Searching Techniques in Problem Solving

CSE 3207 – Artificial Intelligence

Md. Nasif Osman Khansur
Lecturer
Dept of CSE, RUET

Problem Solving – Searching

- AI uses certain problem-solving techniques to give the best optimal result quickly.
- Problem-solving refers to the ability of an AI system to find solutions to complex tasks, challenges, or puzzles using various techniques and algorithms.
- Search techniques are universal problem-solving methods.
- A search problem consists of:
  - **A State Space**. Set of all possible states where you can be.
  - **A Start State**. The state from where the search begins.
  - **A Goal State**. A function that looks at the current state returns whether or not it is the goal state.

Problem Solving – Searching

- A tree representation of search problem is called **Search tree**. The root node corresponds to the initial state.
- Path Cost is a function which assigns a numeric cost to each path.
- **Solution**: An action sequence which leads from the start node to the goal node.
- **Optimal Solution**: If a solution has the lowest cost among all solutions.

Properties of Searching

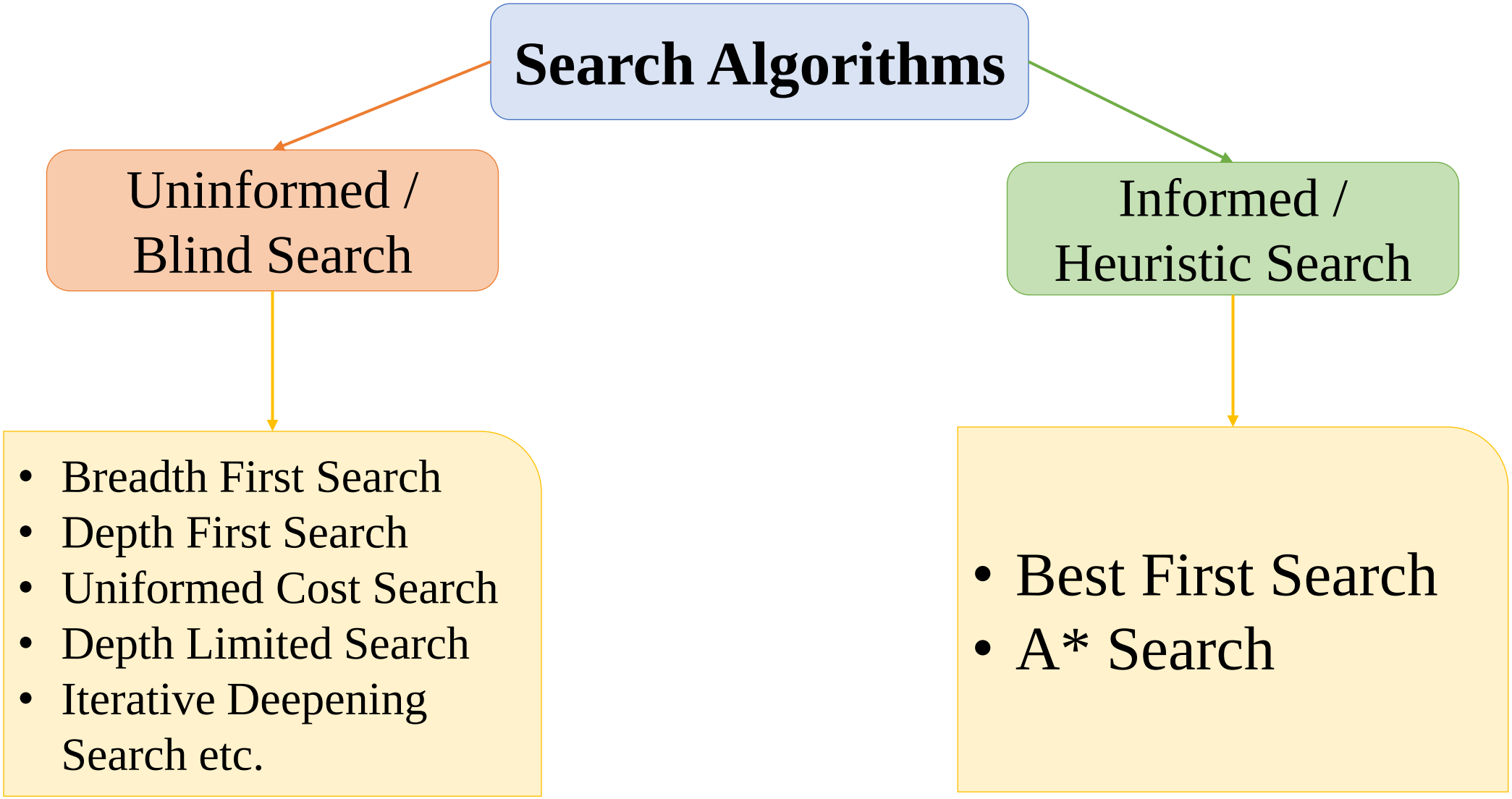
- **Completeness**: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

- **Optimality**: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

- **Time Complexity**: Time complexity is a measure of time for an algorithm to complete its task.

- **Space Complexity**: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of Search Algorithms



Uninformed Search

- Also known as Blind Search.
- They do not contain any domain knowledge such as closeness, the location of the goal.
- A brute-force way: only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- They do not have a heuristic function to guide the search towards the goal.

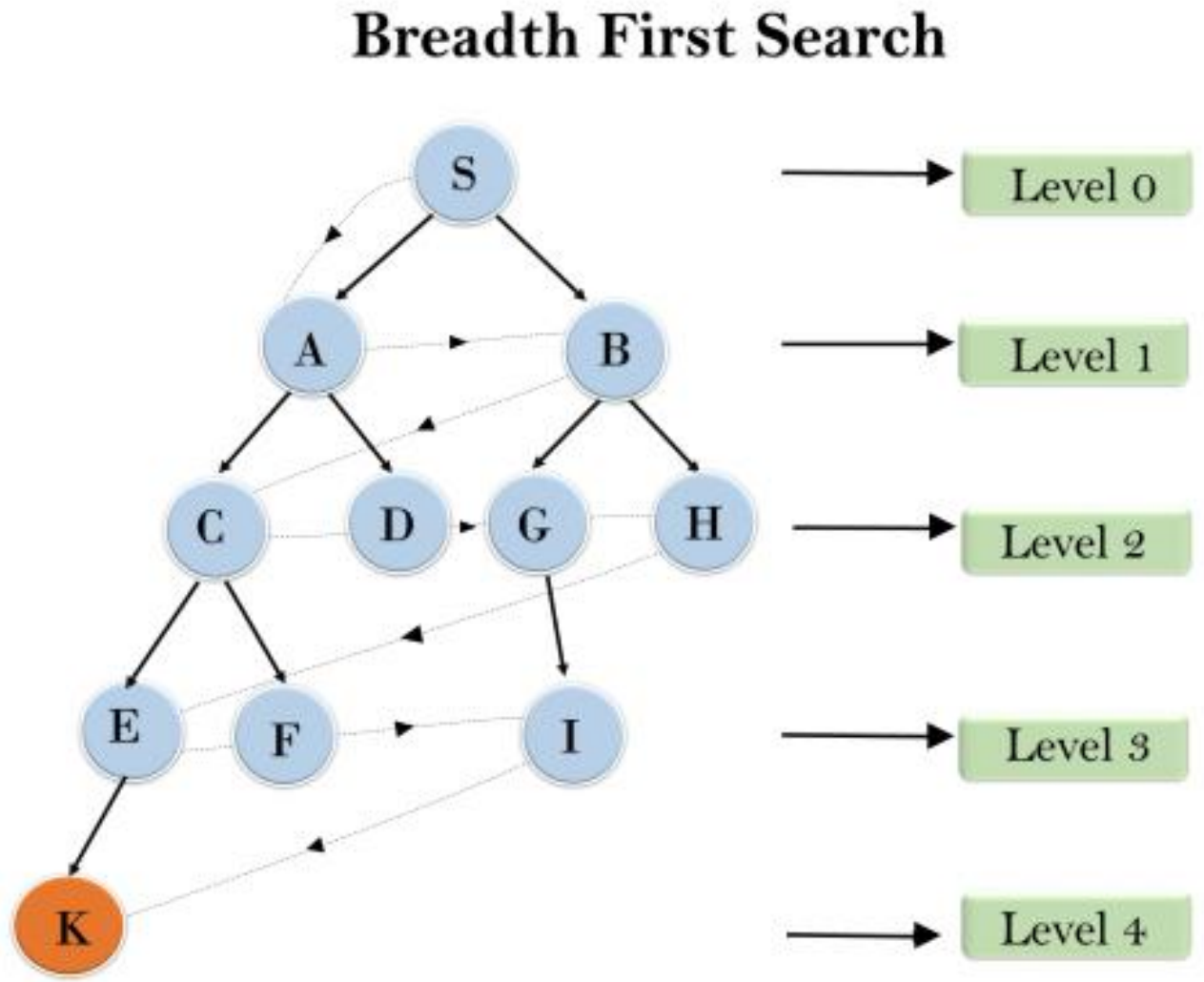
Informed Search

- Also known as Heuristic Search.
- They use a heuristic function h(n) that provides an estimate of the minimal cost from node n to the goal.
- Heuristic function helps the algorithm to prioritize which nodes to explore first based on their potential to lead to an optimal solution.
- Not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Breadth First Search (BFS)

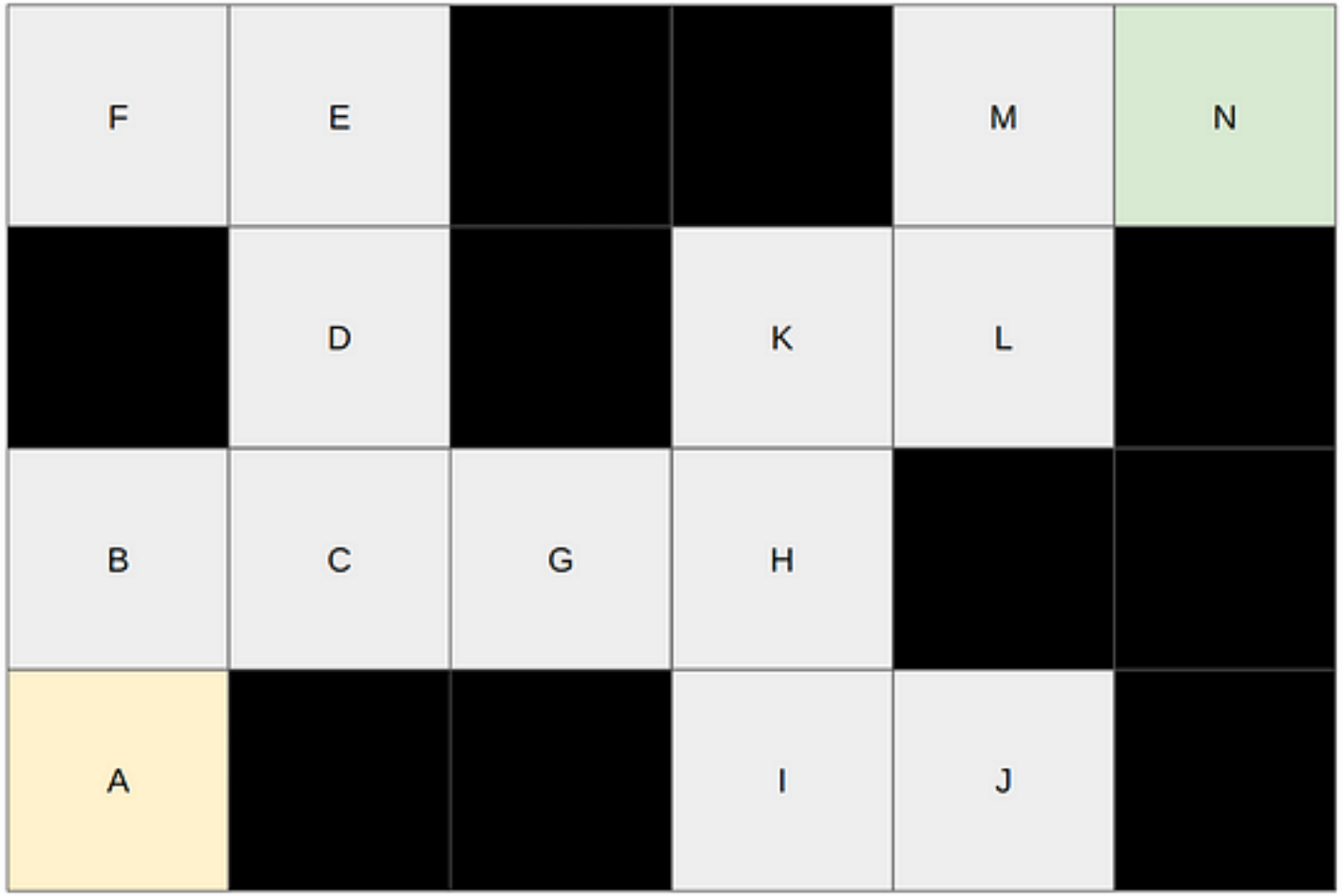
- Let's understand the traversing of a tree using BFS algorithm from the root node S to goal node K.
- BFS search algorithm traverse in layers.
- The traversed path will be:

  S---> A---> B----> C---> D---> G---> H---> E----> F----> I----> K
- Complete and Optimal.
- Complexity: O (b^d)



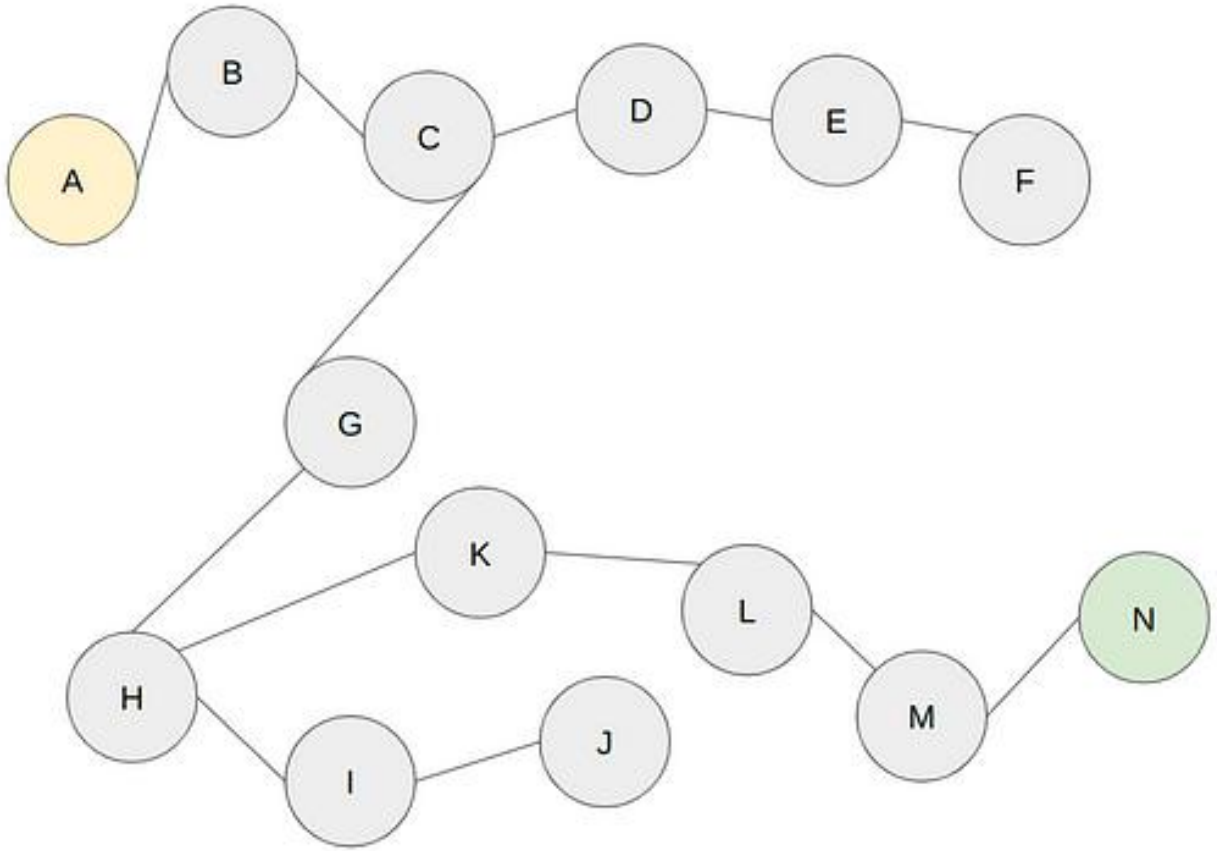
Breadth First Search (BFS)

Let's say we have the following Maze to be solved. What is the **solution** and **cost** of the above Maze problem if we want to move from **A** to **N**?



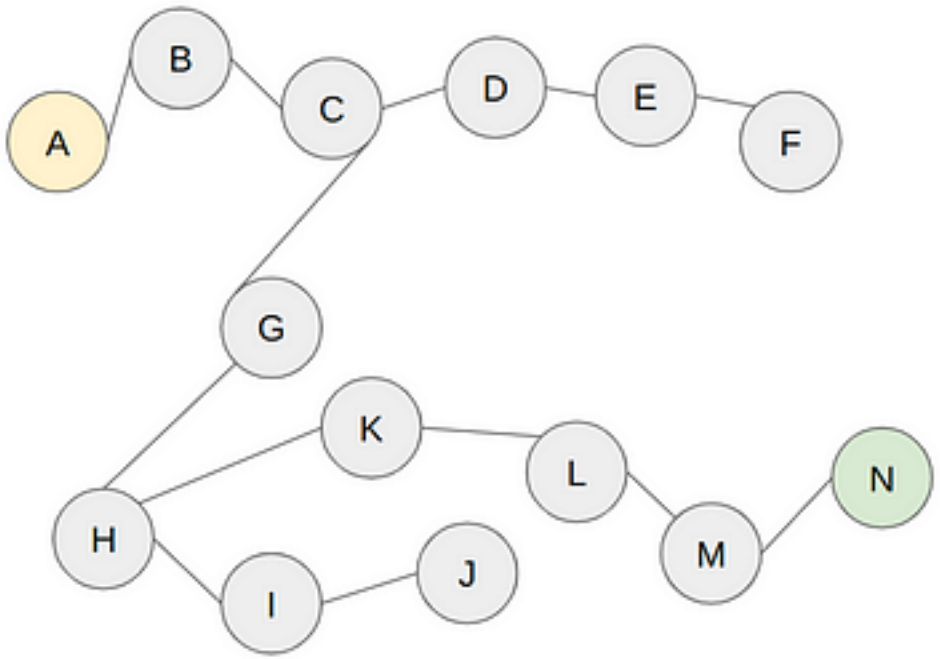
Breadth First Search (BFS)

To make the visualization easier, let’s transform the maze into a graph:



Breadth First Search (BFS)

Append A to the Frontier.



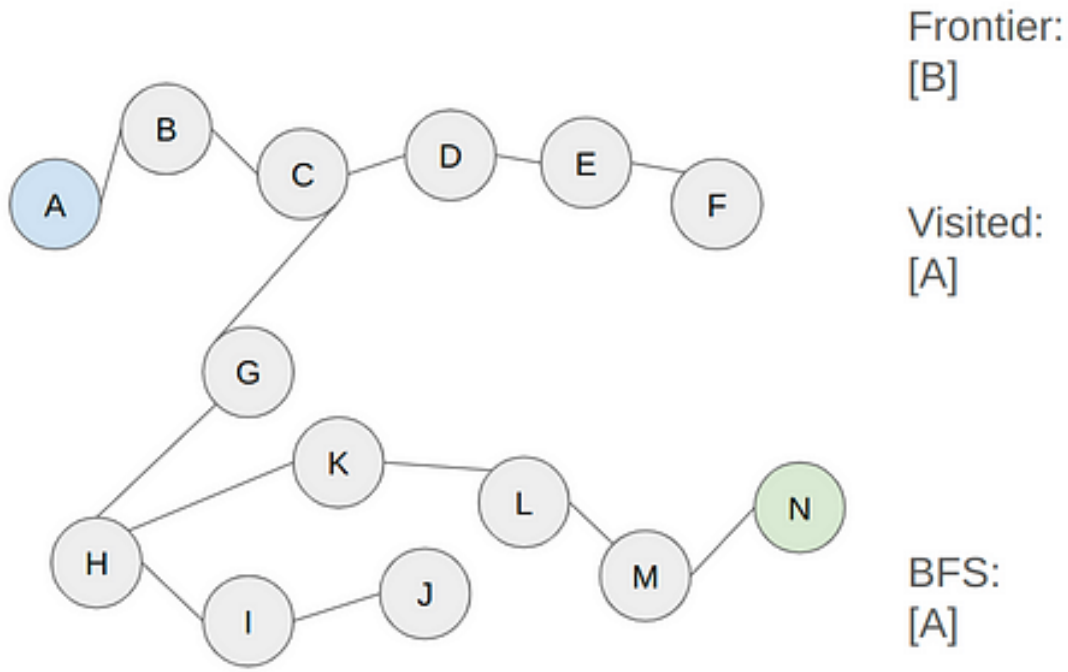
Frontier:
[A]

Visited:
[]

BFS:
[]

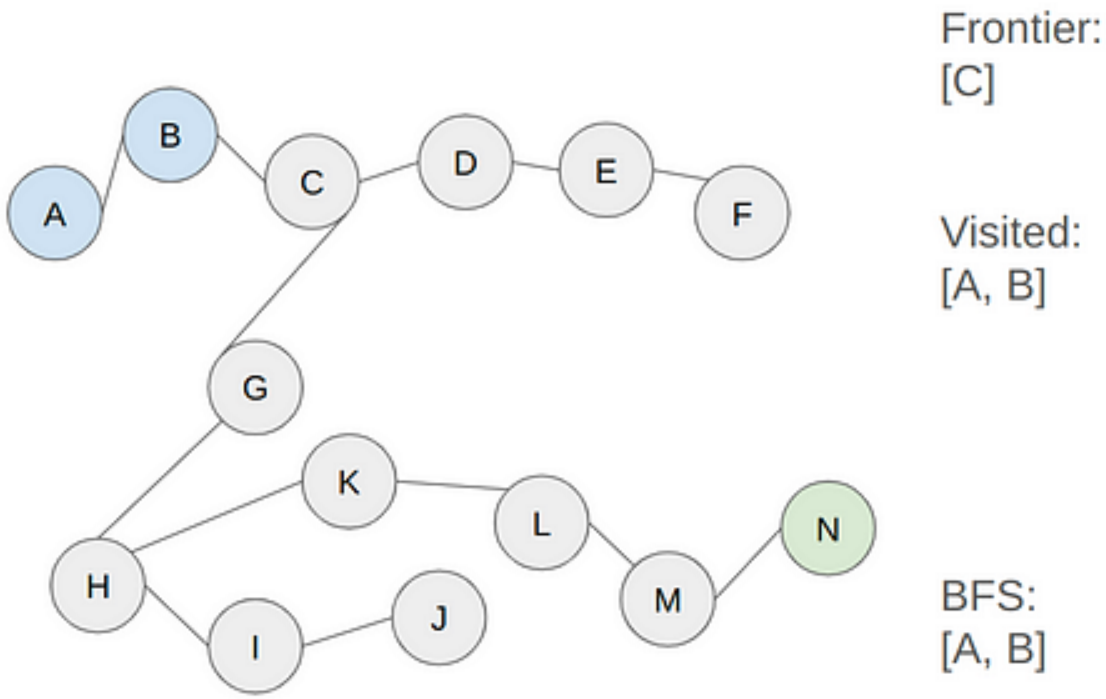
Breadth First Search (BFS)

Append A to the Visited and BFS, then append A's neighbor (B) to the Frontier. Finally, pop A from the Frontier.



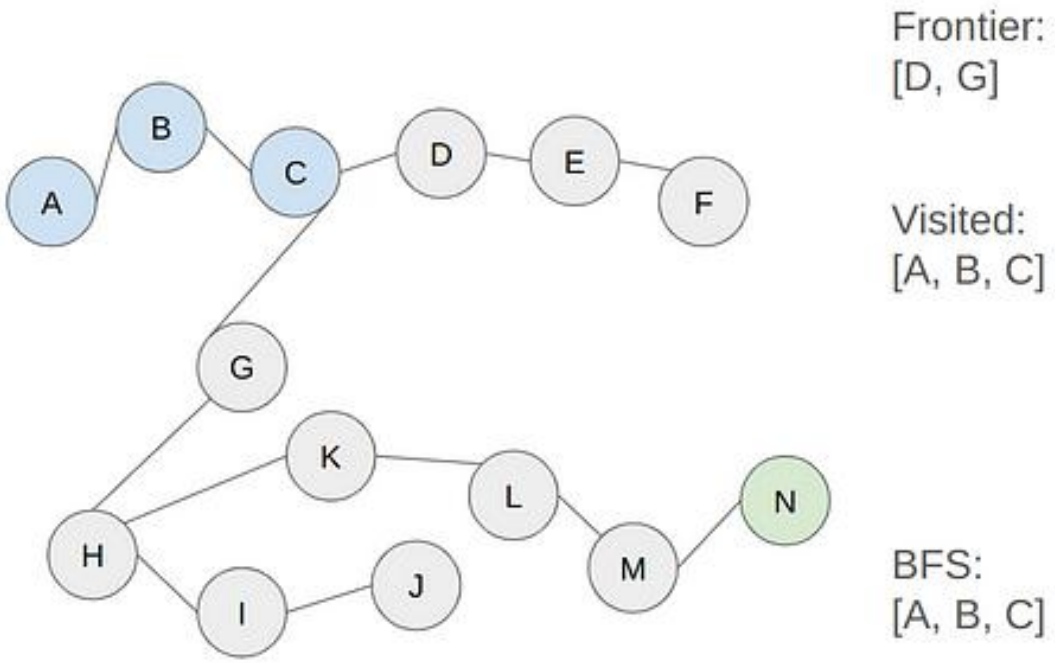
Breadth First Search (BFS)

Append B to the Visited and BFS, then append B’s neighbor (C) to the Frontier. Finally, pop B from the Frontier.



Breadth First Search (BFS)

Append C to the Visited and BFS, then append C’s neighbor (D, G) to the Frontier. Finally, pop C from the Frontier.



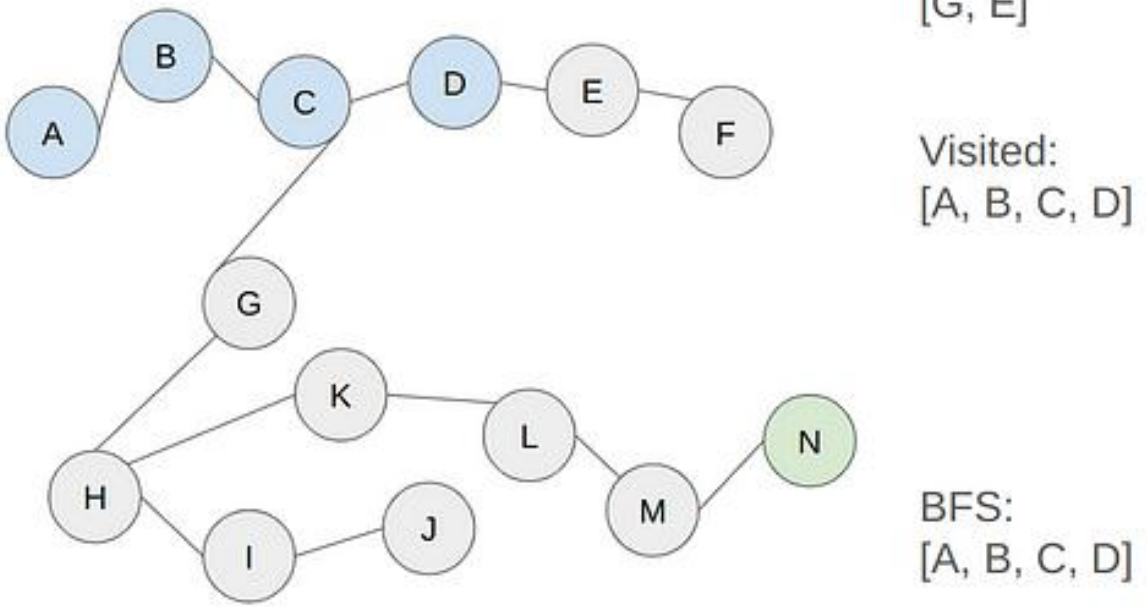
Frontier:
[D, G]

Visited:
[A, B, C]

BFS:
[A, B, C]

Breadth First Search (BFS)

Append D to the Visited and BFS, then append D’s neighbor (E) to the Frontier. Finally, pop D from the Frontier.



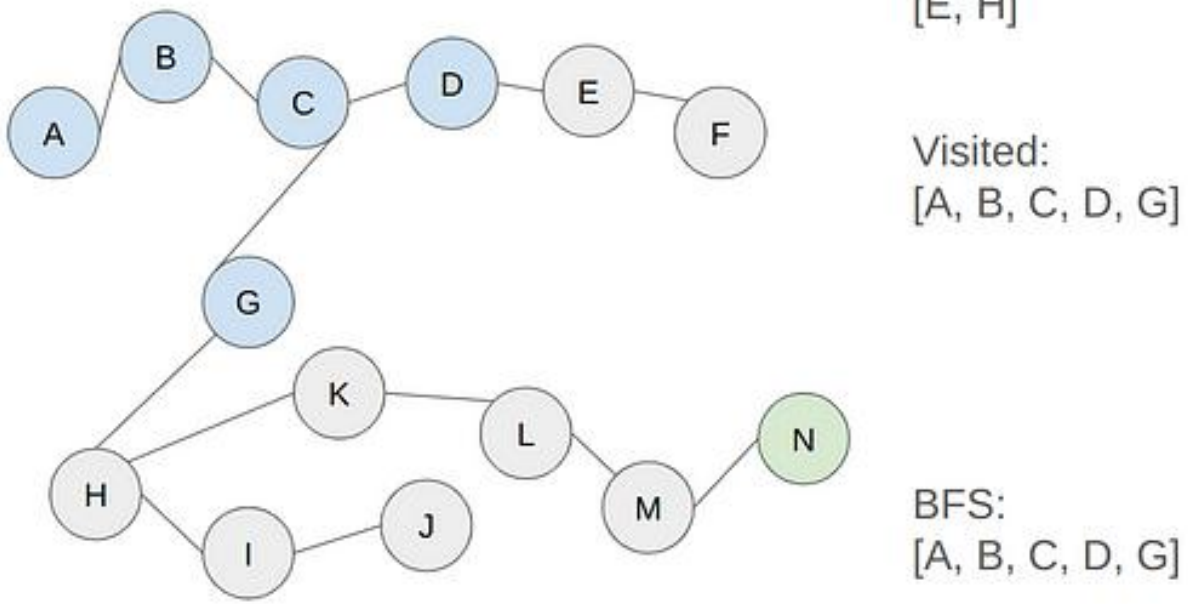
Frontier:
[G, E]

Visited:
[A, B, C, D]

BFS:
[A, B, C, D]

Breadth First Search (BFS)

Append G to the Visited and BFS, then append G's neighbor (H) to the Frontier. Finally, pop G from the Frontier.



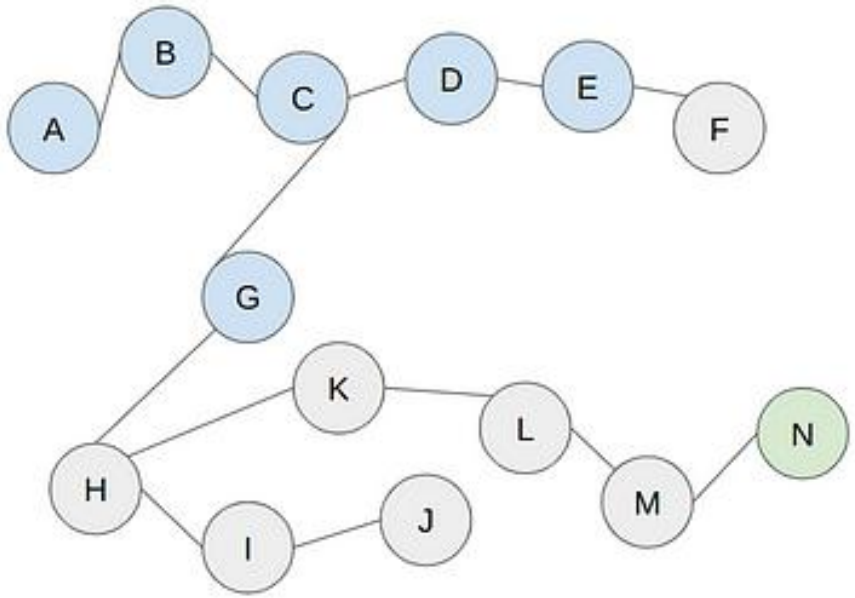
Frontier:
[E, H]

Visited:
[A, B, C, D, G]

BFS:
[A, B, C, D, G]

Breadth First Search (BFS)

Append E to the Visited and BFS, then append E’s neighbor (F) to the Frontier. Finally, pop E from the Frontier.



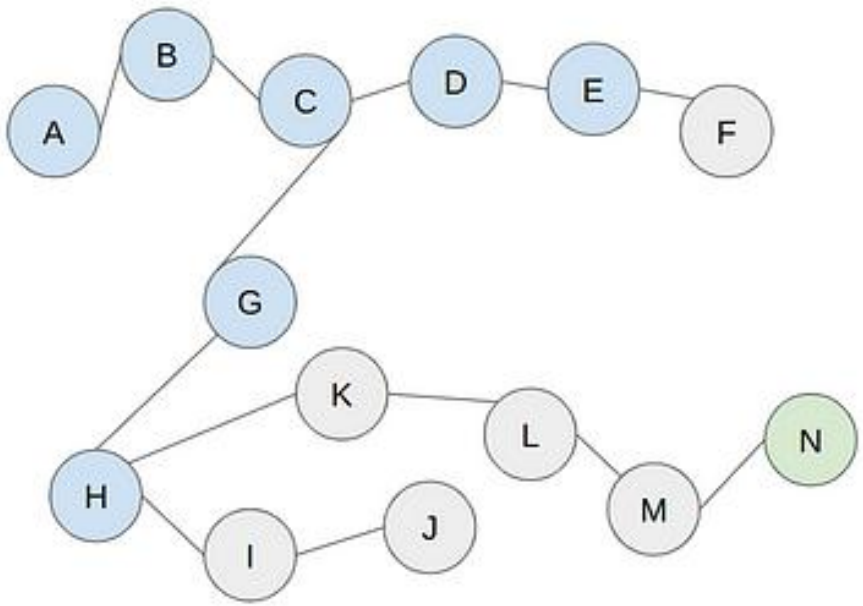
Frontier:
[H, F]

Visited:
[A, B, C, D, G, E]

BFS:
[A, B, C, D, G, E]

Breadth First Search (BFS)

Append H to the Visited and BFS, then append H's neighbor (K, I) to the Frontier. Finally, pop H from the Frontier.



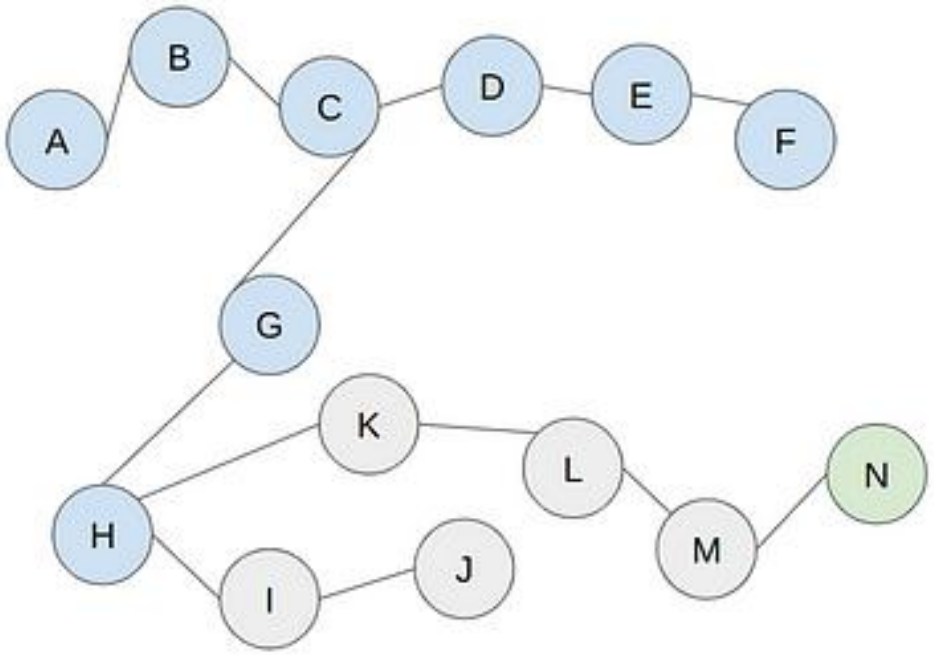
Frontier:
[F, K, I]

Visited:
[A, B, C, D, G, E, H]

BFS:
[A, B, C, D, G, E, H]

Breadth First Search (BFS)

Append F to the Visited and BFS. Finally, pop F from the Frontier.



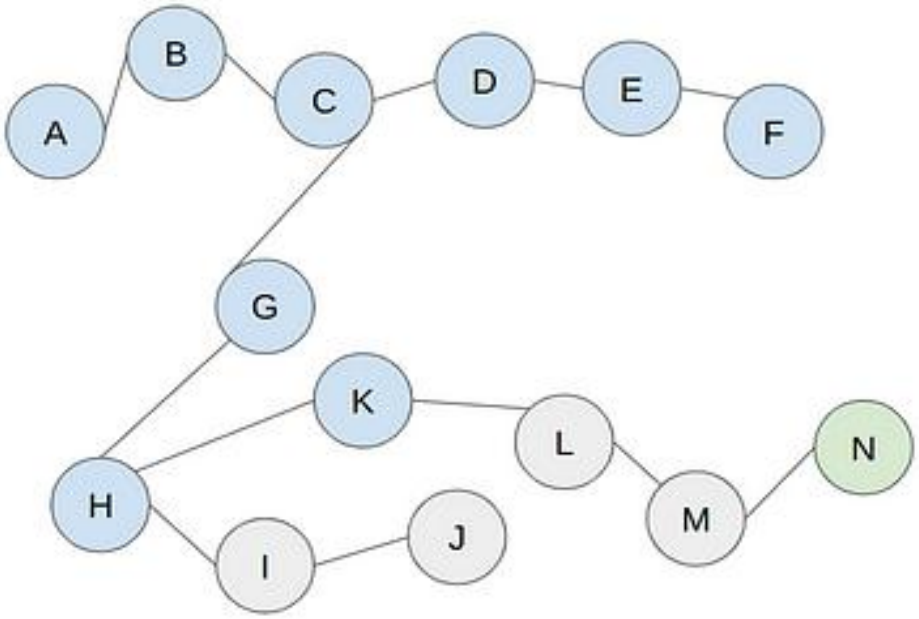
Frontier:
[K, I]

Visited:
[A, B, C, D, G, E, H, F]

BFS:
[A, B, C, D, G, E, H, F]

Breadth First Search (BFS)

Append K to the Visited and BFS, then append K's neighbor (L) to the Frontier. Finally, pop K from the Frontier.



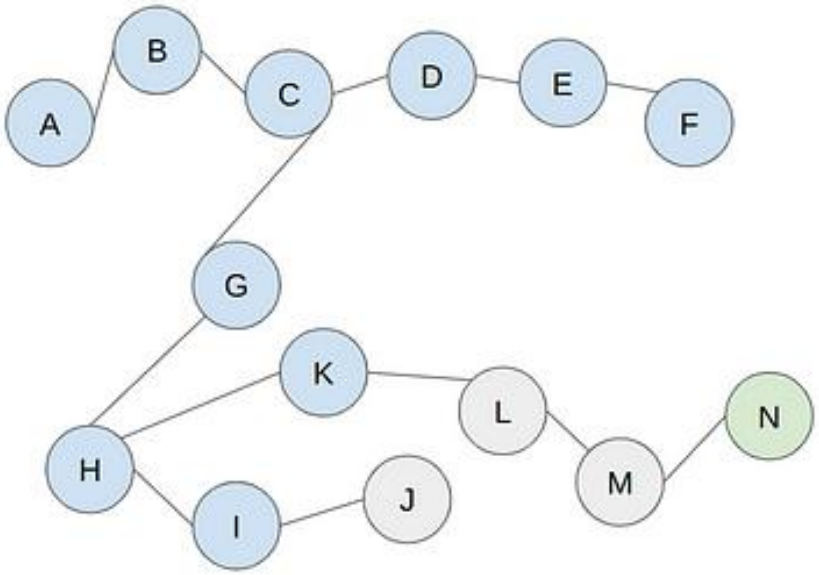
Frontier:
[I, L]

Visited:
[A, B, C, D, G, E, H, F, K]

BFS:
[A, B, C, D, G, E, H, F, K]

Breadth First Search (BFS)

Append I to the Visited and BFS, then append I's neighbor (J) to the Frontier. Finally, pop I from the Frontier.



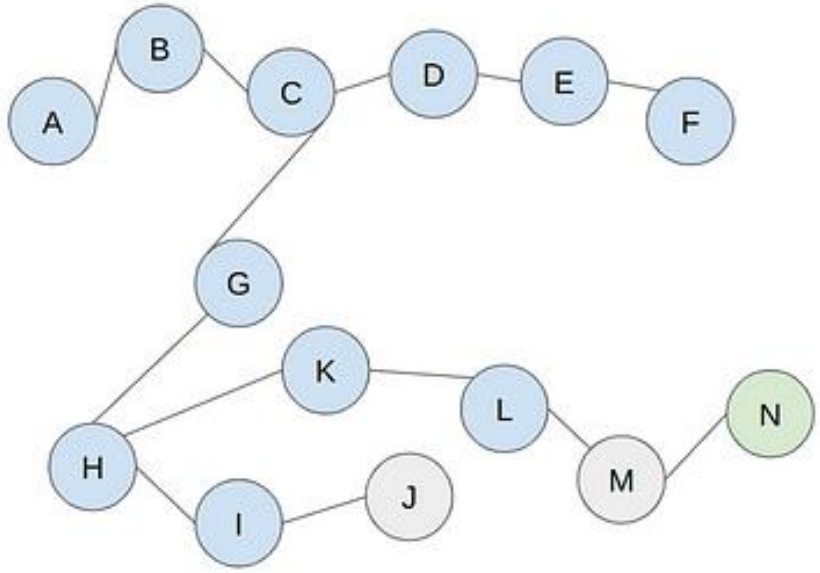
Frontier:
[L, J]

Visited:
[A, B, C, D, G, E, H, F, K, I]

BFS:
[A, B, C, D, G, E, H, F, K, I]

Breadth First Search (BFS)

Append L to the Visited and BFS, then append L's neighbor (M) to the Frontier. Finally, pop L from the Frontier.



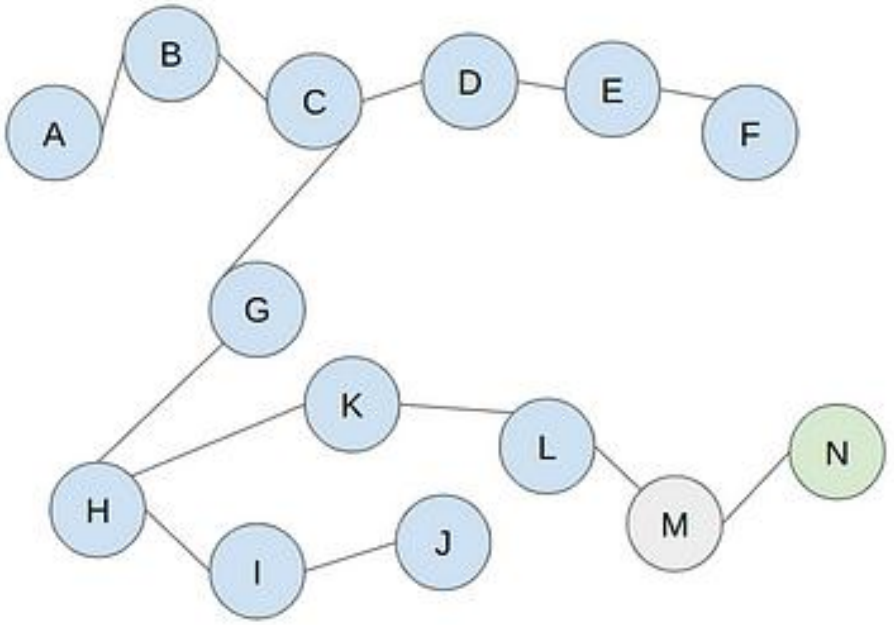
Frontier:
[J, M]

Visited:
[A, B, C, D, G, E, H, F, K, I, L]

BFS:
[A, B, C, D, G, E, H, F, K, I, L]

Breadth First Search (BFS)

Append J to the Visited and BFS. Finally, pop J from the Frontier.



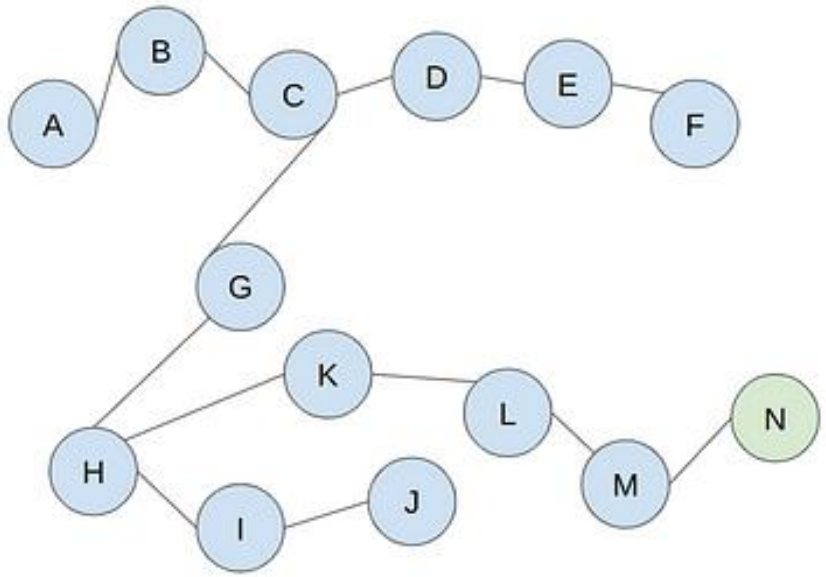
Frontier:
[M]

Visited:
[A, B, C, D, G, E, H, F, K, I, L, J]

BFS:
[A, B, C, D, G, E, H, F, K, I, L, J]

Breadth First Search (BFS)

Append M to the Visited and BFS, then append M's neighbor (N) to the Frontier. Finally, pop M from the Frontier.



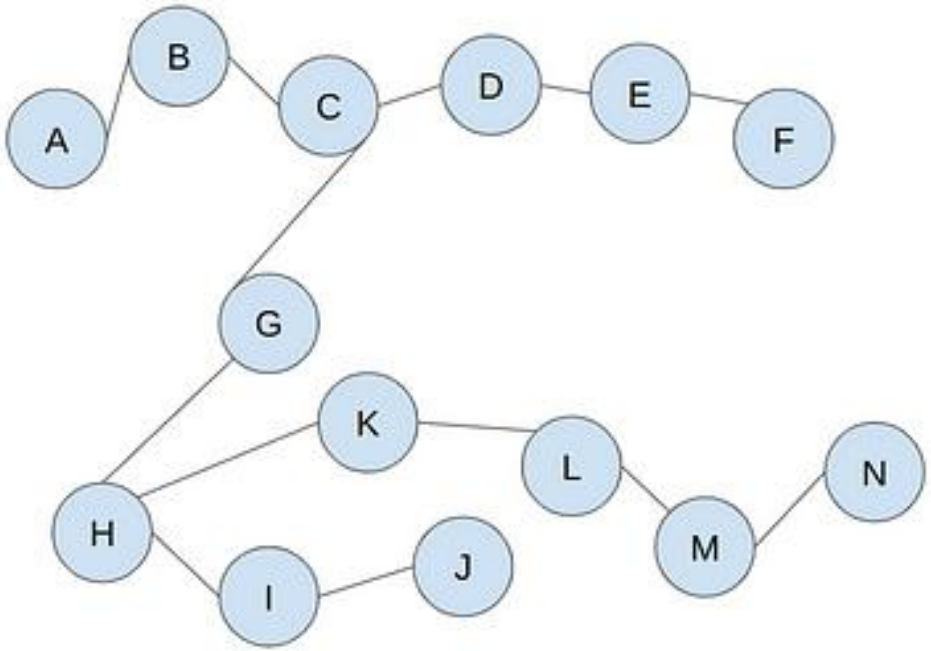
Frontier:
[N]

Visited:
[A, B, C, D, G, E, H, F, K, I, L, J, M]

BFS:
[A, B, C, D, G, E, H, F, K, I, L, J, M]

Breadth First Search (BFS)

Append N to the Visited and BFS. Finally, pop N from the Frontier.



Frontier:
[]

Visited:
[A, B, C, D, G, E, H, F, K, I, L, J, M, N]

BFS:
[A, B, C, D, G, E, H, F, K, I, L, J, M, N]

Breadth First Search (BFS)

Solution: [A, B, C, D, G, E, H, F, K, I, L, J, M, N]

Costs: 13 (there are 13 nodes to step to reach the end node from the start node)

Breadth First Search (BFS)

Algorithm:

1. **Initialize** a queue with the root node.
2. While the queue is **not empty**:
   - **Dequeue** the front node.
   - Check if it is the goal state: If yes, return **the node** (or **True**).
   - Otherwise, **enqueue its children** *(left and right for a tree, all neighbors for a graph).*
3. If all nodes are explored and goal is not found, return **False**.

Breadth First Search (BFS)

Function:

```
from collections import deque

def bfs_search(root, goal):
    if not root:
        return False

    queue = deque([root])

    while queue:
        node = queue.popleft()
        if node.value == goal:
            return True
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

return False
```

Breadth First Search (BFS)

Advantages:

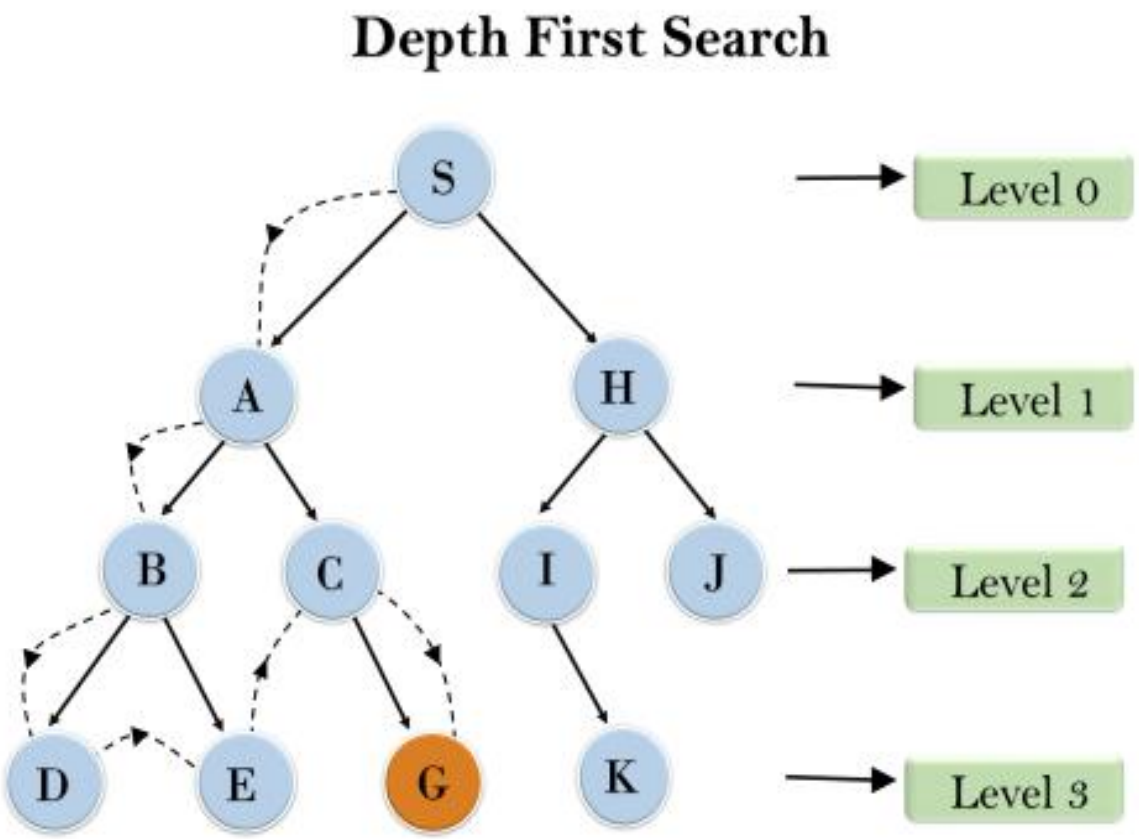
- ✓ Simplest search strategy
- ✓ BFS is complete. If there is a solution, BFS guarantees to find it.
- ✓ If there are multiple solutions, then a minimal solution will be found.

Disadvantage:

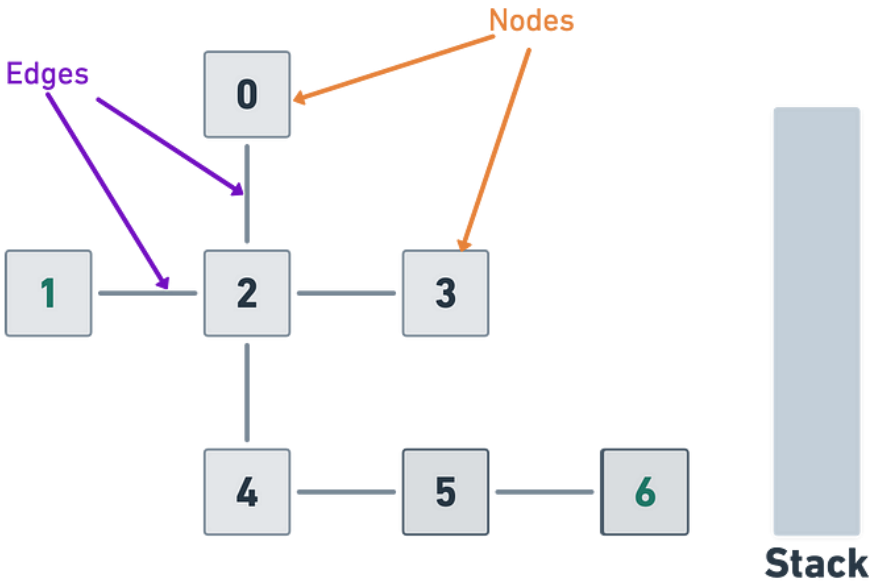
- ✓ BFS need the search space to be quite small to be used effectively.

Depth First Search (DFS)

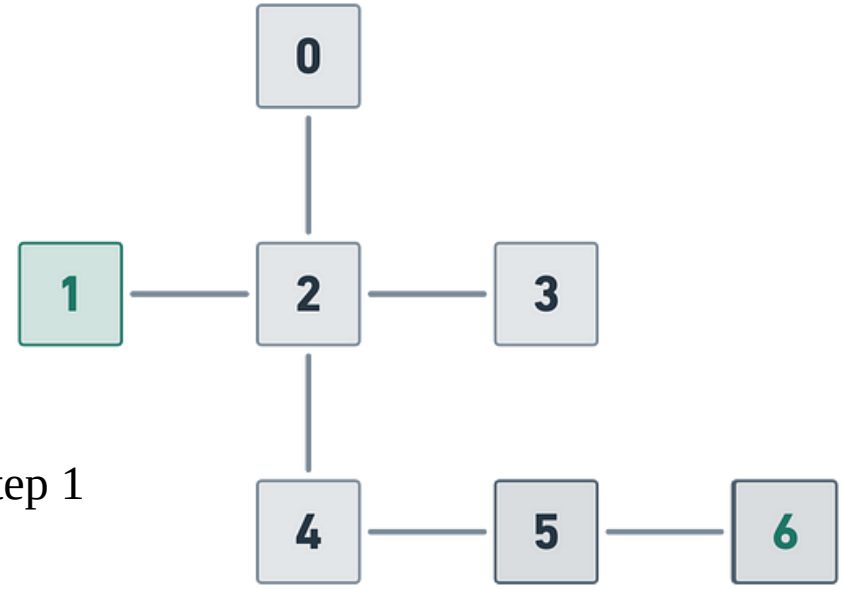
- Let's understand the traversing of a tree using DFS algorithm from the root node S to goal node G.
- DFS explores a graph by selecting a path and traversing it as deeply as possible before backtracking.
- The traversed path will be:
  S---> A---> B----> D---> E ---> C---> G
- It is not cost optimal.
- Incomplete.
- Complexity: O (b^d)



Depth First Search (DFS)

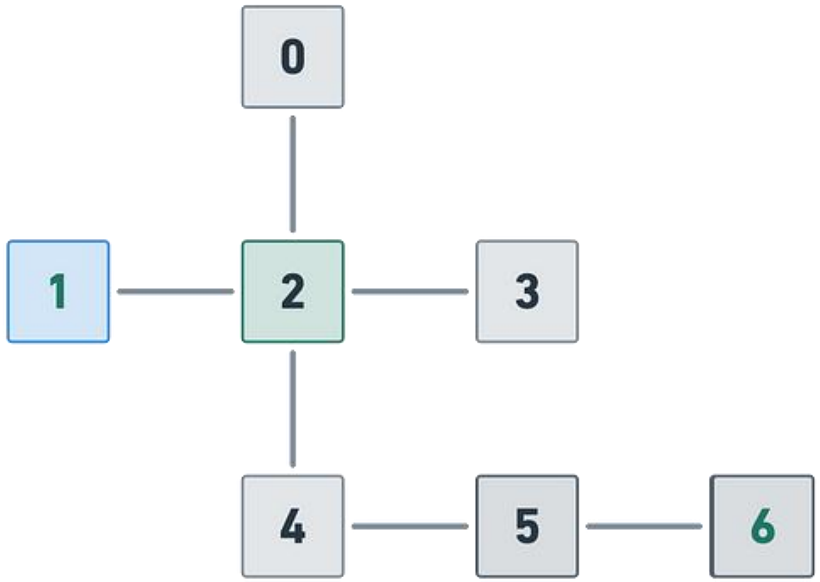


Step 0

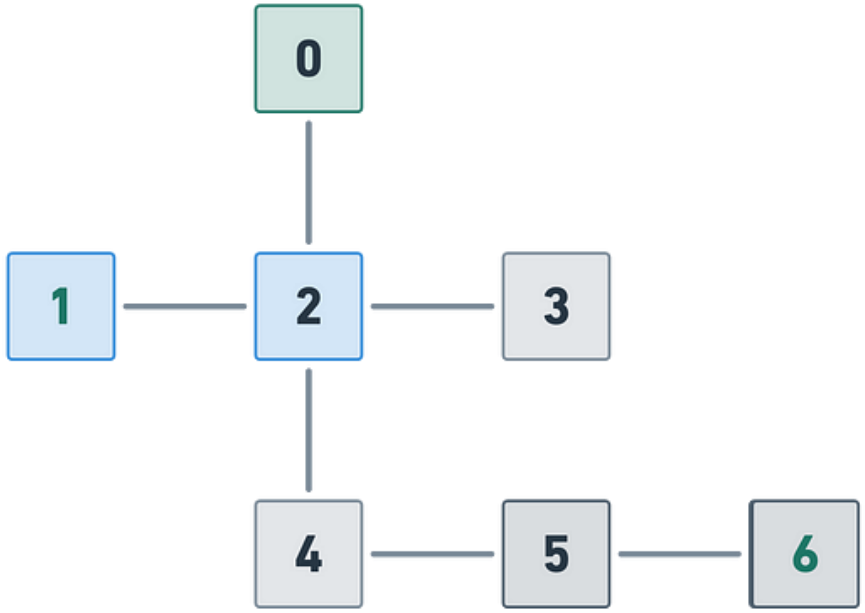


Step 1

Depth First Search (DFS)

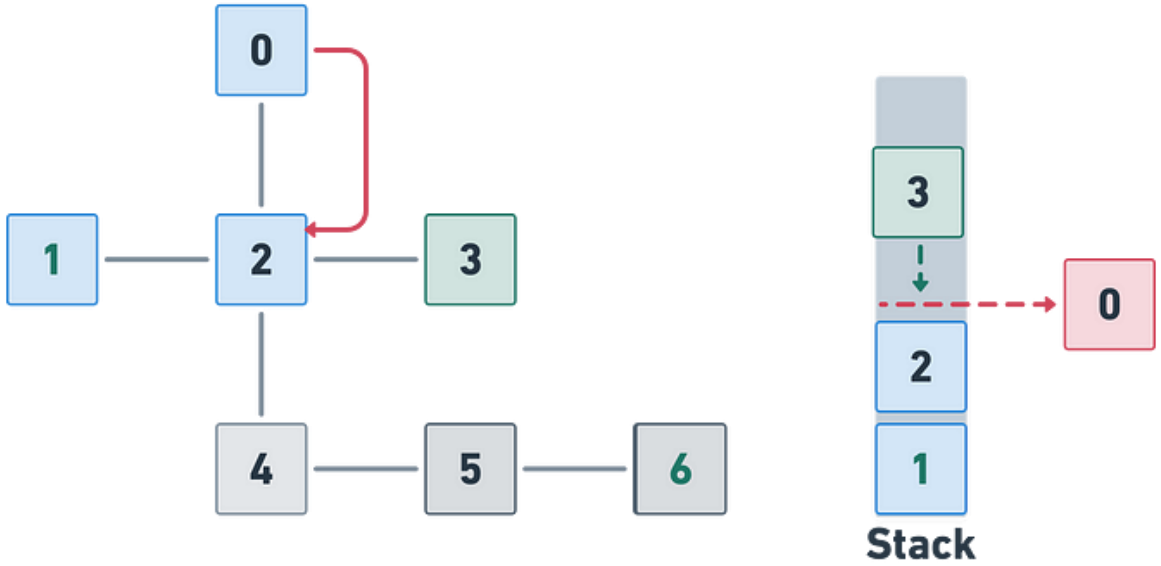


Step 2

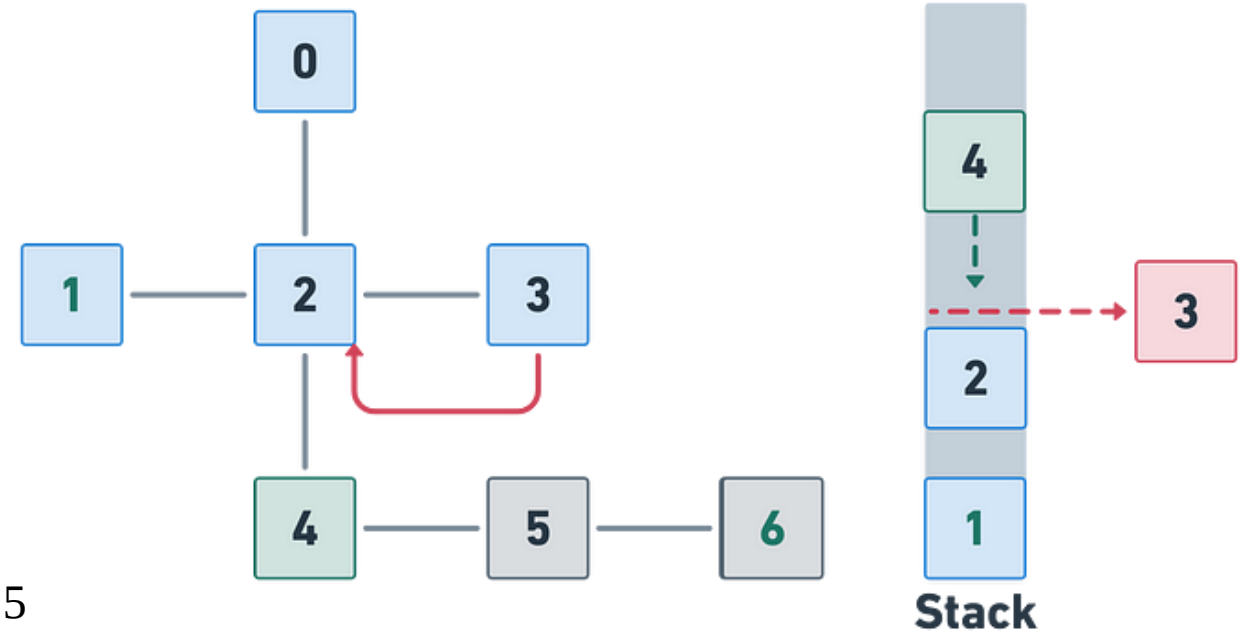


Step 3

Depth First Search (DFS)

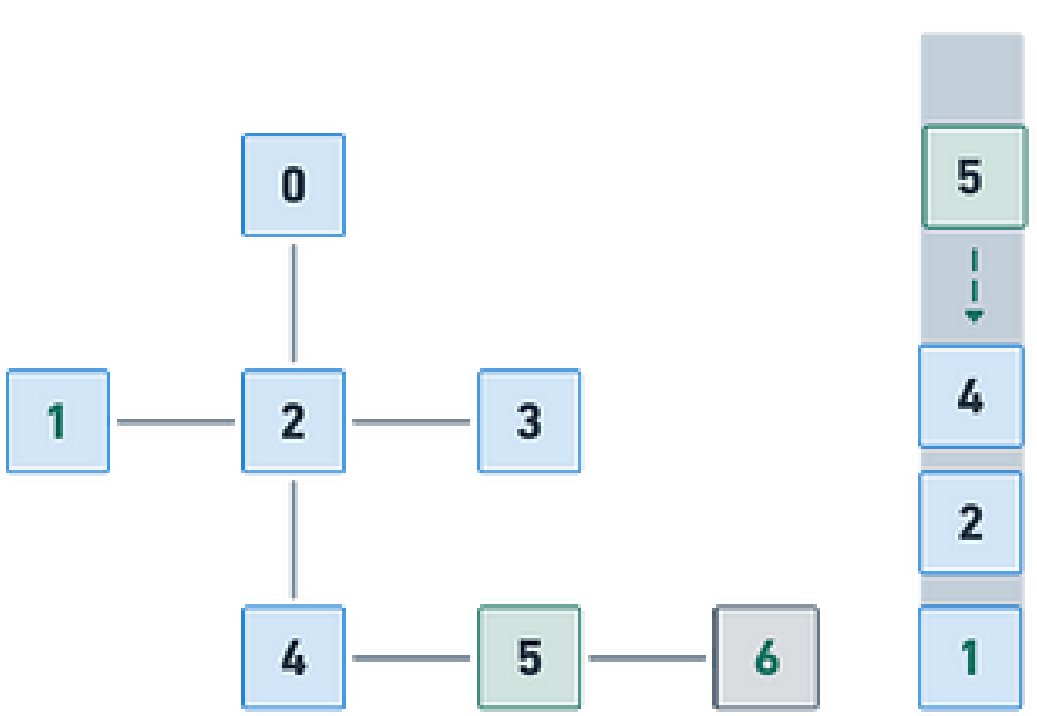


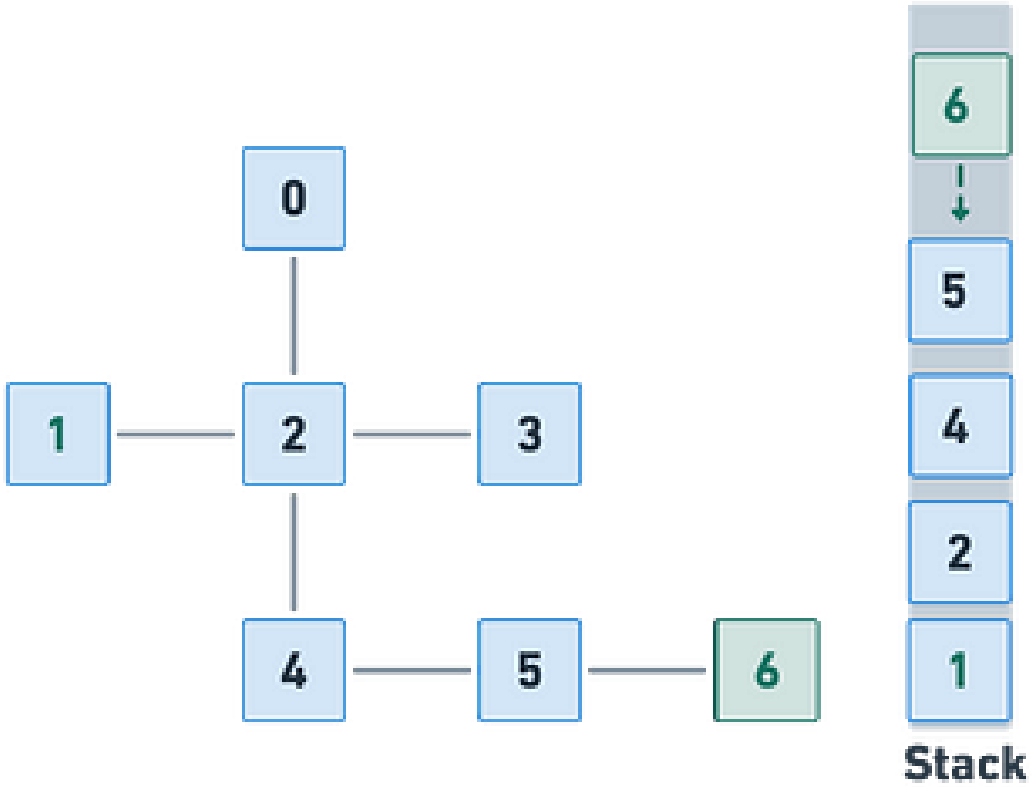
Step 4



Step 5

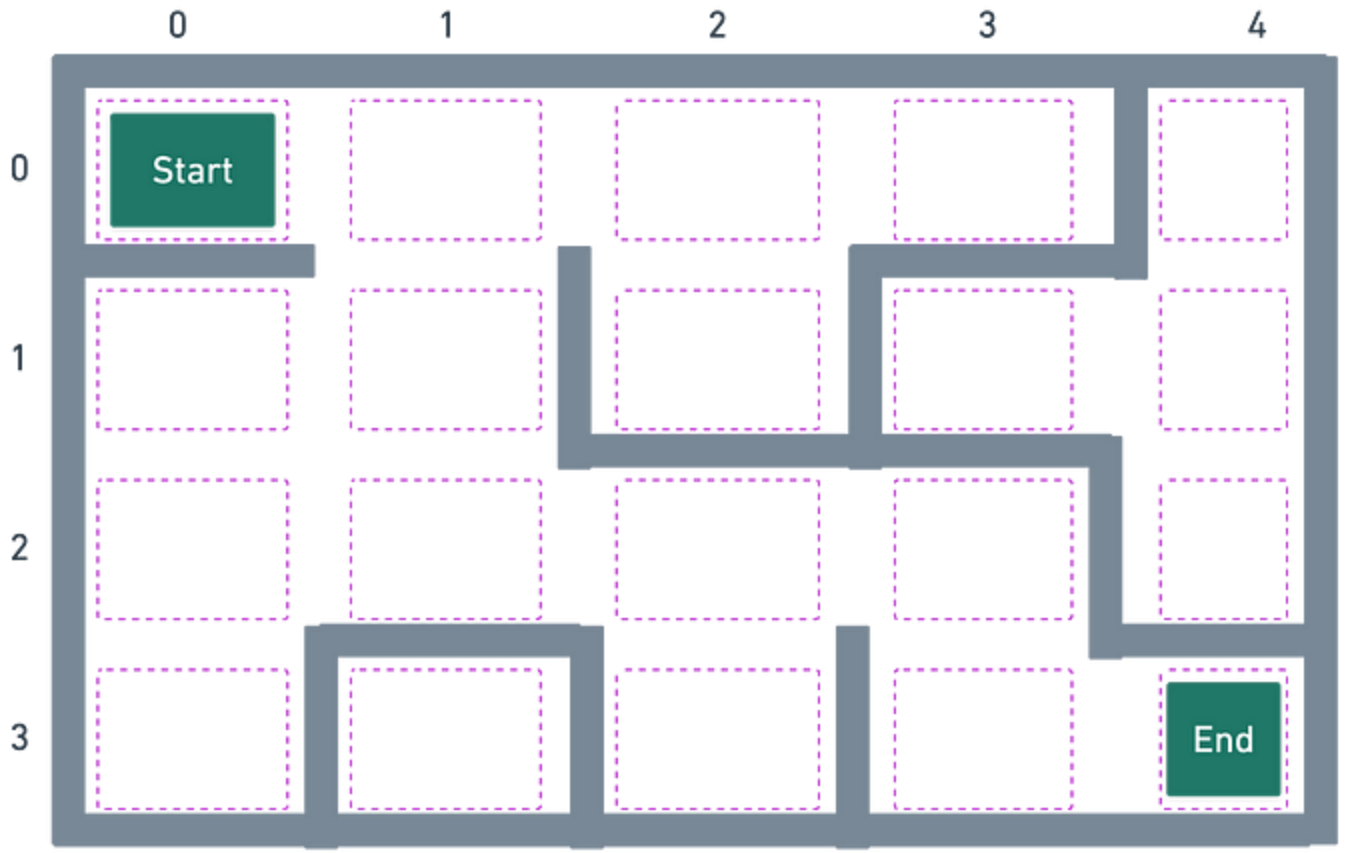
Depth First Search (DFS)





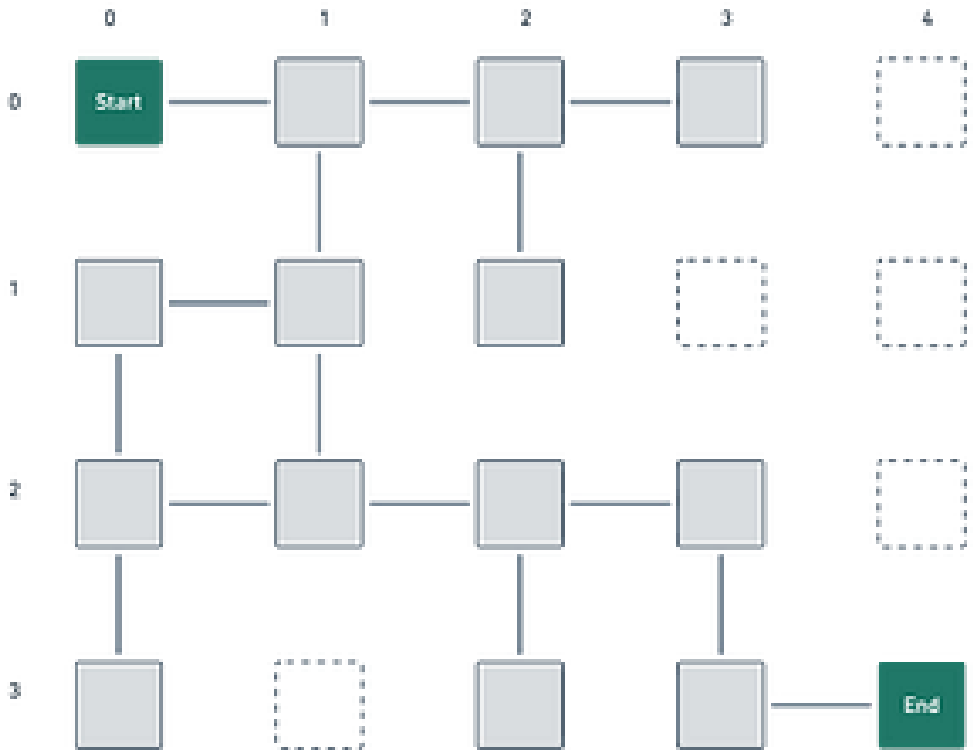
Step 6 and 7

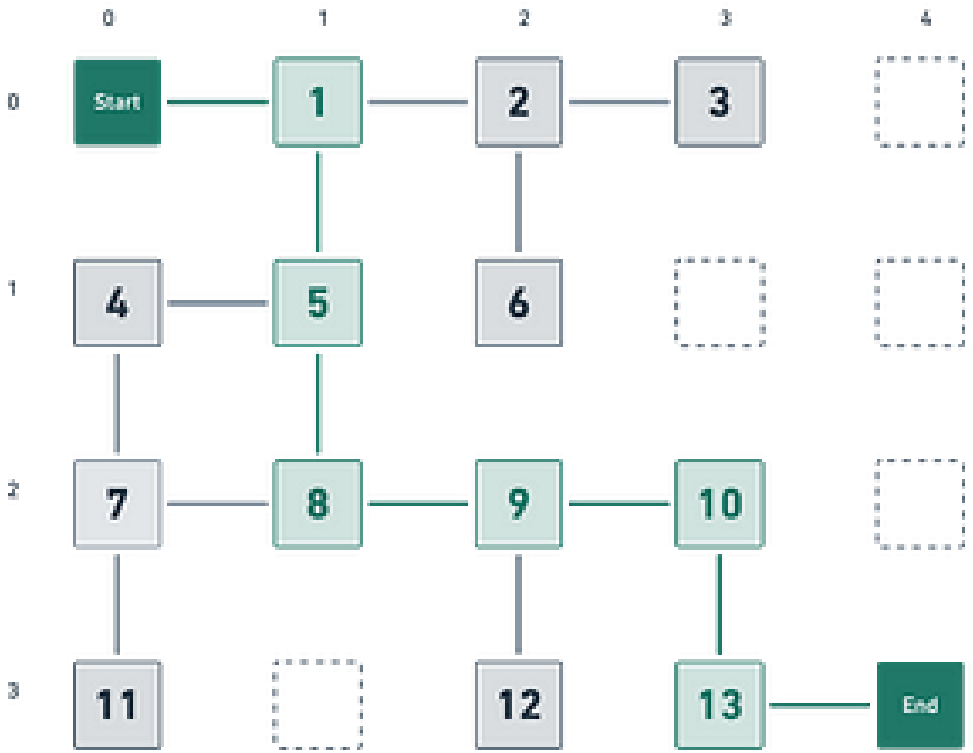
Depth First Search (DFS)



What about the Maze States??

Depth First Search (DFS)





Depth First Search (DFS)

- Depth-First Search (DFS) can be implemented in two ways:
  - Recursive DFS (using function calls and implicit recursion stack)
  - Iterative DFS (using an explicit stack)

Recursive DFS :

It uses recursion to traverse the tree/graph, relying on the **call stack** to keep track of visited nodes.

- **Algorithm:**
  - Base Case: If the node is None, return.
  - Process the Node: Print or perform an action on the node.
  - Recur on Left Child (if exists).
  - Recur on Right Child (if exists).

Depth First Search (DFS)

- Depth-First Search (DFS) can be implemented in two ways:
  - Recursive DFS (using function calls and implicit recursion stack)
  - Iterative DFS (using an explicit stack)

Iterative DFS :

Instead of recursion, we use an explicit stack (LIFO) to control the traversal.

- **Algorithm:**
  - **Initialize** a Stack with the root node.
  - While Stack is **Not Empty:**
    - **Pop** the top node.
    - **Process** the node.
    - **Push** the right child (if exists).
    - **Push** the left child (if exists).
  - Repeat Until Stack is Empty.

Depth First Search (DFS)

Function:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def dfs_recursive(node):
    if node is None:
        return
    print(node.value)
    dfs_recursive(node.left)
    dfs_recursive(node.right)
```

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

dfs_recursive(root)
```

Depth First Search (DFS)

Function:

```
def dfs_iterative(root):
    if root is None:
        return
    stack = [root]
    while stack:
        node = stack.pop()
        print(node.value)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

dfs_iterative(root)
```

Depth First Search (DFS)

Advantages:

- ✓ Requires less memory since only the nodes in the current path are stored.
- ✓ DFS may find a solution without exploring the much of the search space at all.

Disadvantages:

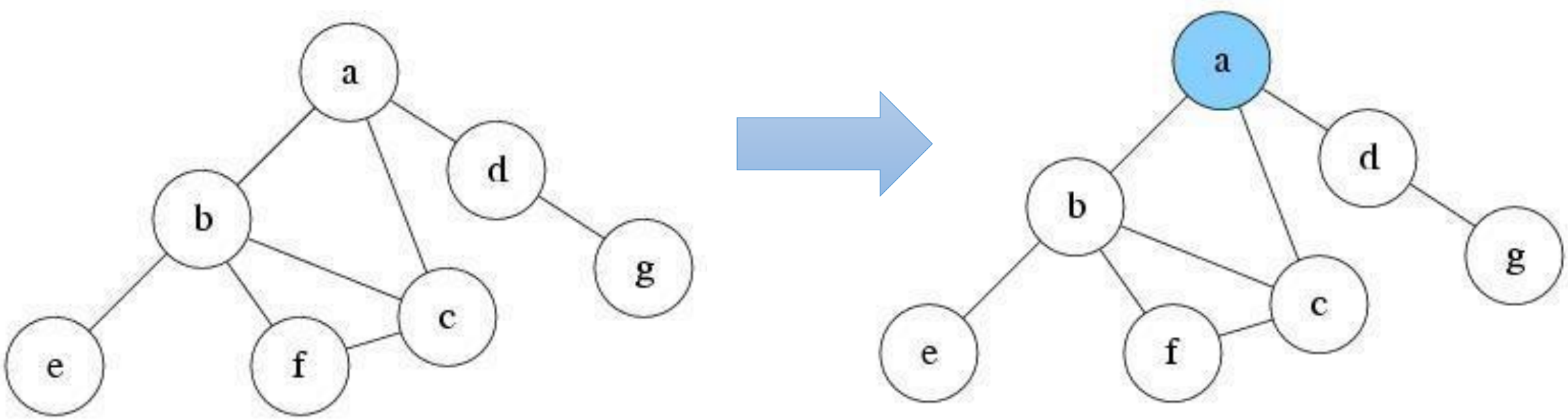
- ✓ DFS may find a sub-optimal solution (one that is deeper or most costly than the best solution).
- ✓ Incomplete: without a depth bound, the solution may not be found even if there exists a solution.

Depth Limited Search (DLS)

- The depth-limited search (DLS) method is almost equal to depth-first search (DFS)
- But DLS can work on the infinite state space problem because it bounds the depth of the search tree with a predetermined limit L.
- Nodes at this depth limit are treated as if they had no successors.
- Completeness depends on limit L.
- Time Complexity: O(b^L) (limited to depth L)
- Space Complexity: O (L)

Depth Limited Search (DLS)

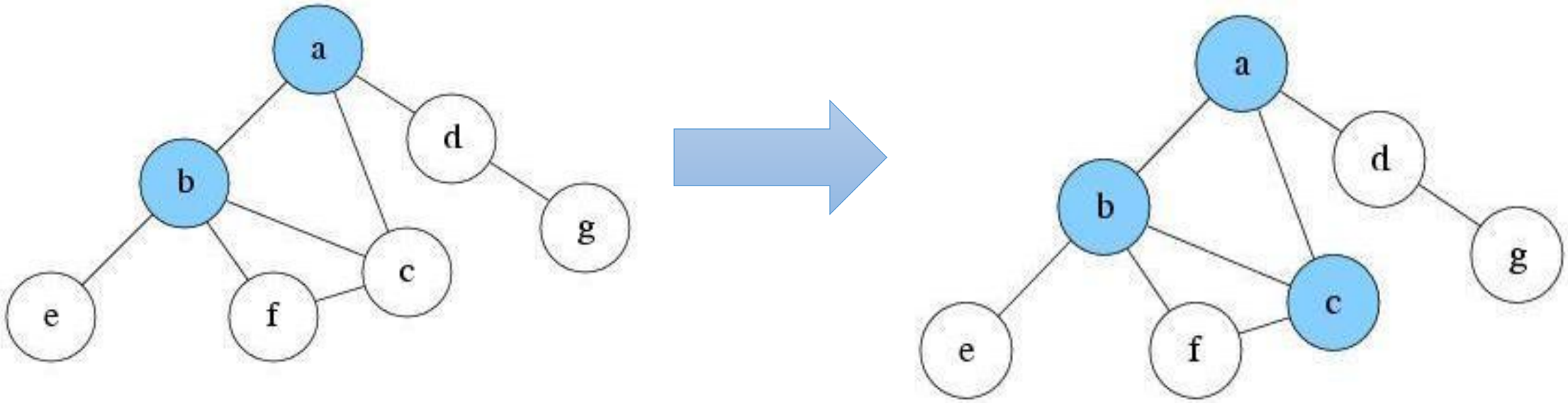
- Below is the graph we will traverse using DLS with L=1.
- Suppose the source node is node a. Initially, stack S1: empty.



- At first, the only reachable node is a. So, push it into stack S1 and mark as visited. Current level is 0. Hence, S1: a

Depth Limited Search (DLS)

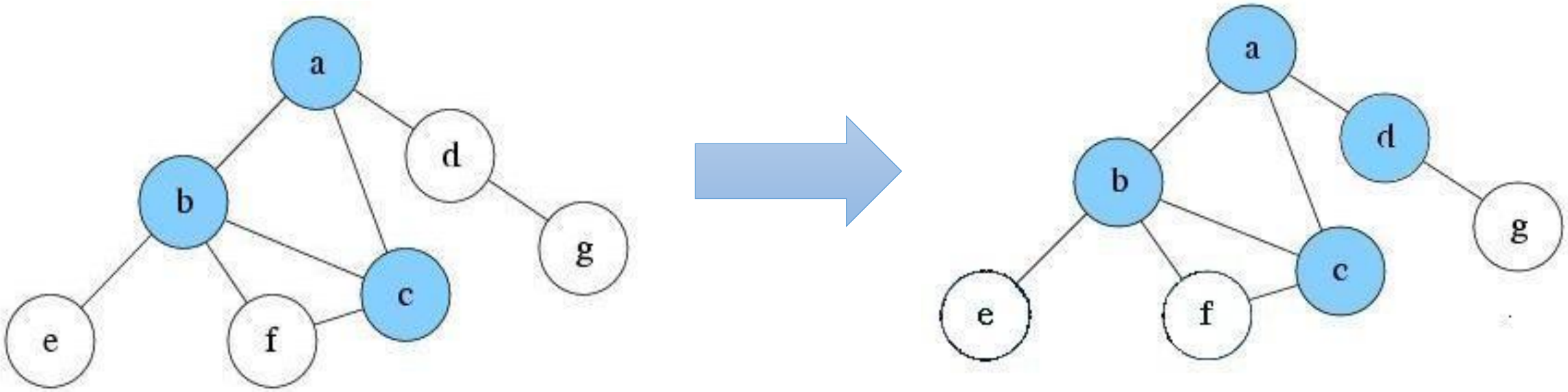
- After exploring a, now there are three nodes reachable: node b, c and d. Suppose we pick node b to explore first. Push b into S1 and mark it as visited. Current level is 1. S1: b, a



- Since current level is already the max depth L. Node b will be treated as having no successor. So, there is nothing reachable. Pop b from S1. Current level is 0. S1: a
- Explore a again. There are two unvisited nodes c and d that are reachable. Suppose we pick node c to explore first. Push c into S1 and mark it as visited. Current level is 1. S1: c, a

Depth Limited Search (DLS)

- Since current level is already the max depth L. Node c will be treated as having no successor. So there is nothing reachable. Pop c from S1. Current level is 0. S1: a
- Explore a again. There is only one unvisited node d reachable. Push d into S1 and mark it as visited. Current level is 1. S1: d, a



- Explore d and find no new node is reachable. Pop d from S1. Current level is 0. S1: a
- Explore a again. No new reachable node. Pop a from S1. S1: empty
- As, S1 is empty, DLS has finished.

Depth Limited Search (DLS)

Algorithm :

Input: A starting node, goal node, and depth limit L.

Output: **True** if the goal is found within depth L, otherwise **False**.

Steps:

1. **Base Case: If** the node is **None**, return **False**.
2. **Check if the node is the goal**: **If** yes, return **True**.
3. **If** depth limit is reached, return **False** *(cutoff condition).*
4. Recur for each child:
   - Call DLS on each child with **depth incremented by 1.**
   - **If** any recursive call returns True, return **True**.
5. **If** all children are explored and the goal is **not found**, return **False**.

Iterative Deepening Search (IDS)

- Iterative Deepening Search (IDS) is a combination of Depth-First Search (DFS) and Breadth-First Search (BFS).
- It **repeatedly runs Depth-Limited Search (DLS)** with increasing depth limits until the goal is found.
- Each iteration explores nodes at a specific depth limit, progressively increasing it.
- **When to use IDS:**
  - **When the depth of the solution is unknown**.
  - **When memory is limited** (like DFS, it only stores nodes of a single path).
  - **When BFS is too memory-intensive** (BFS requires O(b*d) space, while IDS uses O(d)).
  - **When a solution exists at a shallow depth** but we don’t know where.

Iterative Deepening Search (IDS)

❑ **Algorithm**:

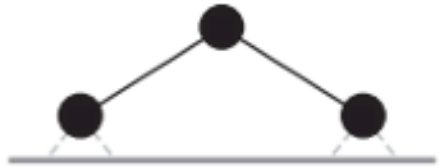
1. For each depth limit L = 0,1,2,... :
   - Run **Depth-Limited Search (DLS)** with depth limit L.
   - **If** the goal is found, return **True**.
   - **If** the goal is not found, increase L and repeat.

2. If the search exhausts all levels and the goal is not found, return **False**.

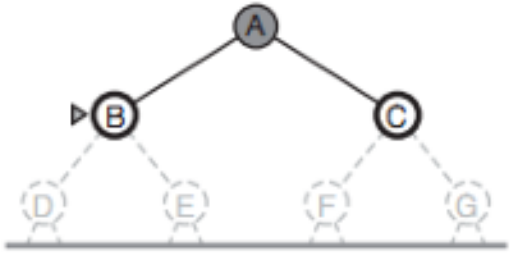
Iterative Deepening Search (IDS)

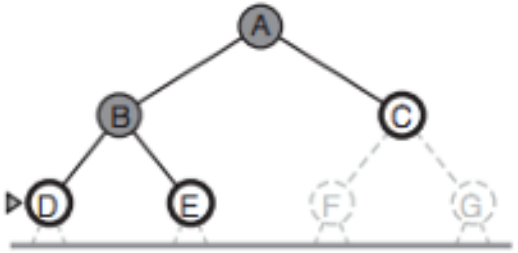
Limit = 0

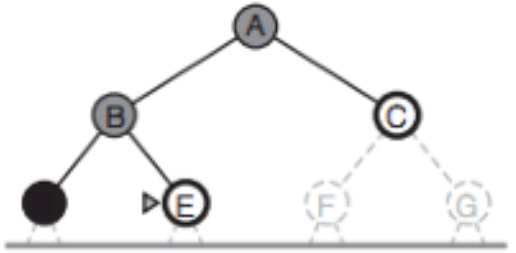
Limit = 1

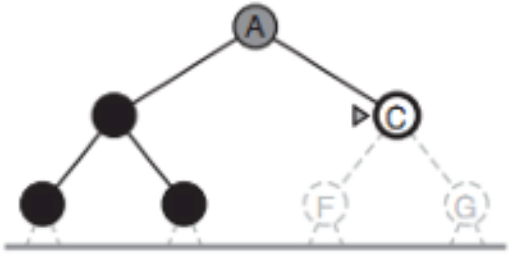


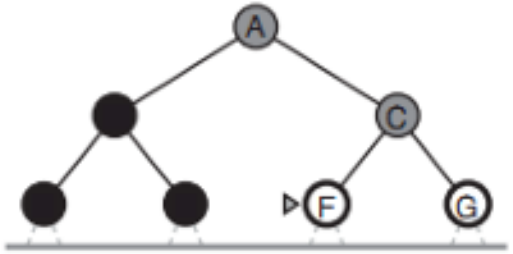
Limit = 2

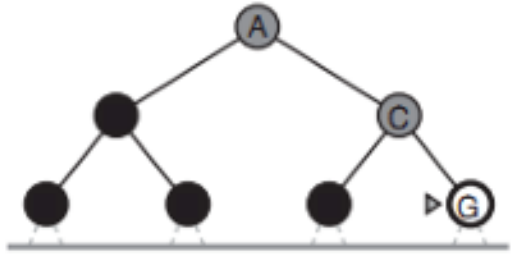


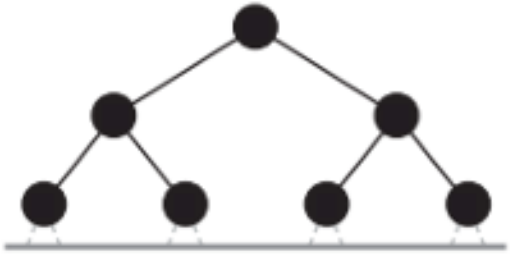




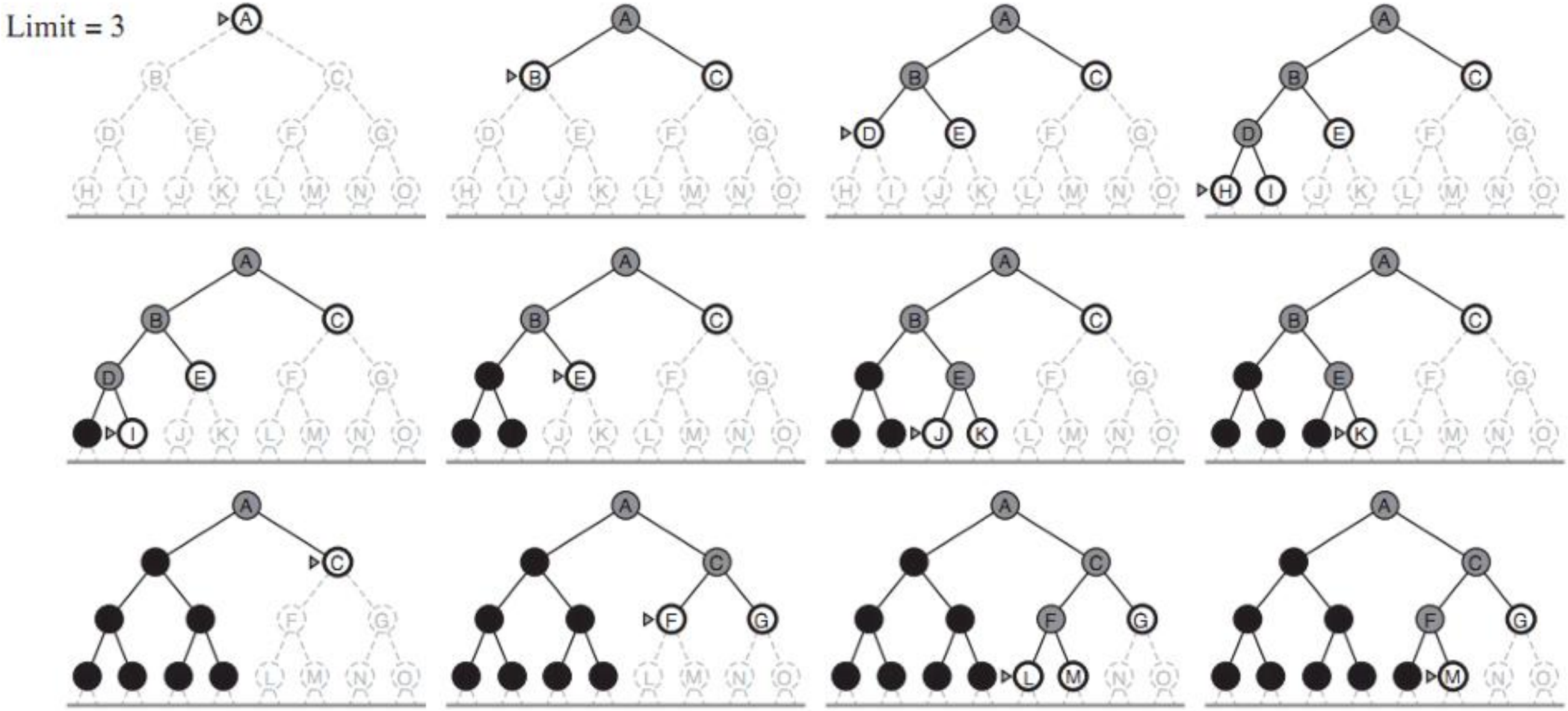








Iterative Deepening Search (IDS)



Uniform Cost Search (UCS)

- **Uniform Cost Search (UCS):** UCS is a search algorithm used to find the least-cost path from a start node to a goal node in a graph. It works by exploring the node with the lowest cumulative cost first, expanding nodes based on their path cost from the start.

- **Dijkstra's Algorithm**: Dijkstra's algorithm is often described as a specific form of UCS, particularly when the goal is to find the shortest path from a start node to all other nodes in the graph. It also expands nodes based on the lowest cumulative path cost, which is the defining characteristic of UCS.

The main difference is that Dijkstra's algorithm is typically used to find the shortest path from a start node to all nodes, whereas UCS is more flexible and can be applied to any search problem, typically with a goal node in mind.

What is Heuristic?

- The word **heuristic** comes from the Greek word *"heuriskein"*, which means **"to discover"** or **"to find".**
- A **heuristic** is a rule of thumb or a smart guess that helps in decision-making, especially when exact solutions are too expensive to compute.
- The heuristic function gives an estimate of the cost to reach the goal from a node, helping the algorithm "guess" which paths are more promising.
- It's called heuristic because it's not an exact measure—just a clever guide to speed up the search.
- The key is that the heuristic function estimates **how close a node is to the goal**, guiding the search efficiently.

How to measure Heuristic?

- **Grid-Based Pathfinding (e.g., in games, maps):**
  - Context: 2D or 3D maps, robots, game AI
  - Common Heuristics:
    - 4-direction (up/down/left/right): Manhattan Distance

$$h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$$

      - Example: If you're at (2,3) and the goal is (5,1) the heuristic is:

$$h = |2 - 5| + |3 - 1| = 3 + 2 = 5$$

    - 8-direction (add diagonals): Octile / Chebyshev

$$h(n) = \max(|x_n - x_{goal}|, |y_n - y_{goal}|)$$

      - Example: If you're at (2,3) and the goal is (5,1) the heuristic is:

$$h = \max(|2 - 5|, |3 - 1|) = \max(3, 2) = 3$$

How to measure Heuristic?

- **Grid-Based Pathfinding (e.g., in games, maps):**
  - Any-angle (free space): Euclidean Distance

$$h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2}$$

    - Example: If you're at (2,3) and the goal is (5,1) the heuristic is:

$$h = \sqrt{(2-5)^2 + (3-1)^2} = \sqrt{9+4} = \sqrt{13} \approx 3.61$$

- **Puzzles (like 8-puzzle, 15-puzzle):**
  - Context: Solving sliding puzzles
  - Heuristics: Misplaced Tiles : Count tiles not in goal position
    Manhattan Distance: Sum of distances of each tile to its goal

How to measure Heuristic?

- **Navigation in Road Networks (e.g., GPS):**
  - Context: Finding shortest driving route
  - Heuristics: Straight-line Distance, Travel Time Estimate etc.
- **Adversarial Games (e.g., Chess, Tic-Tac-Toe):**
  - Heuristics
    - Chess: Material count, piece positioning, mobility
    - Tic Tac Toe: Number of 2-in-a-rows, threats blocked
- **Resource Optimization:**
  - Context: Assigning tasks, minimizing cost or space
  - Heuristics: Estimated remaining work, Unused space left etc.
- **Traveling Salesman Problem**
  - Heuristics
    - Minimum spanning tree (MST) of remaining cities
    - Nearest neighbor distance sum
    - Straight-line distance to nearest unvisited

Best First Search

- Best First Search (**Greedy BFS**) is a search algorithm that explores a graph by selecting the most promising node based on a given heuristic function $h(n)$.
- It chooses the node that appears to be closest to the goal.
- It is a greedy algorithm, using only the heuristic $h(n)$, not the cost from the start node.
- It uses a **priority queue** to pick the next node with the ***lowest*** *h(n)*.
- Not guaranteed to find the shortest path, unless the heuristic is perfect.
- The formula used: $f(n) = h(n)$; $h(n)$ = Estimated cost from node n to goal

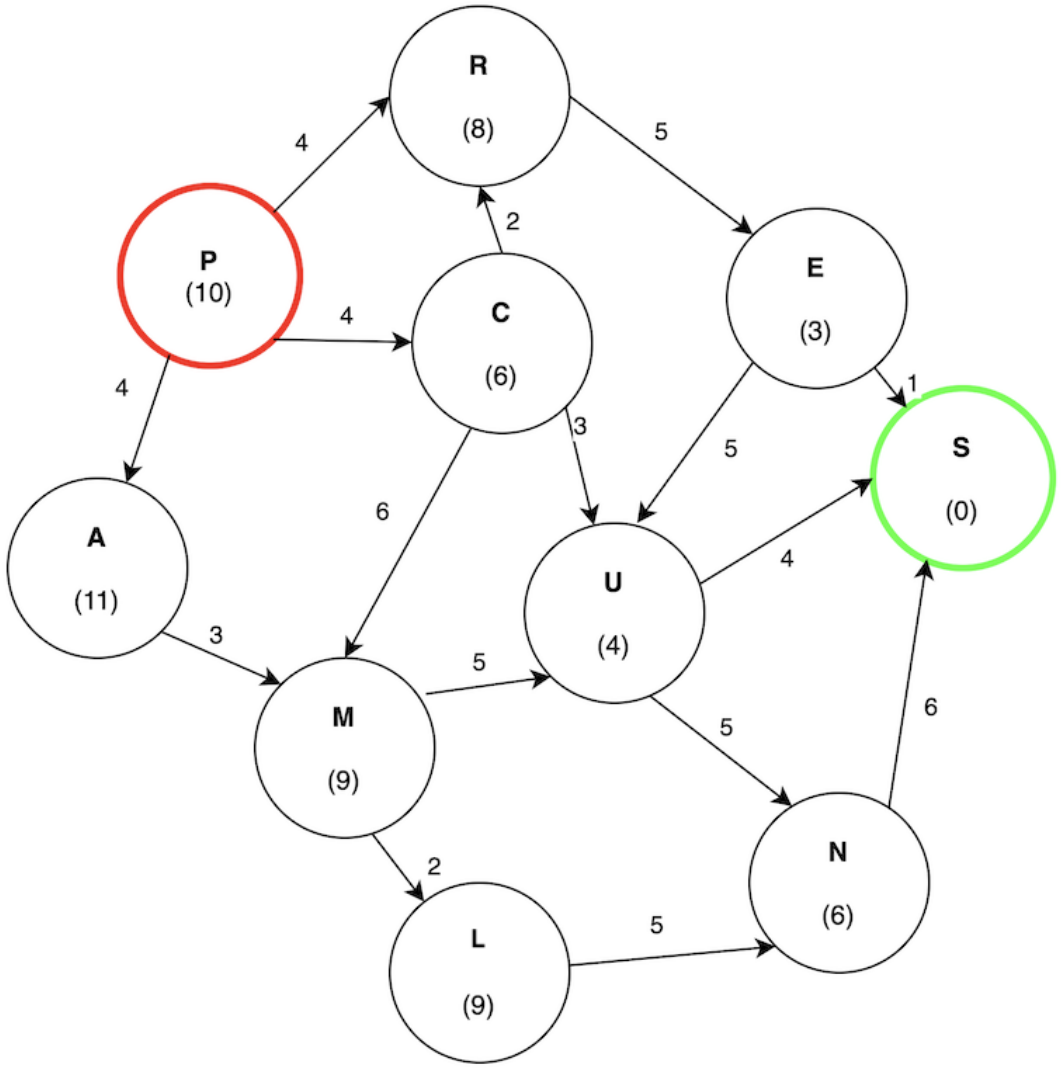
Best First Search

❑ Algorithm:

1. Initialize a priority queue called *open_list*
   Add the start node *S* to *open_list* with priority = *h(S)*
2. Initialize an empty set called *closed_list*
3. While *open_list* is not empty:
   a. Remove the node *n* with the lowest *h(n)* from *open_list*
   b. If *n* is the goal node:
      Return the path from *S* to *G* by backtracking through parents
   c. Add *n* to *closed_list*
   d. For each neighbor *m* of *n*:
      If *m* is not in *open_list* and not in *closed_list*:
      - Set parent of *m* to *n*
      - Add *m* to *open_list* with priority = *h(m)*
4. If goal is not found:
   Return "Failure – no path exists"

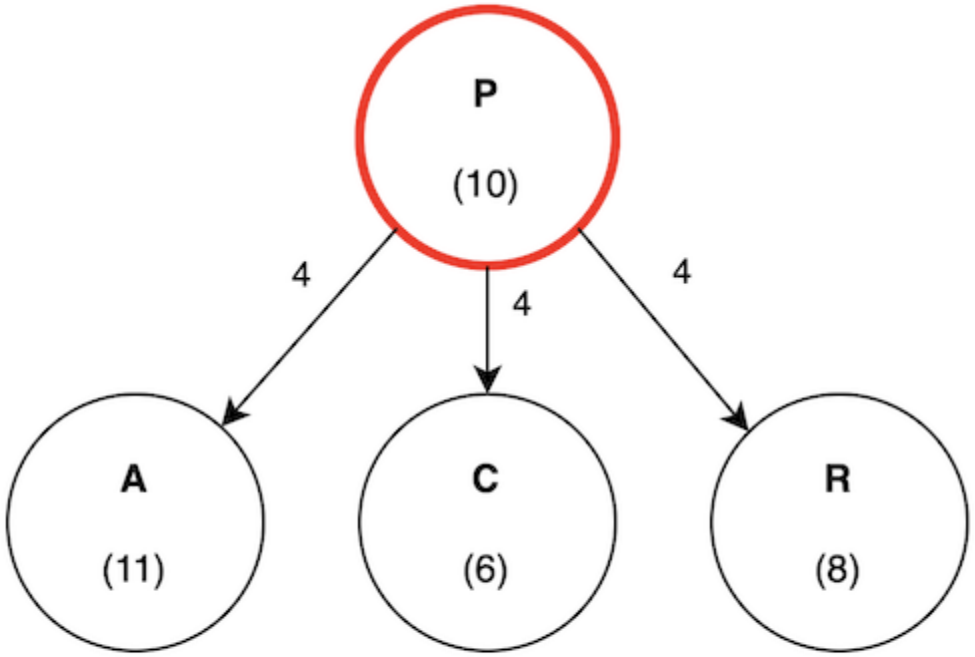
Best First Search – Example

- Consider finding the path from P to S in the following graph:
- The cost is measured strictly using the heuristic value.
- In other words, how close it is to the target.



Best First Search – Example

- C has the lowest cost of 6.

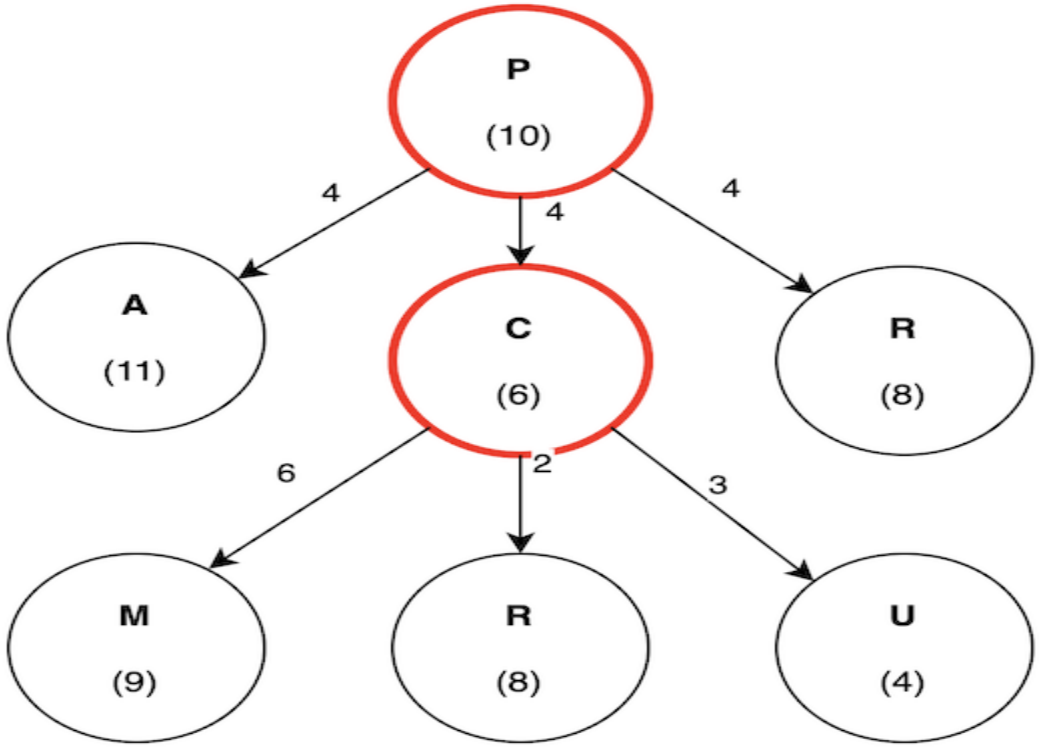


Node[cost]
A[11]
C[6]
R[8]

Closed List
P

Best First Search – Example

- C has the lowest cost of 6. The search will continue like so:

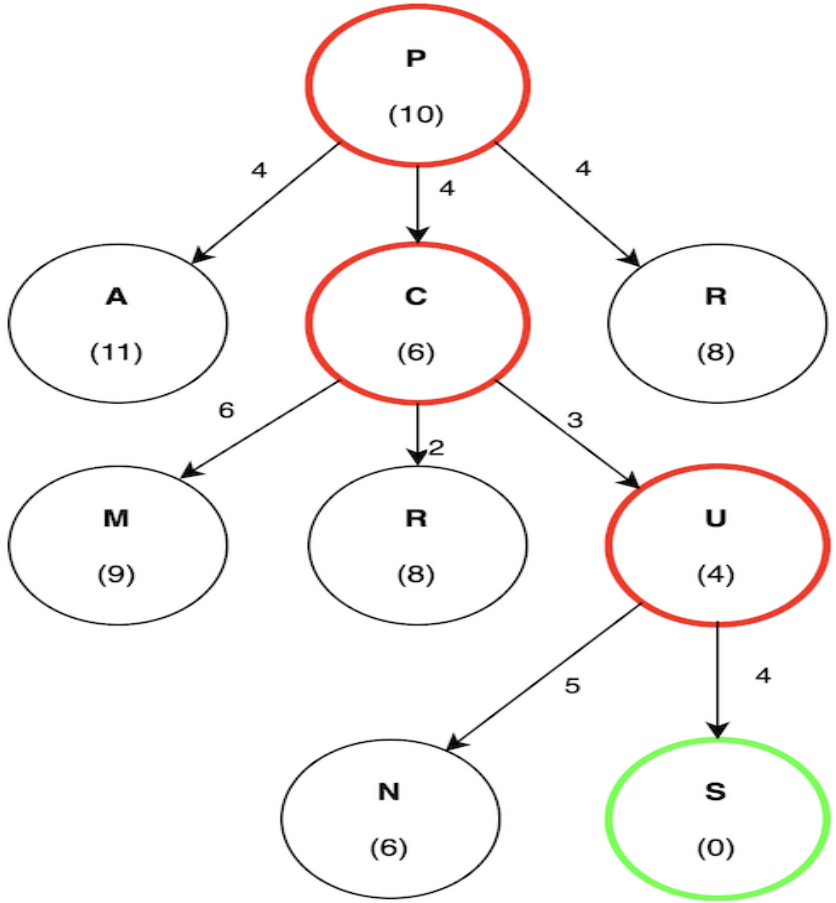


Node[cost]
M[9]
R[8]
U[4]

Closed List
P
C

Best First Search – Example

- U has the lowest cost compared to M and R, so the search will continue by exploring U. Finally, S has a heuristic value of 0 since that is the target node:



Node[cost]
N[6]
S[0]

Closed List
P
C
U

- The total cost for the path (**P** -> **C** -> **U** -> **S**) evaluates to 11. The potential problem with a greedy best-first search is revealed by the path (**P** -> **R** -> **E** -> **S**) having a cost of 10, which is lower than (**P** -> **C** -> **U** -> **S**). Greedy best-first search ignored this path because it does not consider the edge weights.

A* Search Algorithm

- The A* algorithm is called **“A-star”** because of the way it was named in the original paper by *Peter Hart*, *Nils Nilsson*, and *Bertram Raphael* in *1968*.
- The "A" was simply a label chosen by the authors—like an arbitrary identifier.
- The "*" (star) symbolizes that it is optimal, in the sense of being the best possible version in its class of algorithms under certain conditions (specifically, when the heuristic used is **admissible** and **consistent**).
- **Ideal properties of a heuristic:**
  - **Admissibility**: It should never *overestimate* the true cost to the goal.
  - **Consistency** (or Monotonicity): The estimated cost to reach the goal should never exceed the cost of getting to a neighbor plus the cost from that neighbor to the goal.
- **A good heuristic** makes A* search **much faster** because it helps the algorithm focus on promising paths.
- If you were using a random heuristic (i.e., guessing values with no pattern), A* would degenerate to something like Dijkstra’s algorithm, which explores all paths and ends up being inefficient.

A* Search Algorithm

- The word "**admissible**" comes from general English meaning "acceptable or allowed."
- In AI and pathfinding, it means a heuristic is "**allowed**" because it guarantees that A* will find the optimal path.
- It's considered "safe" or "permissible" to use in optimal search.
- A heuristic *h(n)* is **admissible** if: $h(n) < h^*(n)$
  - *h(n)* is the estimated cost from n to the goal,
  - $h*(n)$ is the actual minimum cost from n to the goal.
- The **heuristic helps A***:
  - ✓ **Prioritize** nodes that are likely to lead to the goal faster.
  - ✓ **Reduce search space**, making it more efficient.
  - ✓ **Balance** between actual cost so far and estimated future cost.
- Without a heuristic, A* becomes **Dijkstra's algorithm**, which explores all possible paths until it finds the shortest one.
- The formula used: $f(n) = g(n) + h(n)$

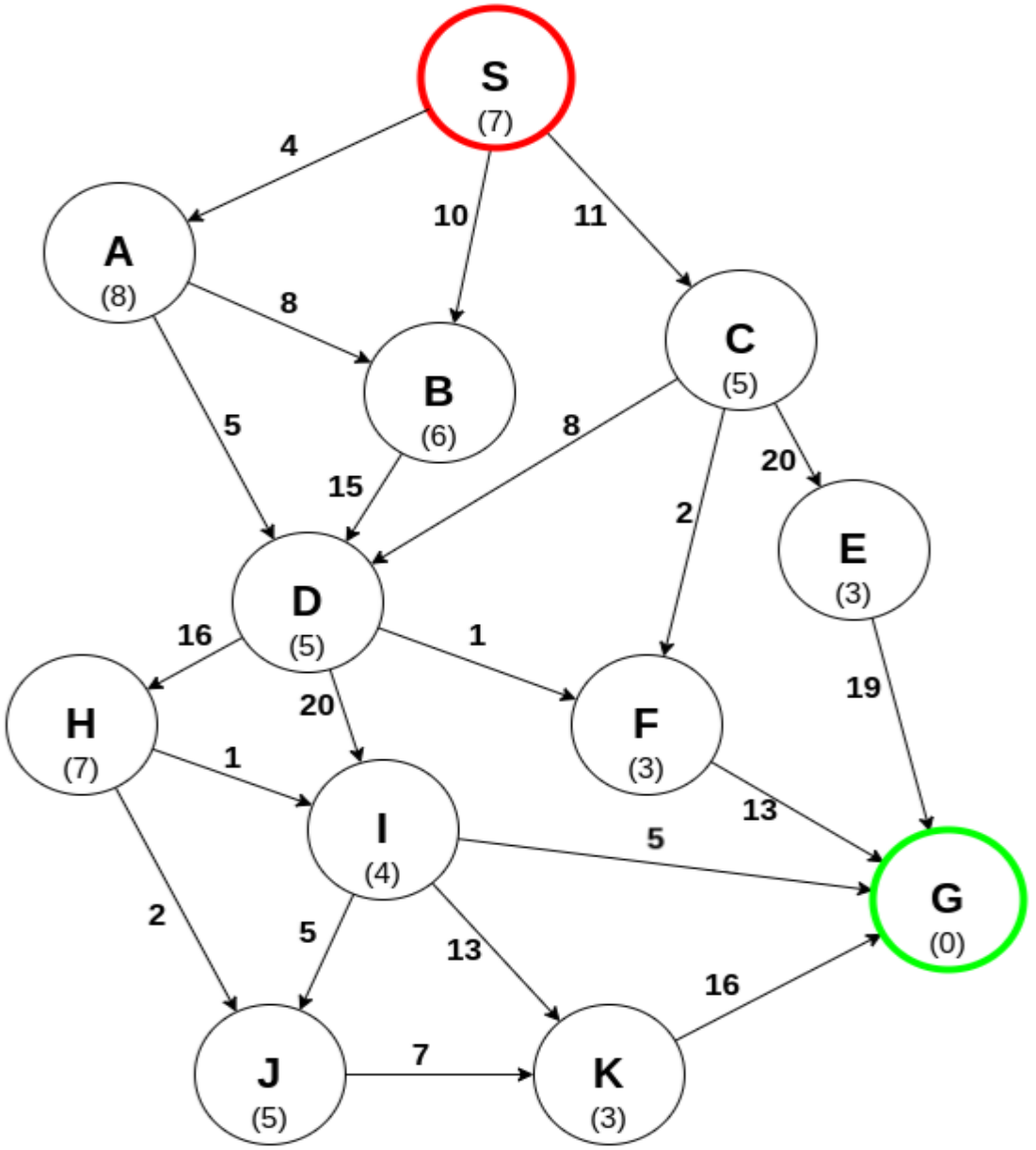
A* Search Algorithm

❑ Algorithm:

1. Initialize a priority queue called open_list
   Add the start node S to open_list with:
   g(S) = 0 and f(S) = g(S) + h(S)
2. Initialize an empty set called closed_list
3. Initialize a parent map to reconstruct the path
4. While open_list is not empty:
   a. Remove the node n with the lowest f(n) from open_list
   b. If n is the goal node G:
      Return the path by backtracking from G to S using the parent map
   c. Add n to closed_list
   d. For each neighbor m of node n:
      - Compute tentative_g = g(n) + cost(n, m)
      - If m is in closed_list and tentative_g >= g(m), skip m
      - If m is not in open_list OR tentative_g < g(m):
        - Update g(m) = tentative_g
        - Calculate f(m) = g(m) + h(m)
        - Set parent[m] = n
        - If m is not in open_list, add it to open_list
5. If open_list is empty and goal not found:
   Return "Failure – no path exists"

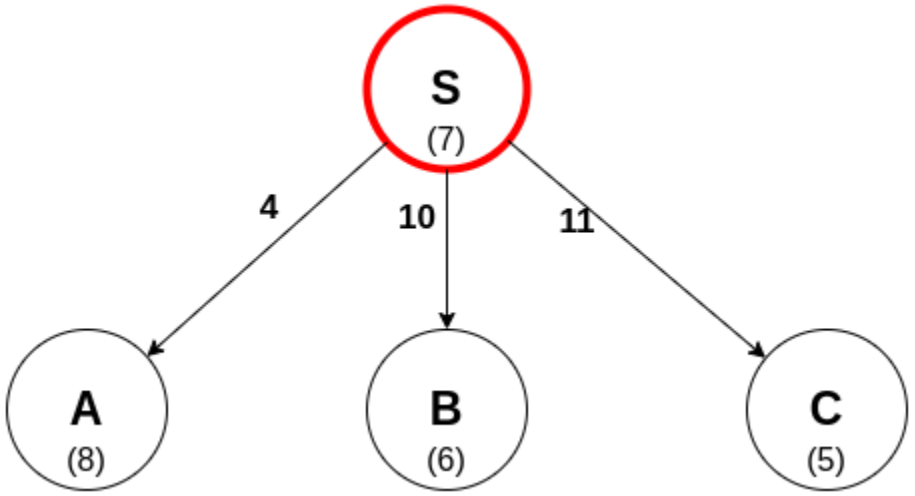
A* Search Algorithm – Example

- Consider the following example of trying to find the shortest path from **S** to **G** in the following graph.
- Each edge has an associated weight, and each node has a heuristic cost (in parentheses).
- An open list is maintained in which the node S is the only node in the list.



A* Search Algorithm – Example

- Exploring **S**.

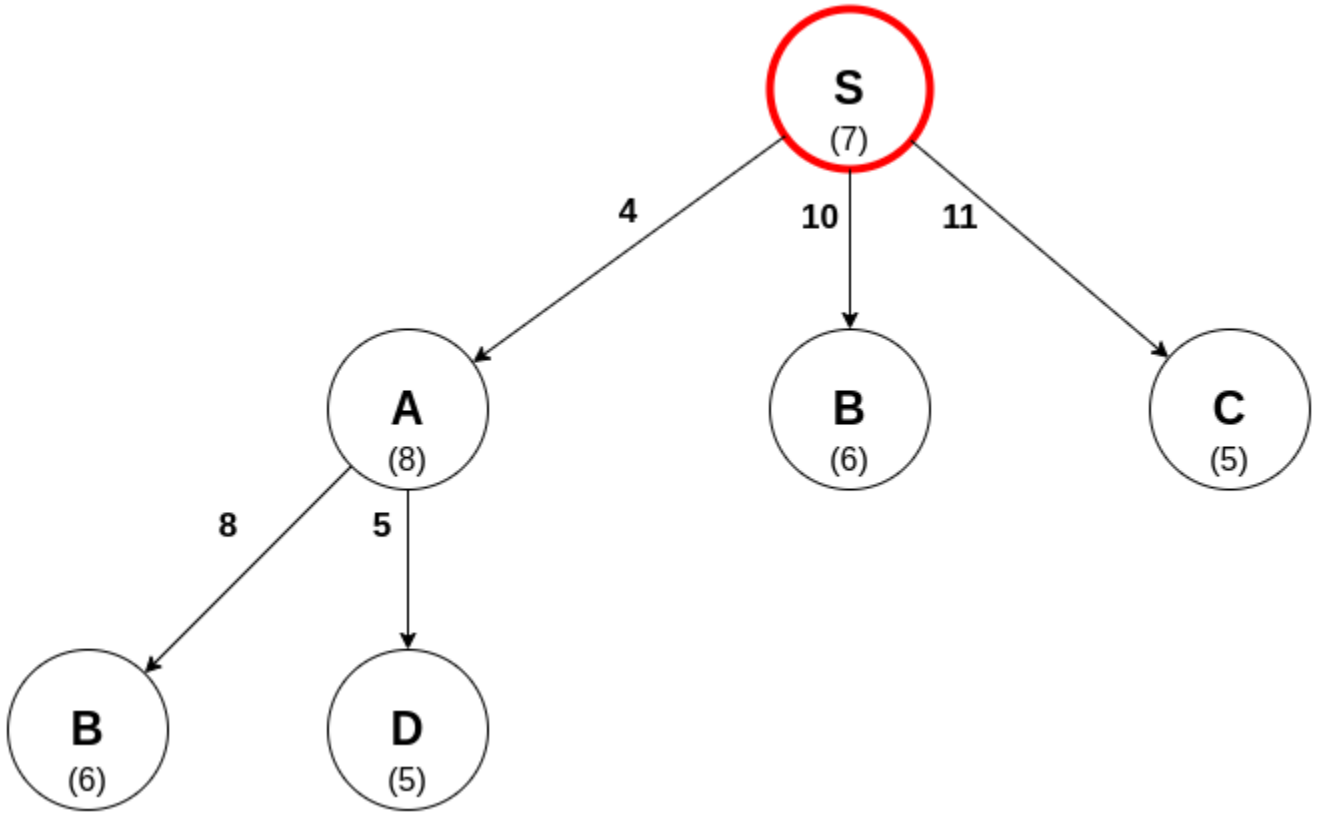


Node[cost]
A[12]
B[16]
C[16]

Closed List
S

A* Search Algorithm – Example

- **A** is the current most promising path, so it is explored next:

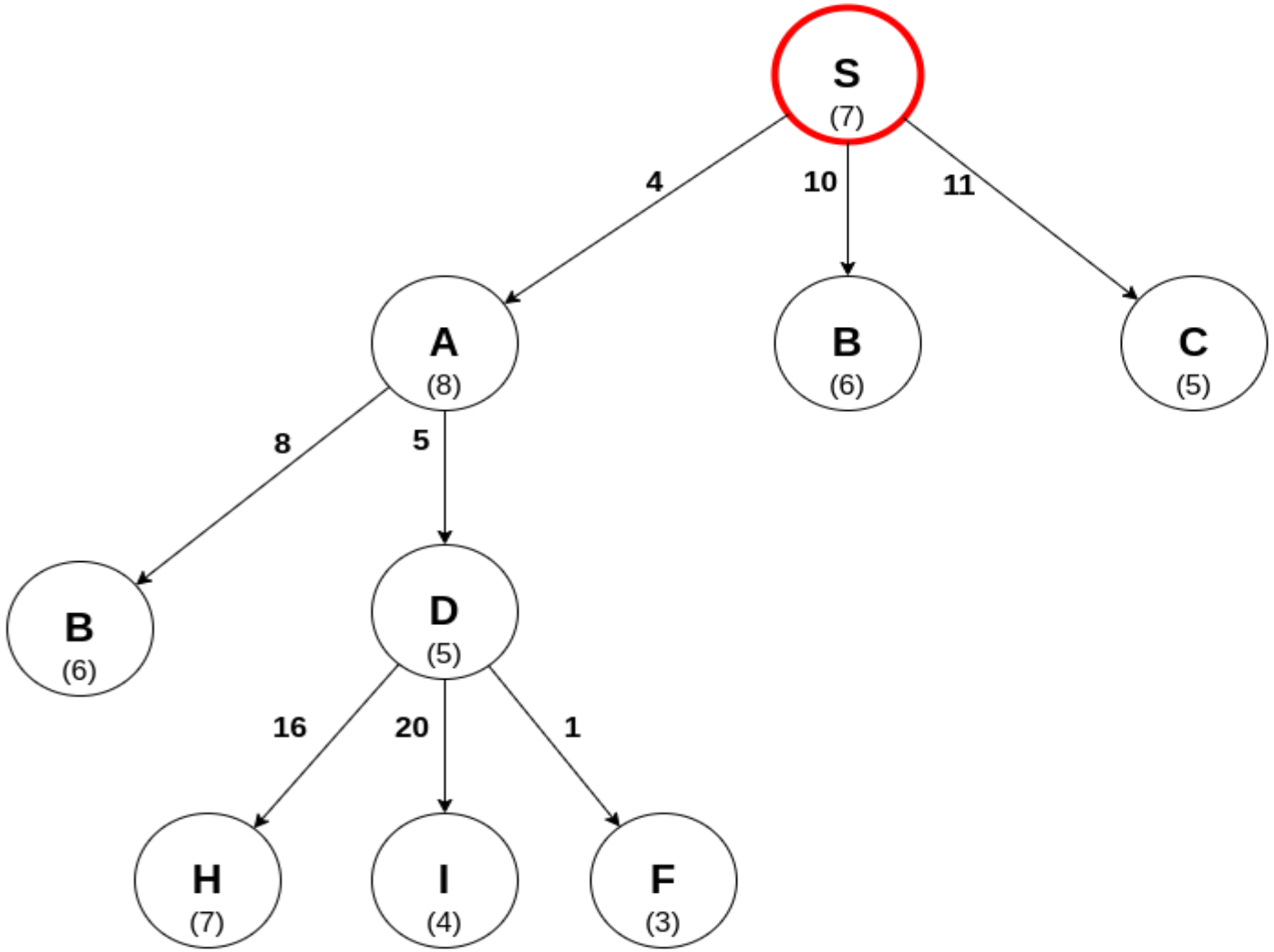


Node[cost]
D[14]
C[16]
B[16]
B[18]

Closed List
S
A

A* Search Algorithm – Example

- Exploring **D**:

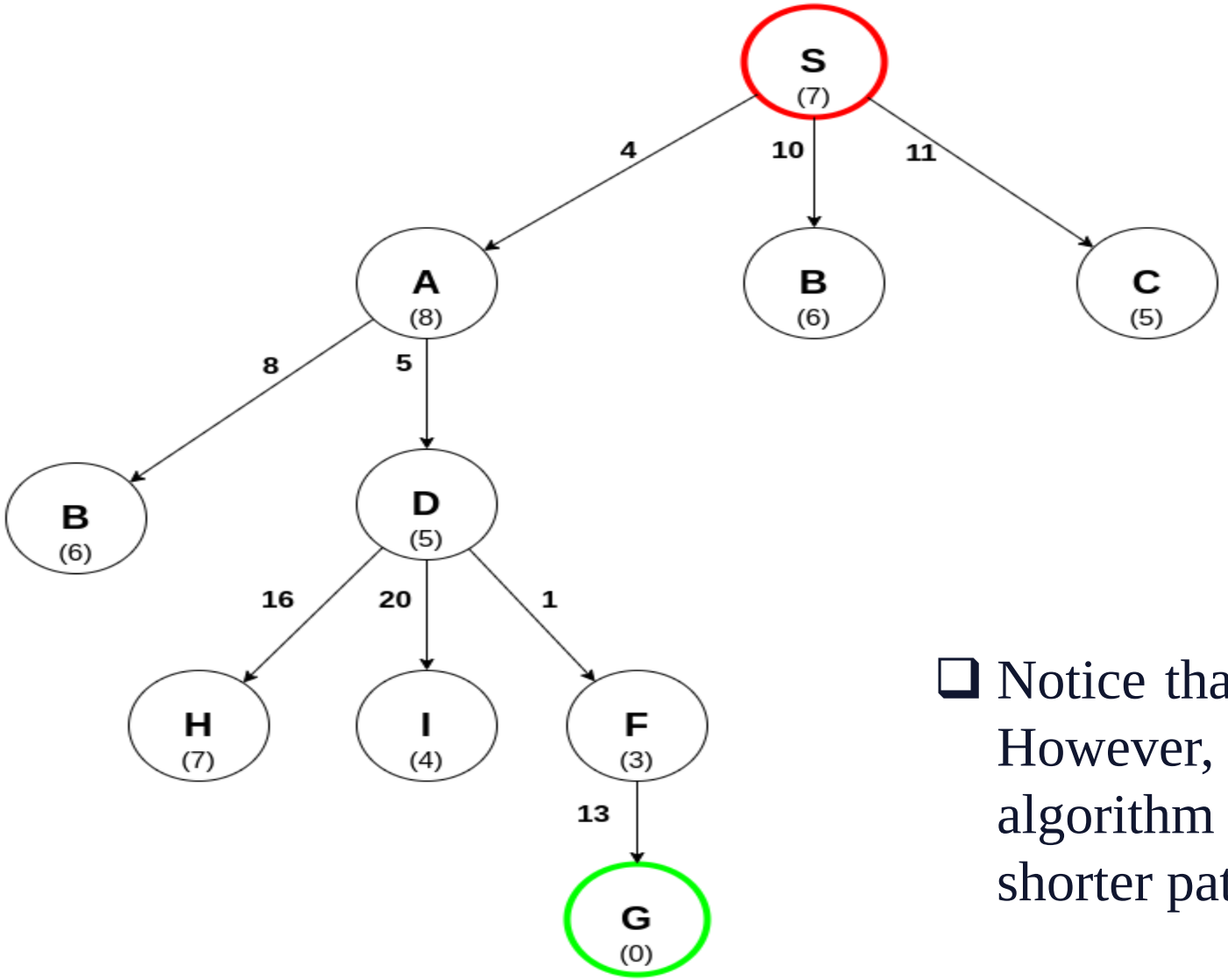


Node[cost]
F[13]
C[16]
B[16]
H[32]
I[33]
B[18]

Closed List
S
A
D

A* Search Algorithm – Example

- Exploring **F**:



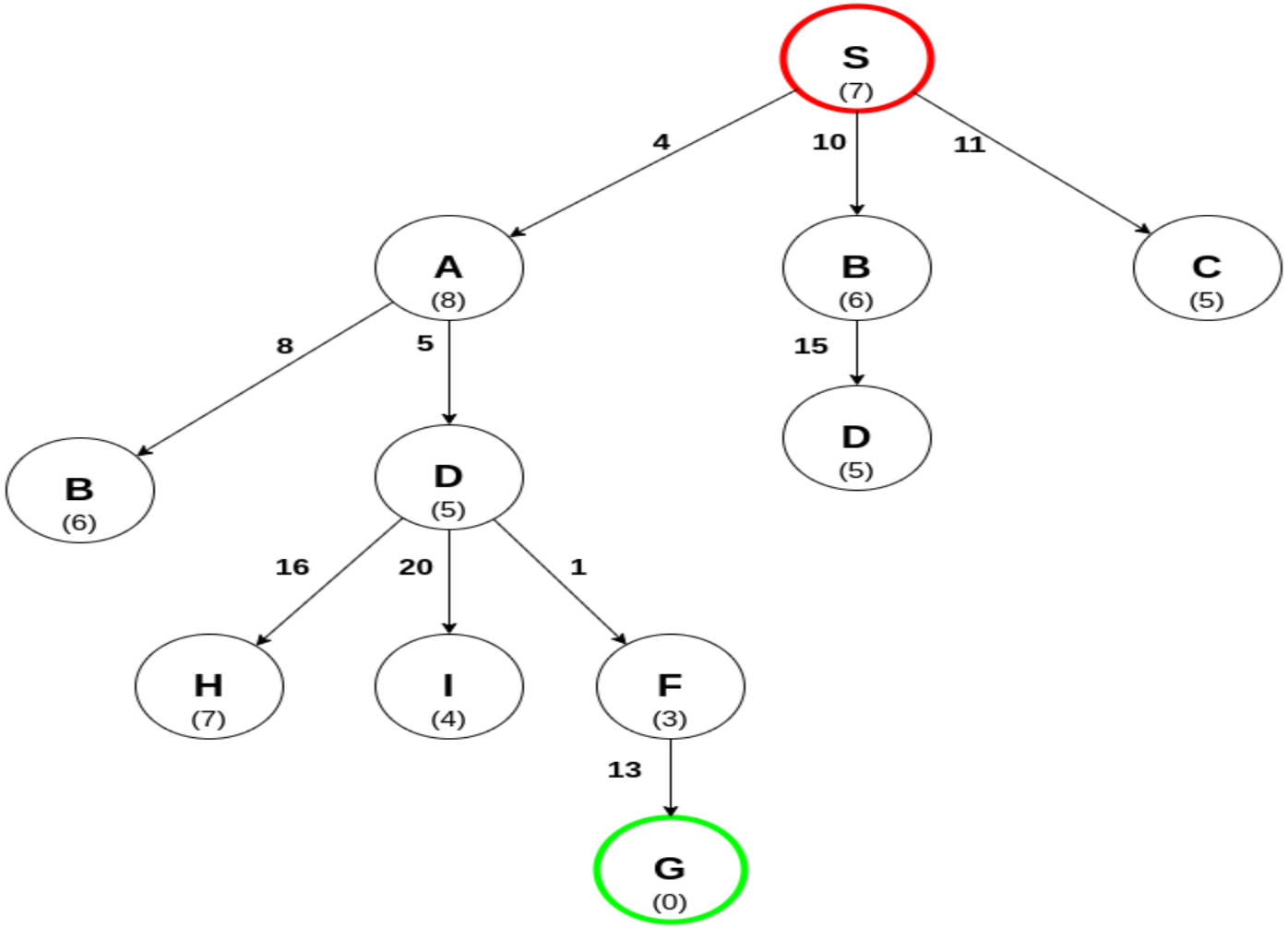
Node[cost]
B[16]
C[16]
B[18]
H[32]
I[33]
G[23]

Closed List
S
A
D
F

- Notice that the goal node **G** has been found. However, it hasn't been explored, so the algorithm continues because there may be a shorter path to G.

A* Search Algorithm – Example

- The node **B** has two entries in the open list: one at a cost of 16 (child of **S**) and one at a cost of 18 (child of **A**). The one with the lowest cost is explored next:

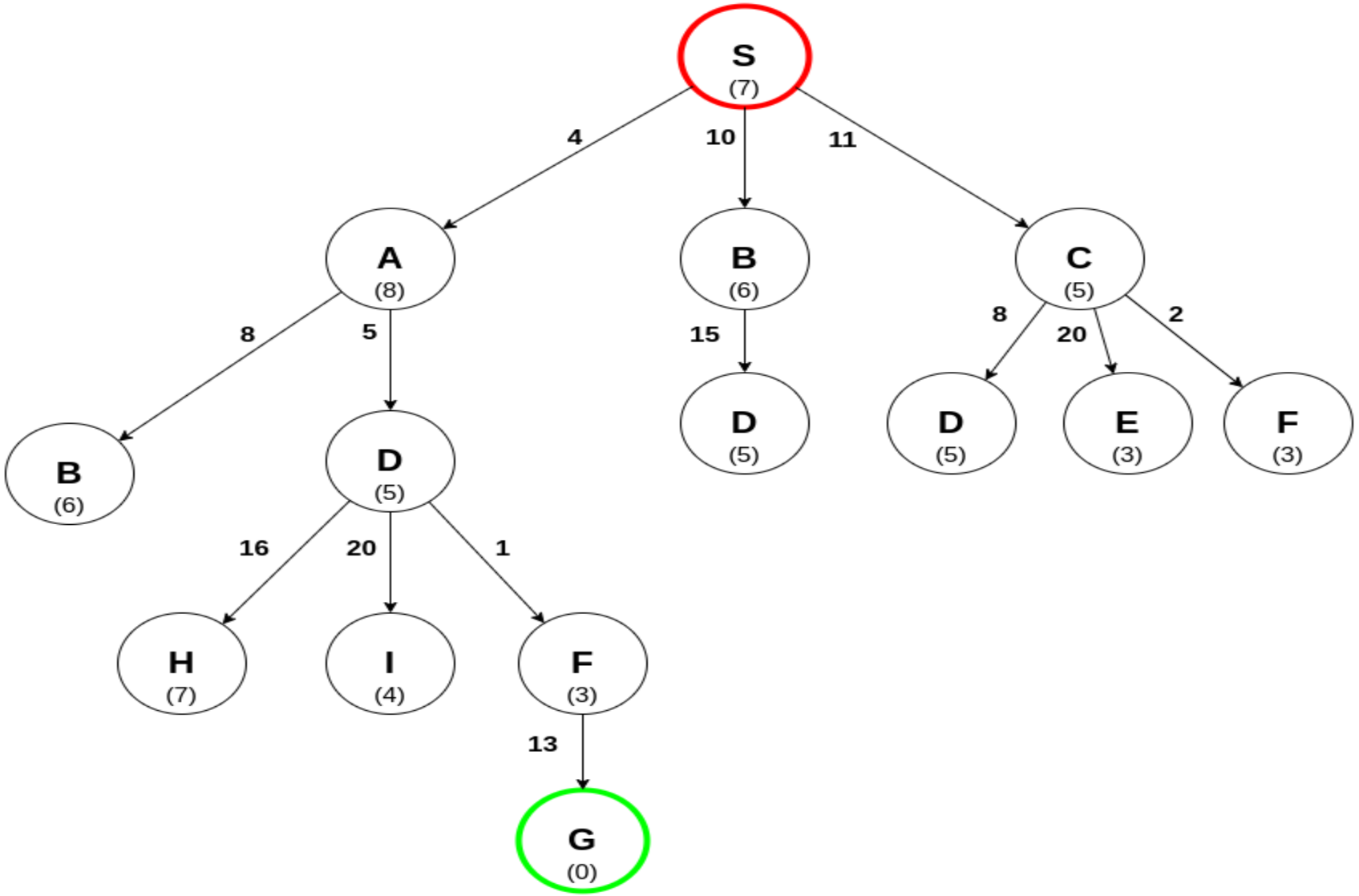


Node[cost]
C[16]
G[23]
B[18]
H[32]
I[33]

Closed List
S
A
D
F
B

A* Search Algorithm – Example

- Exploring **C**:

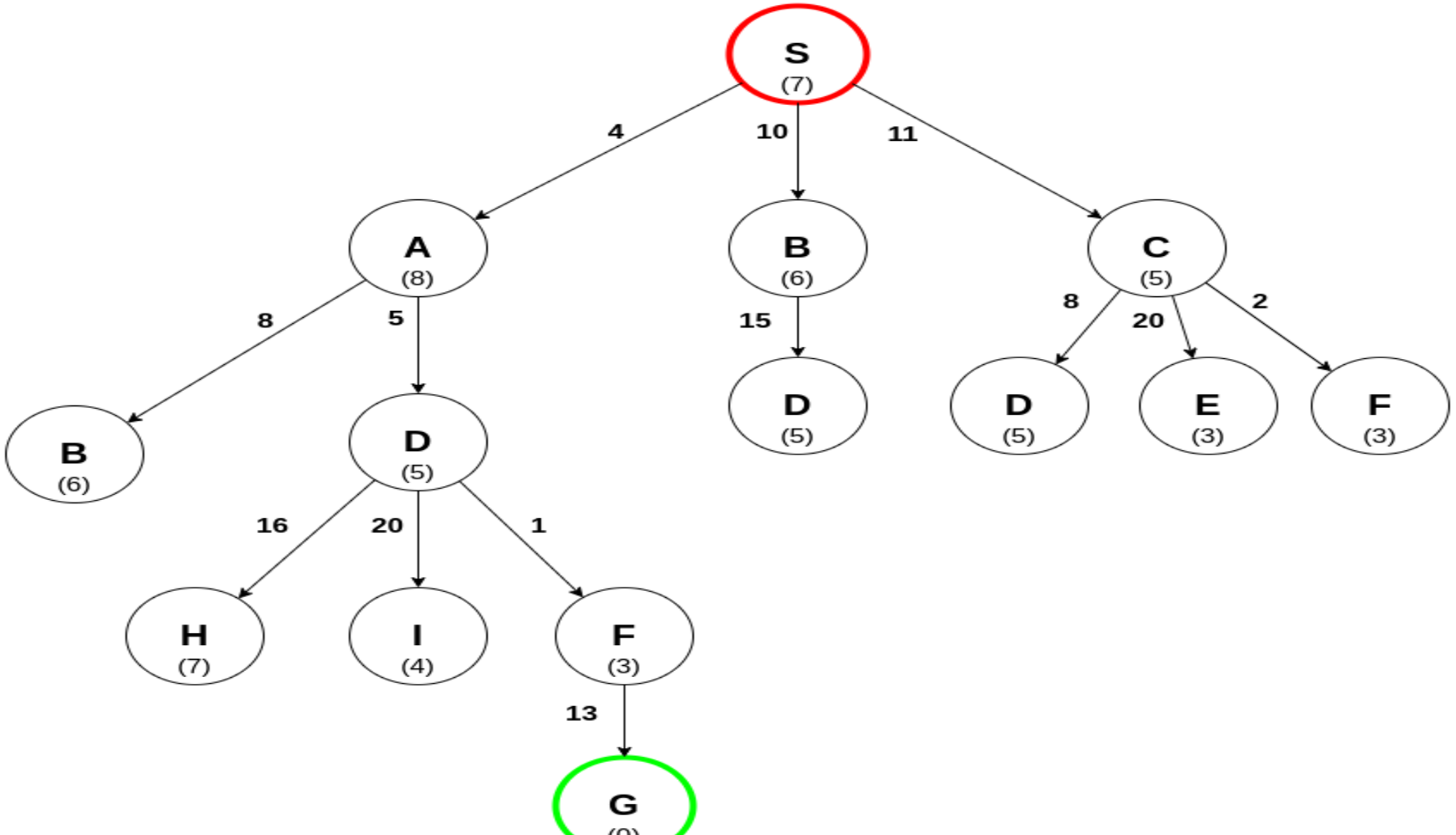


Node[cost]
B[18]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

A* Search Algorithm – Example

- The next node in the open list is again B. However, because B has already been explored, meaning the shortest path to B has been found, it is not explored again, and the algorithm continues to the next candidate.

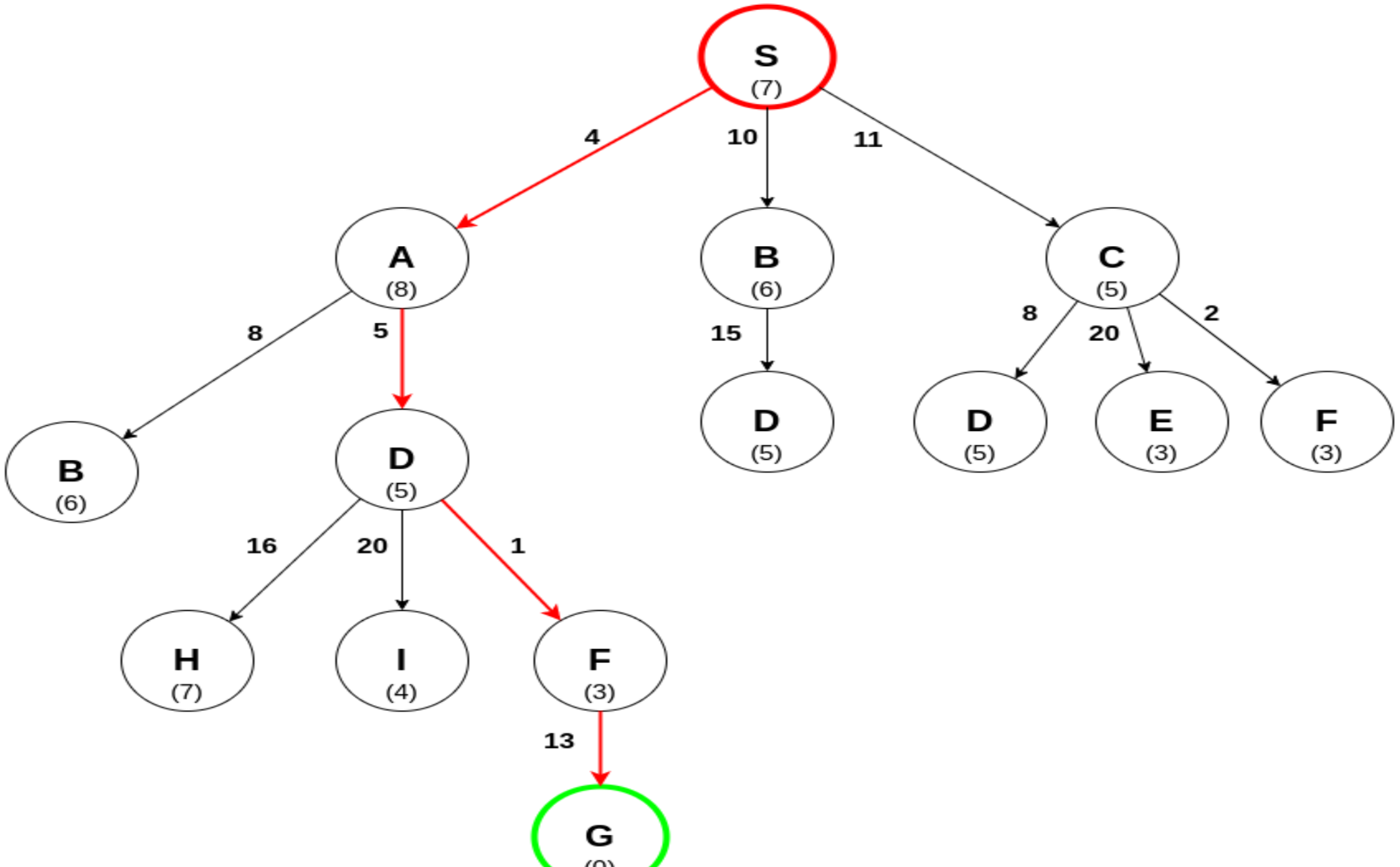


Node[cost]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

A* Search Algorithm – Example

- The next node to be explored is the goal node G, meaning the shortest path to G has been found! The path is constructed by tracing the graph backwards from G to S:



Node[cost]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C
G

Application of Heuristic Search Algorithms

Best First Search	A* Search
• Route planning • Game AI • Web Crawling • Robotics (Exploration) • Information Retrieval	• GPS & Maps • Game Development • Robotics Navigation • Solving Puzzles • Natural Language Processing (NLP) • Logistics & Supply Chain • Network Routing
• Useful when you need **speed over perfection**.	• Use A* when you want the **best (optimal) path** and can afford more computation.


thank
you!