Graphs

Edwin Ahlstrand

Fall 2022

Introduction

This assignment covers graphs and their usefulness in dealing with maps, especially when finding the shortest path between two points. A map of distances between cities will be introduced, together with three different algorithms to search for the shortest path between two cities.

The map

The points on the map are cities which consist of a name and a list of connections to other cities, where the bidirectional connections represent edges.

To add a connection between cities, they are simply added to each others list of existing connections.

To add two cities with a connection to the map, both cities names are first looked up on the two dimensional array working as a map, where the names are hashed and searched through their buckets for the city. If a city isn't found, it is created. Lastly the two cities are given a connection to each other. The code for adding cities to the map is on the following page.

To add two cities with a distance to the map:

```
// Adds two cities and their distance apart to the map
public void add_cities(String c1_name, String c2_name, int distance){
    City c1 = lookup(c1_name);  // Check if c1 exists on map
    City c2 = lookup(c2_name);  // Check if c2 exists on map
    if(c1 == null){  // C1 isn't already on map
        c1 = new City(c1_name);  // Create new c1
        add_city(c1);  // Add c1 to map
    }
    if(c2 == null){  // C1 isn't already on map
        c2 = new City(c2_name);  // Create new c1
        add_city(c2);  // Add c2 to map
    }
    c1.add_connection(c2, distance);  // Add connection between cities
}
private void add_city(City city){  // Adds a single city to the map
    int index = hash(city.name);  // Retrieve it's hashed index
    if(cities[index] == null){  // No bucket at index
        cities[index] = new City[] {city};  // Create new bucket with c1
    }
    else{  // Bucket exists
        City[] tmp = new City[cities[index].length+1];  // Tmp one larger
        for(int i = 0; i < cities[index].length; i++){  // Iterate bucket
            tmp[i] = cities[index][i];  // Copy
        }
        tmp[cities[index].length] = city;  // Add given city
        cities[index] = tmp;  // Set new bucket as tmp
    }
}
```

Shortest path from A to B

To calculate the shortest path between two cities, three different algorithms are used. The first being a rather naive way, the second keeping track of paths used and the third is an improvement of the second by also keeping track of the max allowed distance. All of the algorithms calculate the same destinations.

The naive way

The first algorithm searches every path it can take as long as it doesn't run out of a preset max time, this means that it can go back and forth between two stations.

A snippet of the code is following which all the algorithms are similar to:

```
for (int i = 0; i < from.neighbours.length; i++) {  // Iterate neighbours
    if (from.neighbours[i] != null) {  // Neighbour exists
        Connection conn = from.neighbours[i];  // Get connection
        Integer tmp = shortest1(conn.city, to, max-conn.distance);  // Get path
        if(tmp == null)  // No result
            tmp = conn.distance;  // tmp is distance
        else  // Result
            tmp += conn.distance;  // Adds distance to tmp
        if (shrt == null)  // Currently shortest is empty
            shrt = tmp;  // tmp is new shortest
        else if (tmp < shrt)  // Shortest exists and if tmp is smaller
            shrt = tmp;  // tmp is new shortest
    }
}
```

Following table is the results from the benchmark:

From-To	Travel time (*min*)	Execution time (*ms*)
Malmö-Göteborg	153	2
Göteborg-Stockholm	211	0
Malmö-Stockholm	273	35
Stockholm-Sundsvall	327	21
Stockholm-Umeå	401	21
Göteborg-Sundsvall	501	1522
Sundsvall-Umeå	190	0
Umeå-Göteborg	705	705
Göteborg-Umeå	(gave up)	(gave up)

Table 1: Naive algorithm to calculate distance between two cities and its execution time. Time is in milliseconds and travel time is in minutes.

The last entry wasn't calculated since it would take too long time to complete, with an estimate of around six hours to complete. This is because Umeå doesn't have a lot of options towards Göteborg, meanwhile Göteborg has way more increasing options.

Using paths

A better solution for larger maps is keeping track of the current path, where the current city is checked if it already has been passed through, if it has, then the current branch is terminated.

From-To	**Path time** (*min*)	**Execution time** (*ms*)
Malmö-Göteborg	153	184
Göteborg-Stockholm	211	77
Malmö-Stockholm	273	156
Stockholm-Sundsvall	327	108
Stockholm-Umeå	517	152
Göteborg-Sundsvall	515	135
Sundsvall-Umeå	190	390
Umeå-Göteborg	705	134
Göteborg-Umeå	705	192

Table 2: Path algorithm to calculate distance between two cities its the execution time. Time is in milliseconds and travel time is in minutes..

As can be seen in the table above, it is slower to calculate shorter distances, but quicker to calculate longer ones compared to the previous algorithm.

Using improved paths

An improvement to the previous algorithm is to keep track of the currently shortest distance to reach the goal, so that if a distance between two cities is longer than the shortest distance to the goal, the branch is terminated.

From-To	**Path time** (*min*)	**Execution time** (*ms*)
Malmö-Göteborg	153	31
Göteborg-Stockholm	211	52
Malmö-Stockholm	273	64
Stockholm-Sundsvall	327	95
Stockholm-Umeå	517	137
Göteborg-Sundsvall	515	113
Sundsvall-Umeå	190	391
Umeå-Göteborg	705	71
Göteborg-Umeå	705	175

Table 3: Improved path algorithm to calculate distance between two cities and its execution time. Time is in milliseconds and travel time is in minutes..

This improved algorithm is almost always much faster than the previous.