Day 4

Done	☑
Topic	Cheapest Flights within K Stops, Connecting Cities with Minimum Cost
Languages	JavaScript
Difficulty	⭐⭐⭐
Date Started	@March 4, 2023 10:30 PM
Date Completed	@March 5, 2023 1:00 AM
Related to Progress (Days)	Your Progress

What I Learned Today

Cheapest Flights Within K Stops(Hard) - LeetCode

Connecting Cities with Minimum Cost(Medium) - LeetCode

Key Concepts

Quick Links

Video Tutorial

Documentation

Video Tutorial

Documentation

Cheapest Flights Within K Stops

Initial configuration:

- **Queue:** Define a Queue that would contain pairs of the type {stops, {node,dist} }, where ‘dist’ indicates the currently updated value of the distance from the source to the ‘node’ and ‘stops’ contains the number of nodes one has to traverse in order to reach node from src.
- **Distance Array:** Define a distance array that would contain the minimum cost/distance from the source cell to a particular cell. If a cell is marked as ‘infinity’ then it is treated as unreachable/unvisited.
- **Source and Destination:** Define the source and the destination from where the flights have to run.

The Algorithm consists of the following steps :

- Start by creating an adjacency list, a queue that stores the distance-node and stops pairs in the form {stops,{node,dist}} and a dist array with each node initialized with a very large number (to indicate that they’re unvisited initially) and the source node marked as ‘0’.
- We push the source cell to the queue along with its distance which is also 0 and the stops are marked as ‘0’ initially because we’ve just started.
- Pop the element at the front of the queue and look out for its adjacent nodes.
- If the current dist value of a node is better than the previous distance indicated by the distance array and the number of stops until now is less than K, we

update the distance in the array and push it to the queue. Also, increase the stop count by 1.

- We repeat the above three steps until the queue becomes empty. Note that we **do not** stop the algorithm from just reaching the destination node as it may give incorrect results.
- Return the calculated distance/cost after we reach the required number of stops. If the queue becomes empty and we don't encounter the destination node, return '-1' indicating there's no path from source to destination.

Connecting Cities with Minimum Cost

There are n cities and there are roads in between some of the cities. Somehow all the roads are damaged simultaneously. We have to repair the roads to connect the cities again. There is a fixed cost to repair a particular road. Find out the minimum cost to connect all the cities by repairing roads. Input is in matrix(city) form, if city[i][j] = 0 then there is not any road between city i and city j, if city[i][j] = a > 0 then the cost to rebuild the path between city i and city j is a. Print out the minimum cost to connect all the cities. It is sure that all the cities were connected before the roads were damaged.

Method: Here we have to connect all the cities by path which will cost us least. The way to do that is to find out the Minimum Spanning Tree(**MST**) of the map of the cities(i.e. each city is a node of the graph and all the damaged roads between cities are edges). And the total cost is the addition of the path edge values in the Minimum Spanning Tree.

Prerequisite: MST Prim's Algorithm

Code Snippets

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class CheapestFlights {
    static class Edge {
        int src;
        int dest;
        int wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    static void createGraph(int[][] flight, ArrayList<Edge> graph[]) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        for (int i = 0; i < flight.length; i++) {
            int src = flight[i][0];
            int dest = flight[i][1];
            int wt = flight[i][2];

            Edge e = new Edge(src, dest, wt);
            graph[src].add(e);
        }
    }
```

```
    static class FlightInfo {
        int vertex;
        int cost;
        int stops;

        public FlightInfo(int vertex, int cost, int stops) {
            this.vertex = vertex;
            this.cost = cost;
            this.stops = stops;
        }
    }

    public static int cheapestFlight(int n, int[][] flights, int src, int dest, int k) { // O(V+ElogV) -> Priority Queue
        ArrayList<Edge> graph[] = new ArrayList[n];
        createGraph(flights, graph);
        int dist[] = new int[graph.length];
        for (int i = 0; i < graph.length; i++) { //dis[i] -> src to i
            if (i != src) {
                dist[i] = Integer.MAX_VALUE; //+infinity for neighbour of src
            }
        }
//        boolean[] vis = new boolean[graph.length];
        Queue<FlightInfo> pq = new LinkedList<>();
        pq.add(new FlightInfo(src, 0, 0));
        //loop
        while (!pq.isEmpty()) {
            FlightInfo curr = pq.remove();
            if (curr.stops > k) {
                break;
            }
            //neighbours ke liye check karlo
            for (int i = 0; i < graph[curr.vertex].size(); i++) {
                Edge e = graph[curr.vertex].get(i);
                int u = e.src;
                int v = e.dest;
                int wt = e.wt;

                if (curr.cost + wt < dist[v] && curr.stops <= k) {
                    dist[v] = curr.cost + wt;
                    pq.add(new FlightInfo(v, dist[v], curr.stops + 1));
                }
            }
        }
        //distance[destination]
        if (dist[dest] == Integer.MAX_VALUE) {
            return -1;
        } else {
            return dist[dest];
        }
    }


    public static void main(String[] args) {
        int n = 4;
        int[][] flights = {{0, 1, 100}, {1, 2, 100}, {1, 3, 600}, {2, 0, 100}, {2, 3, 200}};
        int src = 0, dest = 3, k = 1;
        System.out.println(cheapestFlight(n, flights, src, dest, k));
    }
}
```

```
import java.util.PriorityQueue;

public class ConnectingCities {
    static class Edge implements Comparable<Edge> {
        int dest;
        int cost;

        public Edge(int dest, int cost) {
            this.dest = dest;
            this.cost = cost;
        }

        @Override
        public int compareTo(Edge e2) {
```

```
            return this.cost - e2.cost;
        }
    }

    public static int connectCities(int[][] cities) {
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        boolean visited[] = new boolean[cities.length];

        pq.add(new Edge(0, 0));
        int finalCost = 0;
        while (!pq.isEmpty()) {
            Edge curr = pq.remove();
            if (!visited[curr.dest]) {
                visited[curr.dest] = true;
                finalCost += curr.cost;

                for (int i = 0; i < cities[curr.dest].length; i++) {
                    if (cities[curr.dest][i] != 0) {
                        pq.add(new Edge(i, cities[curr.dest][i]));
                    }
                }
            }
        }
        return finalCost;
    }

    public static void main(String[] args) {
        int cities[][] = {{0, 1, 2, 3, 4},
                          {1, 0, 3, 0, 7},
                          {2, 5, 0, 6, 0},
                          {3, 0, 6, 0, 0},
                          {4, 7, 0, 0, 0}};
        System.out.println(connectCities(cities));
    }
}
```

Challenges Experienced

Challenged Experienced in Dijkstra’s Algorithm which is applied on Cheapest Flight

Resources Used

Youtube, Alpha, ChatGPT, GeeksForGeeks, TakeUForward