Graphs

Ayda Ghalkhanbaz

Fall 2023

Introduction

In this assignment, we study a fundamental data structure known as a graph. We then apply this data structure to find the shortest path between two cities using various approaches. In later sections, the execution time of these approaches will be compared and discussed.

Graph

A graph is a data structure that consists of nodes and edges. Edges define a connection between nodes and can be linked together. These connections can be directed, for instance a connection from A to B but not B to A, or undirected, where direction is meaningless. Graph structures are fundamental solutions to problems involving paths, networks etc.

For this assignment, creating a graph included implementing two helper classes; the `City` class and the `Connection` class. The `Connection` class represents edges in the graph and defines only two attributes; a city and its corresponding distance.

The `City` class represents the nodes in the graph and includes two attributes; the city's name and an array containing its neighbours. Additionally, it features a method named `connect()` which takes two arguments (a city and a distance), creates an instance of the `Connection` class with the arguments and finally adds it to the neighbour array of the city.

The graph creation occurs within a class called `Map`. The `Map` class has two attributes; an array, which later will be filled with cities, and a `final` integer named `mod`, which has value of 541 and will be used for hashing of the array.

This class also has 3 methods; `hash()`, `find()` and `lookupAdd()`. The `hash`() method calculates the hash value of a city, which comes handy when adding a new city to the graph and searching for existing ones.

The `find()` method follows a simple logic; it calculates the sought city's hash value and locates the corresponding entry in the array. If the entry is `null` then `find`() returns `null`; meaning that the entry is empty and the

sought city is not in the array, otherwise, it returns the city we've searched for. However, since we use open addressing for handling collisions in hashing, we repeat the searching process using a `while` loop until we reach the end of the array or the sought city is found.

The `lookupAdd()` method is really similar to the `find()` method, with the distinction that if the sought city is not found in the array, it adds the city to the appropriate entry. Below, you'll find the code for the `lookupAdd()` method.

```
    public City lookupAdd(String name){
            Integer hashKey = hash(name);
            while(true){
                if(cities[hashKey] == null){
                    City city = new City(name);
                    cities[hashKey] = city;
                    return city;
                }
                if(cities[hashKey].name.equals(name)){
                    return cities[hashKey];
                }
                hashKey = (hashKey + 1) % mod;
            }
        }
```

Shortest Path

In this task we will find the shortest path between two cities using three approaches. The first approach is the easiest one but not the best. The second approach has a better algorithm than the first one. Lastly, the third approach is an improved version of the second approach which gives the best result.

First Approach

The first approach utilises a depth-first search through the graph to find the best path between two cities. To avoid getting stuck in an infinite loop, we set a maximum allowed path as well. The code snippet provided below demonstrates the implementation of this algorithm.

```
public static Integer shortest(City start, City destination,
                                                Integer max){
```

```
    if(max < 0) // base case
        return null;
    if(start == destination)
        return 0;

    Integer minPath = null;
    for(Connection conn : start.neighbours){
        Integer dist =
        shortest(conn.city, destination, max-conn.distance);
        if(dist != null){
            if(minPath == null || minPath > dist + conn.distance)
                minPath = dist + conn.distance;
        }
    }
    return minPath;
}
```

As shown in the code above, the `shortest()` method handles two base cases. It also investigates every neighbour of the `start` city, and for each of these neighbours, it calls the `shortest()` method recursively to find a path to the `destination` city. The `max` variable sets a limitation to keep the searching safe from being trapped in a loop forever.

For instance, when finding the shortest path from Malmö to Stockholm with a maximum path length of 300, the method starts by investigating the immediate neighbours of Malmö. Consequently, it searches for the shortest path from each neighbour to Stockholm by calling the method recursively. Each time the method investigates a neighbour, it has to update the `max` value so that it holds the time it is left for the rest of searching.

Second Approach

The drawback of the first approach is that it potentially wastes the time by entering in a loop and consuming the maximum path before realising that the time is running out and it has to explore another path. To fix this issue, instead of having a maximum path, we add the cities we've visited during our search to an array. If a city is already visited, then we don't bother visiting it again, instead we check other neighbours. All modification of the `shortest()` method for this part of the task is shown below.

```
private Integer shortest(City start, City destination){
        if(start == destination)
            return 0;
```

```
        // already visited? return null
        for (int i = 0; i < sp; i++) {
          if (path[i] == start)
            return null;
        }
        // add the city we are in to the array
        path[sp++] = start;
        Integer minPath = null;

        for(Connection conn : start.neighbours){
            Integer dist = shortest(conn.city, destination);
            if(dist != null){
                if(minPath == null || minPath > dist + conn.distance)
                    minPath = dist + conn.distance;
            }
        }
        path[sp--] = null;
        return minPath;
    }
```

Third Approach

As mentioned before, this approach is an improved version of the second approach. While still keeping track of already visited cities, we set the maximum limitation to the first found path. In other words, we set the maximum path to the first found path and assume that this path is the shortest and all other paths should be longer than this, but if we find a shorter path then we update the `max` accordingly. The code snippet below illustrates the implementation of this approach.

```
private Integer shortest(City start, City destination,
                                        Integer max){
        if(max != null && max < 0)
            return null;
        if(start == destination)
            return 0;
        for (int i = 0; i < sp; i++) {
          if (path[i] == start)
            return null;
        }
        path[sp++] = start;
        Integer minPath = null;
```

```
        for(Connection conn : start.neighbours){
            Integer dist = shortest(conn.city, destination,
                        (max != null) ? max-conn.distance: null);
            if(dist != null){
                if(minPath == null || minPath > dist + conn.distance){
                    minPath = dist + conn.distance;
                if(((minPath != null) && max == null)|| minPath < max)
                    max = minPath;
                }
            }
        }
        path[sp--] = null;
        return minPath;
    }
```

Result

Table 1 demonstrates the results of benchmarking the first approach. While table 2 presents a comparison between the second and third approaches. These benchmarks highlight the execution time for finding the shortest path between given cities using mentioned approaches.

Route	**Travel Time[min]**	**Run Time[ms]**
Malmö to Göteborg	153	1.08
Göteborg to Stockholm	211	1.56
Malmö to Stockholm	273	5.08
Stockholm to Sundsvall	327	11.00
Stockholm to Umeå	517	7.55
Göteborg to Sundsvall	515	11.82
Sundsvall to Umeå	190	0.02
Umeå to Göteborg	705	1.25
Göteborg to Umeå	705	4529.00

Table 1: Execution time of finding the shortest path between two cities using the first approach

Route	Travel Time	2nd appr.	3rd appr.
Malmö to Göteborg	153	160	0.05
Göteborg to Stockholm	211	88	0.04
Malmö to Stockholm	273	139	0.02
Stockholm to Sundsvall	327	110	3.2
Stockholm to Umeå	517	150	9.53
Göteborg to Sundsvall	515	131	4.9
Sundsvall to Umeå	190	394	259
Umeå to Göteborg	705	146	0.08
Göteborg to Umeå	705	185	36.33
Malmö to Kiruna	1162	599	90

Table 2: Execution time of finding the shortest path between two cities before improvement (second approach) vs after improvement (third approach) in ms

Discussion

In the first approach, the implementation doesn't memorise the path that it visited before so every time it searches for the shortest path, it may get stuck in unnecessary loops. For instance, when trying to find a way from Malmö to Stockholm, it can start by visiting one of the immediate neighbours to Malmö, and since it doesn't remember where it has started, it can loop between that city and Malmö. These redundant calculations and lack of memorisation cause the unpredictable results, which is shown in table 1.

Although it seems that the implementation has a notable performance when finding a path from Umeå to Göteborg, it took so long to find a path in the opposite direction, i.e. from Göteborg to Umeå. This might be because of the more number of neighbours nearby Göteborg. Meaning that the more neighbours a city has, it is more likely to get stuck in silly loops that will not end up towards the destination.

It is also interesting to see how the second implementation (see table 2) is slower than the first approach (table 1). This might be because of memory access patterns or optimisations done by the compiler but even though we get much more stable and predictable results from the second implementation. The second implementation is more likely preferable due to its ability to memorise the path it has been through.

Although the second implementation can memorise the path, there are further algorithms that can give better results, such as the third implementation. As shown in table 2, the third approach outperforms the second one by aborting the paths that have longer distances. This helps us to always check only the paths that can potentially be better candidates than the first found path.