Graphs

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved implementing the graph data-structure, which consists of nodes and edges, where nodes are the fundamental units and edges connect these units. The edges can also be directed, meaning that the connection only goes one way. Graphs are very useful for representing real-life networks such as paths in a city or connections on a social media. In this assignment the graphs were used to find shortest path between two cities in Sweden using the railway network. This was done by first creating the graph by adding all connections and then implementing different versions of the method to decrease the run-time of finding the shortest paths.

2 The map

The first task of the assignment was to create the layout of the map. This was done using the already provided file containing 52 cities and 75 connections. Each entry is in the form of the starting city, the destination city and the time taken to travel between them. The connections are also bidirectional, meaning that the path goes both ways. The first step was creating two classes city and connection along with constructors.

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

The next step was to create two methods - add and lookup to be in the.

2.1 Hash

To create the lookup method, a hash method was first implemented. This method was responsible for returning the index to store a particular city in.

This was calculated by the result of calculating the modulus with a specific number, in this case a prime number (541) to minimise collisions. A section of the code provided can be seen below:

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

2.2 Look-up method

The purpose of the look-up method was to check if a specific city existed in the array. This was done by using the hash function to get the index for the city in question. The method then checked if the index was null, in which case a new city was created, otherwise the existing city was returned. This method was also then used in another method to add connections between two cities. A section of the code can be seen below:

```
if (city == null) {
    city = new City(cityName);
    cityTable[index] = city;
}
return city;
```

3 Shortest path - naive

The next task of the assignment was to find the shortest path between cities using the map. The first implementation was a very simple method that starts off by taking the two cities as parameters and a max time value. The max value is to avoid getting stuck in loops as the the program won't take into account paths that have distances longer than the value. The method first has a base case of when to and from cities are the same, in which case as there exist no path in between. Otherwise the method iterates through all the connections and uses recursive calls to find different paths while calculating the total time. A variable "min" keeps track of the total time and gets updated as shorter paths are found.

```
if (max < 0)
        return null;
    if (from == to)
        return 0;
```

3.1 Results

Value (n)	Run-time	Total time
Malmö to Göteborg	1300	153
Göteborg to Stockholm	1500	211
Malmö to Stockholm	2100	273
Stockholm to Sundsvall	3500	327
Stockholm Umeå	33476200	517
Göteborg to Sundsvall	21633700	515
Sundsvall to Umeå	2700	190
Umeå to Göteborg	4200	705
Göteborg to Umeå	Interrupted	

Table 1: Run-time and total time for paths between different cities

The table above (ref.table 1) shows the different total time calculated between the different as well as the different run-times taken to find the paths. As seen above (ref.table 1) the last the run-times were significantly inconsistent, particularly the paths from Stockholm to Umeå and Göteborg to Sundsvall. Additionally the run-time for the last path was interrupted due to it taking too long to excavate. Therefore to combat this certain improvements were made.

4 Shortest path - Improved

The first implementation was less efficient since the method would go through all nodes as the visited nodes were not kept track of. Not keeping track could also have led to being stuck in loops until the max value was reached. Therefore as improvements the method was modified to keep track of visited paths. This was done by creating a new array that kept track of the current path and if a city being visited was found in the path array the search would be terminated.

```
for (int i = 0; i < sp; i++) {
    if (path[i] == city) {
    return true;
```

4.1 Results

Value (n)	Run-time	Total time
Malmö to Göteborg	950	153
Göteborg to Stockholm	16300	211
Malmö to Stockholm	2200	273
Stockholm to Sundsvall	3700	327
Stockholm Umeå	9500	517
Göteborg to Sundsvall	3900	515
Sundsvall to Umeå	1200	190
Umeå to Göteborg	930	705
Göteborg to Umeå	5600	705
Malmö to Kiruna	160300	1162

Table 2: Run-time and total time for paths between different cities

5 Shortest path - Further Improved

The results above (ref. table 2) show improvements with the run-time. But for further improvement a dynamic max path was implemented. The method initially starts with no restrictions on the max path (null) and as the paths are explored the max path is adjusted based on the shortest paths found, which narrows the search space making the method more efficient.

```
if (max == null || subPath < max) {
    max = subPath;
}
```

5.1 Results

Value (n)	Run-time	Total time
Malmö to Göteborg	640	153
Göteborg to Stockholm	9430	211
Malmö to Stockholm	1100	273
Stockholm to Sundsvall	2400	327
Stockholm Umeå	8430	517
Göteborg to Sundsvall	2600	515
Sundsvall to Umeå	530	190
Umeå to Göteborg	980	705
Göteborg to Umeå	3900	705
Malmö to Kiruna	90540	1162

Table 3: Run-time and total time for paths between different cities

The results above show the most improvement out of all methods as the dynamic max path significantly increases efficiency for finding different paths.