Task-1

a)The code reads an input file, assuming it contains data about a weighted graph. It extracts the number of nodes (N) and edges (M) from the first line. It initializes an adjacency matrix (N+1 x N+1) with zeros. It then populates this matrix based on the edge information read from the input file, storing the weights of edges between nodes. It prints the adjacency matrix to the console. Finally, it writes the adjacency matrix to an output file. Each row in the matrix represents a node, and elements in each row denote edge weights to other nodes, printing the resulting matrix in both the console and an output file.

b) The code reads from an input file "Input1(b).txt" containing graph information. It extracts the number of nodes (N) and edges (M) from the first line. It initializes an empty adjacency list, assigning an empty list to each node. Then, it populates this adjacency list with edge information, storing tuples containing the adjacent node and edge weight for each node. The code prints the adjacency list to the console, displaying each node with its associated edges and weights. Additionally, it reads the contents of an output file "Output1(b).txt" into a list named `output_lines` for further processing or comparison.

Task-2

This code utilizes BFS (Breadth-First Search) to traverse a graph represented as an adjacency list. It reads from an input file ("Input2.txt") containing information about a graph's nodes, edges, and connections. It constructs the graph as an adjacency list, performs a BFS starting from node 1, and writes the resulting traversal path to an output file ("Output2.txt"). The BFS function explores nodes level by level, storing the traversal order in `bfs_result`. Finally, it closes both the input and output files after the operations are completed.

Task-3

This code implements Depth-First Search (DFS) to traverse a graph represented as an adjacency list. It reads from an input file ("Input3.txt") containing information about a graph's nodes, edges, and connections. The code constructs the graph using the adjacency list representation, performs DFS starting from node 1, and stores the traversal path in `dfs_path`. The DFS algorithm explores as far as possible along each branch before backtracking. It prints the nodes visited in the DFS traversal. Additionally, it reads the expected output from an output file ("Output3.txt") to compare against the computed DFS traversal path.

Task-4

This code checks for cycles within a directed graph using Depth-First Search (DFS). It defines a function `has_cycle` that employs a modified DFS approach with recursion stacks to identify cycles. The code reads an input file ("Input4.txt") to construct an adjacency list representing the graph. It then utilizes the `has_cycle` function to determine if the graph contains a cycle.

Finally, it reads the expected output from an output file ("Output4.txt") and compares it against the computed result. If the graph has a cycle and the expected output is "YES" or if the graph doesn't have a cycle and the expected output is "NO," it prints the corresponding result. If there's a mismatch between the computed and expected output, it indicates a discrepancy.

Task-5

This code employs Breadth-First Search (BFS) to determine the shortest path from node 1 to a specified destination node D in an undirected graph. It reads an input file ("Input5.txt") containing details about the graph, constructs an adjacency list representation, and utilizes the `bfs_shortest_path` function to find the shortest path from node 1 to the destination node D. The function returns both the time taken to traverse the shortest path and the nodes visited along the path.

After the BFS traversal, the code prints the time taken and the nodes in the shortest path. Additionally, it reads the expected time and path from an output file ("Output5.txt") for comparison against the computed shortest path and time taken.

Task-6

This code aims to find the maximum number of diamonds a person can collect in a grid by moving only up, down, left, or right (not diagonally). It defines a depth-first search (DFS) function `dfs_max_diamonds` to explore paths within the grid. The algorithm considers each cell in the grid as a potential starting point, checks for paths that collect diamonds (denoted as 'D') while avoiding obstacles ('#'), and calculates the maximum number of diamonds that can be collected.

The code reads from an input file ("Input6.txt") containing the grid layout and dimensions, initializes the grid, and executes the DFS algorithm for each cell that represents a valid starting point. Finally, it prints the maximum number of diamonds collected. It also reads the expected maximum diamond count from an output file ("Output6.txt") for comparison.