Artificial Intelligence
AI 2002

Lecture 4

Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Recap

- Problem Solving Agents
- Problem Abstraction
- Search Tree
- Tree Search Algorithm
- Graph Search Algorithm

Search Strategy

A search strategy is defined by picking the order of node expansion

- In Artificial Intelligence, search technique are the universal problem solving method.

Search Algorithm Terminologies

- Searching is a step by step procedure to solve a search problem in a given search space.
- A search problem have three main factors

1. Search state

   Set of possible solution a system may have

2. Start state

   State where the agents begin the search

3. Goal Test

   Function which observed the current state and returns whether the goal state achieve or not.

Properties of Search Strategies

Strategies are evaluated along the following dimensions:

1. Completeness

   Strategy is said to be complete if it guarantee to return the solution if at least the solution exist

2. Optimality

   Solution for the problem is guarantee to be the best solution (least cost solution) among all other solutions.

3. Time Complexity

   Complexity is measure in time in which need to complete the task (no of nodes generated)

4. Space Complexity

   Maximum amount of storage space required at any point during the search (maximum number of nodes in memory)

Search Strategy (Cont…)

- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be ∞)

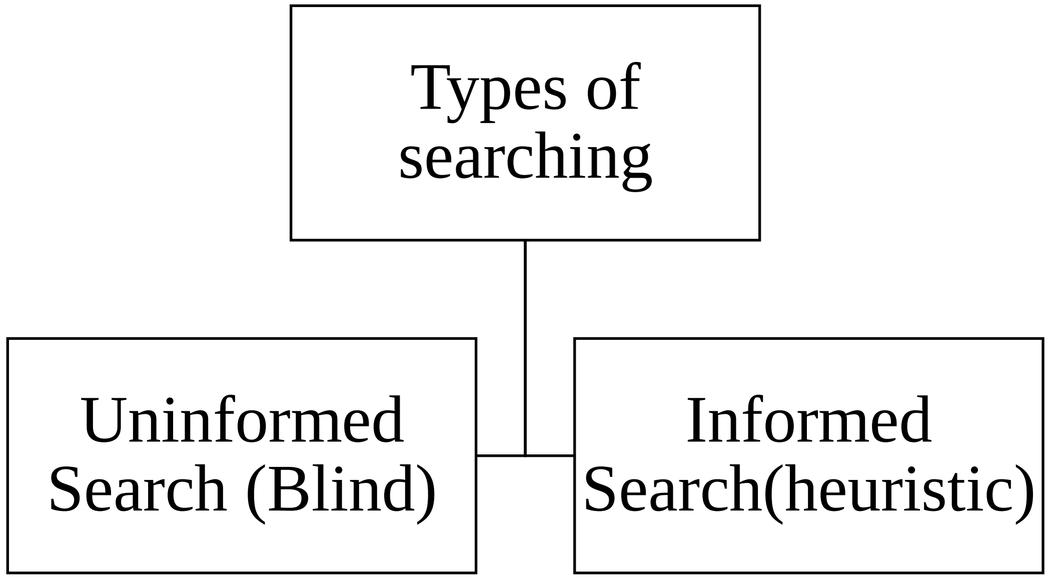
- Search cost (time)
- total cost (time+space)

Types of Search Strategy

- Based on search problems we classify it in two types

- **<u>Uninformed search (blind search)</u>**
  strategies use only the information available in the problem definition

- **<u>Informed Search (Heuristic Search)</u>**
  Strategies that know whether one non-goal state is better than another are called informed search or heuristic search



Types of Uninformed Search Strategy

- General uninformed search strategies:

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening search

Breadth First Search

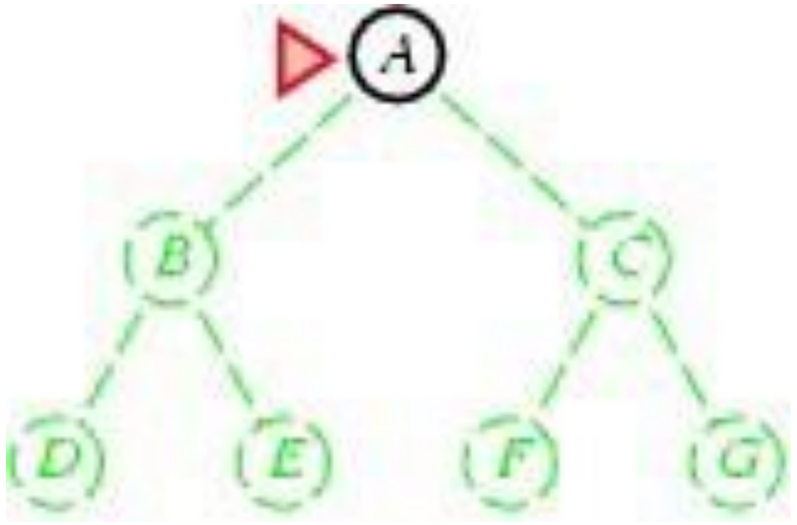
- Breadth first search is a simple strategy in which
  - the root node is expanded first then
  - All the successors of the root node are expanded next
  - their successors, and so on.
- In General, all the nodes are expanded at a given depth in the search tree before any node at the next level are expanded.

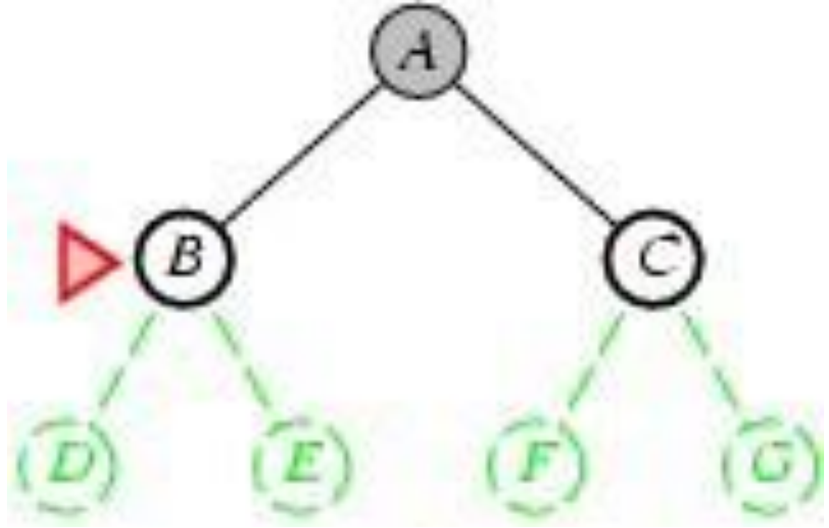
Breadth First Search (Cont…)

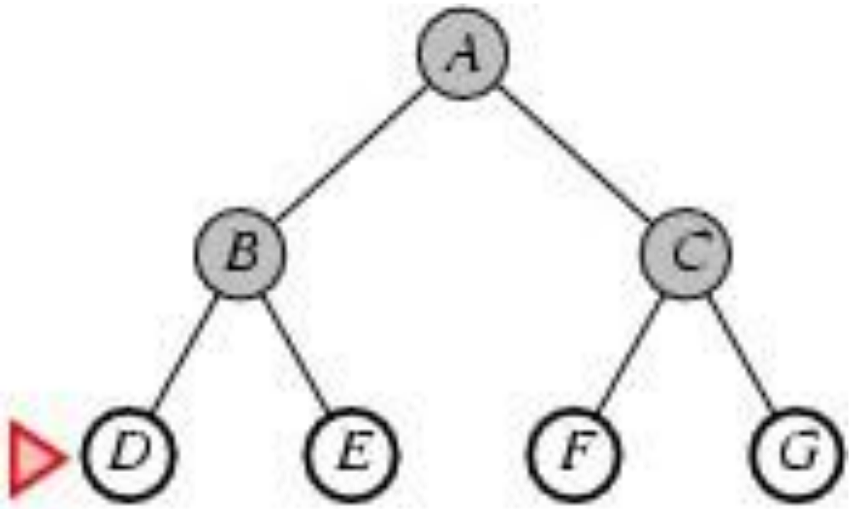
Expand shallowest unexpanded node

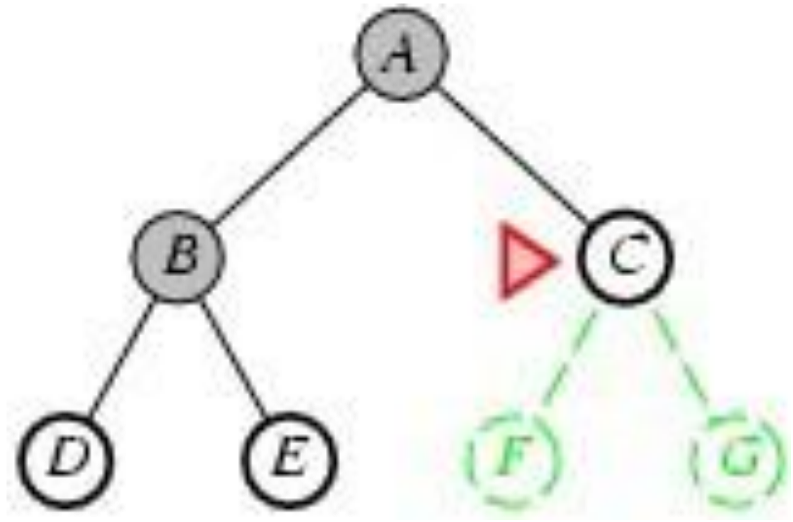
- **Implementation:**

Frontier is a FIFO queue, i.e., new successors go at end

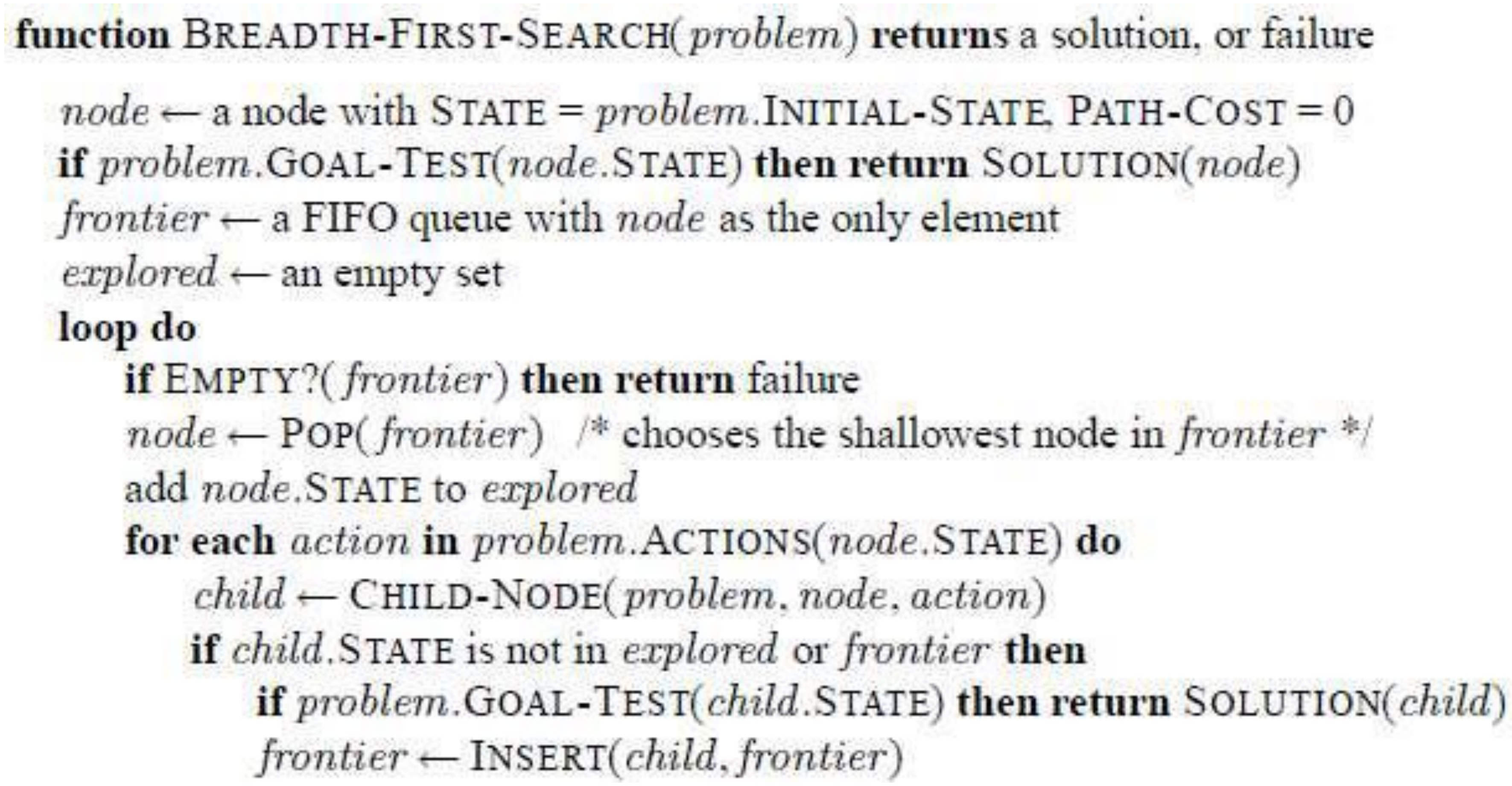

A
B
C
D
E
F
G


A
B
C
D
E
F
G


A
B
C
D
E
F
G


A
B
C
D
E
F
G

Breadth First Search on Graph



```
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)
```

Working Detail:

- We use the queue data structure to implement BFS.
    1. Add the initial state to a queue.
    2. While the queue is not empty, dequeue the first node.
    3. If the node is the goal state, return it.
    4. If the node is not the goal state, add all its neighbors to the end of the queue.
    5. Repeat steps 2-4 until the goal state is found or the queue is empty.

Analysis of BFS

- Complete?

  Yes
- Time?

  $b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space?

  $O(b^d)$
- Optimal?

  Yes if step costs are all identical or path cost is a non decreasing function of the depth of the node

- **Space** is the bigger problem (more than time)
- **Time** requirement is still a major factor

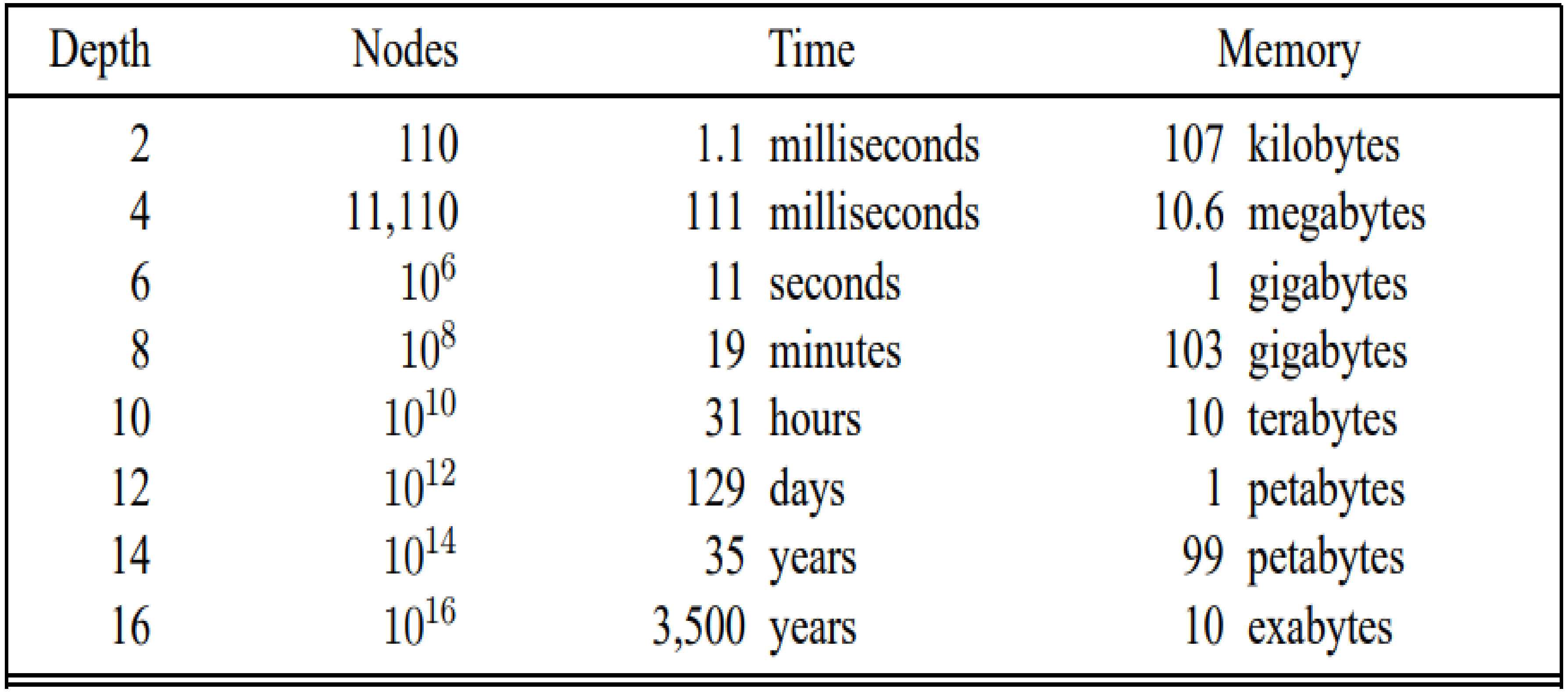
How bad a BFS?

- An exponential complexity bound such as O(b^{d}) is scary.
- It lists the time and memory required for a breadth-first search with branching factor b = 10 for various values of the solution depth d.
- The table assumes that **100,000 nodes can be generated per second** and that a node **requires 1000 bytes of storage.**
- Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer

Time and Memory Requirements for BFS

The numbers shown assume branching factor

- b = 10
- 100,000 nodes/second
- 1000 bytes/node



Depth	Nodes	Time	Memory
2	110	1.1 milliseconds	107 kilobytes
4	11,110	111 milliseconds	10.6 megabytes
6	10^6	11 seconds	1 gigabytes
8	10^8	19 minutes	103 gigabytes
10	10^{10}	31 hours	10 terabytes
12	10^{12}	129 days	1 petabytes
14	10^{14}	35 years	99 petabytes
16	10^{16}	3,500 years	10 exabytes

Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

Advantages

- **Completeness:**
  - BFS is guaranteed to find the goal state if it exists in the search space, provided the branching factor is finite.
- **Optimal solution:**
  - BFS is guaranteed to find the shortest path to the goal state, as it explores all nodes at the same depth before moving on to nodes at a deeper level.
- **Simplicity:**
  - BFS is easy to understand and implement, making it a good baseline algorithm for more complex search algorithms.
- **No redundant paths:**
  - BFS does not explore redundant paths because it explores all nodes at the same depth before moving on to deeper levels.

Disadvantages

- **Memory-intensive:**
  - BFS can be memory-intensive for large search spaces because it stores all the nodes at each level in the queue.
- **Time-intensive:**
  - BFS can be time-intensive for search spaces with a high branching factor because it needs to explore many nodes before finding the goal state.
- **Inefficient for deep search spaces**:
  - BFS can be inefficient for search spaces with a deep depth because it needs to explore all nodes at each depth before moving on to the next level.

Depth First Search

Expand deepest unexpanded node

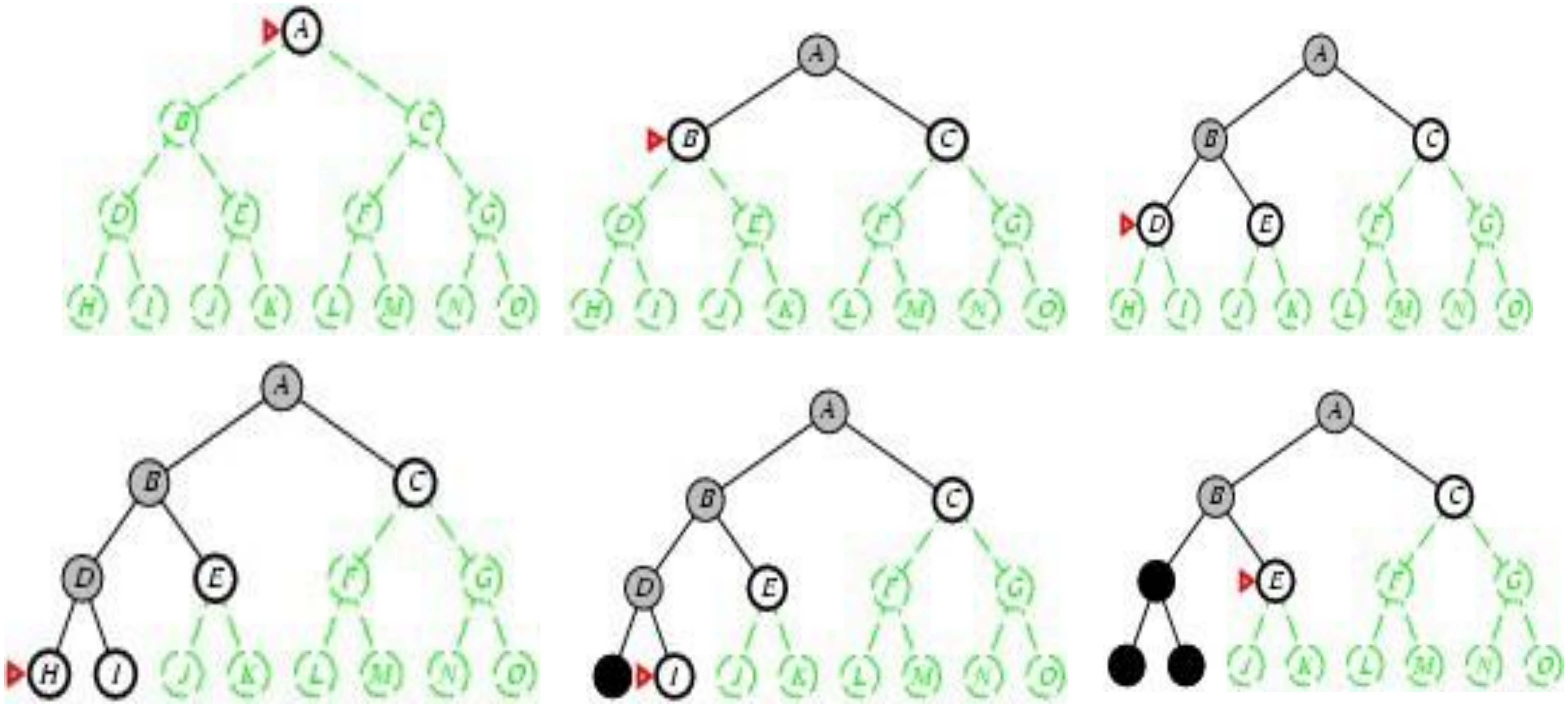
- Implementation:
  - frontier = LIFO queue, i.e., put successors at front
  - Or a recursive function

- **Properties of DFS**
  - Properties of DFS depend strongly on whether the graph-search or tree-search version is used

Overview:

- **Traversal Order:**
  - DFS starts at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking. It goes deep before going wide.
- **Data Strctures:**
  - DFS uses a **stack** (either explicitly or via recursion) to remember which vertices to visit next.
- **Completeness**
  - DFS is generally not complete for infinite graphs or graphs with loops, as it can get stuck in a loop.
- **Optimality**
  - DFS does not guarantee that it will find the shortest path to a goal.

Example:



Analysis of DFS with Tree

- Complete?
  - No: fails in infinite-depth spaces, or spaces with loops
  - Modify to avoid repeated states along path
    - complete in finite spaces
- Time?
  - $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space?
  - $O(bm)$, i.e., linear space!
- Optimal?
  - No

Analysis of DFS with Graph

- Complete?
  - No: also fails in infinite-depth spaces
  - Yes: for finite state spaces
- Time?
  - $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space?
  - Not linear any more, because of explored set
- Optimal?
  - No

Backtracking

- Backtracking search is a variant of DFS
- Only one successor is generated at a time rather than all successors
- Each partially expanded node remembers which successor to generate next

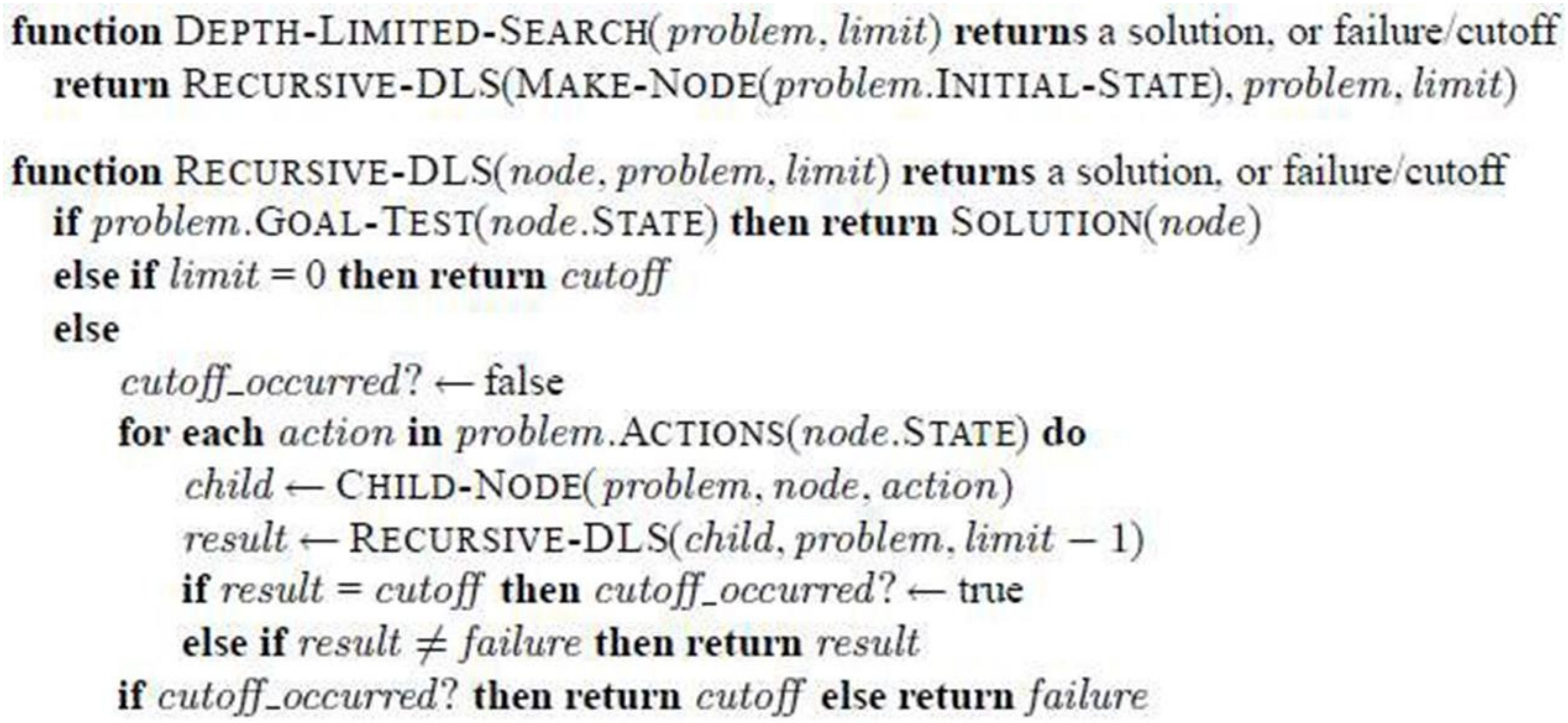
- Memory requirement: O(m) vs. O(bm)

Depth Limited Search

Depth-limited search (DLS) is a variant of depth-first search (DFS) that limits the depth of the search tree to a certain level.

- DLS is commonly used in artificial intelligence (AI) to explore a **search space efficiently** while **avoiding the problems of infinite loops** and **memory overflow** that can arise in regular DFS.

Depth Limited Algorithm with Limit 1



```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if $limit = 0$ then return cutoff
    else
        cutoff_occurred? $\leftarrow$ false
        for each action in problem.ACTIONS(node.STATE) do
            child $\leftarrow$ CHILD-NODE(problem, node, action)
            result $\leftarrow$ RECURSIVE-DLS(child, problem, $limit - 1$)
            if $result = cutoff$ then cutoff_occurred? $\leftarrow$ true
            else if $result \neq failure$ then return result
        if cutoff_occurred? then return cutoff else return failure
```

Working:

1. **Initialize the search:**

   Start by setting the initial depth to 0 and initialize an empty stack to hold the nodes to be explored. Enqueue the root node to the stack.

2. **Explore the next node:**

   Dequeue the next node from the stack and increment the current depth.

3. **Check if the node is a goal:**

   If the node is in a goal state, terminate the search and return the solution.

4. **Check if the node is at the maximum depth:**

   If the node's depth is equal to the maximum depth allowed for the search, do not explore its children and remove it from the stack.

5. **Expand the node:**

   If the node's depth is less than the maximum depth, generate its children and enqueue them to the stack.

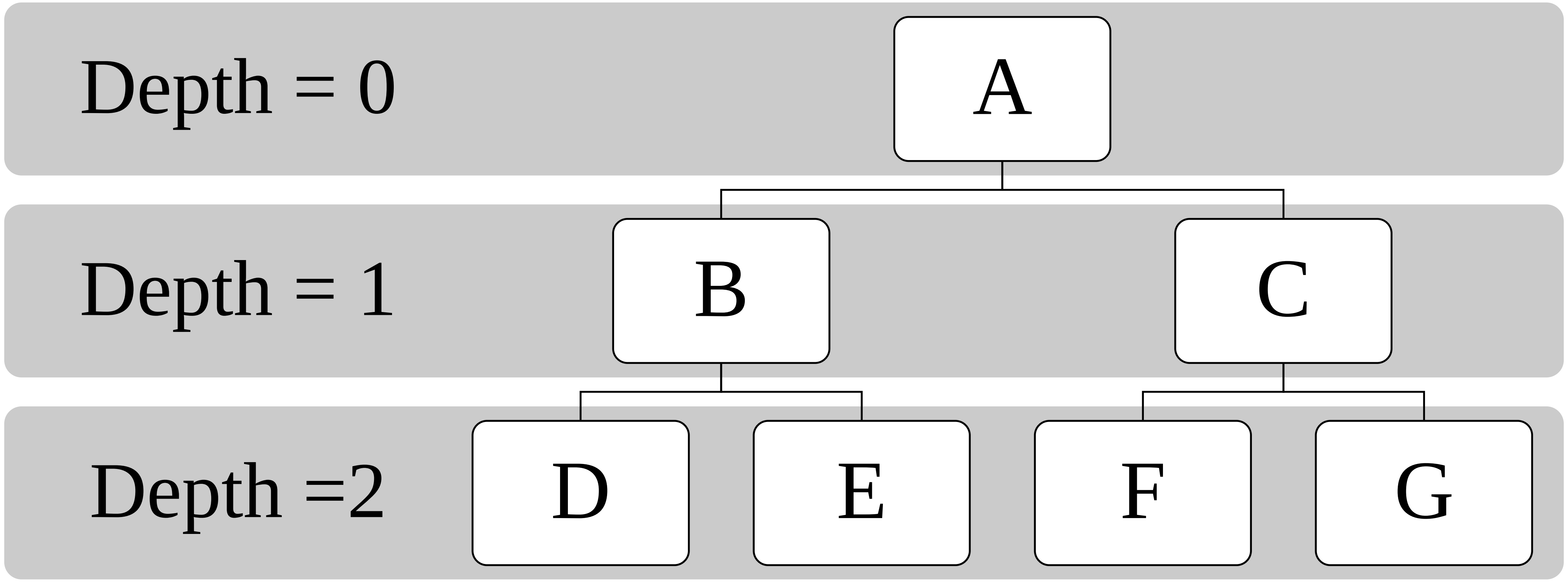
6. **Repeat the process:**

   Go back to step 2 and repeat the process until either a goal state is found or all nodes within the maximum depth have been explored.

Analysis

- Incompleteness:
  - The DEPTH-LIMITED SEARCH depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is not unlikely when d is unknown.)
- Non Optimal:
  - Depth-limited search will also be nonoptimal if we choose $\ell > d$. Its time complexity is $O(b^{\ell})$ and its space complexity is $O(b^{\ell})$. Depthfirst search can be viewed as a special case of depth-limited search with $\ell = \infty$.

Example: find the node with the value "G" using a Depth-Limited Search with a depth limit of 2.

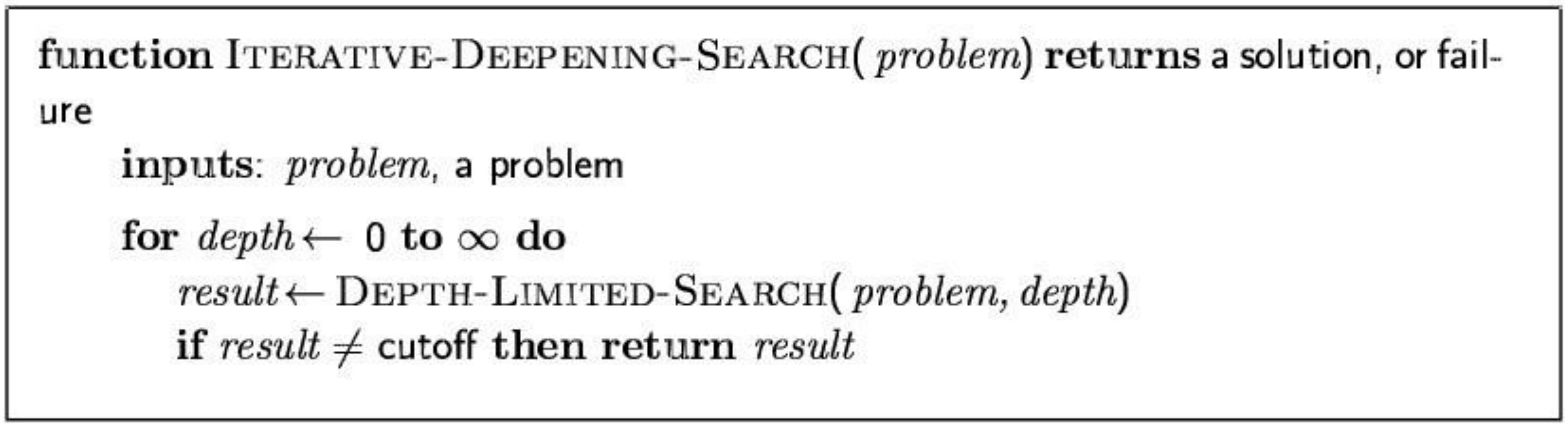


Solution:

- Starting at the root node "A," we can expand its children "B" and "C." Since we have a depth limit of 2, we cannot expand any further, and we backtrack to the parent node "A."
- Next, we expand the other child of "A," which is "C." We can expand its children "F" and "G." Since the node with the value "G" is found at a depth of 2, we stop the search and return the node with the value "G".
- Therefore, the output of the Depth-Limited Search with a depth limit of 2 in this example is the node with the value "G."
- Note that if the depth limit was set to 1, the search would not have found the node with the value "G" as it would have stopped at a depth of 1 without exploring the subtree rooted at "G."

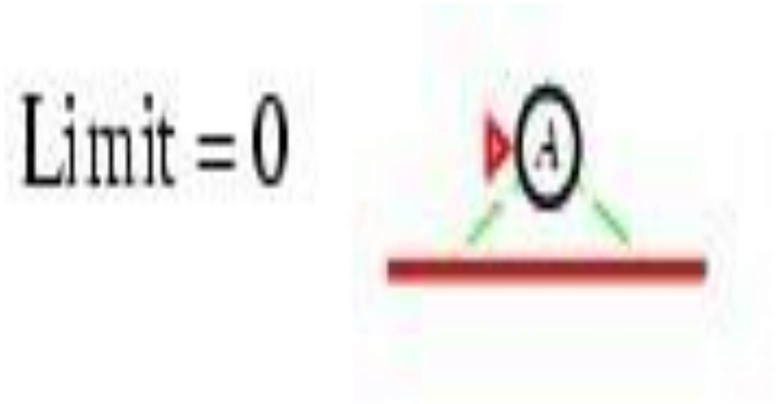
Iterative Deepening DFS Search

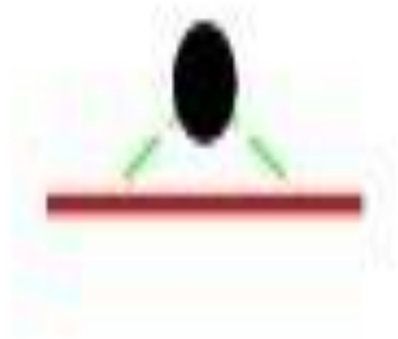
- Gradually increase the depth limit until a goal is found or all the nodes are visited
- Combines the benefits of depth-first and breadth- first search



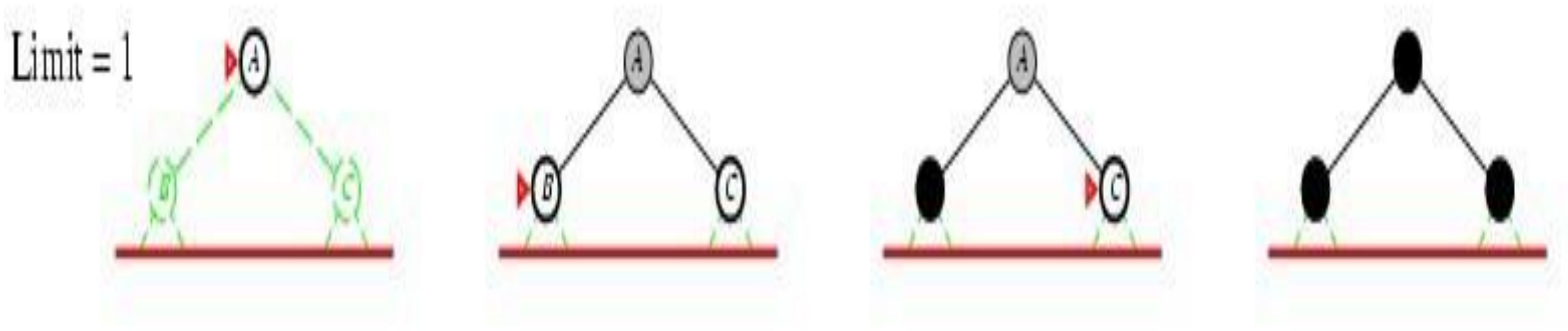
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Depth Limit 0

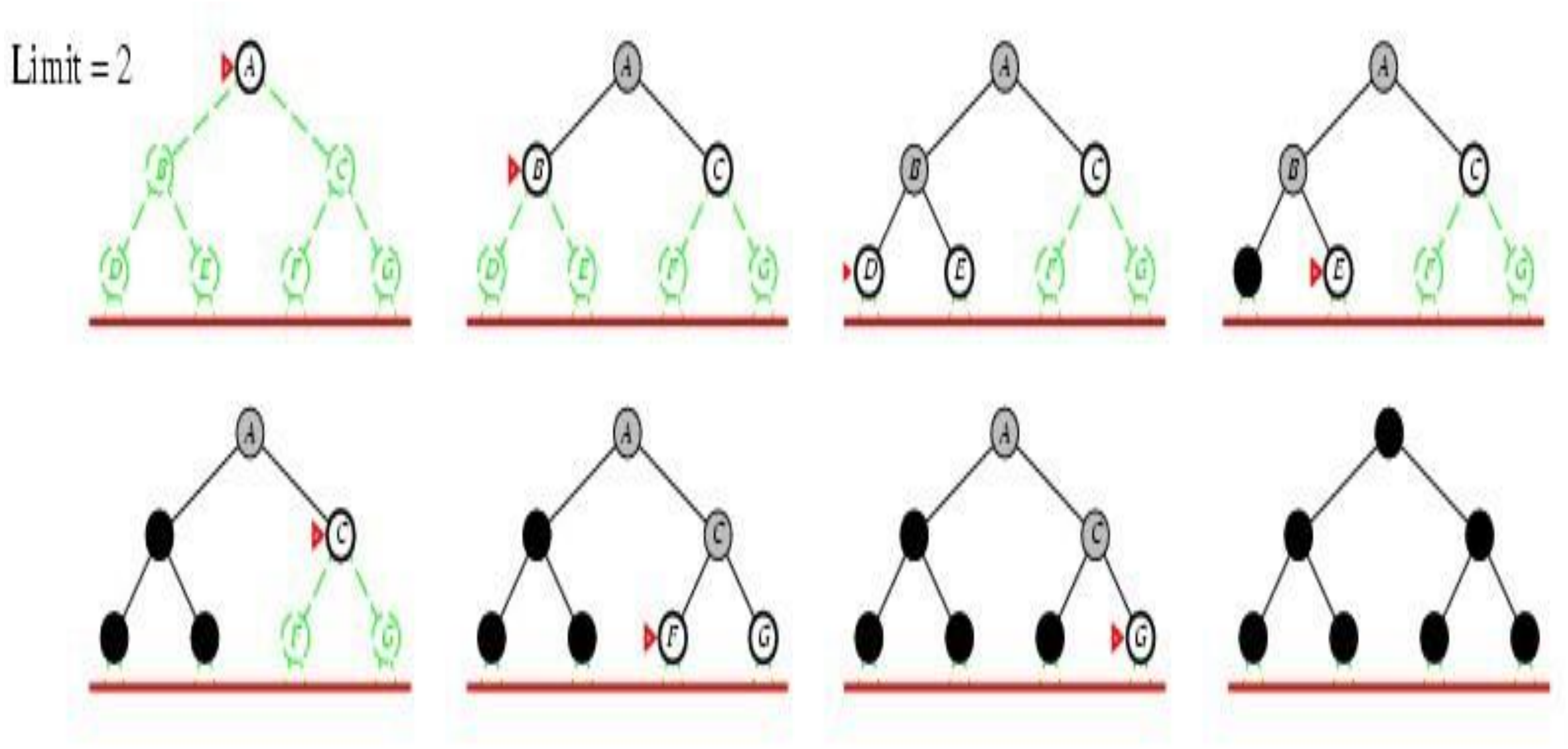




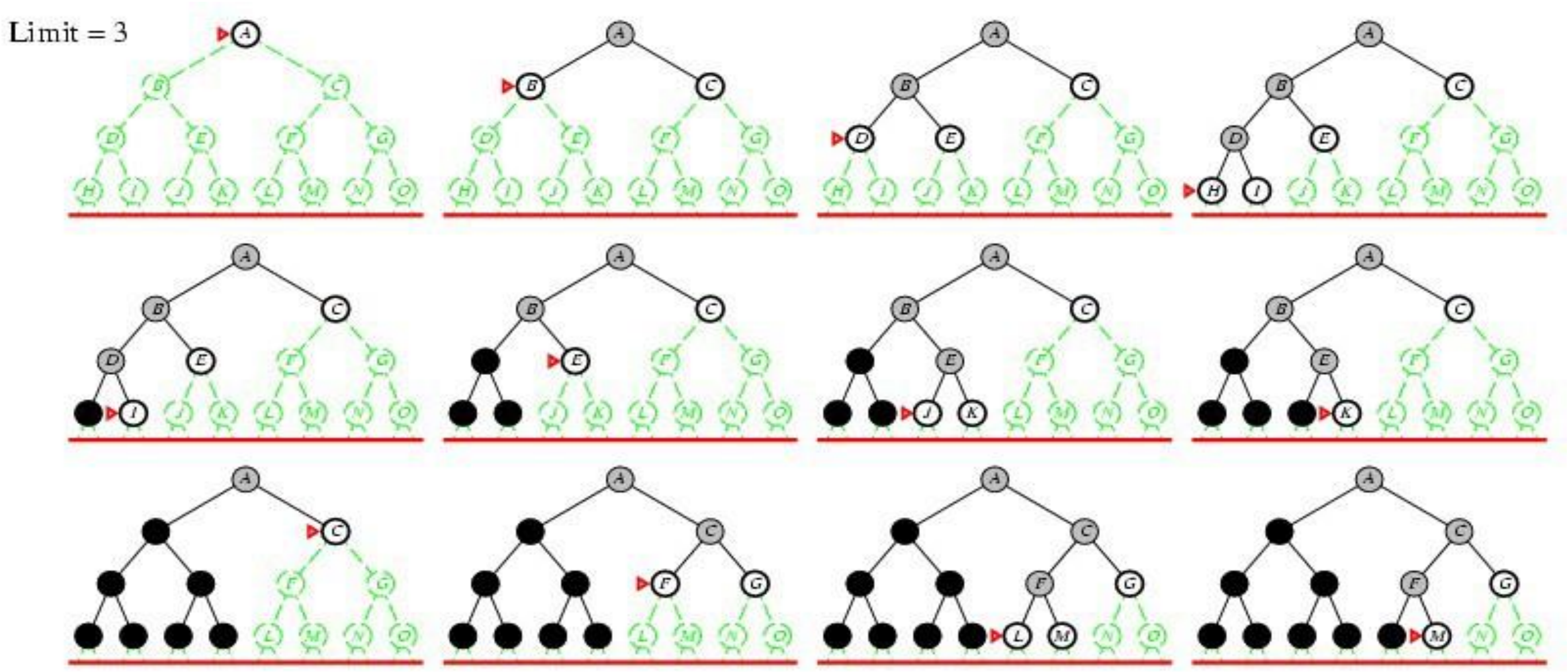
Depth Limit 1



Depth Limit 2



Depth Limit 3



Analysis

- Complete?
  - Yes
- Time?
  - $d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$ (tree search version)
- Optimal?
  - Yes, if step costs are identical or path cost is a nondecreasing function of the depth of the node

Uniform Search Algorithm

Expand least-cost unexpanded node

- Implementation:
- Frontier = priority queue ordered by path cost g(n)

- Example, shown on board, from Sibiu to Bucharest

Uniform search Algorithm

- The uniform cost search expands the node n with the lowest path cost g(n)

UCS guarantees to find the optimal solution to the problem provided that the cost function is non-negative.

- BFS vs UCS
- Uniform search has two significant differences from BFS

1. The goal test is applied to a **nodes when it is selected for expansion rather than when it is first generated**
2. The goal test is added in case **s** to a node currently on the forntier.

Uniform search Algorithm (Cont…)

- **Principle:**
  - UCS explores paths in the **increasing order of cost**. It prioritizes nodes based on the **cumulative cost from the start node to the current node, ensuring that the paths with the lowest total cost are explored first**.
- **Data Structure:**
  - UCS typically **uses a priority queue** to keep track of nodes, where the nodes are prioritized by their path cost from the start node.
- **Optimality:**
  - UCS is guaranteed to find the optimal solution if the cost function satisfies the condition of non-negativity. It will explore all paths with cost less than the cost of the optimal path to ensure that the solution found is indeed the best one.

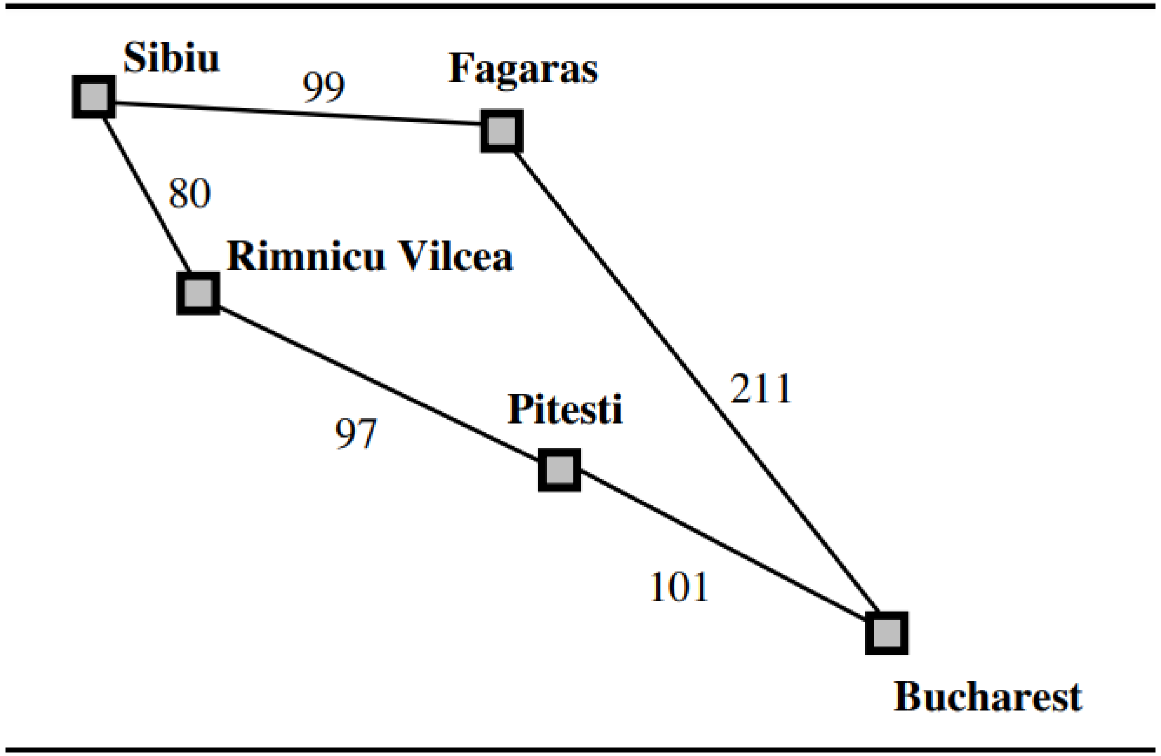
Uniform search Algorithm (Cont…)

- **Completeness:**
  - UCS is complete, meaning that if a solution exists, UCS will find it.
- **Time and Space Complexity:**
  - The time and space complexity of UCS can be high, depending on the branching factor and the depth of the optimal solution. It can be inefficient if there are many paths with similar costs.
- **Usage:**
  - UCS is often used in scenarios where the path cost is a critical factor, and the goal is to find the least costly path without any heuristic information about the goal's location.

Example 1:

The successors of **Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99 r**espectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding **Pitesti with cost 80+97 = 177.** The least-cost node is now Fagaras, so it is expanded, adding **Bucharest with cost 99+211 = 310**. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost **80+97+101 = 278**

Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

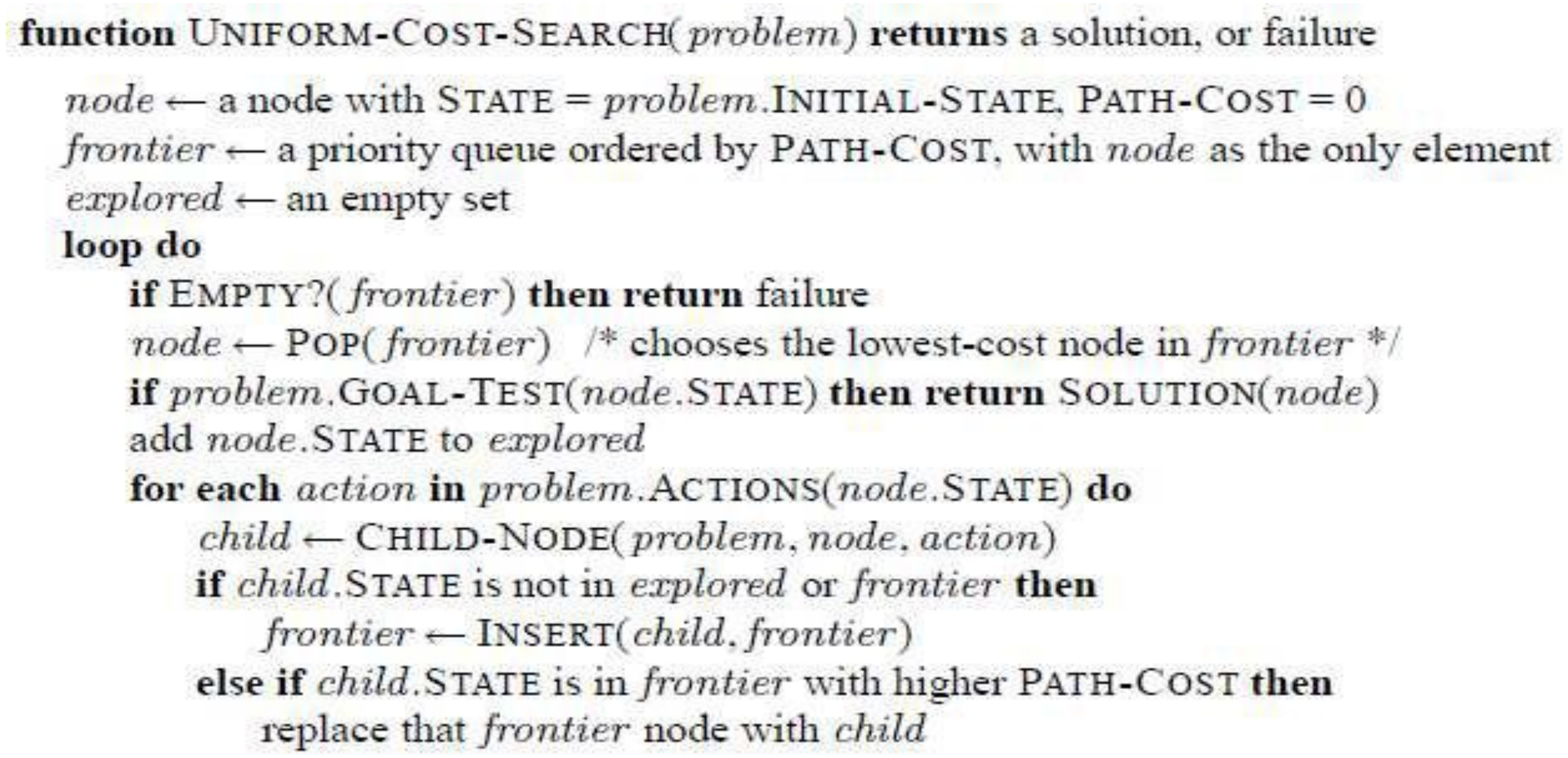


A portion of the Romania state space, selected to illustrate uniform-cost search.

Uniform Cost Search (Cont…)

- Almost equivalent to breadth first
  - If step costs are all equal
  - Except the **BFS stops as soon as it generate goal**, whereas Uniform search **examine all the nodes at the goal depth** to see if one has a lower cost.
- It is implemented by **queue order by path cost rather than depth**
- Therefore, its complexity cannot be easily characterized in term of **b and d.**

Uniform Cost Search 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
```

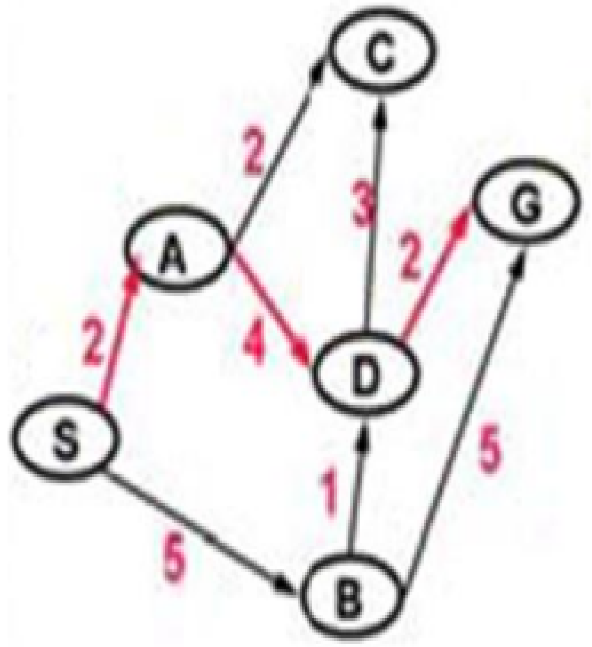
Working:

- Initialize the frontier with the start node, and set the cost of the start node to zero. The frontier in a search algorithm contains states that have been seen, but their outgoing edges have not yet been explored
- While the frontier is not empty, select the node with the lowest cost and remove it from the frontier.
- If the selected node is the goal node, return the solution.
- Otherwise, expand the selected node and add its children to the frontier with the corresponding path cost.
- If a child node is already in the frontier with a lower path cost, update its cost with the lower value.
- Repeat steps 2-5 until the goal node is found or the frontier is empty.

Example 2:

- Pick the best path
- Add extension in Q

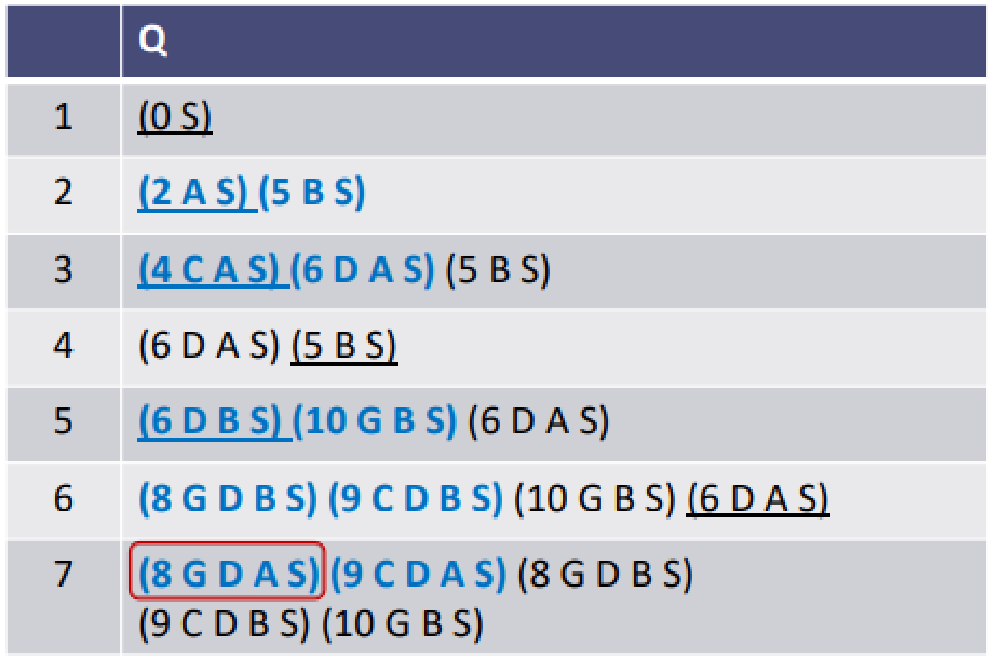
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)



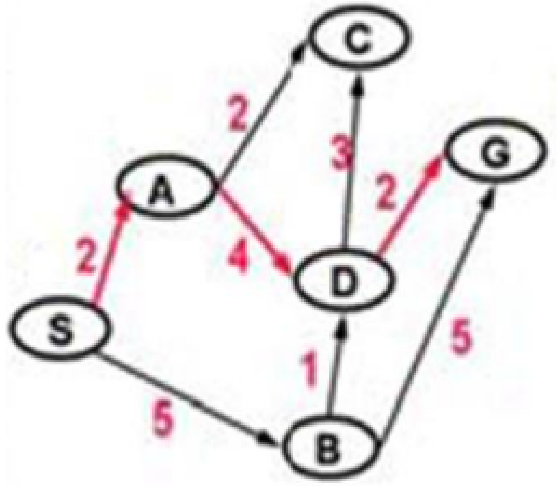
- Blue color represent the chosen path
- Underline path are chosen from extension

Example 2:

- Pick the best path
- Add extension in Q



	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



- **Blue color represent** the chosen path
- **Underline path** are chosen from extension

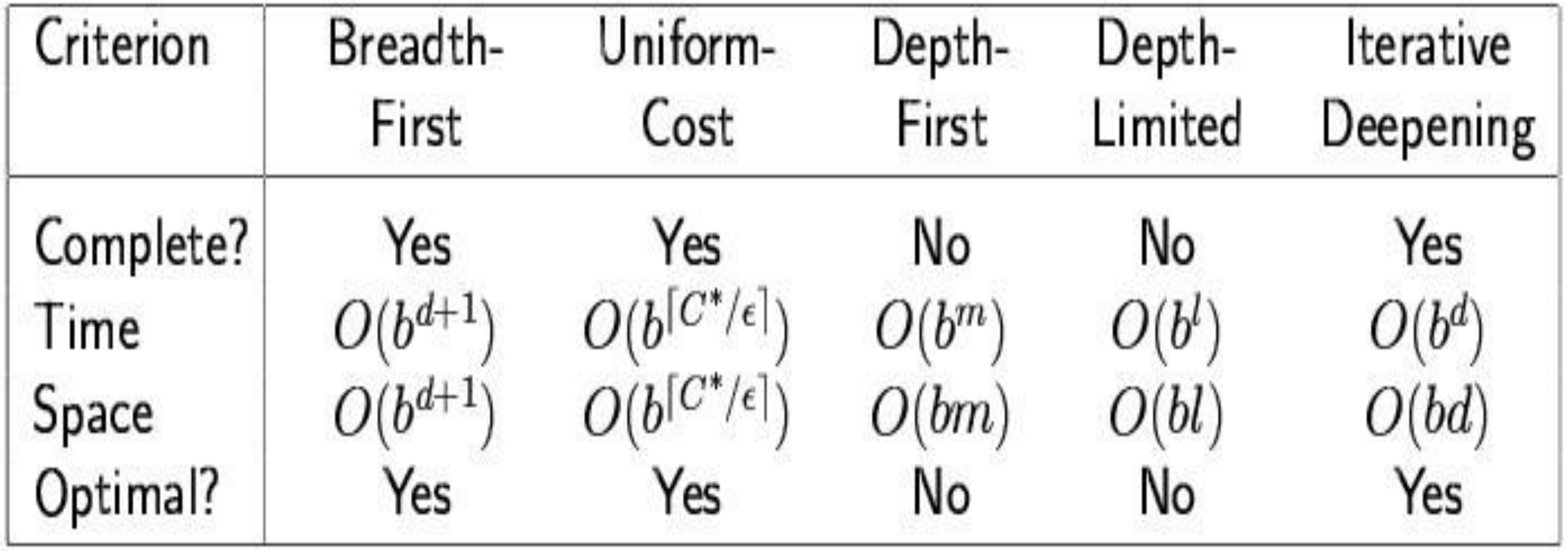
Analysis:

- Complete?
  - Yes,
- Time?
  - $O(b^{1+\text{ceiling}(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution, if step cost $\geq \varepsilon$, ε cost of every action.
  - $O(b^{d+1})$ when all steps cost are equal
  - # of nodes with $g \leq$ cost of optimal solution
- Space?

  $O(b^{\text{ceiling}(C^*/\varepsilon)})$

  - # of nodes with $g \leq$ cost of optimal solution
- Optimal?
  - Yes – nodes expanded in increasing order of $g(n)$

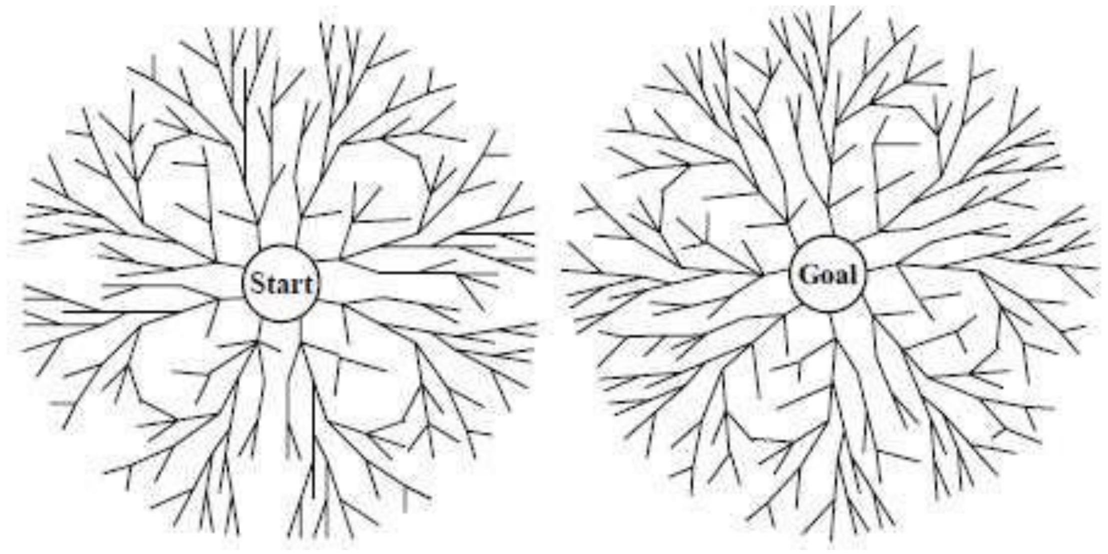
Summary of Uninformed Search Stargtegy



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Bidirectional Search Strategy

- Runs two simultaneous searches
  - Forward from initial state
  - Backward from goal state



Working:

- Two Frontiers:
  - The algorithm maintains two frontiers, one for the forward search from the start and one for the backward search from the goal.
- Meeting Point:
  - The search continues until there is a common node that appears in both frontiers. This common node is the meeting point, and the path from the start to the goal can be constructed by concatenating the paths from the start to the meeting point and from the meeting point to the goal.
- Efficiency:
  - By searching from both directions, the algorithm can significantly reduce the search space, often leading to faster search times compared to unidirectional methods like BFS or DFS.
- Completeness:
  - Bidirectional Search is complete, meaning it will find a solution if one exists.
- Optimality:
  - If both searches are done using BFS (or a variant that ensures the shortest path), then the solution found will be optimal.