Lab 14 Dijkstra to our rescue

Weichen Chai

Fall 2022

Introduction

This assignment will be a continuation of the graphs assignment done before. Continuing our goal of finding the shortest paths to cities in Sweden by train rides. We will be looking at Dijkstra's algorithm in this implementation.

Background

Dijkstra's algorithm's main goal is to find the shortest path from the starting node continuously eliminate longer paths. It will solve the problem in which the old solution in "Graphs" had to run the same paths over and over again because it could not remember where it had been.

Tasks

Sortest path first

In this task, we will the descriptions given and turn them into a graph that can be used to find the shortest path between cities. The file we refer to is given in the task as a CSV file. In order to approach this we keep an array which stores a shortest, where only one entry is kept, when it finds a shorter path, the previous is removed.

a priority queue

In this priority queue, we will be utilizing codes shown in the Graphs assignment. The insert and remove method used here ins the same as previous tasks, as well as the shortest method. The insert and remove will be bubble and sink respectively, which we have discussed already in the previous task, therefore, no codes will be inserted here. Since we are using a priority queue, the shortest paths are given the highest priority, which means it will be stored as the top element.

an identity sequence

The task suggest for us to identify cities with numbers so the program will run faster. In order to identify the cities in integer form, we have need to alter the city class.

```
    String name;
    Connection[] connections;
    int p;

    public City(String name) {
        this.name = name;
        this.connections = new Connection[0];
    }

    ...
```

Following the changes in City class, we will also need to change the lookup function to find the number instead of the name for each city.

```
...
        Integer num = city[look];
        while (cities[num] != null) {
            if (cities[num].dist < look.dist) { // lookup
  ...
        cities[num] = look;

...
```

the path entry

In this task, we will be updating the shorter path to the priority queue. In order to make it efficient, we should bubble the short paths to the root, so that it can be added efficiently to the priority queue. This should be quicker than the implementation we have performed before, where we traverse through the nodes to find a path that is shorter than the current.

In this implementation, we should create a class of newCities, where Integer dist should be initialized, representing the distance from the starting city. The Dijkstra implementation includes 52 elements of cities in our CSV file, and the city from and city to. As newCities suggests, we will have something like: newCities(Malmö,0,null) as the starting point if we follow the example given to us in the task in priority queue.

```
    this.from = cityFrom;
    this.to = cityTo;
    this.shortest = new newCities[52];
...
    add(new newCities(from,0,null));
```

Furthermore, if we want to find the shortest path, we can keep comparing the the distance between from and neighbouring city to find the shortest path, by doing this continuously, we will finally find the shortest path between two given cities.

```
    newCities t = new newCities(num.city.neighbors[i].to, num.d + num.city.neighbors

    if (cCity [t.city] == null){
        cCity[t.city.index] = t.d;
        add(t);
    }
    ..
    if( t.d < cCity[t.city.index]){
        cCity[t.city.index] = t.d;
        add(t);
    }
```

Benchmarks

The benchmark is as following for Malmö to Kiruna with three different implementations for graphs. It is shown in milliseconds.

Original implementation	**Max value implemented**	**Dijkstra**
862	178	0.38

This table shows the time to find the shortest time from Malmö to Kiruna, it is clear that Dijsktra implementation takes the least amount of time. The amount of times improved from the implementation with original implementation is around 3315. By running to find the distance for all cities from Malmö, it is visible that Kiruna is one of the furthest, and Dijstra's method calculates the time taken to get there by cumulatively adding time as it traverses through nodes. So Kiruna will always have a longer time than any other city. In order to show a comparison, Malmö's run time to other cities will also be shown in the table below.

The data chosen above are ones that have a consistent increase in distance from Malmö from top to bottom. The data received from Dijkstra is clearly more efficient in comparison to our previous implementations. From

Malmö to	**Graphs µs**	**Dijkstra µs**
Göteborg	621	175
Stockholm	732	229
Kalmar	890	247
Uppsala	926	271
Sundsvall	4129	342
Umeå	25082	357
Luleå	56702	369
Kiruna	86030	382

the graph above, we can say that the Dijkstra's time complexity looks logarithmic, however, we realize from testing the bubble and sink operations that their time complexity is O(Log e) where e is the number of edges, and a complexity of O(v) where V is the number of vertices (Cities). We conclude that the time complexity for the Dijkstra is something similar to O(V + loge). This relation can sort of be seen in the table above. The worst case scenario, in this case, would be if every city will be connected to all other cities, where the complexity is O(v2) (v squared) where v is the number of cities. However, as we have discussed previously in graphs, each city only has around 2 connections, so this cannot be the case.