Name: *Sankalp Rajshekhar Chakre*

Roll No: *332004* *TY_B1*

PRN: *22210615*

Assignment: 01

- **Title:**

  Compare different algorithms and evaluate their performance/cost. E.g. depth- first search (DFS) to heuristic algorithms such as Monte Carlo tree search (MCTS. Since all of the studied algorithms converge to a solution from a solvable deal, effectiveness of each approach to be measured by how quickly a solution was reached, and how many nodes were traversed until a solution was reached.

- **Objectives**:

  1. To understand algorithm characteristics.
  2. To Analyse Efficiency and Effectiveness and Compare Algorithm Performance
  3. Identify Strengths and Limitations

- **Theory:**

1. Algorithm Basics

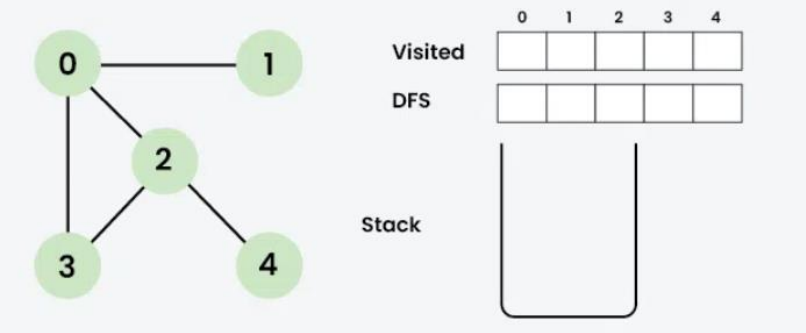
- Depth-First Search (DFS):
  - Search Strategy: DFS explores a branch of the search tree as deeply as possible before backtracking. It uses a stack (either explicitly or implicitly through recursion) to keep track of nodes to be explored.
  - Traversal Pattern: DFS traverses nodes from the root node down to the leaf nodes, exploring each branch fully before moving to the next branch.
  - Data Structures: Stack or recursion stack.

Working of DFS:

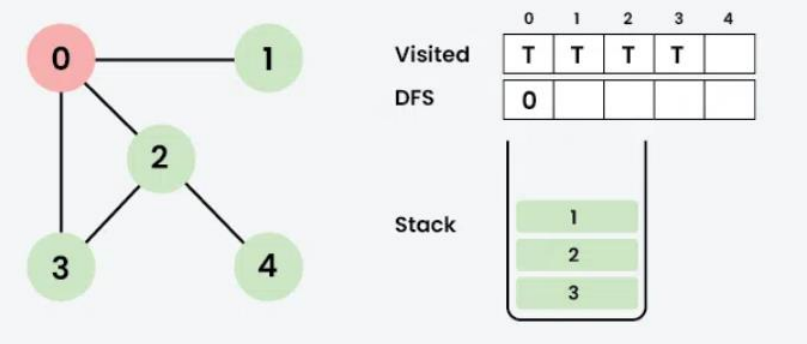
i. First, create a stack with the total number of vertices in the graph.
ii. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
iii. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
iv. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
v. If no vertex is left, go back and pop a vertex from the stack.
vi. Repeat steps 2, 3, and 4 until the stack is empty.

Example:

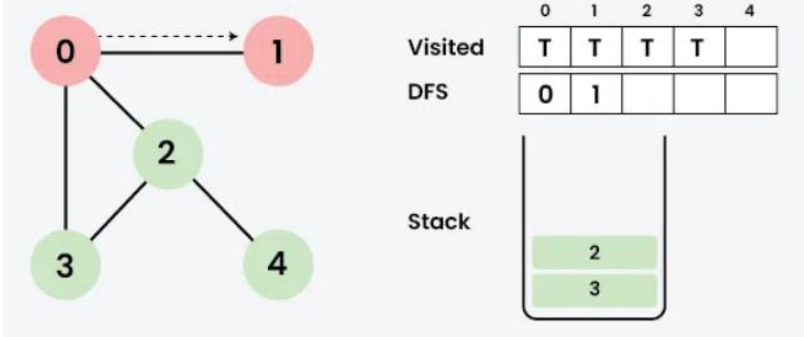
i. Initially stack and visited array are empty.



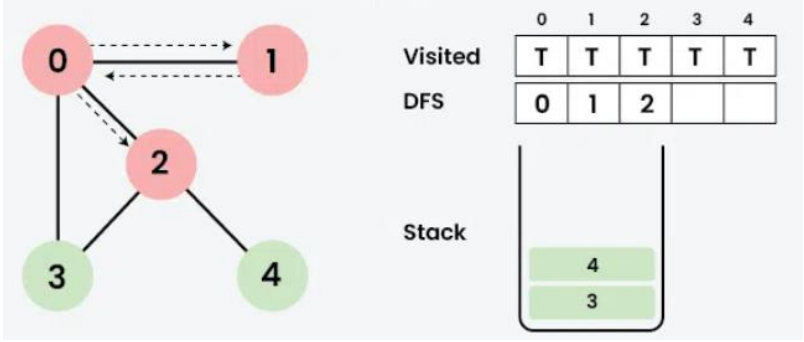
ii. Visited 0 and put its adjacent nodes which are not visited yet into the stack.



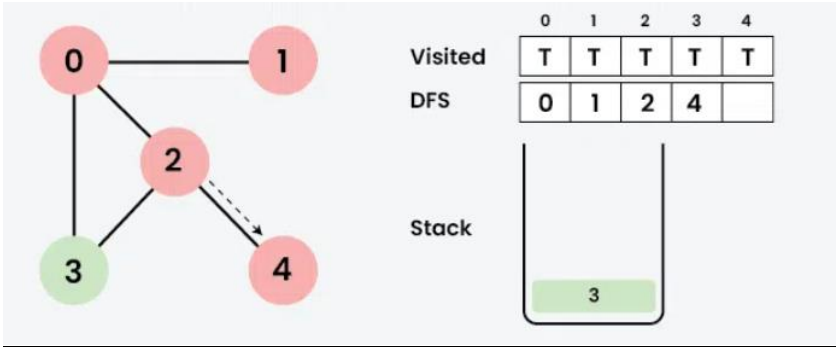
iii. Now, node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



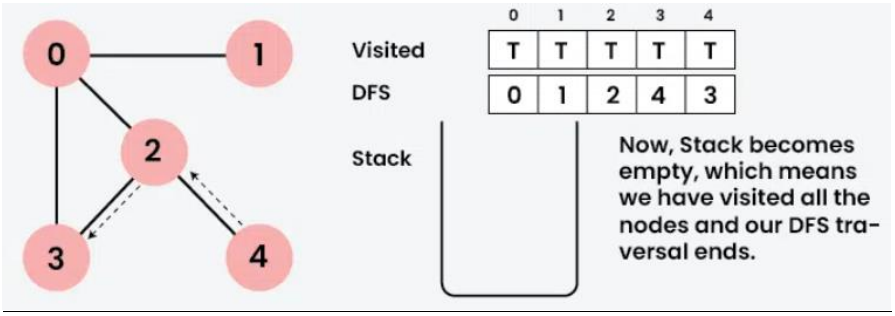
iv. Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes that are not visited in the stack.



v. Now, node 4 at the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



vi. Now, node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes that are not visited in the stack.



- Breadth-First Search (BFS):

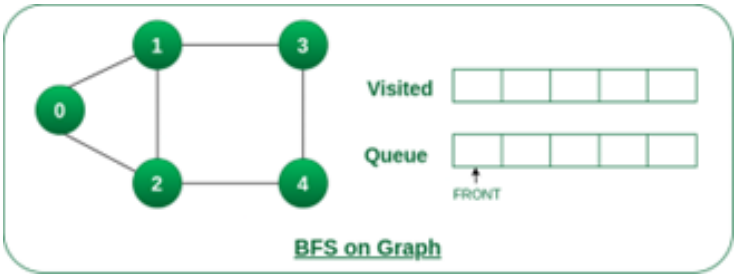
  - Search Strategy: BFS explores the search tree level by level, starting from the root node and moving outward. It visits all the nodes at the present depth level before moving on to nodes at the next depth level. BFS uses a queue to keep track of nodes to be explored.
  - Traversal Pattern: BFS traverses nodes in a breadthwise manner, exploring all nodes at the current level before moving on to the next level. It starts at the root and explores each neighbour fully before moving on to their neighbours.
  - Data Structures: Queue (FIFO - First In, First Out) is used to manage the order of node exploration. Nodes are enqueued as they are discovered and dequeued as they are processed.

Working of BFS:

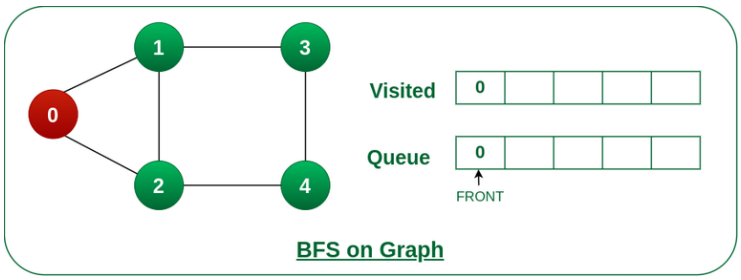
i. Initialize: Create a queue and a set of visited vertices
ii. Start Traversal: Choose a starting vertex, enqueue it, and mark it as visited.
iii. Process Queue: While the queue is not empty, dequeue a vertex, and for each adjacent unvisited vertex, enqueue it and mark it as visited.
iv. Finish: Continue until the queue is empty.

Example:

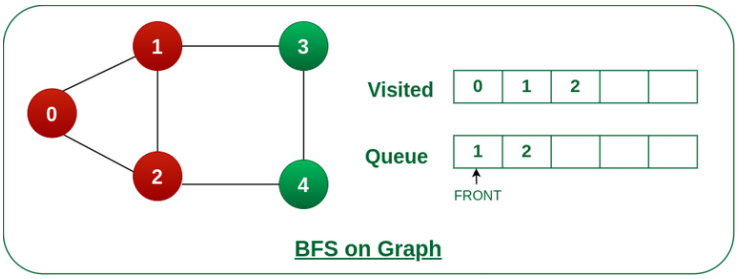
i. Initially queue and visited arrays are empty.



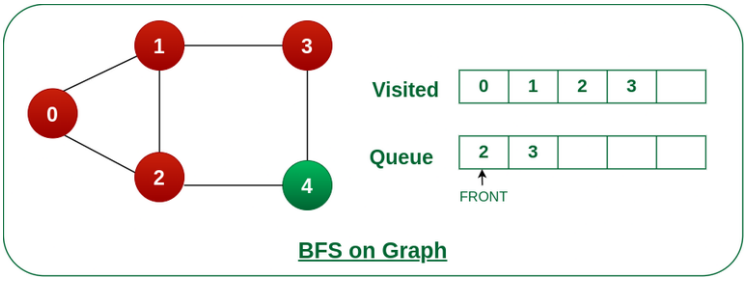
ii. Push 0 into a queue and mark it visited.



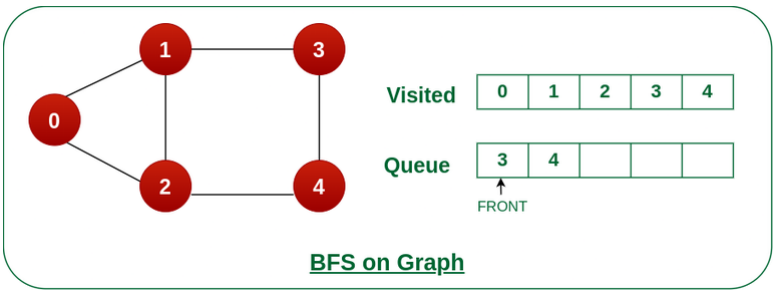
iii. Now Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



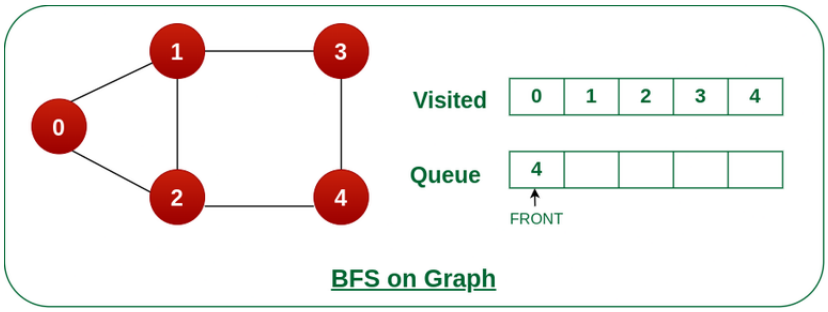
iv. Now, remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



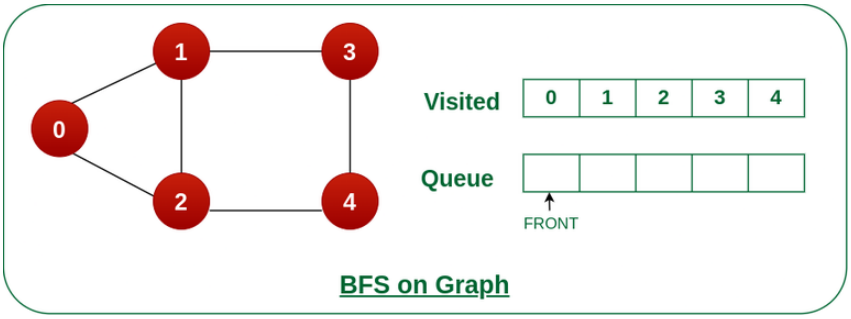
v. Now, remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



vi. Now, remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue. As we can see that every neighbour of node 3 is visited, so move to the next node that are in the front of the queue.



vii. Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue. As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue



Now, Queue becomes empty, So, terminate these process of iteration.

- Monte Carlo Tree Search (MCTS):

  In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. The process of Monte Carlo Tree Search can be broken down into four distinct steps, viz., selection, expansion, simulation and backpropagation. Each of these steps is explained in details below:

  Working of Monte Carlo Tree Search:

  - Selection: In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value.

$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

where;

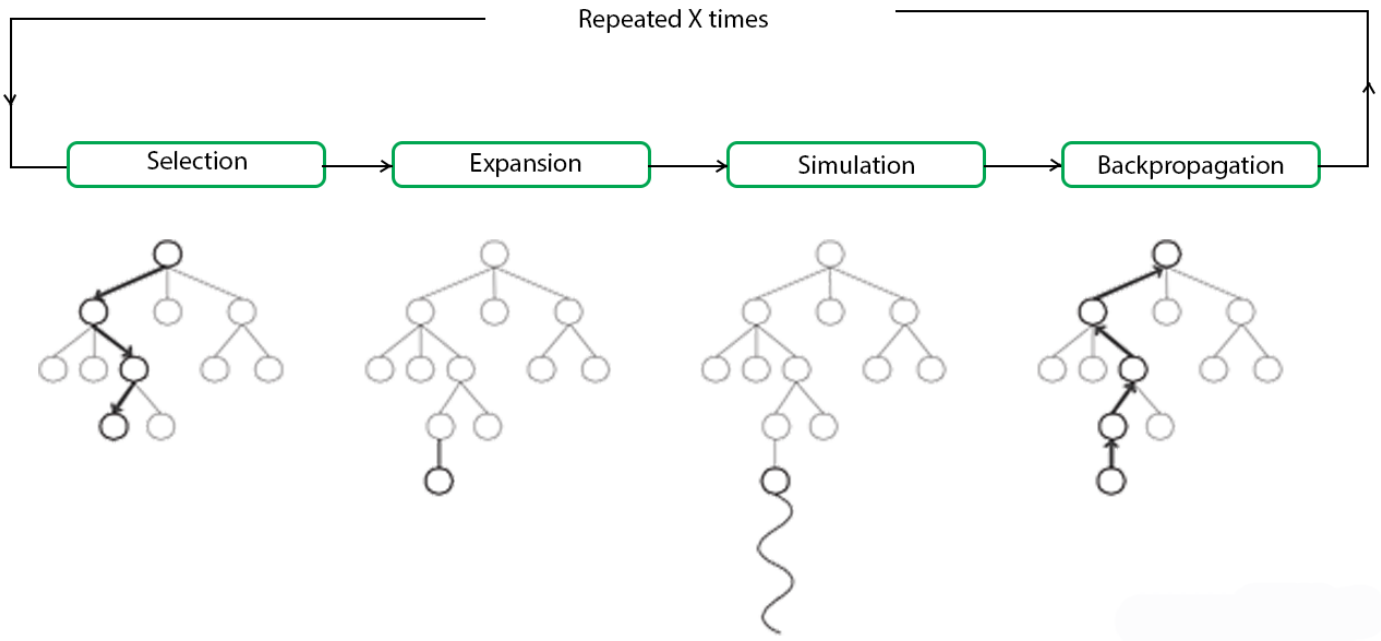
Si = value of a node i

xi = empirical mean of a node i

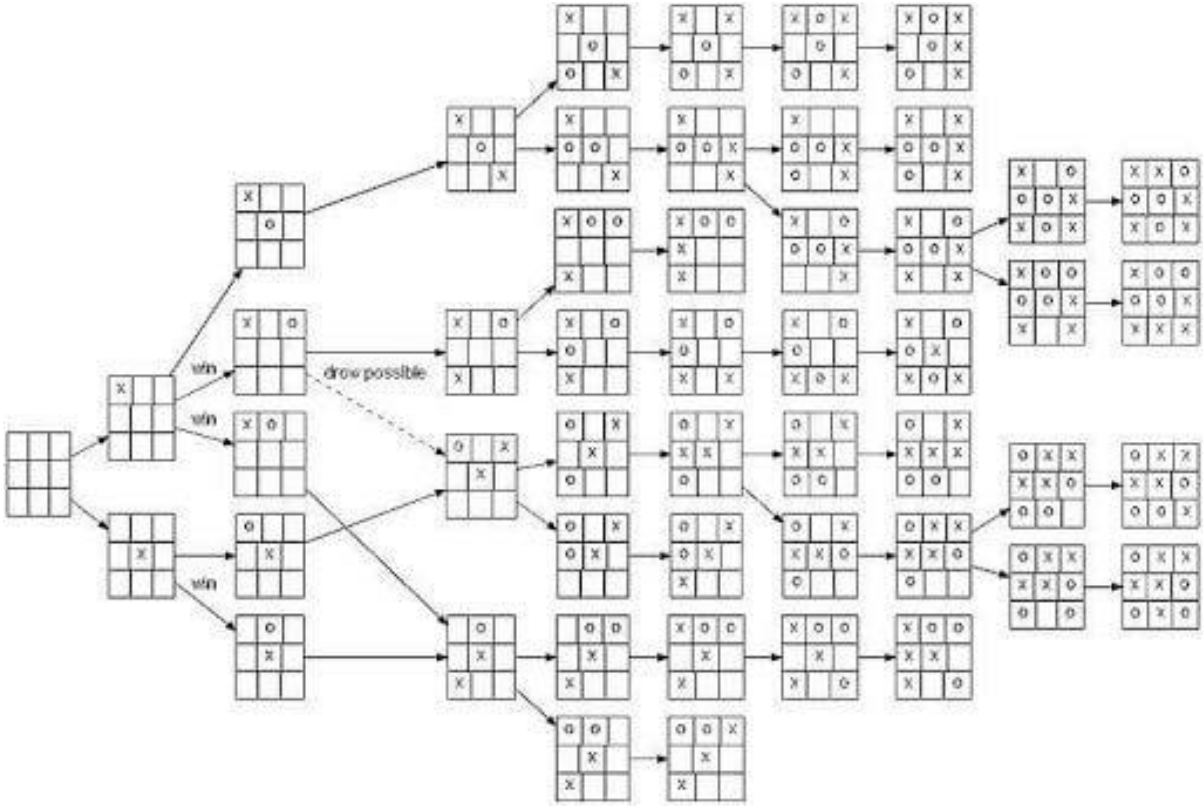
C = a constant

t = total number of simulations

- Expansion: In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.
- Simulation: In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.
- Backpropagation: After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it backpropagates from the new node to the root node. During the process, the number of simulation stored in each node is incremented. Also, if the new node's simulation results in a win, then the number of wins is also incremented.



Example: Game tree of Tic-Tac-Toe



- A* Search:

  - Search Strategy: Uses a priority queue to explore nodes with the lowest estimated total cost (f(n) = g(n) + h(n)). It balances between actual path cost (g(n)) and heuristic estimate (h(n)).
  - Traversal Pattern: Expands nodes based on the lowest f(n) value, focusing on the most promising paths towards the goal.
  - Data Structures:
    a. Priority Queue (Open Set) to manage nodes by their f(n) values.
    b. Closed Set to track fully explored nodes and avoid revisits.

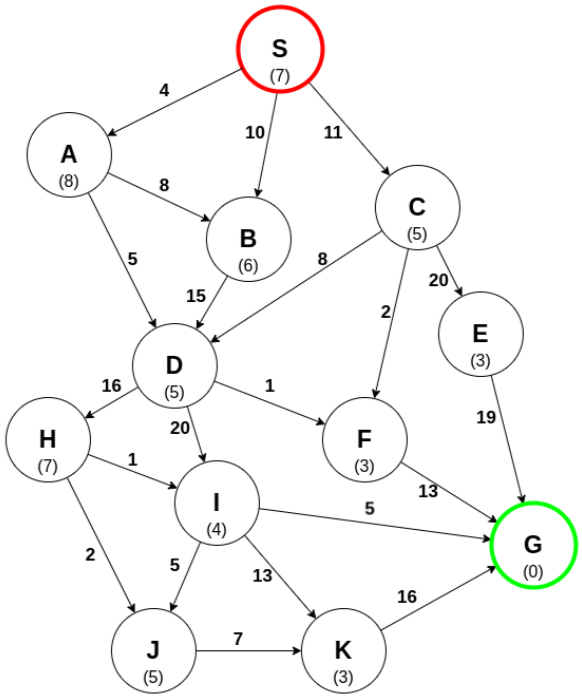
  Working of A*:

  a. Initialize:
     i. Create an openSet (priority queue) with the start node.
     ii. Create a closedSet (empty set) to track evaluated nodes.
     iii. Set gScore for each node (cost from start) and fScore (estimated total cost = gScore + heuristic).
     iv. Initialize gScore of start node to 0 and fScore to the heuristic estimate to the goal.
     v. Use cameFrom to track the path.
  b. Algorithm:
     ✓ While openSet is not empty:
       a. Remove the node with the lowest fScore from openSet (current node).
       b. If the current node is the goal, reconstruct and return the path.
       c. Move current node to closedSet.
       d. For each neighbor of the current node:
          i. Skip if in closedSet.
          ii. Calculate tentative gScore for the neighbor.

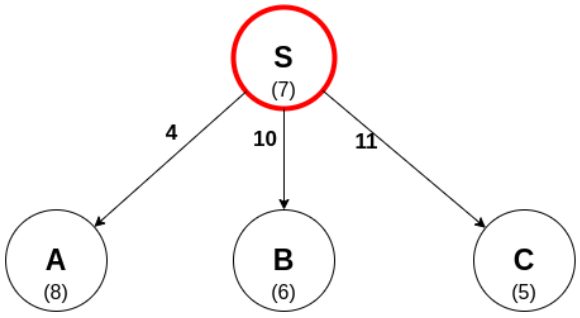
iii. If the neighbor is not in openSet or the new gScore is better:
iv. Update gScore, fScore, and record the path in cameFrom.
v. Add the neighbor to openSet.

c. Finish:
i. If openSet is empty and goal is not reached, no path exists.
ii. Otherwise, reconstruct the path from cameFrom.

Example:



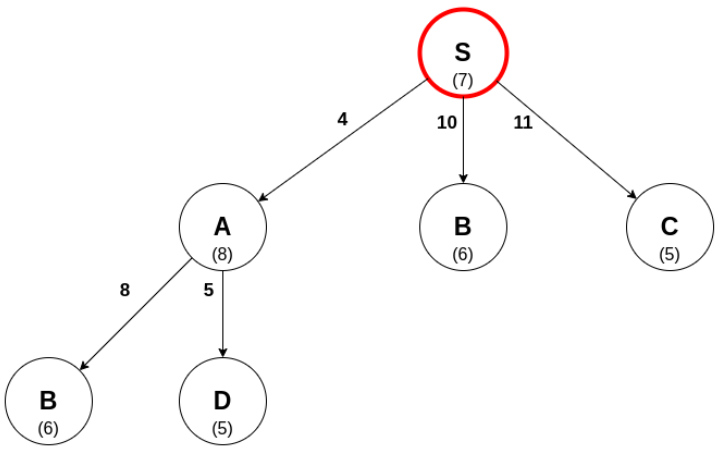
i. Exploring **S**:



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

Closed List
S

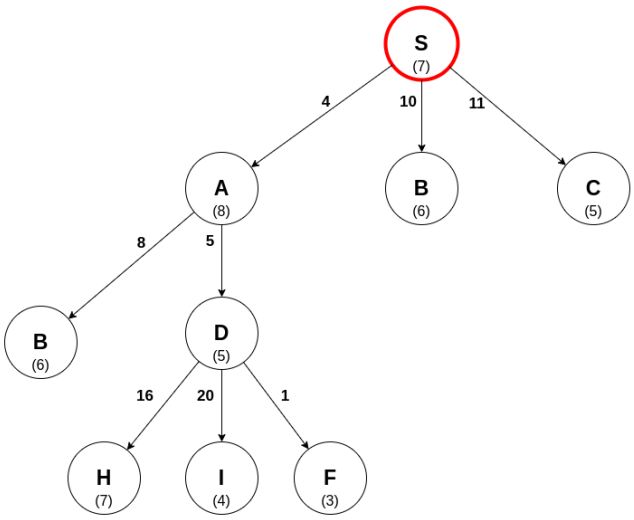
A is the current most promising path, so it is explored next:



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

Closed List
S
A

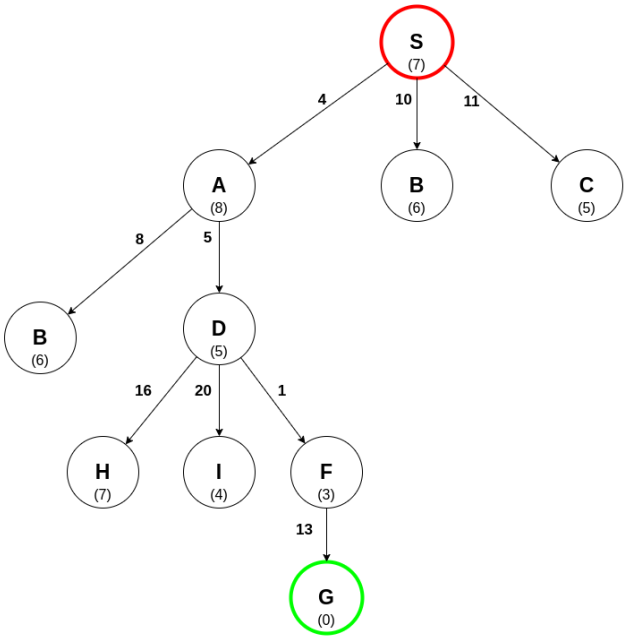
ii. Exploring **D**:



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

Closed List
S
A
D

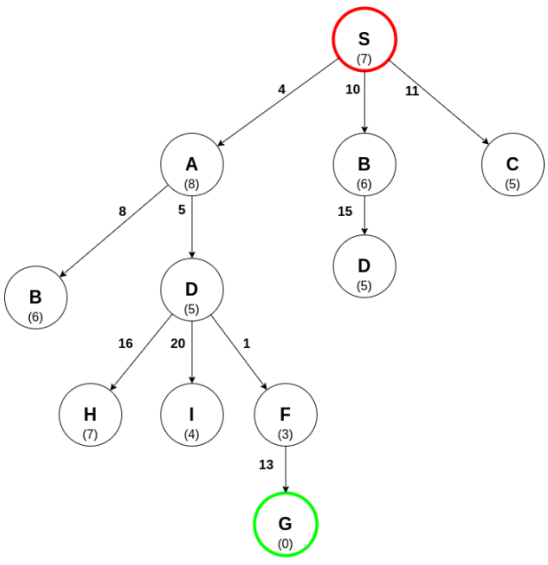
iii. Exploring **F:**



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

Closed List
S
A
D
F

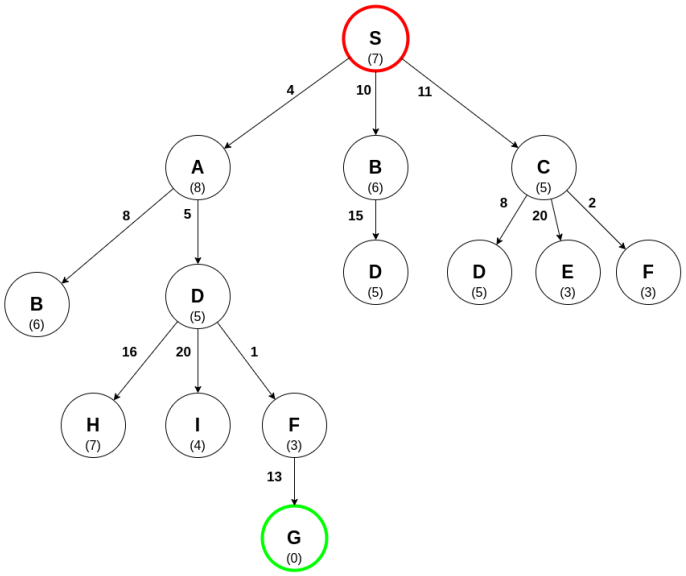
Notice that the goal node **G** has been found. However, it hasn't been explored, so the algorithm continues because there may be a shorter path to G. The node **B** has two entries in the open list: one at a cost of 16 (child of **S**) and one at a cost of 18 (child of **A**). The one with the lowest cost is explored next:



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

Closed List
S
A
D
F
B

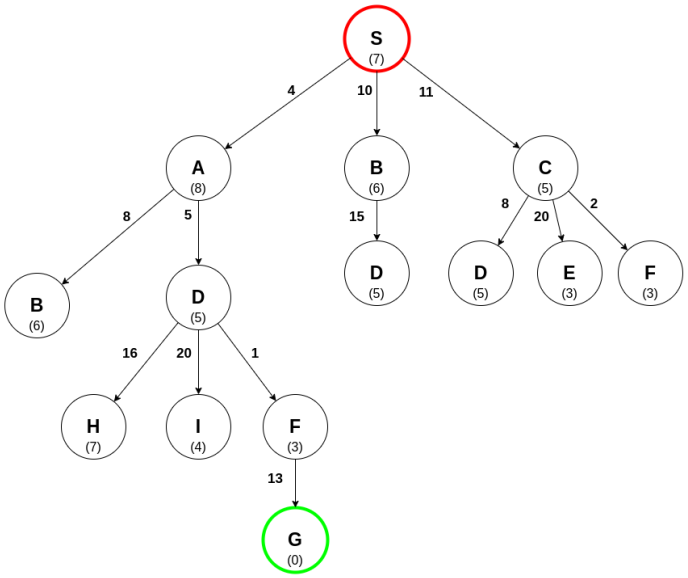
iv. Exploring **C**:



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

Closed List
S
A
D
F
B
C

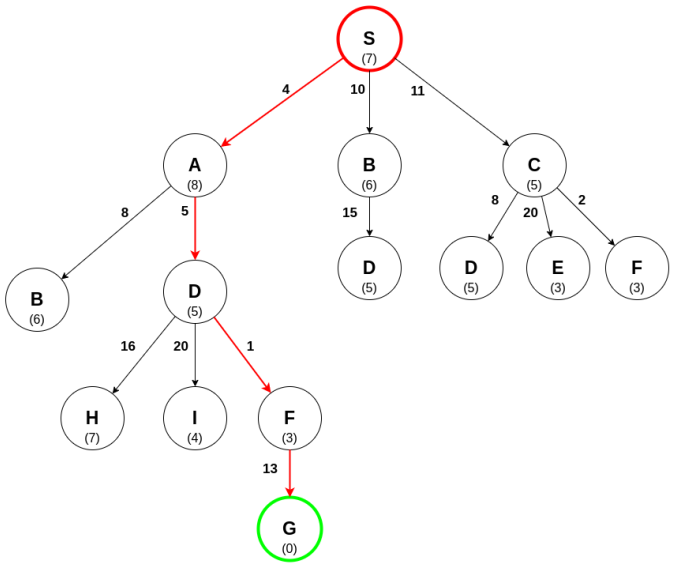
The next node in the open list is again **B**. However, because **B** has already been explored, meaning a shortest path to **B** has been found, it is not explored again and the algorithm continues to the next candidate.



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

Closed List
S
A
D
F
B
C

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 backward from **G** to **S**:



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

Closed List
S
A
D
F
B
C
G

- **Comparison:**

Parameter	DFS	BFS	A*	MCTS
Strategy	Depth-first	Breadth-first	Best-first (heuristic-guided)	Monte Carlo simulation
Completeness	Not guaranteed	Guaranteed (for unweighted graphs)	Guaranteed (with an admissible heuristic)	Not guaranteed (approximates solutions)
Optimality	Not guaranteed	Guaranteed (for unweighted graphs)	Optimal (with an admissible heuristic)	Not guaranteed (depends on simulations)
Time Complexity	O(b^d)	O(b^d)	O(b^d) with heuristic optimization	O(b^d) with simulation depth
Space Complexity	O(b*d)	O(b^d)	O(b^d) with heuristic optimization	O(b*d) or less (depending on simulations)
Memory Usage	Low (depth-controlled)	High (stores all nodes at current level)	High (stores nodes with their scores)	Variable (depends on number of simulations)
Suitability	Simple, good for smaller spaces	Good for unweighted graphs	Effective for finding shortest paths in weighted graphs	Effective for large and complex spaces with uncertainty
Handling Cycles	May revisit nodes (can use visited set)	Handles cycles well	Handles cycles well	Can handle cycles if properly managed

- **Conclusion:**

  DFS is straightforward and memory-efficient for problems with a well-defined and manageable search space. It can be slow and inefficient for complex problems due to its depth-first nature, leading to potential inefficiencies if the solution is not deep or if there are many branches.

  BFS guarantees the shortest path in unweighted graphs by exploring nodes level-by-level. It's complete and straightforward but can consume a lot of memory, especially in large graphs.

  MCTS is well-suited for large and complex search spaces, especially when dealing with uncertainty or incomplete information. It approximates optimal solutions by focusing on promising areas of the search space through simulations, though it may require significant computation time and memory,

  A* algorithm finds the shortest path efficiently by combining actual cost and heuristic estimates, ensuring optimal solutions if the heuristic is admissible. However, it can be memory-intensive due to storing extensive node information.