Graphs

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the tenth and last assignment ”Graphs” on the course ”ID2021” TIDAB, KTH. The subject of this assignment is to explore a more general data structure than in previous assignments called graphs. A graph is a set of vertices that is connected through edges, the edges can be directed. Graphs can be linked into many types of arrangements. When they are connected one to one like a linked list it is called a path. They can also be connected in circular and treelike patterns. Similarly to the precious assignment to have data to work with we will make use of a file. The file in this case contains a list of cities and their train connections and is ordered in the pattern of ”from, to , minutes” where minutes is the distance to ride the train between the two cities. When adding the connections they will be directed from both cities to the other so one connection ”from, to” and one ”to, from”. First a structure for the vertices and nodes will be implemented, following this we will create a code structure to map them together. Finally some implementations of graph operations will be implemented to test and benchmark how a graph works.

Vertices and Edges

The vertices in this graph implementation will be class called ”City” and will have a name, an array of connecting cities and how far away they are and also a value for for many neighbouring cities we have found called ”size”. It has one function which is used to add a new connecting city. The code for this is given below:

```
public void addConnection(City name, int distance){
    Connection newC = new Connection(name, distance);
    neighbours[size] = newC;
    size++;
}
```

The edges are represented by another class called "Connection" which holds two variables, a "City" variable called "destination" which holds the name of the connecting city and a value for the distance in minutes from a city to this destination.

Mapping

To map the graph containing "City" vertices with "Connection" edges we read from the the file. There are some tasks that we need to handle when we do this, firstly we need to see if a read "City" object from the file has already has been added or if it is the first time we see it and need to ad it to the program. Secondly every time we read two cities that should have a connection we set up connecting edges both ways between the cities and pair that with the corresponding distance. Since we learned about hashing in the previous assignment we will use our knowledge of fast lookup methods and create a hash function that is based on the string name of a city. The map class will have an array of "City" objects which will correspond to all the cities from the data file.

The lookup operation takes the name of a city and looks through the city array to see if it has already been added. Another method is implemented for adding a new city to the city array that we are mapping. Since we are hashing we will need to handle collisions, in this case the collision handling is done using the "linear probing" method learned in the previous assignment. The code for this is given below:

```java
private void addCity(City city) throws Exception {
    int index = hash(city.name);

    while (index < cities.length) {
        if (cities[index] == null) {
            cities[index] = city;
            return;
        }
        index++;
    }
    throw new Exception("No empty index to insert new city");
}
```

Shortest Path

To make use of our newly made graph the task is to implement a program that searches for the shortest path between any two given cities in the graph. The method "shortest" takes in the value of the two cities and also a maximum value for how long it is allowed to search for a path. The method will be called recursively and for each recursive call the distance of the path that we have already traveled will be deducted from this max value and if we go further the function will return ”null”. Since the method will try all possible paths to all the cities neighbours we will in some cases get circular movements that wont lead anywhere, here the maximum value hedges against infinite loops that could get the program stuck. We then continuously update the value of which path is the shortest. As soon as a path is not the shortest or does not meet the requirement o the max value conditional statements protect the variable holding the shortest path from changing. The code for this implementation is given below:

```java
private static Integer shortest(City from, City to, Integer max){
    if (max < 0){
        return null;
    }
    if (from == to){
        return 0;
    }
    Integer shrt = null;
    Integer traversed;
    Integer traversedTot;
    for (int i = 0; i < from.neighbours.length; i++){
        if (from.neighbours[i] != null){
            Connection conn = from.neighbours[i];
            traversed = shortest(conn.destination, to, max - conn.distance);
            if (traversed != null){
                traversedTot = traversed + conn.distance;
                if (shrt == null || traversedTot < shrt){
                    shrt = traversedTot;
                }
            }
        }
    }
    return shrt;
}
```

To bench mark this some cities are given to try to find the shortest path bewteen:

1. Malmö to Göteborg

   Shortest: 153 min (took the program 21 ms).

2. Göteborg to Stockholm

   Shortest: 211 (took the program 19 ms).

3. Malmö to Stockholm

   Shortest: 273 (took the program 20 ms).

4. Stockholm to Sundsvall

   Shortest: 327 min (took the program 32 ms).

5. Sotckholm to Umeå

   Takes too long to find.

6. Göteborg to Sundsvall

   Takes too long to find.

7. Sundsvall to Umeå

   Shortest: 190 (took the program 0 ms).

8. Umeå to Göteborg

   Shortest: 705 min (took the program 46 ms).

9. Göteborg to Umeå

   Takes too long to find.

From this bench marking we can see that the problem with too long run-times is not only caused by long train routes but also from which way we take them. For some reason it seems to go a lot faster bench marking from the north of Sweden and down southward than the other way around. The reason for this from how to algorithm is written could be that there are far more connections in the beginning when you start down south and the possibility of getting stuck in circular loops of having to check a very high amount of paths is a lot higher then if we start along a very linear path up north and then have fewer options as we move down south.

Loop detection

Since the loops seem to be a part of the problem of getting high time complexities the next task is to implement the algorithm in a way so that this problem is eliminated. In this implementation of the code we have a separate array that keeps track of our path to see if we have already visited a city before, if we encounter such a case we return null and abort the operation. Apart from this the code is very similar to the previous implementation. Below are the benchmarks re-done with this new implementation:

1. Malmö to Göteborg

   Shortest: 153 min (took the program 0 ms).

2. Göteborg to Stockholm

   Shortest: 211 (took the program 0 ms).

3. Malmö to Stockholm

   Shortest: 273 (took the program 0 ms).

4. Stockholm to Sundsvall

   Shortest: 327 min (took the program 1 ms).

5. Sotckholm to Umeå

   Shortest: 517 min (took the program 35 ms).

6. Göteborg to Sundsvall

   Shortest: 515 min (took the program 36 ms).

7. Sundsvall to Umeå

   Shortest: 190 (took the program 2 ms).

8. Umeå to Göteborg

   Shortest: 705 min (took the program 3 ms).

9. Göteborg to Umeå

   Shortest: 705 min (took the program 50 ms).

As we can see from the results we get an overall better performance and with the removal of the loops we have no two cities that takes so long that we did not find a value. Still there is some difference in direction so we still have the fact that we get less complicated search patterns when starting up north by ”burning” possible paths faster.

Even attempting the longest possible path (1162 min) which would be form ”Malmö” to ”Kiruna” takes 2380 ms. The task is now to improve upon this further by constantly updating the maximum value when we find a path between the cities eliminating paths that are longer than the one we have already found. This improvement to the algorithm shortened the run time down to 254 ms when doing the same path from ”Malmö” to ”Kiruna” which is an extraordinary improvement. This last optimisation really shows the value of carefully considering your code before implementing an algorithm.