**Report:**

**Title: Implementing BFS Algorithm for Finding Shortest Path in a Graph Introduction:**

This report outlines the steps taken to implement the Breadth-First Search (BFS) algorithm to find the shortest path between two nodes in a graph. The report explains each function used in the algorithm and the problems faced during the implementation.

**Steps:**

1. Reading the Graph:

The first step was to read the graph data from a file using the read\_graph() function. The function reads the graph data from the input file and stores it in a dictionary format, where each node is represented as a key, and its connected nodes are stored in a list.

1. Implementing BFS Algorithm:

The BFS algorithm is implemented using the shortest\_path() function. The function takes three arguments, graph, node1, and node2, where the graph is a dictionary of nodes and their edges, and node1 and node2 are the starting and ending nodes. The function returns the shortest path between the two nodes.

1. Breadth-First Search:

The BFS algorithm works by exploring all the neighboring nodes of the current node before moving on to the next level. The shortest\_path() function uses this approach to explore all the possible paths from the starting node to the ending node.

1. Path Finding:

The shortest\_path() function keeps track of the previous nodes to avoid visiting them again. It also maintains a list of paths to explore, starting from the starting node. The function creates new paths by copying the current path and appending the next node. If the ending node is found in the next nodes, the function returns the completed path.

1. Time Around a Day.

**Functions:**

1. read\_graph(fp):

This function takes a file path as input and returns a dictionary of nodes and their edges.

1. shortest\_path(graph, node1, node2):

This function takes a graph, starting node, and ending node as input and returns the shortest path between them.

**Problems Faced:**

The major challenge faced during the implementation was handling large graphs. The BFS algorithm has a time complexity of O(V+E), where V is the number of nodes, and E is the number of edges in the graph. Hence, for large graphs, the algorithm may take a lot of time to execute. To overcome this challenge, we used various optimization techniques, such as using sets to keep track of previously visited nodes and reducing redundant computations.

**Mapper function:**

Reads each line of the input file from standard input.

Removes leading and trailing whitespaces from the line.

Skips the lines starting with "#" character.

Splits the line into fields separated by ",".

Creates a string containing the source and target nodes separated by "\t" and the distance as the value.

Emits the string as a key-value pair where the key is the source and target nodes separated by "\t" and the value is 1.

**Reducer function:**

Initializes a variable for the current node and a dictionary for the distances from the current node to other nodes.

Reads each line from standard input and removes leading and trailing whitespaces.

Splits the line into fields separated by "\t".

Parses the node and edges information from the fields.

Checks if the current node is the same as the parsed node. If not, prints the distances for the previous node.

Initializes the distances dictionary for the current node.

Loops over the edges and updates the distances dictionary with the minimum distance to each target node.

Prints the distances for the final node after processing all the input lines.

Time taken : 5 mins