Experiment 7A

Student Name: Tanmaya Kumar Pani **UID: 22BCS12986**

Branch: BE-CSE **Section/Group: IOT-613B**

Semester: 5 **Date of Performance: 17/09/2024**

Subject Name: AP Lab **Subject Code: 22CSH-311**

1. TITLE:

Breadth First Search: Shortest Reach

2. AIM:

Consider an undirected graph where each edge weighs 6 units. Each of the nodes is labelled consecutively from 1 to n. You will be given a number of queries. For each query, you will be given a list of edges describing an undirected graph. After you create a representation of the graph, you must determine and report the shortest distance to each of the other nodes from a given starting position using the breadthfirst search algorithm (BFS). Return an array of distances from the start node in node number order. If a node is unreachable, return -1 for that node.

3. Objective

Complete the bfs function in the editor below. If a node is unreachable, distance is .
bfs has the following parameter(s):

- int n: the number of nodes
- int m: the number of edges
- int edges[m][2]: start and end nodes for edges
- int s: the node to start traversals from

Returns

int[n-1]: the distances to nodes in increasing node number order, not including start node (-1 if a node is not reachable)

4. Algorithm

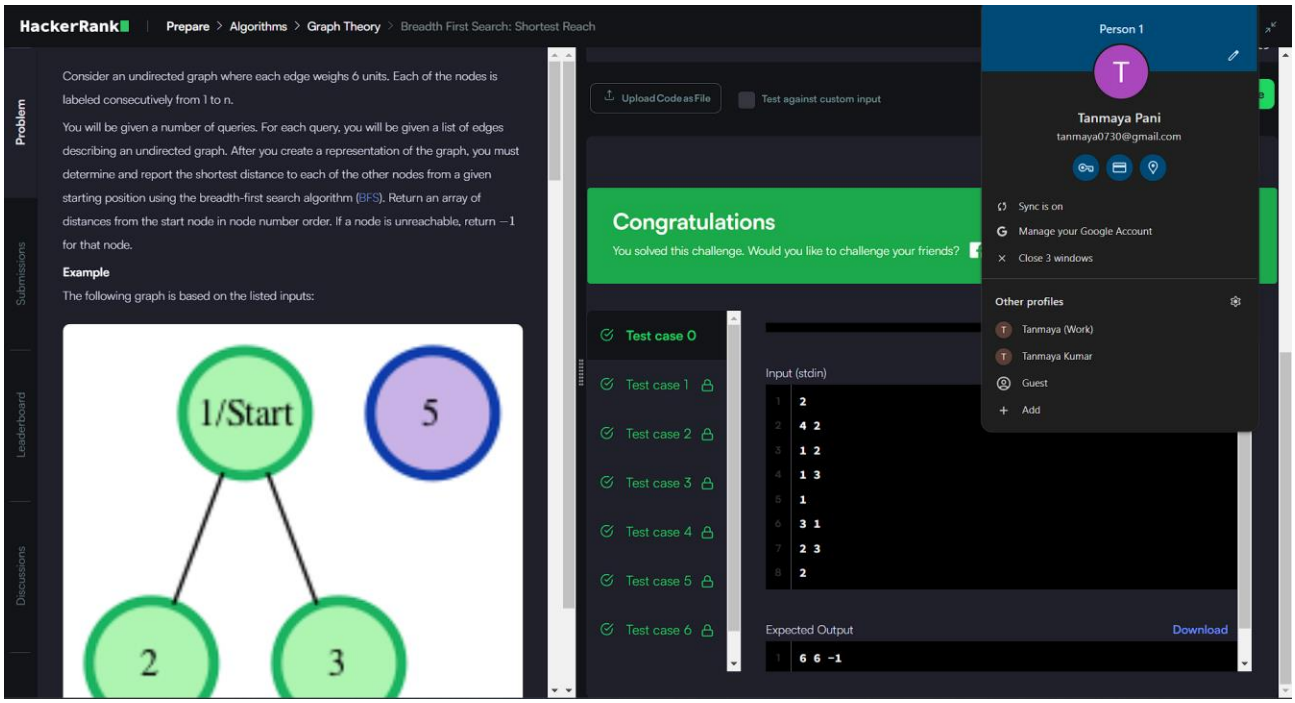
1. Create an adjacency list from the input edges.
2. Initialize distances to all nodes as -1.
3. Use a queue to perform BFS starting from the given node.
4. Set the distance of the start node to 0.
5. For each node, visit its unvisited neighbors and update their distances.
6. Continue BFS until all reachable nodes are visited.
7. Exclude the start node from the result and return the distances.

5. Implemetation/Code

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

vector<int> bfs(int n, int m, vector<vector<int>> edges, int s) {
    // Initialize distances to -1 for all
    vector<int> distances(n + 1, -1);
    // Create adjacency list for graph
    unordered_map<int, vector<int>> graph;
    for (auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // BFS first value
    queue<int> q;
    distances[s] = 0;
    q.push(s);

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

        // Traverse all neighbors of current node
        for (int neighbour : graph[current]) {
            if (distances[neighbour] == -1) {  // If not visited
                distances[neighbour] = distances[current] + 6;  // Update distance
                q.push(neighbour);
            }
```

```
            }
        }

        // Prepare the result
        vector<int> result;
        for (int i=1; i<=n; ++i) {
            if (i!=s) {
                result.push_back(distances[i]);
            }
        }

        return result;
    }

    int main()
    {
        int t;
        cin >> t;
        while (t--)
        {
            int n, m;
            cin >> n >> m;
            int edges[m][2];
            for (int i = 0; i < m; ++i)
                cin >> edges[i][0] >> edges[i][1];
            int s;
            cin >> s;
            int result[1001];
            int result_size = bfs(n, m, edges, s, result);
            for (int i = 0; i < result_size; i++)
                cout << result[i] << " ";
            cout << endl;
        }

        return 0;

      }
```

6. Output:



7. Time Complexity : O(n+m)

8. Space Complexity : O(n+m)

9. Learning Outcomes:-

1. Understand how BFS can be used to find shortest paths in an unweighted graph.
2. Learn how to implement an adjacency list using linked lists.
3. Handle multiple test cases efficiently with a queue-based approach.

Experiment 7B

Student Name: Tanmaya Kumar Pani **UID: 22BCS12986**

Branch: BE-CSE **Section/Group: IOT-613B**

Semester: 5 **Date of Performance: 17/09/2024**

Subject Name: AP Lab **Subject Code: 22CSH-311**

1. TITLE:

Snakes and Ladders: The Quickest Way Up

2. AIM:

Markov takes out his Snakes and Ladders game, stares at the board and wonders: "If I can always roll the die to whatever number I want, what would be the least number of rolls to reach the destination?"

Rules The game is played with a cubic die of 6 faces numbered 1 to 6.

1. Starting from square 1 , land on square 100 with the exact roll of the die. If moving the number rolled would place the player beyond square 100 , no move is made.
2. If a player lands at the base of a ladder, the player must climb the ladder. Ladders go up only.
3. If a player lands at the mouth of a snake, the player must go down the Snake and come out through the tail. Snakes go down only.

3. Objective

Complete the quickestWayUp function in the editor below. It should return an integer that represents the minimum number of moves required.

quickestWayUp has the following parameter(s):

ladders: a 2D integer array where each ladders[i] contains the start and end cell numbers of a ladder

snakes: a 2D integer array where each ladders[i] contains the start and end cell numbers of a snake

4. Algorithm

- Initialize ladders, snakes, and visited arrays to -1 or 0.
- Input the number of ladders and update the ladders array with start and end points.
- Input the number of snakes and update the snakes array with start and end points.
- Begin BFS from position 1 with 0 rolls and mark it as visited.
- For each dice roll (1 to 6), calculate the next position.
- Check if the next position is affected by a ladder or snake.
- If the position is 100, print the number of rolls and exit.
- If not visited, mark the position as visited and continue BFS. If no solution, print -1.

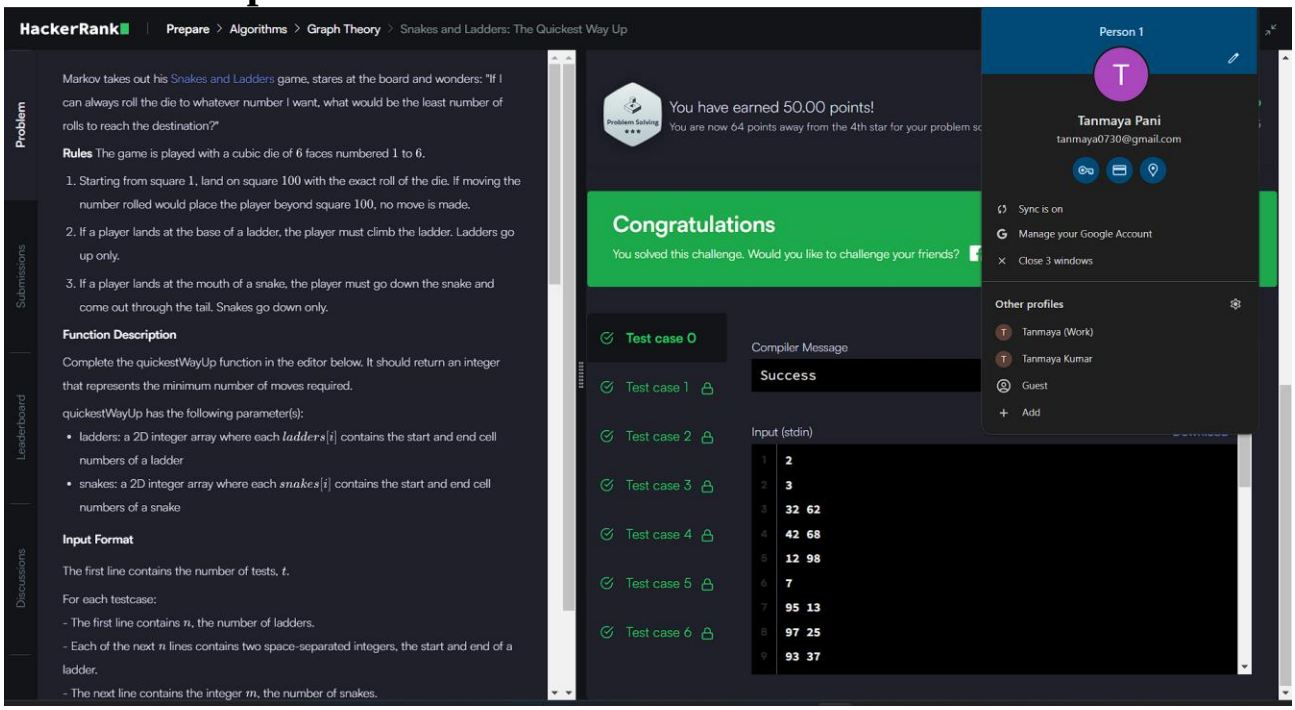
5. Implemetation/Code

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

int main() {
   int t;
   cin >> t;

   while (t--) {
      int ladders[101], snakes[101],
visited[101] = {0};

      // Initialize ladders and snakes
arrays
      for (int i = 0; i < 101; ++i) {
         ladders[i] = -1;
         snakes[i] = -1;
      }

      // Input ladders
```

```
    int num_ladders, num_snakes,
start, end;
    cin >> num_ladders;
    for (int i = 0; i < num_ladders;
++i) {
        cin >> start >> end;
        ladders[start] = end;
    }

    // Input snakes
    cin >> num_snakes;
    for (int i = 0; i < num_snakes;
++i) {
        cin >> start >> end;
        snakes[start] = end;
    }

    // BFS to find the minimum
number of rolls
    queue<pair<int, int>> q;
    q.push({1, 0});
    visited[1] = 1;
    bool found = false;

    while (!q.empty() && !found) {
        auto [pos, rolls] = q.front();
        q.pop();

        for (int dice = 1; dice <= 6;
++dice) {
            int next_pos = pos + dice;
```

```
            if (next_pos > 100)
continue;

            // Check for ladders and
snakes
            if (ladders[next_pos] != -1)
next_pos = ladders[next_pos];
            if (snakes[next_pos] != -1)
next_pos = snakes[next_pos];

            // If we reach position 100
            if (next_pos == 100) {
                cout << rolls + 1 << endl;
                found = true;
                break;
            }

            // Visit the next position
            if (!visited[next_pos]) {
                visited[next_pos] = 1;
                q.push({next_pos, rolls +
1});
            }
        }
    }

    if (!found) cout << -1 << endl;
  }

  return 0;
}
```

6. Output:



7. Time Complexity : O((t * 101 * 6)

8. Space Complexity : O(1)

9. Learning Outcomes:-

1. Learn how to implement Breadth-First Search (BFS).
2. Understand how to use queues for level-order traversal.
3. Apply ladders and snakes logic to simulate movement in a game board.
4. Using arrays and conditional branching in algorithms.