**EXPERIMENT NO. 1**

**Title:** Implementation of Breadth First Search (BFS) using Python in Online Compiler Programiz

**Aim:** To implement the Breadth-First Search (BFS) algorithm using Python.

**Theory:**

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm used to systematically explore nodes and edges in a graph. It starts at a selected node (often called the 'root') and explores all neighboring nodes at the current depth level before moving on to nodes at the next depth level.

Breadth-First Search (BFS) is a graph traversal algorithm that explores all nodes at the present depth level before moving on to nodes at the next depth level. BFS is implemented using a queue data structure. It is used in shortest path algorithms, network broadcasting, and AI problem-solving techniques.

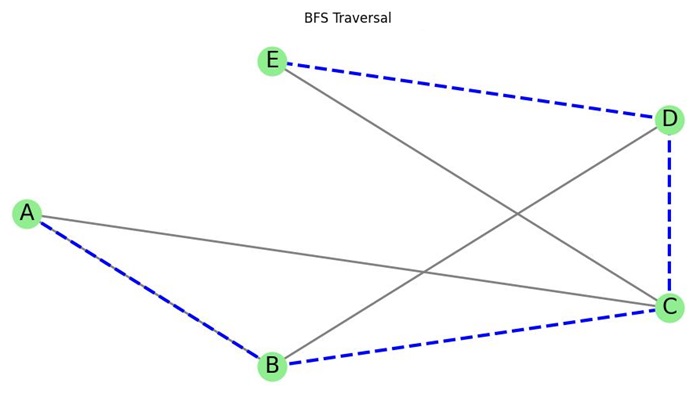
It is particularly useful in scenarios such as finding the shortest path in an unweighted graph, determining the shortest distance between nodes, and exploring the reachability of nodes in a graph.

**BFS Algorithm**

**The BFS algorithm follows these basic steps −**

* **Initialize a queue and mark the starting node as visited.**
* **While the queue is not empty, dequeue a node from the queue.**
* **Visit all unvisited neighboring nodes of the current node, mark them as visited, and enqueue them.**
* **Repeat this process until all reachable nodes are visited.**

**Let us look at an example to better understand BFS. Consider the following graph with nodes labeled A, B, C, D, and E −**

****

**We will perform BFS starting from node A. The traversal proceeds as follows −**

* **Start at node A. Mark it as visited and enqueue it.**
* **Dequeue A, visit its neighbors (B, C), mark them as visited, and enqueue them.**
* **Dequeue B, visit its neighbor (D), mark it as visited, and enqueue it.**
* **Dequeue C, visit its neighbor (E), mark it as visited, and enqueue it.**
* **Dequeue D and E. No further unvisited neighbors are found.**

**Thus, the BFS traversal order is A → B → C → D → E.**

**Applications of BFS**

**BFS is used in various real-world applications, such as −**

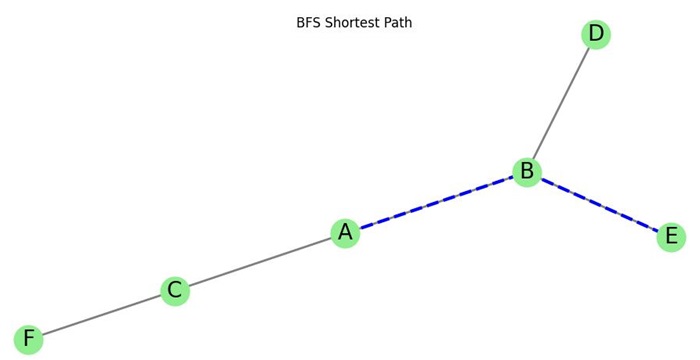
* **Shortest Path in Unweighted Graphs: BFS is often used to find the shortest path between two nodes in an unweighted graph. Since BFS explores level by level, the first time it reaches a node, it does so via the shortest path.**
* **Connected Components: BFS can help identify connected components in an undirected graph. By performing BFS from an unvisited node and marking all reachable nodes, you can determine the entire connected component.**
* **Level Order Traversal in Trees: BFS can be used to traverse trees in level order, ensuring that all nodes at each level are visited before moving to the next level.**
* **Network Broadcasting: In networking, BFS can be used to model the process of broadcasting messages from a central node to all other nodes in a network.**

**BFS in Finding the Shortest Path**

**One of the most important applications of BFS is finding the shortest path in an unweighted graph. Since BFS explores all nodes at a given depth before moving on to the next level, the first time it reaches a node, it does so via the shortest path.**

**For example, given an unweighted graph and two nodes, A and E, BFS can be used to determine the shortest path from A to E. The algorithm proceeds level by level, ensuring that the first time it reaches node E, it does so via the minimum number of edges.**

**Here is an example of finding the shortest path from node A to node E in an unweighted graph −**

****

**The shortest path from A to E is A → B → E, with two edges traversed. BFS ensures that this is the shortest path because it explores the graph level by level, visiting nodes closest to the starting point first.**

**Complexity of BFS**

**The time complexity of BFS is O(V + E), where V is the number of vertices and E is the number of edges in the graph. This is because BFS processes each vertex and edge exactly once.**

**The space complexity is O(V), as BFS requires storage for the queue and a visited list, both of which require space proportional to the number of vertices in the graph.**

**BFS performs efficiently on sparse and dense graphs, making it suitable for various graph-related problems.**

**BFS with Edge Weights**

**While BFS is not suitable for graphs with weighted edges when looking for the shortest path, it can still be used to find the shortest path in graphs where all edges have equal weight. In such cases, BFS guarantees finding the shortest path because it explores nodes level by level.**

**For weighted graphs, algorithms like Dijkstra's algorithm are more appropriate as they account for varying edge weights.**

**BFS Pseudocode**

**The pseudocode for BFS in python goes as below:**

**create a queue Q**

**mark v as visited and put v into Q**

**while Q is non-empty**

**remove the head u of Q**

**mark and enqueue all (unvisited) neighbors of u**

**Program :**

**(note :Attach printout of this program)**

**# Define the graph as a dictionary where keys are nodes and values are lists of connected nodes**

**graph = {**

**'2' : [], # Node '2' has no neighbors**

**'3' : ['2', '4'], # Node '3' is connected to '2' and '4'**

**'4' : ['8'], # Node '4' is connected to '8'**

**'5' : ['3','7'], # Node '5' is connected to '3' and '7'**

**'7' : ['8'], # Node '7' is connected to '8'**

**'8' : [] # Node '8' has no neighbors**

**}**

**visited = [] # List to keep track of visited nodes during BFS**

**queue = [] # Initialize an empty queue to hold nodes to be processed in BFS**

**# Function to perform Breadth-First Search (BFS)**

**def bfs(visited, graph, node):**

**visited.append(node) # Mark the starting node as visited**

**queue.append(node) # Add the starting node to the queue**

**while queue: # Loop until all nodes have been processed**

**m = queue.pop(0) # Pop the first node from the queue for processing**

**print(m, end = " ") # Print the current node (in BFS order)**

**# Loop through all the neighbors of the current node**

**for i in graph[m]:**

**if i not in visited: # If the neighbor hasn't been visited**

**visited.append(i) # Mark the neighbor as visited**

**queue.append(i) # Add the neighbor to the queue to be processed later**

**# Driver Code - Call the BFS function starting from node '5'**

**print("Following is the Breadth-First Search")**

**bfs(visited, graph, '5')**

**Output:**

**Conclusion:**  
BFS successfully traverses a graph level by level, ensuring all nodes at a given depth are visited before proceeding to the next depth.