Dijkstra

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open a window onto the continuation of the previous 'Graphs' assignment (which also represents the starting point of this experiment), namely by implementing Dijkstra's algorithm in order to find the shortest path from A to B. Once again, 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 134 European cities and their connections has been used as railway data source.

Dijkstra's algorithm step by step

Before starting programming Dijkstra's algorithm, time was spent understanding its goal and analyzing real-life examples. Having said that, after having done so, the following steps can be concluded:

- start with identifying the indicated starting point;
- analyze its neighbors and calculate the distances in between them;
- choose the neighbor holding the smallest distance calculated so far and start looking at its own neighbors;
- add the distance obtained so far with the distance held by the new neighbors and so on and so forth;
- once an end point is met, the path is abandoned and the neighbor holding the second smallest distance is given a try;
- new distances are now calculated and replaced only if are smaller than the ones previously obtained;
- the steps above are repeated until the final destination is met;

Implementing Dijkstra: a quick review of the previous assignment

Similar to the previous assignment, the whole graph has been represented using a Map class made out of cities and their own connection. The Paths class has been also reused, but adapted (will explain more on this topic soon) and last but not least, the cities have been hashed using the same clustering method into an array.

Dijkstra related new updates

The main update consists of creating a new class called Dijkstra which is made out of **a list** where visited cities are marked as the algorithm runs, **a priority queue** where neighbors are added and ordered from smallest to largest by their distance, **the map** and **several methods**. It should be taken into consideration that the list is created generic of type City (holds City objects), while the priority queue is defined in a different file and has been also created generic so that it can store objects of type Path. Since we have to prioritize in the queue elements (objects of type Path) with the smallest distances, comparisons have to be made. Therefore, the priority queue and the Path class implement the Comparable interface.

```
@Override
public int compareTo(Path other) {
    return this.dist.compareTo(other.dist);
}
```

Moreover, the *compareTo method* had to be overrode in order to access the 'distance' field in the Path objects so that they can be compared and ultimately ordered in the queue.

Apart from its attributes, the Dijkstra class also consists of the following three methods:

- the constructor: the map, priority queue and the list are initialised;

```
public Dijkstra(Map map) {
     this.map = map;
     queue = new PriorityQueue<>();
     visited = new ArrayList<>();
 }
```

- the 'begin' method (its parameters are the start and end point of the trip): it adds the start point to the queue and initializes the main algorithm;

```
 public void begin(String start, String stop) {
    City city_start = map.lookup(start); //turn the strings to City objects
    City city_stop = map.lookup(stop);
    Path p = new Path(city_start);
    queue.add(p); //add the start point to the queue
    dijkstra(city_stop); //call the method that initializes the algorithm
}
```

- the 'dijkstra' method (its parameter is the City object of the trip's endpoint):
  - it runs as long as the priority queue is NOT empty;
  - it removes the elements step by step from the queue and converts them to a City object so that the list can be used;
  - if the removed object from queue has already been visited the execution of the next steps is skipped and the next element is retrieved; otherwise it marks it as 'visited';
  - for every city removed from the queue, we run trough its neighbors and calculate the new distances, by storing this information in a Path object so that we don't overrun the old information collected from the data base *(very important)*;
  - it stops once the city removed from the queue is the final destination;

```
public void dijkstra(City city_stop) {
     while (!queue.isEmpty()) {

        Path current = queue.remove();
        String name = current.city.name;
        City cityObj = map.lookup(name);
        //check if the city has been previously visited
        if (visited.contains(cityObj)) {
            continue;
        }

        visited.add(cityObj);
        if (cityObj.name.equals(city_stop.name)) {
            System.out.println(city_stop.name + ": " + current.dist + " min");
            return;
        }
```

```
    //run through the neighbors and calculate the distances
        for (int i = 0; i < cityObj.neighbors.length; i++) {
            if (cityObj.neighbors[i] != null) {
                City neighborCity = cityObj.neighbors[i].city;
                //avoid overriding initial data
                Path addcity = new Path(neighborCity);
                //calculate the new distance
                addcity.dist = current.dist + cityObj.neighbors[i].distance;
                //add the neighbor to the queue
                queue.add(addcity);
            }
        }
    }
}
```

Benchmarking Dijkstra: The shortest path from Malmö to Kiruna

In order to see the improvement obtained by implementing Dijkstra's algorithm, I have run the trip from Malmö to Kiruna and the following have been obtained:

Trip	TripTime[min]	OldExecutionTime[ms]	Dijkstra[ms]
Malmö to Kiruna	1162	868	33

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

As it can be noticed Dijkstra's algorithm outperforms the initial implementation and according to my benchmark it is ***26 times faster***.
Note: The 'OldExecutionTime' refers to the time obtained using the 'Paths' approach in the previous assignment.

Now, 12 European cities will be benchmarked as final destination points, all starting from Bucharest. The total trip time, as well as the execution time and the total number of visited cities (the start and end point are also counted) in order to establish the path will be included. One thing I tried to implement, but didn't manage to do so, was to deliver step by step the shortest trip in terms of the cities that have to be followed in order to create it.

Trip	TripTime[min]	Execution[ms]	VisitedCities
Bucharest to Constanta	120	23	2
Bucharest to Budapest	900	24	3
Bucharest to Rome	1904	26	45
Bucharest to Berlin	1557	28	17
Bucharest to Paris	1747	24	35
Bucharest to Madrid	2261	27	100
Bucharest to London	1793	30	39
Bucharest to Copenhagen	1938	23	49
Bucharest to Malmö	1981	25	54
Bucharest to Oslo	2362	27	114
Bucharest to Stockholm	2254	39	99
Bucharest to Kiruna	3143	32	133

Table 2: Analysis of different trips whose starting point is Bucharest.

When choosing the end points, I tried to go step by step, further and further away from Bucharest in order to see if the distance (therefore more possible routes) affect the execution time. I was surprised to see that the execution time is rather constant (I am not completely sure it should be like this since I was not expecting it to be constant - some trips have more possible routes that have to be taken into consideration). Maybe the graph is a rather small one and this why the results are the way they are. However, judging by the shortest trip time in minutes I get, the results look correct (I also checked on Google Maps in order to approximate). Below it can be seen the way I measure the execution time in the main method:

```
long t0 = System.nanoTime();
dij.begin("Bukarest", "Kiruna");
long time = (System.nanoTime() - t0)/1000000;
System.out.println("execution time[ns]: " + time);
```

Conclusion

All in all, Dijkstra's algorithm is undoubtedly more efficient than my previous implementation and makes use of so many concepts learnt throughout this course.