DAA Lab Assignment 8
Devam Desai IIT2022035

Q1

1.1) Code:

```
#include <bits/stdc++.h>
using namespace std;

void bfs(vector<vector<int>>& graph, int S, vector<int>& par, vector<int>& dist) {
        queue<int> q;
        dist[S] = 0;
        q.push(S);

        while (!q.empty()) {
        int node = q.front();
        q.pop();

        for (int neighbour : graph[node]) {
        if (dist[neighbour] == 1e9) {
        par[neighbour] = node;
        dist[neighbour] = dist[node] + 1;
        q.push(neighbour);
        }
        }
        }
}

void printShortestDistance(vector<vector<int>>& graph, int S, int D, int V) {
        vector<int> par(V, -1);
        vector<int> dist(V, 1e9);

        bfs(graph, S, par, dist);

        if (dist[D] == 1e9) {
        cout << "Source and Destination are not connected";
        return;
        }

        vector<int> path;
        int currentNode = D;
        path.push_back(D);
        while (par[currentNode] != -1) {
        path.push_back(par[currentNode]);
        currentNode = par[currentNode];
        }
```

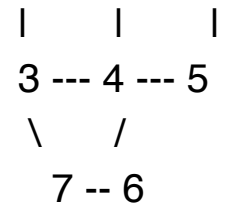
```
        for (int i = path.size() - 1; i >= 0; i--)
        cout << path[i] << " ";
}

int main() {
        int V = 8; // Number of vertices
        int S = 2, D = 6; // Source and Destination vertices

        // Edge list
        vector<vector<int>> edges = { { 0, 1 }, { 1, 2 }, { 0, 3 }, { 3, 4 },
                                { 4, 7 }, { 3, 7 }, { 6, 7 }, { 4, 5 },
                                { 4, 6 }, { 5, 6 } };

        vector<vector<int>> graph(V);
        for (auto edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
        }

        printShortestDistance(graph, S, D, V);
        return 0;
}
```

Step-by-Step Explanation of BFS:

1. **Initialization**: BFS starts by selecting a source node to begin the traversal. This source node is placed in a queue and marked as visited.
2. **Exploration of Neighbours**: BFS explores all the immediate neighbours of the source node before moving on to their neighbours. It explores nodes at each level of the graph before moving deeper.
3. **Marking Visited Nodes**: As BFS explores each node, it marks them as visited to avoid revisiting them again.
4. **Queue Data Structure**: BFS uses a queue data structure to maintain the order of nodes to be visited. Nodes are added to the back of the queue and explored in the order they were added (FIFO - First-In-First-Out).
5. **Level Order Traversal**: BFS visits nodes level by level, starting from the source node. This ensures that it finds the shortest path to each node.
6. **Termination**: BFS continues exploring nodes until the queue becomes empty, indicating that all reachable nodes have been visited.

Example:

0 --- 1 --- 2



Following these steps, BFS will explore all reachable nodes from the source node 0 in level order: 0 -> 1 -> 3 -> 2 -> 4 -> 7 -> 5 -> 6. This demonstrates how BFS works and finds the shortest path in an unweighted graph.

1.2) Time Complexity of BFS for Finding Shortest Path:

- **Time Complexity**

In an unweighted graph, BFS has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. This complexity arises because BFS traverses each vertex once and each edge once.

- **Comparison with Other Algorithms**:

BFS is often the algorithm of choice for finding the shortest path in unweighted graphs due to its simplicity and efficiency. In weighted graphs, where edges have different weights, algorithms like Dijkstra's or Bellman-Ford are used for finding the shortest path. These algorithms have a higher time complexity compared to BFS because they consider the weight of edges.

1.3) Code:

```
#include <bits/stdc++.h>
using namespace std;

void dfs(vector<vector<int>>& graph, int node, int dest, vector<bool>& visited, vector<int>& path,
vector<int>& shortestPath) {
        visited[node] = true;
        path.push_back(node);

        if (node == dest) {
        if (shortestPath.empty() || path.size() < shortestPath.size()) {
        shortestPath = path;
        }
        } else {
        for (int neighbour : graph[node]) {
        if (!visited[neighbour]) {
```

```
            dfs(graph, neighbour, dest, visited, path, shortestPath);
            }
            }
            }

            path.pop_back();
            visited[node] = false;
}

void printShortestPathDFS(vector<vector<int>>& graph, int source, int dest, int V) {
            vector<bool> visited(V, false);
            vector<int> path, shortestPath;

            dfs(graph, source, dest, visited, path, shortestPath);

            if (shortestPath.empty()) {
            cout << "No path exists between source and destination\n";
            } else {
            cout << "Shortest path from source to destination using DFS: ";
            for (int node : shortestPath) {
            cout << node << " ";
            }
            cout << "\n";
            }
}

int main() {
            int V = 8;
            int source = 2, dest = 6;
            vector<vector<int>> edges = {{0, 1}, {1, 2}, {0, 3}, {3, 4},
                                    {4, 7}, {3, 7}, {6, 7}, {4, 5},
                                    {4, 6}, {5, 6}};
            vector<vector<int>> graph(V);
            for (auto edge : edges) {
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
            }

            printShortestPathDFS(graph, source, dest, V);
            return 0;
}
```

Comparison between BFS and DFS:

- **Efficiency**:

    - BFS is generally more efficient in finding the shortest path because it systematically explores all nodes at each level before moving to the next level.
    - DFS may not always find the shortest path because it explores one branch of the graph as deeply as possible before backtracking.
- **Path Finding**:
    - BFS guarantees that the first path found from the source to the destination is the shortest path.
    - DFS may find a shorter path if it explores in the right direction first, but it does not guarantee finding the shortest path.

In summary, while BFS is efficient and guarantees the shortest path, DFS may be more memory-efficient and can be used in certain scenarios where the shortest path is not the primary concern.

Q2

2.1) Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
using namespace std;

bool isValid(int x, int y, int rows, int cols, vector<vector<char>>& maze) {
	return (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] != '#');
}

vector<pair<int, int>> findShortestPath(vector<vector<char>>& maze, pair<int, int>& start, pair<int, int>&
exit) {
	int rows = maze.size();
	int cols = maze[0].size();
	vector<vector<bool>> visited(rows, vector<bool>(cols, false));

	vector<vector<pair<int, int>>> parent(rows, vector<pair<int, int>>(cols, {-1, -1}));
	queue<pair<int, int>> q;
	q.push(start);
	visited[start.first][start.second] = true;
	int dx[] = {-1, 1, 0, 0};
	int dy[] = {0, 0, -1, 1};
	char dir[] = {'U', 'D', 'L', 'R'};

	while (!q.empty()) {
	pair<int, int> current = q.front();
	q.pop();
	if (current == exit) {
```

```
            break;
            }
            for (int i = 0; i < 4; i++) {
            int newX = current.first + dx[i];
            int newY = current.second + dy[i];

            if (isValid(newX, newY, rows, cols, maze) && !visited[newX][newY]) {
            q.push({newX, newY});
            visited[newX][newY] = true;
            parent[newX][newY] = current;
            }
            }
            }

            vector<pair<int, int>> path;
            pair<int, int> current = exit;
            while (current != start) {
            path.push_back(current);
            current = parent[current.first][current.second];
            }
            path.push_back(start);

            reverse(path.begin(), path.end());

            return path;
}

int main() {
            vector<vector<char>> maze = {
            {'S', '.', '.', '.', '#', '.', '.', '#', '.', '.'},
            {'.', '#', '.', '.', '.', '.', '#', '.', '.', '.'},
            {'.', '#', '.', '#', '#', '.', '.', '.', '#', '.'},
            {'.', '.', '#', '#', '.', '.', '.', '.', '.', '.'},
            {'#', '.', '#', '.', '.', '#', '.', '#', '.', '#'},
            {'.', '.', '.', '.', '#', '.', '.', '.', '#', '.'},
            {'#', '#', '#', '.', '#', '.', '#', '.', '#', '.'},
            {'#', '.', '.', '.', '.', '.', '.', '#', '.', '.'},
            {'#', '.', '#', '#', '.', '#', '.', '.', '#', 'E'},
            {'.', '.', '.', '#', '.', '#', '.', '.', '.', '.'}
            };

            int start_x = 0, start_y = 0;
            int exit_x = 9, exit_y = 9;

            vector<pair<int, int>> path = findShortestPath(maze, {start_x, start_y}, {exit_x, exit_y});

            if (path.empty()) {
```

```
        cout << "No path found from start to exit!" << endl;
        } else {
        cout << "Shortest path from start to exit: " << endl;
        for (auto& cell : path) {
        cout << "(" << cell.first << ", " << cell.second << ") ";
        }
        cout << endl;
        }

        return 0;
}
```

There are several optimizations and modifications that can be made to improve the efficiency of BFS or DFS for solving the maze-solving problem. Let's discuss some of them along with examples and analyze their impact on performance:

1. Parallelization:

- **Optimization**: Parallelize BFS or DFS traversal using multiple threads or processes to explore different parts of the maze simultaneously.
- **Example**: Divide the maze into smaller regions and assign each region to a separate thread or process for exploration.
- **Impact on Performance**: Parallelization can lead to significant speedup, especially for large mazes, by leveraging multiple processing units concurrently. However, proper synchronization mechanisms are required to avoid data races and ensure correctness.

2. Preprocessing:

- **Optimization**: Preprocess the maze to identify dead-ends or unimportant areas that can be pruned from the search space to reduce the exploration effort.
- **Example**: Use maze analysis techniques to identify cells that are guaranteed to lead to dead-ends and mark them as blocked before starting the search.
- **Impact on Performance**: Preprocessing reduces the size of the search space, leading to faster traversal and shorter pathfinding times. However, it requires additional computation upfront and may not be applicable in all scenarios.

3. Memory Optimization:

- **Optimization**: Optimize memory usage by minimizing the data structures used for bookkeeping during traversal.
- **Example**: Instead of maintaining a separate visited array, mark visited cells directly in the maze grid itself.
- **Impact on Performance**: Memory optimizations reduce memory overhead, especially for large mazes, and can lead to faster traversal times due to reduced memory access overhead.