MCS – 253P ADVANCED PROGRAMMING AND PROBLEM SOLVING

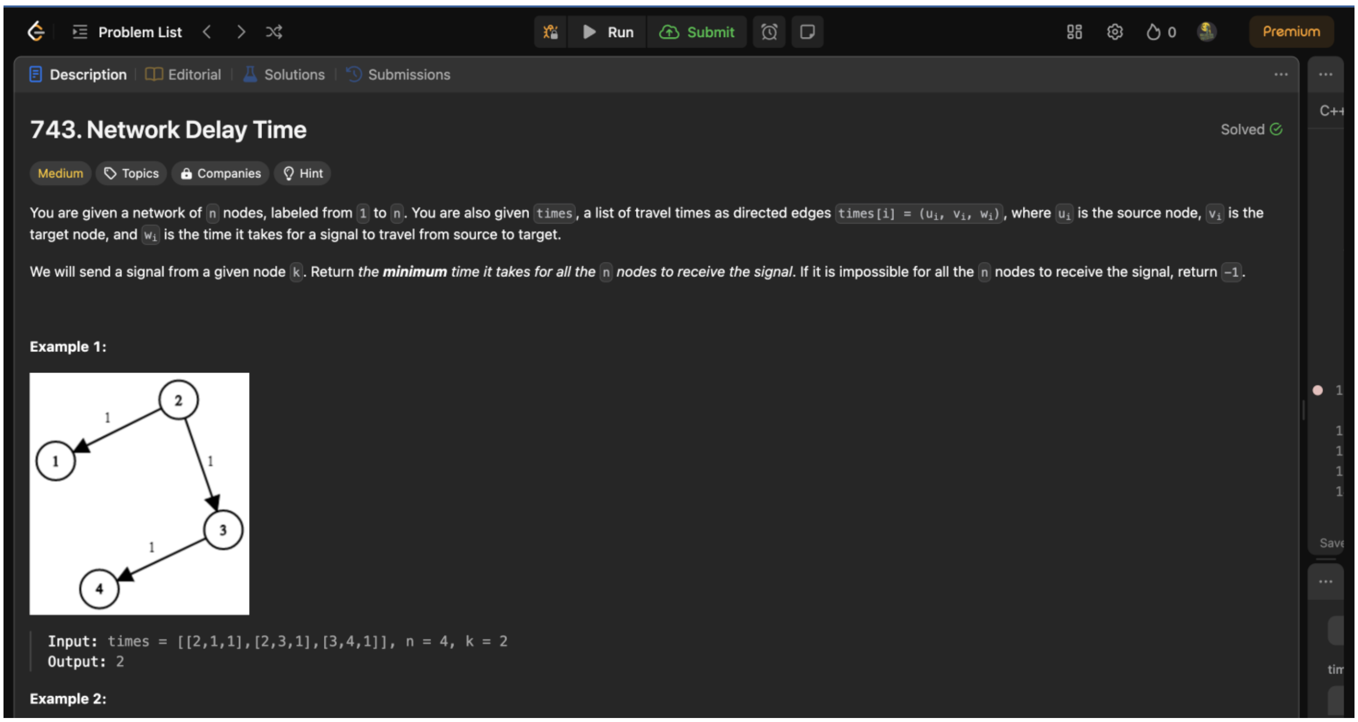
HOMEWORK –6 (Graph Problems)

Aswin Sampath

saswin@uci.edu

(53844684)

Question 1) Network Delay Time (Dijkstra)



Understanding the Problem

The problem presents a network of n nodes labeled from 1 to n. Travel times between nodes are provided as directed edges in the form of times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time taken for a signal to travel from the source to the target.

The task is to send a signal from a specific node k and determine the minimum time it takes for all n nodes to receive the signal. If it's impossible for all nodes to receive the signal, return -1.

Identifying Edge Cases

1. Empty times vector: When there are no directed edges provided, the minimum time for all nodes to receive the signal is not calculable. The code should handle this case appropriately.

2. Single-node network: For a single-node network, sending a signal from that node will result in a minimum time of 0 for all nodes.

Effective Test Cases

A network with multiple nodes and edges:

Input:

times = [[2,1,1],[2,3,1],[3,4,1]]

n = 4

k = 2

Expected Output: 2

A network with unreachable nodes:

Input:

times = [[1,2,1]]

n = 2

k = 2

Expected Output: -1

An empty network:

Input:

times = []

n = 0

k = 0

Expected Output: 0

A single-node network:

Input:

times = []

n = 1

k = 1

Expected Output: 0

Algorithmic Solution

- The algorithm initializes a vector minTime to store the minimum time required for each node to receive the signal, setting all initial times to INT_MAX except for the starting node k, which is set to 0.
- It constructs a graph representation using adjacency lists, where each node's outgoing edges are stored along with their respective weights.
- Using a priority queue (pq), it explores the nodes in the order of their minimum times until all nodes are visited. During traversal, it updates the minimum times for each node if a shorter path is found.
- Finally, it checks the maximum time in minTime and returns it as the minimum time for all nodes to receive the signal.

Time and Space Complexity Analysis

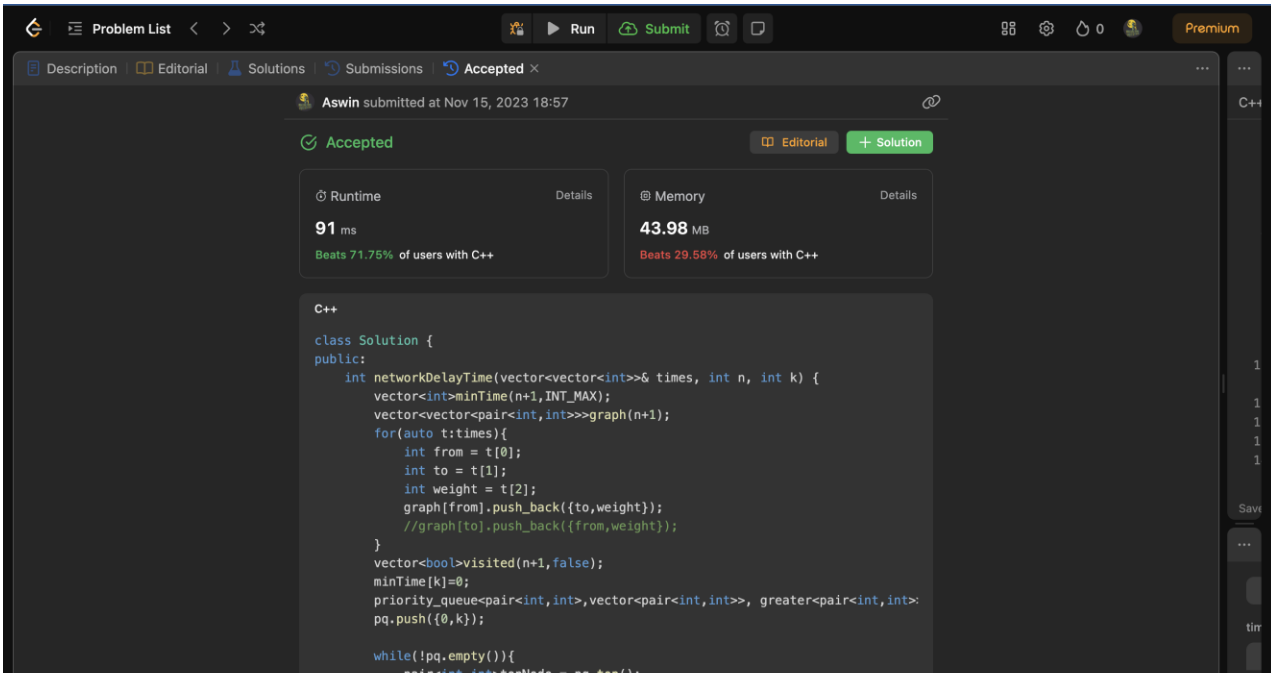
The time complexity of this algorithm is O(E log V), where E is the number of edges and V is the number of vertices (nodes) in the graph. This complexity arises from the priority queue operations in Dijkstra's algorithm.

The space complexity is O(V + E), where V is the number of vertices for the minTime vector and E is the number of edges for the graph representation.

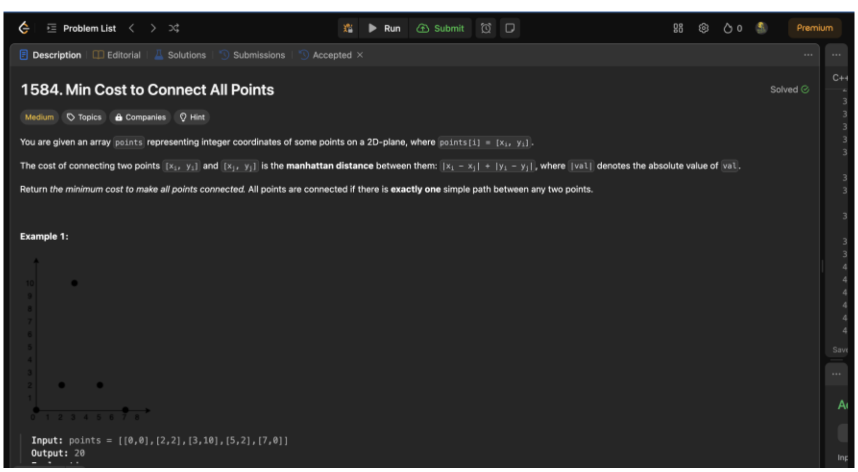
Code

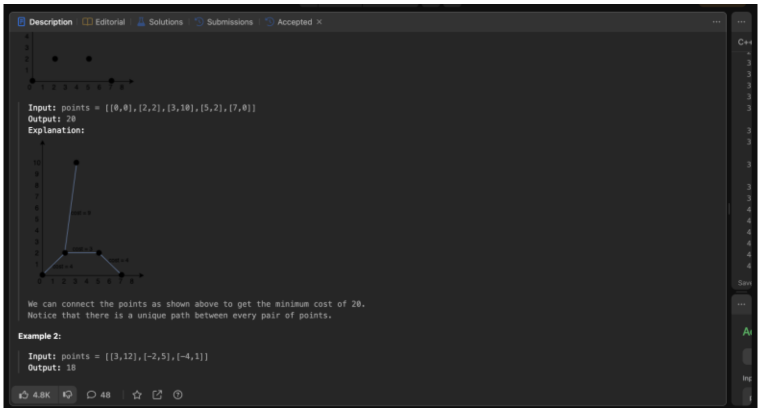
```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<int>minTime(n+1,INT_MAX);
        vector<vector<pair<int,int>>>graph(n+1);
        for(auto t:times){
            int from = t[0];
            int to = t[1];
            int weight = t[2];
            graph[from].push_back({to,weight});
        }
        vector<bool>visited(n+1,false);
        minTime[k]=0;
        priority_queue<pair<int,int>,vector<pair<int,int>>, greater<pair<int,int>>>pq;
        pq.push({0,k});

        while(!pq.empty()){
            pair<int,int>topNode = pq.top();
            int fromVertex = topNode.second;
            pq.pop();
            if(visited[fromVertex])continue;
            for(pair<int,int> edge : graph[fromVertex]){
                int toVertex = edge.first;
                int edgeWeight = edge.second;
                if(minTime[toVertex]> minTime[fromVertex] + edgeWeight){
                    minTime[toVertex] = minTime[fromVertex] + edgeWeight;
                    pq.push({minTime[toVertex],toVertex});
                }
            }
            visited[fromVertex]=true;
        }
        int maxTime = 0;
        for(int i=0;i<minTime.size();i++){
            if(i==0)continue;
            if(minTime[i]==INT_MAX)return -1;
            maxTime = max(maxTime,minTime[i]);
        }
        return maxTime;

    }
};
```

Output:



Question 2) Min Cost to Connect All Points (Kruskal's)





Understanding the Problem

The task involves finding the minimum cost to connect all points given their coordinates on a 2D plane. The cost of connecting two points [xi, yi] and [xj, yj] is determined by the Manhattan distance between them, calculated as |xi - xj| + |yi - yj|.

The goal is to find the minimum cost required to connect all points such that there exists exactly one simple path between any two points.

Identifying Edge Cases

- Minimum input size: When there is only one point, there's no cost involved as there's no connection needed.
- Maximum input size: With the maximum number of points, ensure the algorithm doesn't exceed time or space limitations.

Effective Test Cases

Multiple points in a square pattern:
Input: points = [[0,0],[2,2],[2,0],[0,2]]

Expected Output: 8

A set of points forming a straight line:

Input: points = [[1,1],[2,2],[3,3],[4,4]]

Expected Output: 12

A single point:

Input: points = [[0,0]]

Expected Output: 0

Algorithmic Solution

- The C++ code uses Kruskal's algorithm, implementing Disjoint Set Union (DSU) to find the minimum cost to connect all points:
- It initializes a parent array par to maintain the parent of each node.
- Constructs a weighted graph (adj) with Manhattan distances as weights between all pairs of points.
- Sorts the edges of the graph by weight.
- Iterates through the edges, checking if the endpoints of each edge belong to different sets. If so, merges the sets and adds the edge weight to the total sum.
- Finally, returns the accumulated sum of edge weights as the minimum cost to connect all points.

Time and Space Complexity Analysis

The time complexity for this algorithm is O(E log E), where E is the number of edges (here, O(n^2) due to the pairs of points). The sorting of edges dominates the time complexity.

The space complexity is O(n) for the par array and O(n^2) for the adjacency list.

Code:

```cpp
class Solution {
public:
    int par[1001];

    int find(int a)
    {
        if(par[a] < 0)
            return a;

        return par[a] = find(par[a]);
    }

    void Union(int a, int b)
    {
        par[a] = b;
    }

    int minCostConnectPoints(vector<vector<int>>& arr) {
        int n = arr.size();
        for(int i = 0; i < n; i++) par[i] = -1;
        vector<pair<int, pair<int, int>>> adj;
        for(int i = 0; i < n; i++)
        {
            for(int j = i + 1; j < n; j++)
            {
                int weight = abs(arr[i][0] - arr[j][0]) +
                             abs(arr[i][1] - arr[j][1]);
                adj.push_back({weight, {i, j}});

            }
        }
        sort(adj.begin(), adj.end());
        int sum = 0;
        for(int i = 0; i < adj.size(); i++)
        {
            int a = find(adj[i].second.first);
            int b = find(adj[i].second.second);

            if(a != b)
            {
                sum += adj[i].first;
                Union(a, b);
            }
        }
        return sum;
    }
};
```

Output:

Description | Editorial | Solutions | Submissions | Accepted × | Accepted ×

Aswin submitted at Nov 15, 2023 22:20

Accepted

Editorial | + Solution

Runtime	Details	Memory	Details
366 ms		58.06 MB	
Beats 56.61% of users with C++		Beats 65.35% of users with C++	

C++

```cpp
class Solution {
public:
    int par[1001];

    int find(int a)
    {
        if(par[a] < 0)
            return a;

        return par[a] = find(par[a]);
    }

    void Union(int a, int b)
    {
        par[a] = b;
    }

    int minCostConnectPoints(vector<vector<int>>& arr) {
        int n = arr.size();
        for(int i = 0; i < n; i++) par[i] = -1;
        vector<pair<int, pair<int, int>>> adj;
```