Graphs

Axel Månson Lokrantz

Spring Fall 2022

Introduction

A graph is a set of nodes or vertices that are connected through edges. A linked list can be considered a graph, a very simple one. The same goes for trees. These data structures are non circular and follow a set of rules. The type of graph we are going to work with in this assignment follow no restrictions and will allow the nodes to have a link to any other node in the structure. The following program was written in Java together with Axel Lystam and Björn Formgren at KTH.

Task 1: A map

A CVS file containing the railroad network of Sweden was provided to have some data to work with. Each entry consist of a city, its destination and the distanced between the two cities measured in minutes. All the connections are bidirectional, meaning if Stockholm has a connection to Malmö then Malmö has a connection to Stockholm. To get started we created three classes: city, connections and a map class. The city class containing three fields, a name, a next pointer (for the hash map) and an array with all its connections while the map class holds a collection of all the cities in a hash table.

```
private void addCities(String cityString,
String destinationString, String distanceString){

    City city = addCity(cityString);
    City destination = addCity(destinationString);
    Integer distance = Integer.valueOf(distanceString);
    city.addConnection(destination, distance);
    destination.addConnection(city, distance);
}
```

We chose to build a hash table using the linked list array implementation from the previous assignment. The addCities method receives a CSV string

which is then sliced into three separate strings inside the constructor. The strings are then distributed to their corresponding methods to get added to the hash table.

```
private City addCity(String name){
    City lookUp = lookUp(name);
    if(lookUp == null){
        lookUp = new City(name);
        insert(lookUp);
    }
    return lookUp;
}

public void addConnection(City cityDestination, Integer distance){
    int i = 0;
    while(connections[i] != null)
        i++;
    connections[i] = new Connections (cityDestination, distance);
}
```

addCity and addConnection implementation.

Since the CSV file has multiple entries of the cities we have to make sure we don't get duplicates. This is done inside the addCity method where we use our lookUp method to search for the city in our hash table. If the city is no where to be found we insert it. The connections are then added through a simple while loop.

```
private void insert(City city){
    Integer index = hash(city.name);
    if(cities[index] == null)
        cities[index] = city;
    else{
        City current = cities[index];

        while(current.next != null)
            current = current.next;
        current.next = city;
    }
}
```

Insert method for the map class.

We call the hash function to get a hash index. If the table at the hash index is empty, we insert the given city - else we traverse through the list until we find a free slot.

```
public City lookUp (String name){
    Integer index = hash(name);
    if(cities[index] != null){
        City current = cities[index];
            while(current != null){
                if(current.name.equals(name))
                    return current;
                current = current.next;
            }
    }
    return null;
}
```

Lookup method for the map class.

Similarly to our insert method the lookUp method receives a string which is then converted to a index. We check if the hash index is empty or not and then traverse the list and return the city we are looking for else we return null.

Task 2: The shortest path between X and Y

The next task was to create a program that finds the shortest path between A and B using depth first search. Since the connections are bidirectional, the program will get stuck going back and forth between stations. Therefor we had to create a max variable which was set to 300. The max variable represent the maximal amount of time the program can spend on a single path. Once we reach a total time greater than 300 the path should no long be considered an option.

```
    private static Integer shortest(City from, City to, Integer max) {
        if (max < 0)
            return null;
        if (from == to)
            return 0;
        Integer shortest = null;
        Integer totalDistance, distance;

        for (int i = 0; i < from.connections.length; i++){
            if (from.connections[i] != null) {
            Connections connection = from.connections[i];
            distance = shortest(connection.city, to, max - connection.distance);

                if(distance != null){
```

```
                totalDistance = distance + connection.distance;
                if (shortest == null || totalDistance < shortest)
                    shortest = totalDistance;
            }
        }
    }
    return shortest;
}
```

Method for finding the shortest path between station x and y.

Through recursion the method looks at all possible paths between station x and y. Inside the recursion call we subtract the time it takes between two stations from the max variable. If the time exceeds max we return null and the path is thrown away. All the paths that reaches the destination within the time span of max are then compared inside the if statements below the recursion call continuously comparing with our shortest variable to find the shortest path.

Departure	Destination	Distance (min)	Runtime
Malmö	Göteborg	153	3
Göteborg	Stockholm	211	2
Malmö	Stockholm	273	3
Stockholm	Sundsvall	327	34
Stockholm	Umeå	517	50975
Göteborg	Sundsvall	gave up	gave up
Sundsvall	Umeå	190	5
Umeå	Göteborg	705	7
Göteborg	Umeå	gave up	gave up

Table 1: Data from tests using the shortest method to find the shortest path between station x and y. Runtime in micro seconds.

As seen in table 1, for the most part the method works pretty well. A few paths however like Göteborg to Umeå and Göteborg to Sundsvall the run time is simply too long to wait for. To fix this issue we have to detect the infinite loops that occurs.

```
public Integer shortest (City from, City to, Integer max){
 if(max < 0)
     return null;
 if(from == to)
     return 0;
```

```
        Integer shortest = null;
        for (int i = 0; i < sp; i++){
            if (path[i] == from)
                return null;
        }
        Integer totalDistance, distance;
        path[sp++] = from;
        for (int i = 0; i < from.connections.length; i++){
            if (from.connections[i] != null) {
                Connections connection = from.connections[i];
                distance = shortest(connection.city, to, max - connection.distance);
                // path is valid, did not exceed max value.
                if(distance != null){
                    totalDistance = distance + connection.distance;
                    if (max > totalDistance){
                        max = totalDistance;
                    }
                    if (shortest == null || totalDistance < shortest)
                        shortest = totalDistance;
                }
            }
        }
        path[sp--] = null;
        return shortest;
}
```

Improved method for finding the shortest path between station x and y.

The main idea behind the improved version is to have a stack pointer and a path array which keeps track of all the paths that we already visited. This way we dramatically reduce the number of paths that are available to us while avoiding getting stuck in infinite loops between the same cities. The max value is no longer needed, but can come in handy to make the program even faster. If our max have an initial value of let's say 1000 and we find a path between station x and y in a total of 500 minutes, then there is no longer any point searching for paths longer than 500, therefor we can continuously update the max value making the threshold for a valid path smaller and smaller.

Departure	Destination	Distance (min)	Runtime
Malmö	Göteborg	153	1
Göteborg	Stockholm	211	1
Malmö	Stockholm	273	1
Stockholm	Sundsvall	327	1
Stockholm	Umeå	517	17
Göteborg	Sundsvall	515	16
Sundsvall	Umeå	190	5
Umeå	Göteborg	705	1
Göteborg	Umeå	705	17
Malmö	Kiruna	1162	101

Table 2: Data from tests using the improved shortest method to find the shortest path between station x and y. Runtime in micro seconds.

All paths that we previously gave up on now generate a result. We also added a new path from Malmö to Kiruna which according to our program takes 1162 minutes. For paths relatively close to each other finding the shortest distance is almost instant. However, from Malmö to Kiruna the execution time goes up drastically. The reason is of course that we get more paths to evaluate which raises the complexity. Generally finding a path from somewhere south to a station in the north is much slower than the other way around. This is because the stations in the south have more connections, meaning more paths to evaluate compared to those located in the north. The program will always check every possible path between station x and y which means we only have one case. The time complexity will vary depending on the amount of paths that are available to us.