specific problems or domains. For example, a heuristic for a chess-playing AI might involve evaluating board positions based on the value of the pieces.
4. **Guidance**: In search algorithms, heuristics guide the search process by estimating which paths are more promising, reducing the number of states that need to be explored.

Heuristics Function

A **heuristic function** (often denoted as `h(n)`) is a specific type of heuristic used in search algorithms. It is a function that estimates the cost or distance from a given node (or state) `n` in the search space to the goal state. The heuristic function helps to prioritize which nodes should be explored next based on their estimated proximity to the goal.

Properties of Heuristic Functions:

1. **Admissibility**:
   - A heuristic function is **admissible** if it never overestimates the true cost to reach the goal from any node. In other words, `h(n)` is always less than or equal to the actual lowest cost to reach the goal.
   - Admissible heuristics are crucial in ensuring that search algorithms like A* find the optimal solution.
2. **Consistency (or Monotonicity)**:
   - A heuristic function is **consistent** if, for every node `n` and every successor `n'` of `n`, the estimated cost of reaching the goal from `n` is no greater than the cost of reaching `n'` plus the estimated cost of reaching the goal from `n'`. Mathematically, this means: $h(n) \leq c(n,n') + h(n')$ where `c(n, n')` is the actual cost of moving from node `n` to node `n'`.
   - Consistency ensures that the heuristic function never decreases along a path, which helps in simplifying the implementation of search algorithms.
3. **Informedness**:
   - The **informedness** of a heuristic function refers to how accurately it estimates the true cost to the goal. A more informed heuristic provides estimates closer to the actual costs, making the search more efficient by reducing unnecessary exploration of non-promising paths.

Examples of Heuristic Functions

Manhattan Distance: In grid-based pathfinding problems, such as navigating through a maze, the Manhattan distance (sum of the absolute differences in the x and y coordinates) between the current node and the goal is a common heuristic. It is admissible and often used in algorithms like A*.

$$h(n) = |x_2 - x_1| + |y_2 - y_1|$$

Euclidean Distance: In problems where the cost to move from one point to another is based on straight-line (as-the-crow-flies) distance, the Euclidean distance (the direct distance between two points) can be used as a heuristic.

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Examples of Effective Heuristics

Pathfinding in Grid-Based Environments

- **Heuristic:** Manhattan Distance
- **Why It's Effective:** The Manhattan distance provides a heuristic for grid-based movement where diagonal moves are not allowed. It is simple to compute and provides a good estimate of the actual path cost.

Sliding Puzzle (8-puzzle, 15-puzzle)

- **Heuristic:** Number of Misplaced Tiles or Manhattan Distance
- **Why It's Effective:** Both heuristics provide estimates for the minimum number of moves required to solve the puzzle. The number of misplaced tiles heuristic is simple, while the Manhattan distance gives a more refined estimate by accounting for how far each tile is from its goal position.

Traveling Salesman Problem (TSP)

- **Heuristic:** Minimum Spanning Tree (MST)
- **Why It's Effective:** The MST heuristic estimates the lower bound on the cost of completing the tour by connecting all cities without returning to the start.

Types of Informed Search Strategies

1. **Greedy Best-First Search**
2. **A* Search**

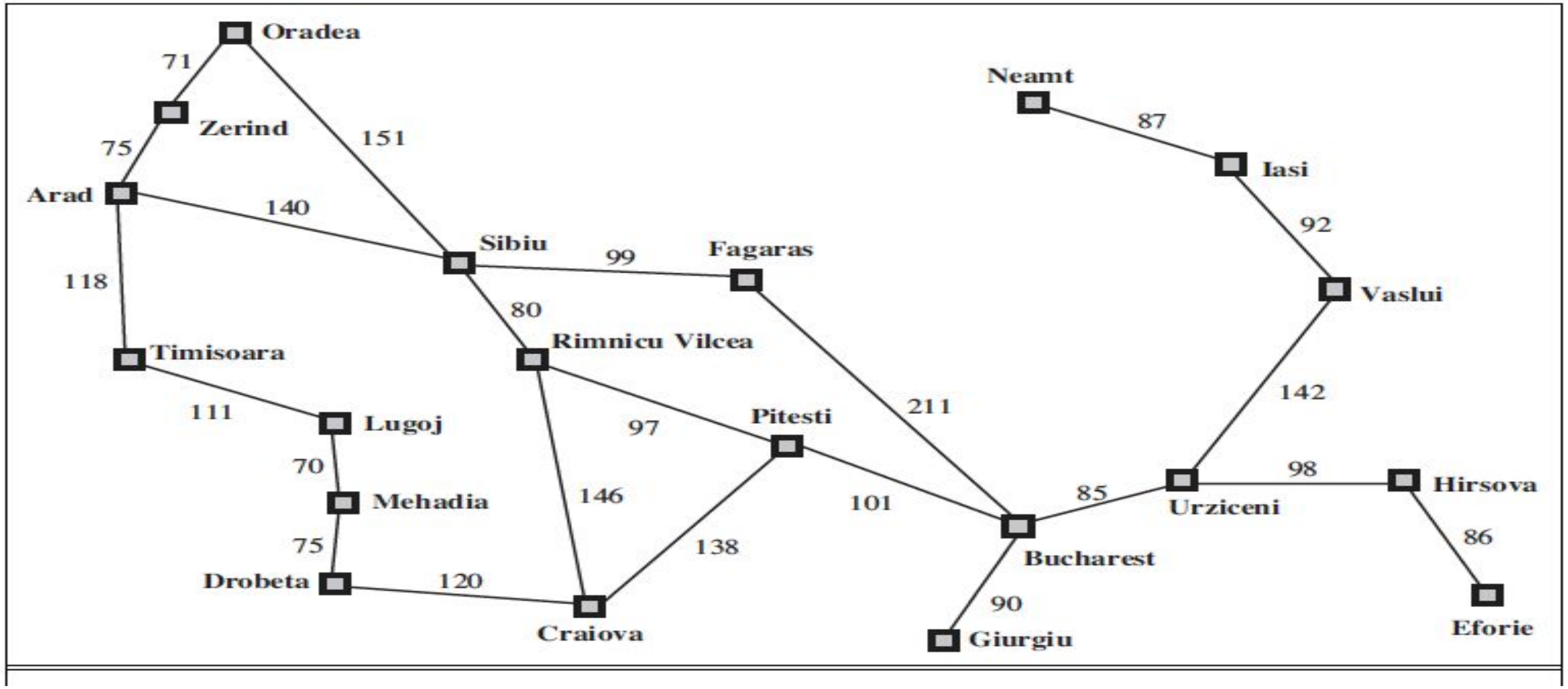
Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that aims to find the most promising path to a goal by expanding the node that appears closest to the goal based solely on a heuristic function. Unlike other search algorithms that consider both the cost to reach a node and the estimated cost to reach the goal, Greedy Best-First Search focuses only on the heuristic estimate, making it "greedy" in its approach.

Here, f(n) = h(n).

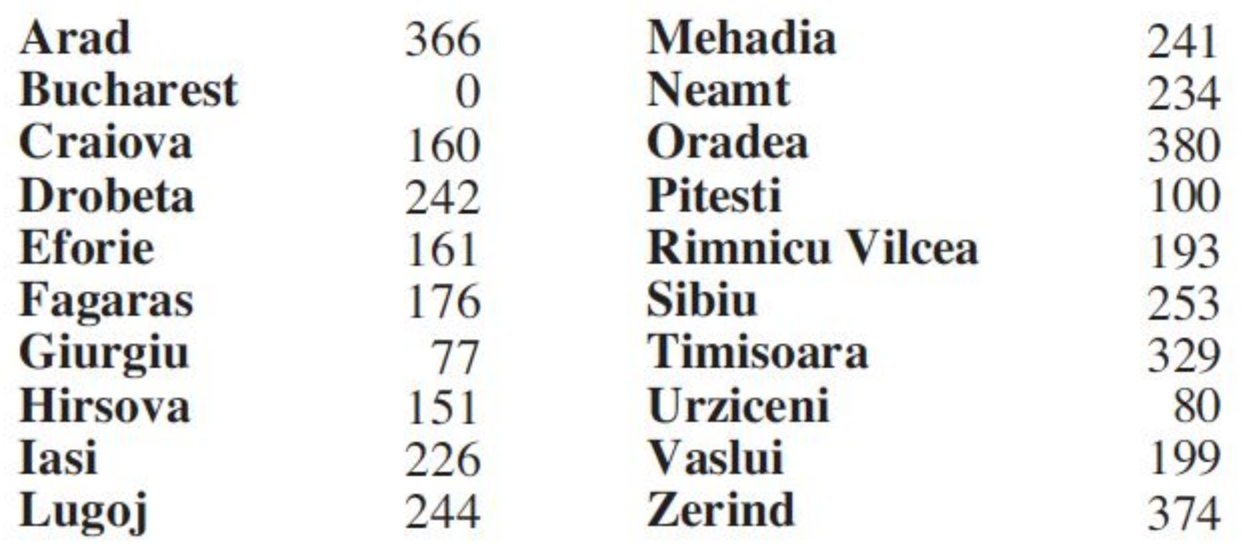
How Greedy Best-First Search Works

1. **Initialization**:
   - Start with an initial node, usually the starting state of the problem.
   - Add this node to a priority queue (often implemented as a min-heap) where nodes are prioritized based on their heuristic value `h(n)`.
2. **Node Expansion**:
   - Extract the node with the lowest heuristic value from the priority queue (this is the "greedy" part, as it selects the node that seems closest to the goal).
   - If this node is the goal, the search ends successfully.
   - If not, expand the node by generating all its successors (i.e., possible states that can be reached from the current state).
3. **Adding Successors to the Queue**:
   - For each successor, compute its heuristic value `h(n)` and add it to the priority queue.
   - The priority queue is then updated to ensure that the node with the lowest `h(n)` will be expanded next.
4. **Repeat**:
   - Continue expanding nodes by repeatedly extracting the node with the lowest `h(n)` and adding its successors to the queue until the goal is found or the queue is empty (indicating that no solution exists).



A simplified road map of part of Romania

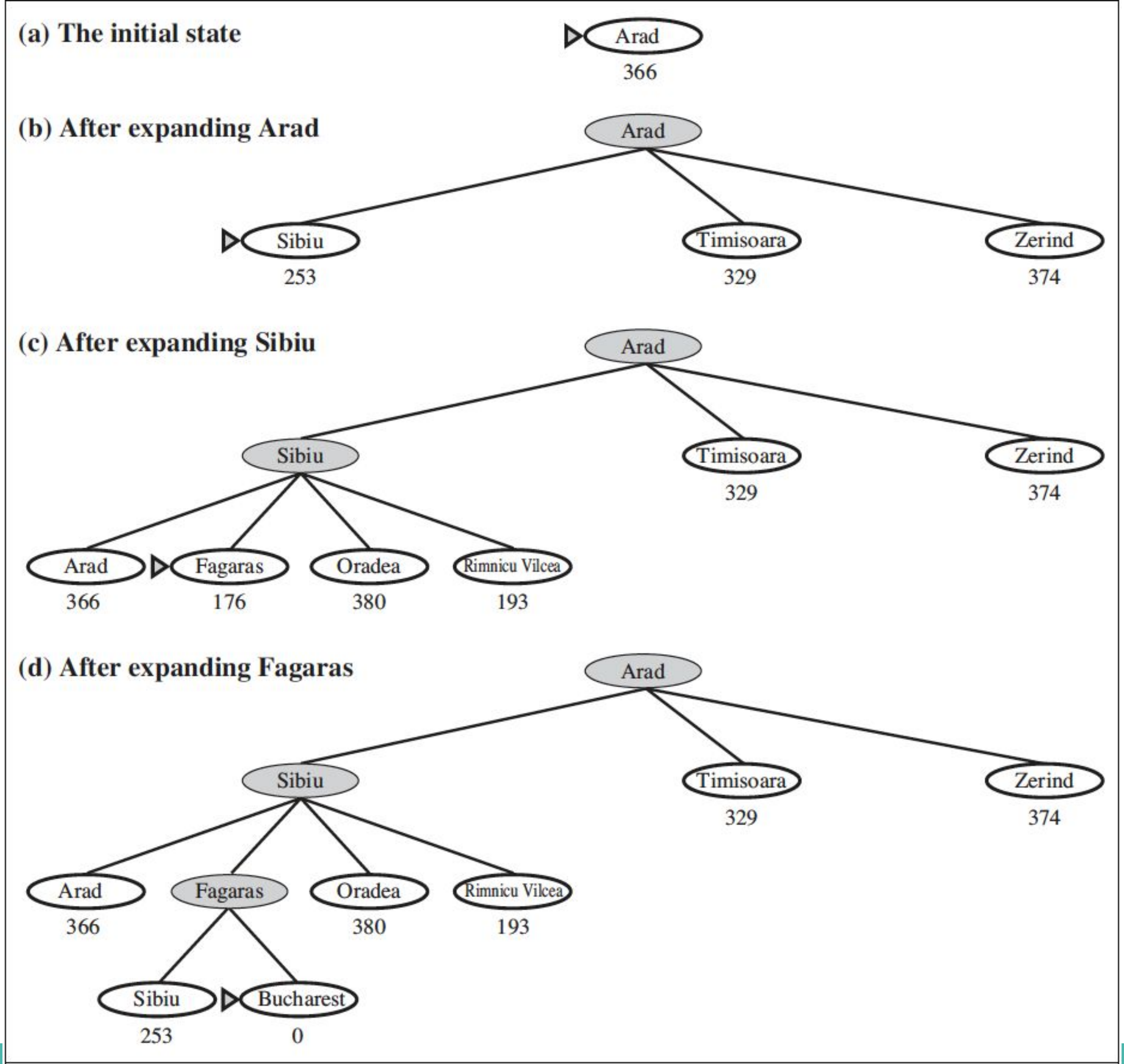
Values of h_{SLD} – straight-line distances to Bucharest



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.



Properties of Greedy Best-First Search

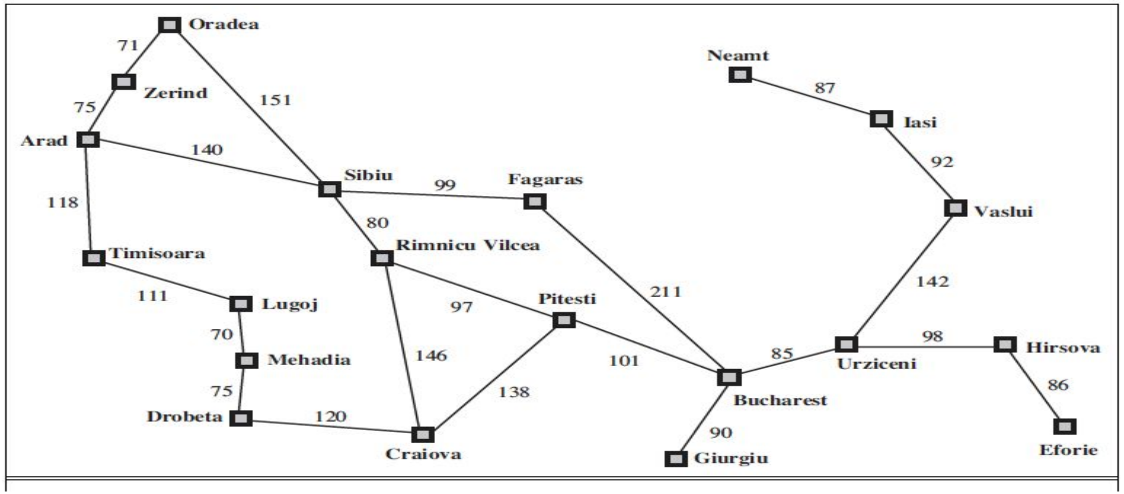
1. **Heuristic-Driven**: GBFS relies entirely on the heuristic function `h(n)` to make decisions. It does not consider the actual cost to reach a node, making it greedy.
2. **Not Optimal**: Since GBFS only considers the heuristic and ignores the actual cost to reach nodes (`g(n)`), it is not guaranteed to find the optimal solution. It may choose a path that looks promising (based on the heuristic) but turns out to be suboptimal.
3. **Not Complete**: GBFS is not complete, meaning it may fail to find a solution even if one exists, especially if the heuristic misguides the search into dead ends or cycles in case of tree-search paradigm, but complete in case of graph-search paradigm.
4. **Efficiency**: GBFS can be faster than algorithms like A* because it often expands fewer nodes. However, this speed comes at the cost of potentially finding suboptimal or incomplete solutions.
5. **Memory Usage**: GBFS stores all generated nodes in memory, like A*. However, because it doesn't keep track of the actual cost `g(n)`, it might require less memory than A* in some cases.

Advantages of Greedy Best-First Search

- **Speed**: GBFS can be very fast, especially in problems where the heuristic is well-chosen and closely approximates the true cost to the goal.
- **Simplicity**: The algorithm is relatively simple to implement, focusing solely on the heuristic to guide the search.

In Greedy Best First Search

- We are ignoring the cost to the current path and only looking to the extra cost to give to the goal
- In GBFS, we don't consider g(n)
- f(n) = h(n)
- Is this ideal?



A simplified road map of part of Romania

- Our objective is to go from the start state to the goal, not from the goal.
- The state ***n*** is somewhere in the middle, but our objective is to go from the start state to the goal and
- as we are taking the minimum, we should not myopically only take the minimum concerning how much cost we are going to pay in the future, but also take minimum over the cost that we have paid so far.