Graphs

Pontus Hector

Fall 2022

Introduction

This report is conducted as the tenth assignment in the course ID1021. In this report one explores graphs and compare different implementations of working them.

The construction of the graph

In the first task one is supposed to extract information from a file containing railroad connections between different Swedish cities. This information is then transferred into a weighted graph in which we implement a method to determine the shortest path between two given cities. The graph is constructed as an object (Map) in which each given connection from the file is transferred into new city nodes, if the given city have not occurred previously. This is compiled and executed through the lookup method.

```
public City lookup(String name) {
        int i = hash(name);
        while (true) {
            if (cities[i] == null) {
                cities[i] = new City(name);
                break;
            } else if (cities[i].name.equals(name)){
                break;
            }
            i = i + 1 % mod;
        }
        return cities[i];
    }
```

The connection, containing the city and the weight (distance), is then added to the corresponding cities and when all the given information from the file has been examined and extracted, the map is ready for usage.

Different implementations of shortest path

In the naive implementation of finding the shortest path, all the nodes branching off the start node is recursively gone through until all paths from the start node has been reviewed, continuously updating the current distance from the start node. Each time we reach the end node, the distance of the current path is compared to the currently shortest previously discovered path (if there exists any) and if the current path to the end node is shorter, we update the return value. Bluntly wandering through all nodes in a undirected graph in this manner may (but in this case almost always) lead to the method getting stuck in a loop. Because of this a maximum distance value is added as a reference to determine whether one is currently visiting a path of considerable distance, and if that is the case the method backtracks and changes path.

```
private 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.length; i++) {
            if (from.neighbors[i] != null) {
                Connections conn = from.neighbors[i];
                Integer dist = shortest(conn.city, to, (max - conn.distance));
                if((dist != null) && ((shrt == null) || (shrt > dist + conn.distance))) {
                    shrt = dist + conn.distance;
                }
            }
        }
        return shrt;
    }
}
```

The following measurements were obtained through benchmarking:

From	To	Shortest path	runtime
Malmö	Göteborg	153 minutes	<1 ms
Göteborg	Stockholm	211 minutes	2 ms
Malmö	Stockholm	271 minutes	2 ms
Stockholm	Sundsvall	327 minutes	28 ms
Stockholm	Umeå	517 minutes	42200 ms
Göteborg	Sundsvall	515 minutes	TBD
Sundsvall	Umeå	190 minutes	<1 ms
Umeå	Göteborg	705 minutes	6 ms
Göteborg	Umeå	705 minutes	TBD

Table 1: Cost of determining of the shortest path

The runtime of determining the shortest path varies considerably for different inputs and this because of several factors. One of the main problems with this method is that the graph is bluntly wandered through without taking notes of where we've been, causing one to visit and wander through loops repeatedly (in worst case). Another observation that can be made is that the runtime between two cities is sometimes of considerable difference depending on which city we evaluate the shortest path from. This is since this method visits **all possible paths** (within the given criterias) from one node to another. Another observation that can be made is that when conducting this operation one can think of it as a node tree, where the root is the city we evaluate the shortest path from. From one direction the tree may branch off in several directions early on which leads to a considerable amount of paths needs to be evaluated. If the same path is evaluated from the opposite direction (further down the original tree), where eventually fewer paths are presented early on, fewer paths would have to be evaluated in which case the runtime changes considerably (Umeå - Göteborg e.g).

To fix the problem with the loops an updated method is implemented in which we save each node that has been passed through and use these nodes as a reference. If the current node is equivalent to any of the previously visited nodes, the program should now exclude the investigation of that node and eventual paths from that node. The following measurements were obtained through benchmarking:

From	To	Shortest path	runtime
Malmö	Göteborg	153 minutes	<1 ms
Göteborg	Stockholm	211 minutes	<1 ms
Malmö	Stockholm	271 minutes	<1 ms
Stockholm	Sundsvall	327 minutes	1 ms
Stockholm	Umeå	517 minutes	2 ms
Göteborg	Sundsvall	515 minutes	2 ms
Sundsvall	Umeå	190 minutes	<1 ms
Umeå	Göteborg	705 minutes	<1 ms
Göteborg	Umeå	705 minutes	16 ms
Malmö	Kiruna	1162 minutes	795 ms

Table 2: Cost of determining of the shortest path

These results confirm that the runtime improved considerably with the new implementation. Another improvement that can be made is that whenever the method finds a path between node A and B it keeps track of the distance further on into the search. Whenever a path with a longer distance than the currently saved distance is explored the current path can be excluded since a shorter path has already been found.

```
private Integer shortest1(City from, City to, Integer max) {
        if (max != null && max < 0){return null;}
```

```
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.length; i++) {
    if (from.neighbors[i] != null) {
        Connections conn = from.neighbors[i];
        Integer time_left;
        if(max != null){time_left = max - conn.distance;}
        else {time_left = max;}
        shrt = shortest1(conn.city, to, time_left);
        if(shrt != null) {
            if(max == null) {max = conn.distance + shrt;}
            else if((conn.distance + shrt) < max){max = (conn.distance + shrt);}
            }
        }
    }
path[--sp] = null;
return max;
```

This implementation gave the following results through benchmarking:

From	To	Shortest path	runtime
Malmö	Stockholm	273 minutes	120μs
Malmö	Uppsala	324 minutes	410μs
Malmö	Sundsvall	600 minutes	4040μs
Malmö	Kiruna	1162 minutes	103000μs

Table 3: Cost of determining of the shortest path

The improvement is considerable when evaluating longer and more complex paths (Malmö - Kiruna e.g). However this method is still of some sort of exponential time complexity ($O(2^n)$) which makes it very inefficient for operating on large and more complex data sets. To find the shortest path between Malmö and Athens with this implementation it would take a considerable amount of time, even if our data set would be extended to Athens with the same, relatively simple, complexity. If one would extend our data set to match all of the actually existing train routes, this implementation is not going to cut it to determine the shortest path.