Graphs

Fatima Mohammad Ali

Fall 28/10/2023

Introduction

The purpose of this assignment was to construct a *graph* using the railroad network of Sweden. Graphs consist of *nodes* (*vertices*) connected to each other through *edges* (*links*). When edges are linked together, they form *paths* that could link nodes that are not directly connected in the structure. The tasks in this assignment specifically explored the properties of linking any node to another by optimizing different methods and preventing *circular* structures.

The Map

City and connection

To define the map, two classes `City` and `Connection` were implemented. A `City` was represented by having a *name* (`String name`) and the *neighbours* as a dynamic array holding direct *connections* (`ArrayList<Connection> neighbours`). Each `Connection` had the *destination city* `City city` and the *distance* `Integer distance` to it. The `City` class also included a method `connect(City nxt, int dst)` that given the destination city and distance, added those as as a new `Connection` to the `neighbour` ArrayList. The array list could simply be implemented from scratch as a dynamic array resembling a dynamic "bucket", but an ArrayList was recommended and hence chosen for simplicity purposes.

Constructing the map

The `class Map` skeleton (given in the assignment) was complemented in this task by adding a *reading file* procedure, a `hash` function, and a `lookup` method. Mainly, the class held an array of cities (`City[] cities`) and a mod value (`mod = 541`). The `hash` method was also given in the instructions.

A *lookup* method was needed when adding a new city to the map. The method returned a city given a name and checked through hashed indices of the *name*. If the name was present in the array of cities, the city holding the name was simply returned. If it was not present, a new city object with the name was created and added to the array.

To read the given railroad file accordingly, two cities were read for each row and a connection was added to each of them.

Collisions

It is worth mentioning that while the `cities` array was sized 541, some collisions still occurred since one city could have several neighbours. At most, the collisions reached 4 for a certain city, which indicates the number of neighbours the city had. For instance, ”Stockholm” had 3 collisions since it had 3 neighbours (”Västerås”, ”Uppsala”, and ”Södertälje”).

Shortest path

In this task, the shortest path from city A to B had to be determined by analysing three implementations of searching in the map and comparing those.

Naive method

The *naive* implementation used *depth-first search* to find the shortest path. To prevent ending up in an endless loop, a `max` value (maximum allowed path e.g. 600 minutes) was set.

The given `shortest` method was complemented in this task (and modified to accommodate an ArrayList as it was permitted) as follows:

```
private static Integer shortest(City from, City to, Integer max) {
    if (max < 0)    //if max was used up
        return null;

    if (from == to) //if the destination was reached
        return 0;

    Integer shrt = null;

    //iterate through all neighbours of a source city
    for (Connection conn : from.neighbours) {
    //recursively find the distance from each neighbour etc. to the destination
        Integer dist = shortest(conn.city, to, max - conn.distance);
        if(dist != null){
            if ((shrt == null) || (shrt > dist + conn.distance)){
                shrt = dist + conn.distance;
                max = shrt; //tighten the range
            }
        }
    }
        return shrt;
    }
```

Recursion was used to find the shortest path and save it in the variable `shrt`. For each recursion, the city on the path was updated to being its neighbour and the distance between this neighbour and the destination city was subtracted from `max` to account for the travelled distance. Hence, each recursive call would return the distance from a city on the path to the destination.

When the destination was reached alternatively if `max` was exceeded, 0 or null would be returned from the recursive call. A backtracking of the recursion would start by checking; if the return value was not null, then the temporary shortest path `shrt` was updated by adding the distance of the backtracked city to it. Additionally, `max` was updated to being `shrt` to avoid searching in one path for too long. Paths were also compared as the backtracking continued to choose the shorter one and save it in `shrt` instead. The recursion procedure would therefore explore all potential paths from the neighbor cities to the destination city within the given `max` range.

Results

From	**To**	**Shortest path (min)**	**Benchmark timestamp [ms]**
Malmö	Göteborg	153	5
Göteborg	Stockholm	211	2
Malmö	Stockholm	273	2
Stockholm	Sundsvall	327	52
Stockholm	Umeå	517	15345
Göteborg	Sundsvall	515	13364
Sundsvall	Umeå	190	5
Umeå	Göteborg	*705	*4
Göteborg	Umeå	**null	**null

Table 1: `max` was set as 600 for all except * where it was set to 800 to accommodate the path size. **Stuck in an cycle, no value detected for long.

Discussion

The naive solution does not keep track of the already visited cities which makes it not able to identify loops. To terminate a search and avoid an endless loop, the method was given a maximum allowed path to use. For instance, going from A to D in less than 300 minutes would require checking the direct connections of A (its neighbours). If A was directly connected to B in a 60 minutes line, then a path from B to D had to be found in the remaining 240 minutes. However, if B itself was connected to other cities, it would visit those as well to check if the shortest path was through them etc.

Finding a the shortest path from Umeå to Göteborg was possible since Umeå was the source city that branched into its neighbours. The naive implementation could find a path within the max value through the neighbours on the way down to Göteborg. Going from Göteborg to Umeå however, required going through other paths and cities that required revisiting other paths and cities, resulting in cycles.

Path method

To detect loops and avoid those, the already visited cities had to be kept track of, which was done in a `Path` solution. The `Path` class assigned a `City[] path` array that was large enough to hold any path (sized 52 since there are 52 cities in the map) and a stack pointer `sp`. The search procedure was done similarly to the `Naive` search but without a `max`:

```
private Integer shortest(City from, City to) {
    :
    //If the city is already in the path, abort the search
    for (int i = 0; i < sp; i++) {
        if (path[i] == from)
            return null;
    }

    //if not already in the path, add to the path/stack
    path[sp++] = from;

    //Iterate through all neighbours to find shortest path
    Integer shrt = null;
    for (Connection conn : from.neighbours) {
        Integer dst = shortest(conn.city, to);
        if(dst != null){
            if ((shrt == null) || (shrt > dst + conn.distance)){
                shrt = dst + conn.distance;
            }
        }
    }
    path[sp--] = null;  //pop city
    return shrt;
}
```

Results

Discussion

The `Path` implementation keeps track of the already visited cities unlike the naive one. The algorithm is able to backtrack by popping cities from the stack and by doing so, it is able to check for other possible paths and save the shortest

From	**To**	**Shortest path (min)**	**Benchmark timestamp [ms]**
Malmö	Göteborg	153	143
Göteborg	Stockholm	211	70
Malmö	Stockholm	273	128
Stockholm	Sundsvall	327	98
Stockholm	Umeå	517	153
Göteborg	Sundsvall	515	120
Sundsvall	Umeå	190	337
Umeå	Göteborg	705	119
Göteborg	Umeå	705	165
Malmö	*Kiruna*	*1162*	*1319*

Table 2: *N*ote; it could be misleading to round the values since it specifies minutes of distance.

one in `shrt`. The operation therefore avoids loops (since each city in the current path is visited once). This makes the path implementation more efficient contrary to the naive implementation. It can be seen in table 2 that Göteborg to Umeå was able to be detected for `Path` unlike the naive solution (table 1). This is since it did not get stuck in a loop that visited a city continuously. Yet, the `Path` benchmark values could still be improved, which was done speedily in the following implementation by incorporating max.

Better Implementation

The `Path` method was improved by using a `max` value in this `Better` solution. The max value was initialized as null and was set to `shrt` accordingly. This improved solution cut down the search even more, hence the shortest path from Malmö to Kiruna was returned as *1162 min* and detected in *153 ms.* This is an improved value compared to `Path` since there is now a constraint of maximum travelled path.

Even though the naive, path, and better solutions returned the shortest path correctly, they are not ideal methods. They consume a lot of time and would be almost useless given a bigger map with more connections. To efficiently find the shortest path, it would be beneficial to remember each path to a city and save those somewhere instead of constantly having to search for the paths over and over again. This could be solved by implementing the so-called *Dijkstra's algorithm*, which is something that was done in the next assignment.