Assignment Solutions : Graph -8 Dijkstra

Q1 Path with Maximum Probability **Leetcode:-1514.**

Solution :

Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <iomanip>

using namespace std;

class Solution {
public:
    double maxProbability(int n, vector<vector<int>>& edges,
vector<double>& succProb, int start, int end) {
        vector<vector<pair<int, double >>> graph(n);
        for (int i = 0; i < edges.size(); ++i) {
            int u = edges[i][0], v = edges[i][1];
            double p = succProb[i];
            graph[u].emplace_back(v, p);
            graph[v].emplace_back(u, p);
        }

        vector<double> probabilities(n, 0.0);
        probabilities[start] = 1.0;

        priority_queue<pair<double, int>> pq;
        pq.push({1.0, start});
        while (!pq.empty()) {
            double cur_prob = pq.top().first;
            int cur_node = pq.top().second;
            pq.pop();

            for (const auto& neighbor : graph[cur_node]) {
                int next_node = neighbor.first;
                double edge_prob = neighbor.second;

                double new_prob = cur_prob * edge_prob;
                if (new_prob > probabilities[next_node]) {
                    probabilities[next_node] = new_prob;
                    pq.push({new_prob, next_node});
                }
            }
        }

        return probabilities[end];
    }
};
```

Q2 Network Delay Time **Leetcode:-743.**

Solution :

Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int N,
int K) {
        vector<vector<pair<int, int>>> graph(N + 1); // 1-
indexed graph
        for (const auto& time : times) {
            int u = time[0], v = time[1], w = time[2];
            graph[u].emplace_back(v, w);
        }
        vector<int> dist(N + 1, numeric_limits<int>::max());
        dist[K] = 0;

        priority_queue<pair<int, int>, vector<pair<int,
int>>, greater<pair<int, int>>> pq;
        pq.push({0, K});

        while (!pq.empty()) {
            int u = pq.top().second;
            int cur_dist = pq.top().first;
            pq.pop();

            for (const auto& neighbor : graph[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        int max_delay = *max_element(dist.begin() + 1,
dist.end());
        return (max_delay == numeric_limits<int>::max()) ?
-1 : max_delay;
    }
};
```

Q3 Path With Minimum Effort **Leetcode:-1631.**

Solution :

Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;
```

```cpp
class Solution {
public:
    int minimumEffortPath(vector<vector<int>>& heights) {
        int rows = heights.size();
        int cols = heights[0].size();

        vector<vector<int>> efforts(rows, vector<int>(cols,
INT_MAX));
        efforts[0][0] = 0;

        priority_queue<pair<int, pair<int, int>>,
vector<pair<int, pair<int, int>>>, greater<>> pq;
        pq.push({0, {0, 0}});

        vector<vector<int>> directions = {{0, 1}, {1, 0},
{0, -1}, {-1, 0}};

        while (!pq.empty()) {
            auto [effort, position] = pq.top();
            pq.pop();
            int row = position.first;
            int col = position.second;

            if (row == rows - 1 && col == cols - 1) {
                return effort;
            }

            for (const auto& direction : directions) {
                int newRow = row + direction[0];
                int newCol = col + direction[1];

                if (newRow >= 0 && newRow < rows && newCol
>= 0 && newCol < cols) {
                    int newEffort = max(effort,
abs(heights[row][col] - heights[newRow][newCol]));
                    if (newEffort < efforts[newRow][newCol])
{
                        efforts[newRow][newCol] = newEffort;
                        pq.push({newEffort, {newRow,
newCol}});
                    }
                }
            }
        }

        return -1; // Should not reach here for a valid
input
    }
};
```