Experiment No-1

Problem Statement:

Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Theory:

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

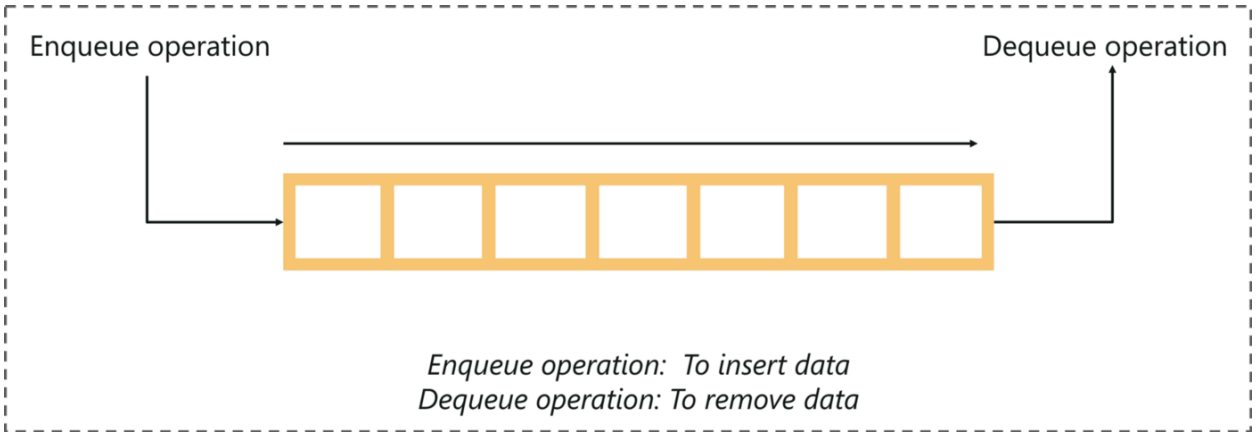
1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

Breadth First Search:

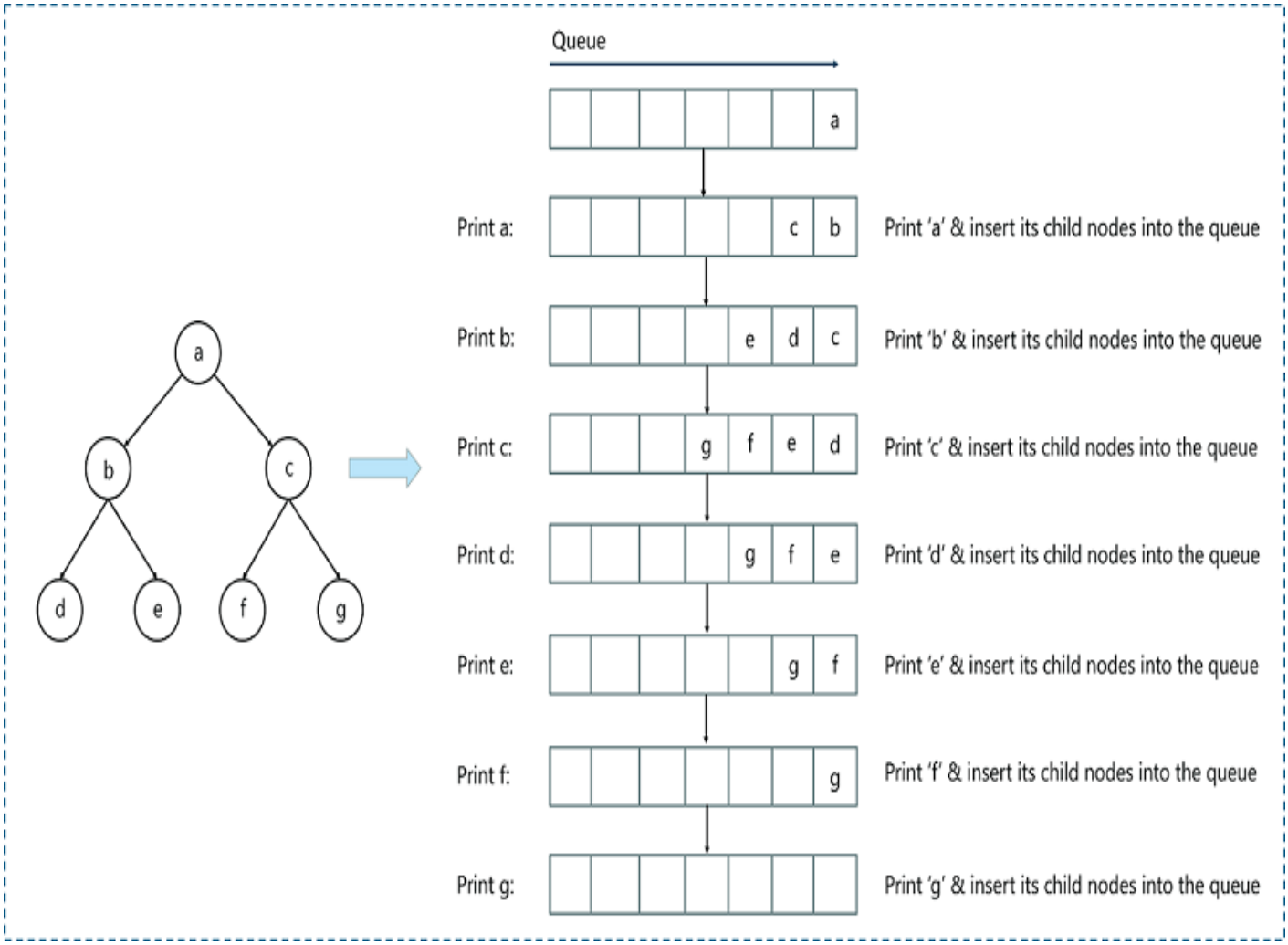
- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor nodes at the current level before moving to nodes of next level.

- Before we get started, you must be familiar with the main data structure involved in the Breadth-First Search algorithm.

- A queue is an abstract data structure that follows the First-In-First-Out methodology (data inserted first will be accessed first). It is open on both ends, where one end is always used to insert data (enqueue) and the other is used to remove data (dequeue)



Example:



The above image depicts the end-to-end process of Breadth-First Search Algorithm. Let me explain this in more depth.

1. Assign 'a' as the root node and insert it into the Queue.

2. Extract node ‘a’ from the queue and insert the child nodes of ‘a’, i.e., ‘b’ and ‘c’.

3. Print node ‘a’.

4. The queue is not empty and has a node ‘b’ and ‘c’. Since ‘b’ is the first node in the queue, let’s extract it and insert the child nodes of ‘b’, i.e., node‘d’ and ‘e’.

5. Repeat these steps until the queue gets empty. Note that the nodes that are already visited should not be added to the queue again.

Properties of BFS

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

  $T(b) = 1+b^2+b^3+\ldots\ldots+ b^d= O(b^d)$

- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Applications of Breadth-First Search Algorithm

1. Crawlers in Search Engines
2. GPS Navigation systems
3. Find the Shortest Path & Minimum Spanning Tree for an unweighted graph
4. Broadcasting
5. Peer to Peer Networking

Depth First Search

- The Depth-First Search is a recursive algorithm that uses the concept of backtracking.
- It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse.
- All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

The DFS algorithm works as follows:

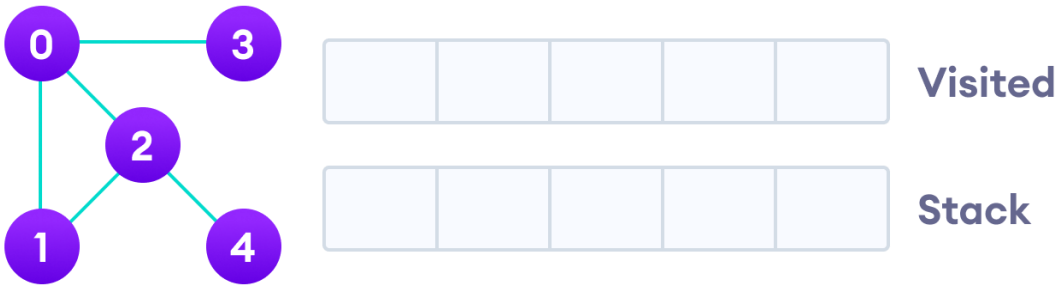
1. Start by putting first node of the graph's vertices on top of a stack

2. Take the top item of the stack and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 and 3 until the stack is empty.

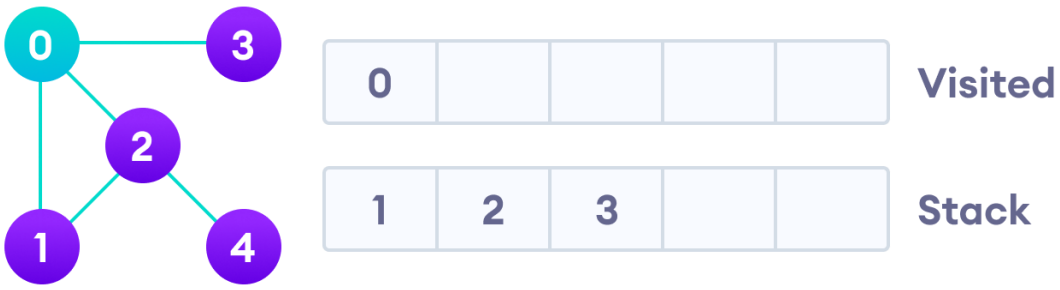
Depth First Search Example



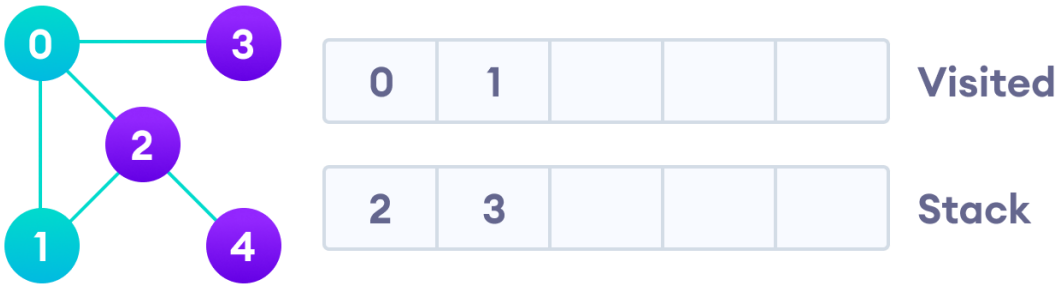
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in

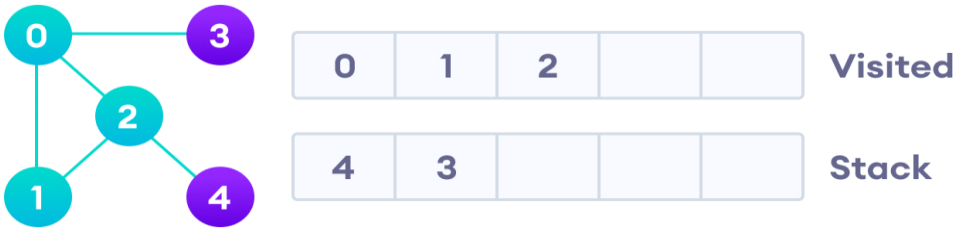
The Visited list and putting all its adjacent vertices in the stack.



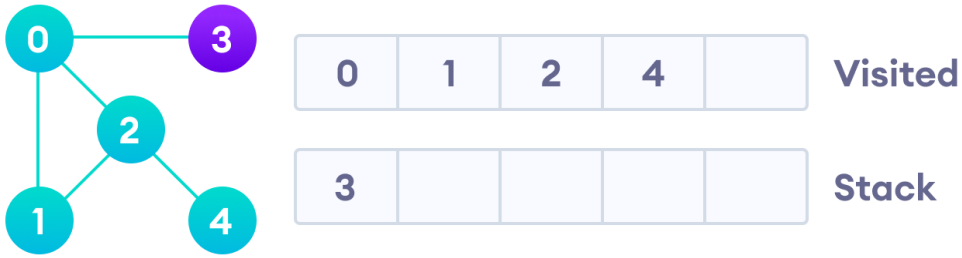
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

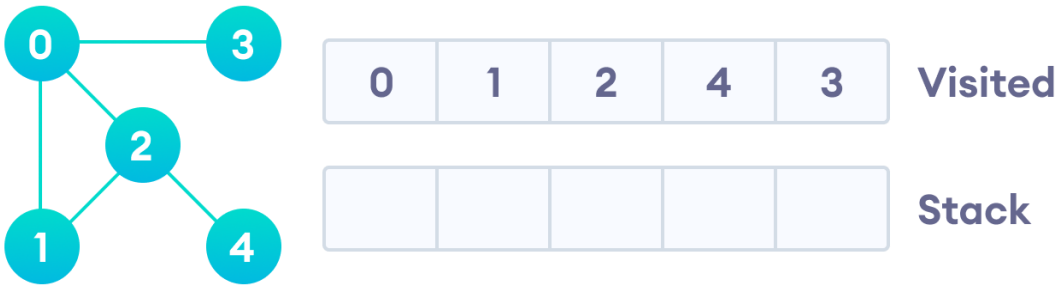


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to
The top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited Adjacent nodes , so we have completed the Depth First traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited Adjacent Nodes, so we have completed the Depth First Traversal of the graph.

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite

3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Properties of DFS

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O (bm)**.

Conclusion:

We have studied Uninformed Search, Breadth first search and Depth First Search.

Experiment No-2

Problem Statement:

Implement A * Algorithm for any game search Problem.

Theory:

The uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.

But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.

This knowledge helps agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

Heuristic function estimates how close a state is to the goal.

It is represented by h (n), and it calculates the cost of an optimal path between the pair of states.

The value of the heuristic function is always positive.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms.

It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list.

In the CLOSED list, it places those nodes which have already expanded.

In the OPEN list, it places nodes which have yet not been expanded.

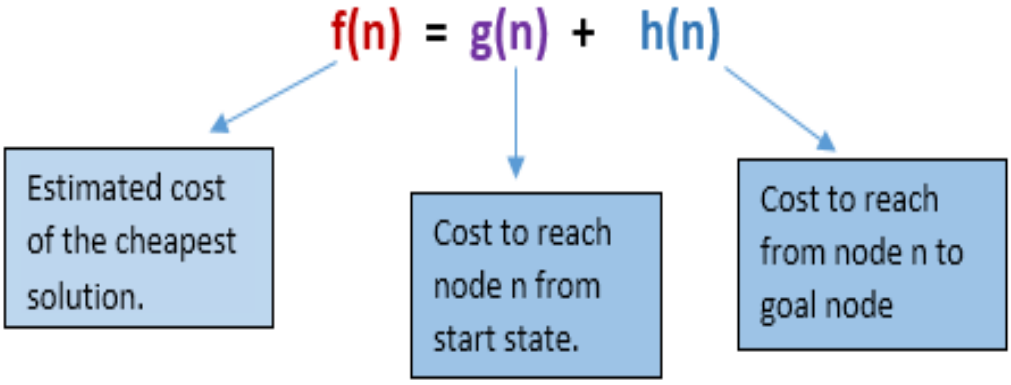
In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

A* Search Algorithm:

- A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g (n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

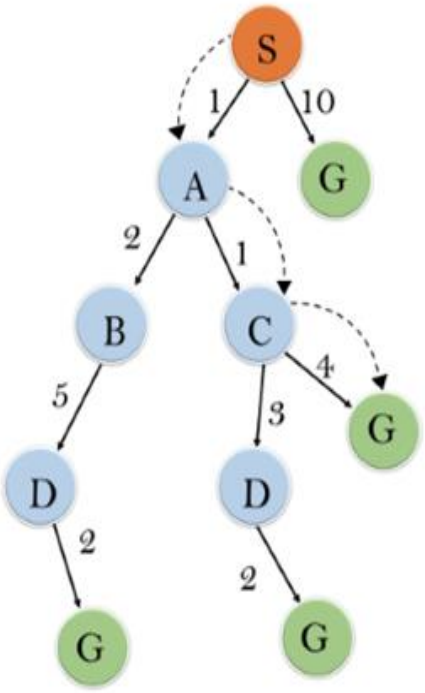
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Initialization: {S, 5}

Iteration 1: Open {(A→4)(G-->10)} Close{(S,5)}

Iteration 2: Open {(G→10)} Close {(S→A, 4)}

Iteration 3: Open {(G→10)(S→A→B,7)(S→A→C,4)} Close{(S→A)}

Iteration 4: Open {(G→10) (S→A→B,7)} Close{(S→A→C,4)}

Iteration 5: Open {(G→10)(S→A→C→D,11)(S→A→C→G, 6)}
Close {(S→A→C, 4)}

Iteration 6: Open {(G→10) (S→A→C→D, 11)}
Close {(S→A→C→G, 6)}

Properties of A* Search →

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O (b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O (b^d)**

Conclusion:

We have Studied the Informed search strategy A* Search with its properties.

Experiment No-3

Problem Statement:

Implement Greedy search algorithm for any of the following application:

I. Selection Sort
II. Minimum Spanning Tree
III. Single-Source Shortest Path Problem
IV. Job Scheduling Problem
V. Prim's Minimal Spanning Tree Algorithm
VI. Kruskal's Minimal Spanning Tree Algorithm
VII. Dijkstra's Minimal Spanning Tree Algorithm

Theory:

What is a Minimum Spanning Tree?

- As we all know, the graph which does not have edges pointing to any direction in a graph is called an undirected graph and the graph always has a path from a vertex to any other vertex.
- A spanning tree is a subgraph of the undirected connected graph where it includes all the nodes of the graph with the minimum possible number of edges.
- Remember, the sub graph should contain each and every node of the original graph.
- If any node is missed out then it is not a spanning tree and also, the spanning tree doesn't contain cycles.
- If the graph has n number of nodes, then the total number of spanning trees created from a complete graph is equal to n^(n-2).
- In a spanning tree, the edges may or may not have weights associated with them. Therefore, the spanning tree in which the sum of edges is minimum as possible then that spanning tree is called the minimum spanning tree.

- One graph can have multiple spanning-trees but it can have only one unique minimum spanning tree.
- There are two different ways to find out the minimum spanning tree from the complete graph i.e Kruskal's algorithm and Prim's algorithm.
- Let us study prim's algorithm in detail below:

What is Prim's Algorithm?

- Prim's algorithm is a minimum spanning tree algorithm which helps to find out the edges of the graph to form the tree including every node with the minimum sum of weights to form the minimum spanning tree.
- Prim's algorithm starts with the single source node and later explores all the adjacent nodes of the source node with all the connecting edges.
- While we are exploring the graphs, we will choose the edges with the minimum weight and those which cannot cause the cycles in the graph.

Prim's Algorithm for Minimum Spanning Tree

Prim's algorithm basically follows the greedy algorithm approach to find the optimal solution.

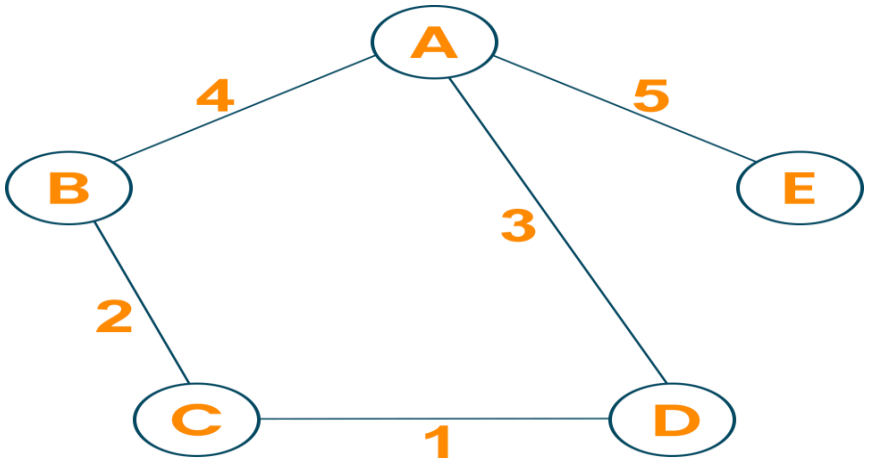
To find the minimum spanning tree using prim's algorithm, we will choose a source node and keep adding the edges with the lowest weight.

The algorithm is as given below:

- Initialize the algorithm by choosing the source vertex
- Find the minimum weight edge connected to the source node and another node and add it to the tree
- Keep repeating this process until we find the minimum spanning tree

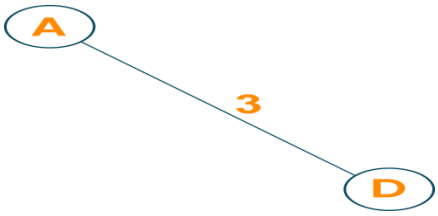
Example

Let us consider the below-weighted graph

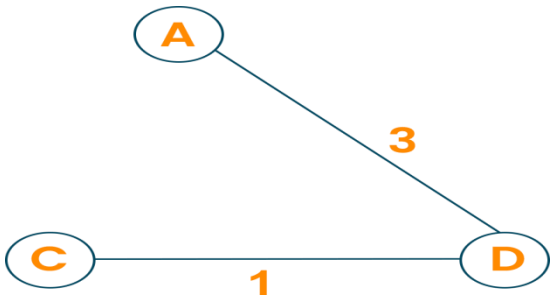


Later we will consider the source vertex to initialize the algorithm

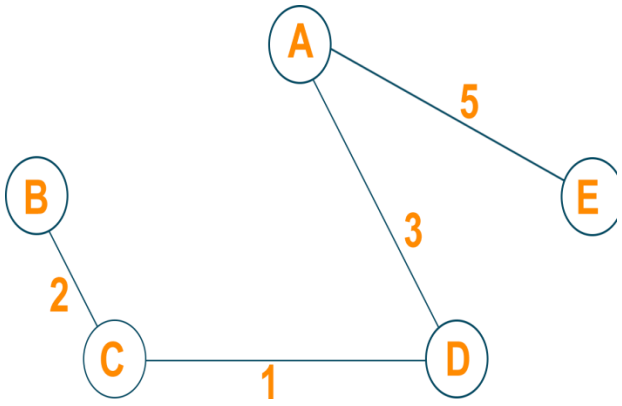
Now, we will choose the shortest weight edge from the source vertex and add it to finding the spanning tree.



Then, choose the next nearest node connected with the minimum edge and add it to the solution. If there are multiple choices then choose anyone.



Continue the steps until all nodes are included and we find the minimum spanning tree.



Applications

- Prim's algorithm is used in network design
- It is used in network cycles and rail tracks connecting all the cities
- Prim's algorithm is used in laying cables of electrical wiring
- Prim's algorithm is used in irrigation channels and placing microwave towers
- It is used in cluster analysis

- Prim's algorithm is used in gaming development and cognitive science
- Path finding algorithms in artificial intelligence and traveling salesman problems make use of prim's algorithm.

Conclusion

We have studied the Prim's Search algorithm for minimum spanning tree.