Graphs in Java

Malte Berg - ID1021

HT 2024

Introduction

This report covers the implementation of a railway network of Sweden using a graph, with nodes connected through edges. The report also covers different ways to find paths with varying performance for each iteration.

Implementing Cities and Connections

The assignment goal is to create a map of the railway network of Sweden, and to start building the graph that will be the map, nodes and edges are needed, which in this assignment will consist of two java classes: `City` and `Connection`.

`City`

The city class contains a String `name`, being the name of the city, and an array `connections` which will be filled with the other nodes connected by edges to the city. The array gets provided with a size of 4, as the provided `.csv` file shows no city has more than 4 connections, and the size will be sufficient to hold the connections.

Apart from the constructor, the city class has one method, `connect()` which adds a connection to the array of connections at the next empty spot.

```java
public void connect(City nxt, Integer dst) {
    int i = 0;
    while (connections[i] != null) i++;
    connections[i] = new Connection(nxt, dst);
}
```

`Connection`

To be able to add connections to the city class, the connection class needed to be implemented as well. The class `Connection` is very simple, and only contains the connected `City`, `destination`, and an Integer of how many minutes it takes to get there, `time`.

Implementing the Map

The assignment asks to choose a way to implement a hash table for storing the cities. For this assignment, a bucket implementation was chosen with the following hash method for strings:

```java
private static Integer hash(String name, int mod) {
    int hash = 7;
    for (int i = 0; i < name.length(); i++) {
        hash = (hash * 31 + name.charAt(i)) % mod;
    }
    return hash;
}
```

The initial hash value is set to 7, as this provided a good hash value distribution. The mod value was set to 61, as this gave a good spread with the biggest collision being 4 on the same hash, and not too many empty places. The constructor looks like this:

```java
public class Map {
    private CityLinkedList[] cities;
    private final int mod = 61;

    public Map(String file) {
        cities = new CityLinkedList[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]);

                one.connect(two, Integer.valueOf(row[2]));
                two.connect(one, Integer.valueOf(row[2]));
            }
        } catch (Exception e) {
            System.out.println("File " + file + " not found.");
        }
    }
    :
```

The method `lookup()` is a method to either return a city that already exist or correctly create a new city added to the map and returning that.

```java
public City lookup(String city) {
    int hashed = HashUtil.hash(city, mod);
    CityLinkedList bucket = cities[hashed];
    if (bucket == null) {
        bucket = new CityLinkedList();
        cities[hashed] = bucket;
    }

    City cityToAdd = bucket.find(city);
    if (cityToAdd != null) return cityToAdd;
    else {
        cityToAdd = new City(city);
        cities[hashed].add(cityToAdd);
        return cityToAdd;
    }
}
```

Finding the Shortest Path

Three methods of finding the shortest path between two cities were created, each one adding better performance for the different paths. First the code and explanation will be provided, then the results compared between the methods.

Regular Depth-First

The first method for finding the shortest path is by providing a max time that the trip can take and then explore all the paths that can be traversed in that time. The class `Naive` has the following method `shortest()`:

```java
public static Integer shortest(City from, City to, Integer max) {
    if (max < 0) return null;
    if (from == to) return 0;
    Integer shortestTime = null;
    for (int i = 0; i < from.connections.length; i++) {
        if (from.connections[i] != null) {
            Connection conn = from.connections[i];
            Integer time = shortest(conn.destination, to, max - conn.time);
            if (time != null) {
                int totalTime = time + conn.time;
                if (shortestTime == null || totalTime < shortestTime) {
                    shortestTime = totalTime;
                }
            }
        }
    }
    return shortestTime;
}
```

Detecting Loops

A new class `Paths` was made building on the `Naive` class, with an added array that is big enough to keep every city. Every time the method is recursively called, the current connection is added to the array, and the method checks if the connection being searched is already in the path array, skipping that attempt, meaning that loops or visits to the same city more than once will not happen.

Dynamic Max Time

The last class `Dynamic` builds on the `Paths` finds any complete path, and then makes sure to search for paths that are faster than the found one.

```
    for (int i = 0; i < from.connections.length; i++) {
        if (from.connections[i] != null) {
            Connection conn = from.connections[i];
            Integer newMax = (max == null) ? null : max - conn.time;
            if (newMax != null && newMax < 0) continue;
            Integer time = shortest(conn.destination, to, newMax);
            if (time != null) {
                int totalTime = time + conn.time;
                if (shortestTime == null || totalTime < shortestTime) {
                    shortestTime = totalTime;
                    if (max == null || totalTime < max) max = totalTime;
                }
            }
        }
    }
```

Results

As seen in Figure 1, `Naive` performs worse when starting from locations with many connections and possible paths. `Paths` performs almost exactly the same no matter the starting position and destination. `Dynamic` performs the best overall except for the path from Sundsvall to Umeå.

In Table 1, the result of the `Dynamic` class method searching from Malmö to furher away cities can be seen. Generally, it can be seen that the further away from Malmö the processing time is longer, except for some outliers.

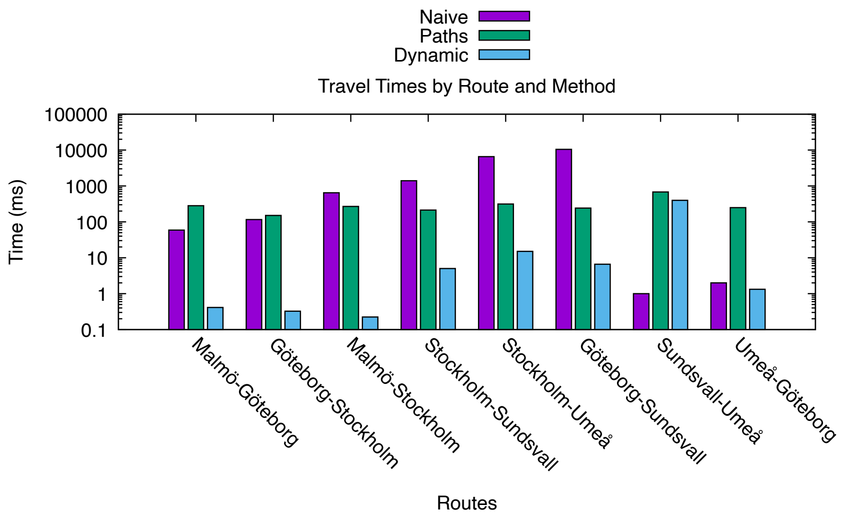

Figure 1: Time taken to find paths using three different methods, in ms

Table 1: Selected Travel and Processing Times for Routes from Malmö (Sorted by Travel Time)

Route	**Travel Time (minutes)**	**Processing Time (milliseconds)**
Malmö - Kristianstad	69	5.538
Malmö - Emmaboda	131	24.218
Malmö - Göteborg	153	0.292
Malmö - Kalmar	160	37.295
Malmö - Linköping	169	0.030
Malmö - Norrköping	193	0.040
Malmö - Södertälje	252	0.376
Malmö - Stockholm	273	0.016
Malmö - Sundsvall	600	1.408
Malmö - Umeå	790	9.689
Malmö - Boden	976	49.308
Malmö - Gällivare	1095	87.108
Malmö - Kiruna	1162	122.296