Assignment 12 Report - Graphs in Java

Dean Tsankov

October 25, 2024

Introduction

In this report I will present one of the more general linked data structures. A graph is a set of vertices connected through edges with each other. The edges can be used to represent paths that connect points that are not directly connected and thus make the graph traversible in some sense. A few of the data structures we have already discussed fall into the category of graphs, such as trees and the linked list type structures. Here we will explore the most general sort of graph i.e. we let go of any restrictions on the structure unlike the mentioned previously explored themes.

In order to visualize the operations over the graph we will be using a set of cities in Sweden connected to each other via train lines. Each connection between two cities is given to us in a CSV formatted file along with the respective time it takes to travel the distance.

Populating the graph with data

The first task we have is to structure the provided set of data into a useful to us graph. First some underlying classes are defined. Those are the City and Connection classes.

A city has a name and a list of connections representing all other cities it has direct train lines to.

```
(...)
public class City {
    String name;
    ArrayList<Connection> conections;

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

    public void connect(City nxt, int dst) {
        this.conections.add(new Connection(nxt, dst));
    }
}
(...)
```

A connection stores a city destination as well as the time to reach it.

```
    (...)
    public class Connection {
        City destination;
        int timeToReach;

        public Connection(City destination, int timeToReach) {
            this.destination = destination;
            this.timeToReach = timeToReach;
        }
    }
    (...)
```

We then go line by line in the CSV file and create the two city vertices or nodes if they have not yet been created, represented by the entry and then "connect" them. Here all links are bidirectional which means that if city A is connected to B then B is also connected to A. The class responsible for initializing the graph then only needs to have properties for the array of cities in the set. But then since we would like this class to provide us with a quick lookup method we would turn this array into a hash table implemented with buckets. Its constructor then has the form:

```
    (...)
    public Map(String file) {
        this.cities = new City[mod];
        this.buckets = (ArrayList<City>[]) new ArrayList[mod];

        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] row = line.split(",");
                City one = lookup(row[0]);
                City two = lookup(row[1]);
                int dist = Integer.valueOf(row[2]);
                one.connect(two, dist);
                two.connect(one, dist);

            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    (...)
```

We want to lookup each city and that makes sure that if the city is already in the graph we only add a connection to it and do not add duplicates with the same name. The lookup method is a bit complex, but does the bulk of the work to setup the proper structure of the graph.

```
    (...)
    public City lookup(String name) {
        int hashedIndex = hash(name, this.mod);
        if (cities[hashedIndex] != null) {
            if (cities[hashedIndex].name.equals(name)) {
                return cities[hashedIndex];
            }else if (buckets[hashedIndex] == null){
                buckets[hashedIndex] = new ArrayList<City>();
                City res = new City(name);
                buckets[hashedIndex].add(res);
                return res;
            }else if(!buckets[hashedIndex]
                    .stream()
                    .anyMatch(c -> c.name == name)) {
                City res = new City(name);
                buckets[hashedIndex].add(res);
                return res;
            } else {
                return buckets[hashedIndex]
                        .stream()
                        .filter(c -> c.name == name)
                        .findFirst()
                        .orElse(null);
            }
        } else {
            cities[hashedIndex] = new City(name);
        }

        return cities[hashedIndex];
    }
    (...)
```

It is very similar in a way to the lookup method implemented in the Hash tables assignment, but now also adds the city node if it is nowhere to be find in the array or its bucket list.

A Naive way to find the shortest path between cities

One very naive solution, when we think about it, to find the shortest path between two cities is to perform a depth first search over the graph beginning from one of the cities and checking in depth the shortest way to reach the other. This has some pit holes we should consider first, though. Knowing that the links are bidirectional it is not hard to notice that it would be very easy to enter an infinite spiral where we go through the same cities without making any progress in our search algorithm. To negate this we should implement a maximum distance traveled constraint, which subtracts the connection's distance each time we traverse it and when the constraint reaches zero we finish the execution.

Filling in the following loop segment in the provided `shortest()` method will make it usable:

```
    (...)
    for (Map.Connection c : from.conections) {
        if (c != null) {
            Integer currDist = shortest(c.destination,
                    to, max - c.timeToReach);
            if (currDist != null) {
                Integer distSum = c.timeToReach + currDist;
                if (shrt == null || distSum < shrt) {
                    shrt = distSum;
                }
            }
        }
    }
    (...)
```

Benchmarking

We are asked to try and run the search for shortest path algorithm on a couple of pairs of cities. But as it should become clear this solution, while functional in theory, becomes quite cumbersome in practice, judging from the couple of x values in the table, of timed out executions. This is due to an underlying exponential complexity of the traversal.

Route	**Travel time (min)**	**Execution time (ms)**
Malmö to Göteborg	153	3
Göteborg to Stockholm	211	4
Malmö to Stockholm	273	4
Stockholm to Sundsvall	327	439
Stockholm to Umeå	x	411
Göteborg to Sundsvall	x	1501
Sundsvall to Umeå	190	1
Umeå to Göteborg	705	1505
Göteborg to Umeå	x	1544

Table 1: Graph DFS benchmark

Spending a bit of time thinking about how our algorithm works currently, makes us realize that there are a lot of multiples of the same operations done over and over again without getting us closer to the result. This happens of course due to the before mentioned loops. A max restraint limits their effect, but does not in any way prevent them from happening in the first place.

Keeping track of the path taken

We implement a separate array of cities which will keep track of the ones we have already been to during our search and as such when we decide to go along a link we will have to check if we already went to the linked city or not yet. Since the `shortest()` method is implemented recursively this path array of cities will perform much like a stack and we will keep track of a stack pointer index in order to push and pop cities onto it as we visit them.

The function now looks like this:

```
(...)
private Integer shortest(Map.City from, Map.City to) {
    if (from.equals(to)) {
        return 0;
    }

    Integer shrt = null;

    for (int i = 0; i < sp; i++) {
        if (path[i] == from)
        return null;
    }
    path[sp++] = from;
    for (Map.Connection c : from.conections) {
        if (c != null) {
            Integer currDist = shortest(c.destination, to);
            if (currDist != null) {
                Integer distSum = c.timeToReach + currDist;
                if (shrt == null || distSum < shrt) {
                    shrt = distSum;
                }
            }
        }
    }
    path[sp--] = null;
    return shrt;
}
(...)
```

Performing the same benchmark tests now gives nicer results and all travel times are able to be computed.

Route	**Travel time (min)**	**Execution time (ms)**
Malmö to Göteborg	153	16
Göteborg to Stockholm	211	1
Malmö to Stockholm	273	2
Stockholm to Sundsvall	327	3
Stockholm to Umeå	517	4
Göteborg to Sundsvall	515	4
Sundsvall to Umeå	190	8
Umeå to Göteborg	705	4
Göteborg to Umeå	705	5
Malmö to Kiruna	1162	19

Table 2: Graph DFS benchmark

Improving further

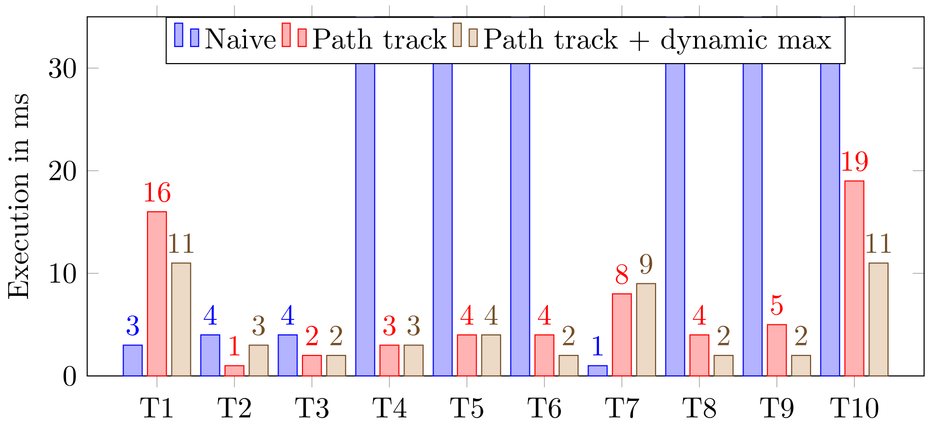
Even still there is another optimization we can implement in hopes of lowering execution even further. We could change the max constraint dynamically. It begins as null, but as soon as we find a path to the destination we know that the one we are searching for will always be no larger than the one we just found, so we set the constraint to that value. This means that with each step along the graph we refine this value until we find our result.

The changes to the new shortest function are as follow:

```
(...)
if (max!=null && max < 0){
    return null;
}
(...)
for (Map.Connection c : from.conections) {
    if (c != null) {
        if (max != null) {
            max -= c.timeToReach;
        }
        Integer currDist = shortest(c.destination, to, max);
        if (currDist != null) {
            Integer distSum = c.timeToReach + currDist;
            if (shrt == null) {
                shrt = distSum;
            }
            if (distSum < shrt) {
                shrt = distSum;
                max = shrt;
            }
        }
    }
}
(...)
```

Comparison

Pinning the three versions over the set of city to city benchmarks yields the chart:



We can notice that our considerations do pay off, but the fact of the matter is that this depth first search for the shortest distance algorithm still has exponential time complexity and would definitely not fare well for larger data sets.

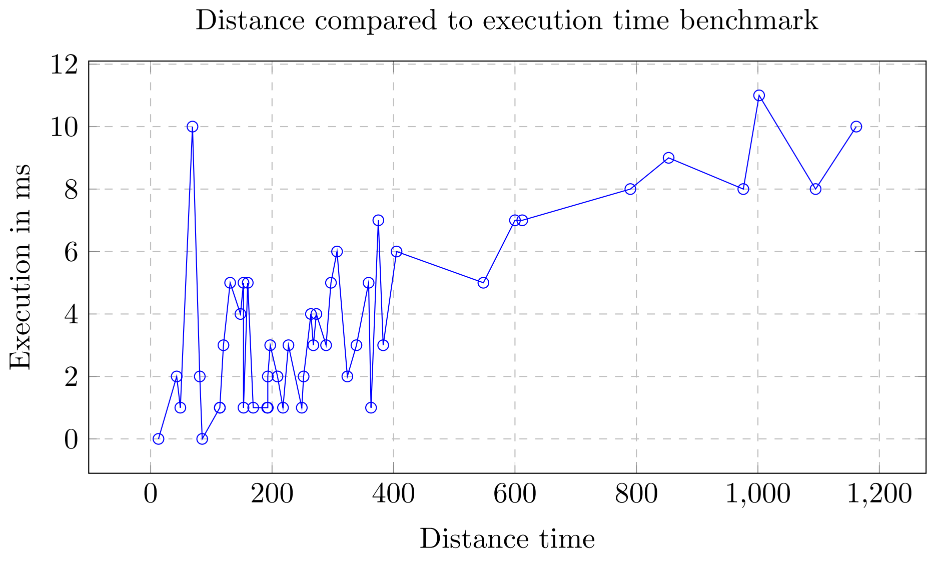


Figure 1: Distance compared to execution time benchmark

With this last graph I tried to show the exponential growth of the execution time with respect to the distance traversed using our most optimized solution so far. To what extent it serves its purpose is questionable. Nevertheless, this was done by computing the shortest distance from Malmö to every other city in the data set and then sorting the results in ascending order based on the distance. The code for this benchmark can be found in the respective github repository for the assignment.

Closing

All in all we now have a solution that works for the small data, but will be very inefficient for larger maps. A large problem with the current algorithm is that it does not in any way keep track of previously performed computations. Finding a way to record our previous work will be key to reaching optimal performance.