Graphs

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open a window onto an real life usage of graphs, during which in order to succeed, we will make use of multiple concepts learnt throughout this course so far, among which recursion, hash tables and trees. The goal is in the end, to build the program such that it delivers the shortest distance by train in between two given cities. In this assignment, a data base of 52 Swedish cities and their connections (75 in total) has been used as railway data source.

Representing the graph, namely the map

In order to be able to draw the conclusions related to shortest possible paths in between given locations, the map had to be first virtually represented. All in all it consists of the following:

- the cities, including their name and their neighbors, which represent the connections;
- the connections themselves, represented by their name and the distance between them and the city they connect to;
- a hashed array of all cities included in the data base (will soon dive into more details);
- two methods used to establish new connections and look for the existence of a particular city;

Hashing the cities as the data base was read

As the file was read, for each line, therefore for each city, the *lookup* method is called. Since the given array was pretty spacious and I knew from the beginning I would be working with 52 cities only (there was no need at some point to resize the array), I decided to handle the collisions by using

the clustering method. First, the hash value is calculated and if the space at that particular index in the array is 'null', a new city is created there, otherwise, we go to the left as long as there is an empty space, check that the city has not previously been added and when an empty spot is found under these circumstances, we create it.

Once the 'else' branch is chosen two consecutive times, it's obvious a collision has occurred, therefore we increment the variable. After having done so, I have concluded that the program came across **4 collisions**, which were then, handled. The written and described *lookup* method looks as follows:

```
public City lookup(String city) {
    Integer hash = hash(city);
    if(cities[hash] == null) {
        cities[hash] = new City(city);
        TotalNoCities++;
    }
    else {
        for(int i = hash; i < cities.length; i++) {
            if(cities[i] != null && cities[i].name.compareTo(city) == 0)
                return cities[i];
            else if(cities[i] == null) {
                    cities[i] = new City(city);
                    TotalNoCities ++;
                    collisions++;
                    return cities[i];
                }
            }
        }

        return cities[hash];
    }
```

Note: I initially counted the total number of cities when I was reading the file lines, however I realised I was actually counting the number of existing connections. I, therefore, created a new variable which increments every time the 'City' constructor above is called (that's the moment when one unique new city is created).

The shortest path from A to B: the naive approach

After having implemented the algorithm and run the trips suggested in the instructions, I have obtained the following results:

Trip	TripStatus	MinTripTime[min]	Execution[ms]
Malmö to Göteborg	found	153	3
Göteborg to Stockholm	found	211	1.6K
Malmö to Stockholm	found	273	1.6K
Stockholm to Sundsvall	found	327	1.1K
Stockholm to Umeå	not found(gave up)	-	-
Göteborg to Sundsvall	found	515	78K
Sundsvall to Umeå	found	190	6
Umeå to Göteborg	found	705	7
Göteborg to Umeå	not found(gave up)	-	-

Table 1: The naive approach: Benchmark analysis of different trips by train in Sweden.

Since we are told from the very beginning in the assignment instructions that the connections are bidirectional, it would make perfect sense to find the trip both from Göteborg to Umeå and from Umeå to Göteborg. However, because of the way Swedish railway system is designed *(more connections in the south of Sweden)*, having the starting point from south to north results in more time spent in loops, therefore the execution time increases. On the other hand, going from north to south is straightforward since there is one path only to do so and loops are avoided.

Detecting paths: the new approach

In order to avoid the loops, the code has been updated so that it marks in an array the cities that have already been visited in order to be avoided in the future ('null' is returned). After having done so, the previous trips have been rerun, the following numbers included in *Table 2* have been obtained and the following have been noticed:

- this new algorithm has quite a constant execution time **(except for the trip from Sundsvall to Umeå)**;
- all routes have been found quite fast, even the ones for which I previously gave up because it took too long time;

Trip	TripStatus	MinTripTime[min]	Execution[ms]
Malmö to Göteborg	found	153	223
Göteborg to Stockholm	found	211	109
Malmö to Stockholm	found	273	198
Stockholm to Sundsvall	found	327	168
Stockholm to Umeå	found	517	218
Göteborg to Sundsvall	found	515	194
Sundsvall to Umeå	found	190	524
Umeå to Göteborg	found	705	204
Göteborg to Umeå	found	705	265

Table 2: The 'paths' approach: Benchmark analysis of different trips by train in Sweden.

A dynamic max depth

Last but not least, a new update has been tried in order to improve the algorithm. In other words, no third parameter is used in the recursive step, instead the 'max' value is initialised to 'null' and updated as new routes are found. If multiple options are found, we remember the smallest value and in the end return it. After having run the benchmark, apart from this version being more convenient than the naive implementation (we don't have to guess a value to begin with), it has not shown significant improvements timewise compared to the 'Paths' approach.

```
private Integer shortest(City from, City to) {
        ...
        Integer shrt = null;
        Integer max = null; //new update - initialize max
        for (int i = 0; i < from.neighbors.length; i++) {
            ...
            //new updates
            if(max == null && shrt != null)
                max = shrt;
                if(max != null && shrt != null)
                    if(shrt < max )
                        max = shrt;
            }
        }

        path[sp--] = null;
        return max; //new update - return max
    }
```

An inefficient algorithm for large data sets

In spite of the fact that the three versions of the algorithm do provide a solution to our problem (they help us find the shortest path by train between two cities), they turn out to be inefficient as far as large data sets are concerned. In order to show this, we will benchmark trips from Malmö to the up north of Sweden.

Trip	MinTripTime[min]	Execution[ms]
Malmö to Lund	13	0
Malmö to Kristianstad	69	492
Malmö to Göteborg	153	230
Malmö to Linköping	169	248
Malmö to Stockholm	273	201
Malmö to Mora	484	345
Malmö to Sveg	645	337
Malmö to Umeå	790	507
Malmö to Kiruna	1162	868

Table 3: Analysis of different trips whose starting point is Malmö.

All in all, by looking at the numbers presented above, it can easily be noticed that as the distance between the starting point increases (we go up north), the execution time does the same. In other words, for a larger map, extended to more than just Sweden, this algorithm would undoubtedly turn out to be inefficient *(which is mostly visible starting with the trip 'Malmö to Stockholm' up until 'Malmö to Kiruna').*

Conclusion

All in all, finding the shortest path between two places is no easy task, however it was so much fun to work on real data from Sweden and to estimate the trip time by train from one city to another.