Graphs

Dhruba Das

Spring Fall 2023

Introduction

This assignment was quite fun to do. We implemented a graph data structure, using the map of trains in Sweden. There are 3 problems in this assignment: implementing the Map class, implementing the Naive class, and implementing the Paths class.

City and Connection classes

A city is a data structure with a String name, and a list of it's neighbors (i.e direct connections which go both directions). The Connection class simply contains a city, and it's distance in minutes. Below are the code snippets

```
public class Connection {
    City city;
    Integer distance;

    public Connection(City c, Integer d){
        this.city = c;
        this.distance = d;
    }

}

import java.util.ArrayList;

public class City {
    String name;
    ArrayList<Connection> neighbors;

    public City(String na){
        this.name = na;
        this.neighbors = new ArrayList<>();
    }

    public void connect(City nxt, int dst){
        Connection c = new Connection(nxt, dst);
        neighbors.add(c);
    }


}
```

Here, instead of using an array, I opted for an ArrayList data structure, for the ease of adding, removing, and retrieving elements from the list compared to an array.

Map class

In this section, I will be going through the methods implemented in the Map class (in this class, we have an array of cities and a mod value of 541).

hash method

This method is used to derive a hash index for the name of a given city. We start at a value of 7 as to pick a prime number (we could also start at 0).

```
private Integer hash(String name) {
        int hash = 7;
        for (int i = 0; i < name.length(); i++) {
            hash = (hash*31 % mod) + name.charAt(i);
        }
        return hash % mod;
    }
```

lookup method

This method a searches for a city, with a given name, in the cities array; if there is no city at the hash index we calculated using the name, it's added to the array. Below is the code snippet:

```
public City lookup(String name){
        Integer indx = hash(name);
        while(true){
            if(cities[indx] == null) {
                size++;
                City city = new City(name);
                cities[indx] = city;
                return city;
            }

            if(cities[indx].name.equals(name))
              return cities[indx];
            System.out.printf("collision: %s and %s both has to %d\n", cities[indx]
            indx = (indx + 1) % mod;
        }
    }
```

Naive class

In this section, I will be going through the methods implemented in the Naive class, and provide a table containing the outputs of the benchmarks I ran.

shortest method

This method starts at a city, and traverses from neighbor to neighbor, in order to find the shortest distance from the starting city, to the destination, within a given time limit (i.e max). It uses the recursive approach to explore all possible paths and find the shortest one. Below is the code snippet:

```
public static Integer shortest(City from, City to, Integer max){
        if(max < 0){
            return null;
        }

        if(from == to){
            return 0;
        }

        Integer shrt = null;

        for(int i = 0; i < from.neighbors.size(); i++){
            if(from.neighbors.get(i) != null){
              Connection conn = from.neighbors.get(i);
              Integer dist = shortest(conn.city, to, max - conn.distance);


            if((dist != null) && ((shrt == null) || (shrt > dist + conn.distance)))
                shrt = dist + conn.distance;

              if((shrt != null) && (max > shrt)) {
                max = shrt;
              }
            }
        }
        return shrt;
    }
```

Benchmarks

Below are the benchmarks which I ran to test the shortest method, of the Naive class.

Starting city	**Ending city**	**Shortest (min)**	**Found in (ms)**
Malmö	Göteborg	153	2
Göteborg	Stockholm	211	4
Malmö	Stockholm	273	3
Stockholm	Sundsvall	327	102
Stockholm	Umeå	517	74694
Göteborg	Sundsvall	515	53379
Sundsvall	Umeå	190	555
Umeå	Göteborg	705	2

Table 1: (Shortest is in minutes; Found-in in milliseconds)

Paths class

In this section, I will be going through the methods implemented in the Paths class, as well as the benchmarks for the same cities as those for Naive class, but using the shortest method implemented in the Paths class.

Paths constructor

The constructor simply contains an array of cities, called path, and a sp or stack pointer Below is the code snippet:

```
public Paths(){
    path = new City[54];
    sp = 0;
   }
```

shortest method

This method has the same function as the previous shortest method, but now uses a stack pointer in order to prevent visiting the same cities over and over again, thus avoiding loops. It also removes the time constraint (max), compared to the previous implementation. Below is the code snippet:

```
public Integer shortest(City from, City to){
    if(from == to){
        return 0;
    }

    for(int i = 0; i < sp; i++){
        if(path[i] == from)
          return null;
    }

    path[sp++] = from;

    Integer shrt = null;

    for(int i = 0; i < from.neighbors.size(); i++){
        if(from.neighbors.get(i) != null){
            Connection conn = from.neighbors.get(i);
            Integer dist = shortest(conn.city, to);

            if((dist != null) && ((shrt == null) || (shrt > dist + conn.distance)))
              shrt = dist + conn.distance;
        }
    }
    path[sp--] = null;
    return shrt;
   }
```

Benchmarks

Below are the benchmarks which I ran to test the shortest method, of the Paths class.

Starting city	**Ending city**	**Shortest (min)**	**Found in (ms)**
Malmö	Göteborg	153	197
Göteborg	Stockholm	211	88
Malmö	Stockholm	273	170
Stockholm	Sundsvall	327	132
Stockholm	Umeå	517	183
Göteborg	Sundsvall	515	158
Sundsvall	Umeå	190	473
Umeå	Göteborg	705	159
Göteborg	Umeå	705	224
Malmö	Kiruna	1162	735

Table 2: (Shortest is in minutes; Found-in in milliseconds)