Graphs

Jonathan Nilsson Cullgert

Fall 2023

Introduction

In this assignment a graph is created. The graph is a data structure which contains nodes and edges. A node is the where data is stored and an edge connects nodes to eachother. Trees and linked lists are a form of graph except that a will only connect to a parent node and/or two children nodes. A linked list will only connect to a next node, or in case of a doubly linked list, the previous and next node. The main thing that differentiates a regular graph from a tree or linked list is that the nodes in a graph can be linked in any way possible. Another important thing to note is that the edges in a graph are directed, meaning that an edge from node A to B does not necessarily create an edge from node B to A.

In this assignment we will import a csv file which contains a from city, a to city and the amount of time it takes to travel between them in minutes. In this assignment all the cities are linked both ways meaning that a connection from Stockholm to Södertälje also creates a connection from Södertälje to Stockholm. We are supposed to create this graph, a map, and some ways to find the shortest distance between two cities.

A train from Malmö

In this csv file there will be 52 cities and 75 connections, meaning that some cities have an edge to more than one other city. As mentioned previously all the connection are also linked both ways. The first part is to create the graph itself. To do this we will create a city class and a connection class. The city object will hold its name as well as an array of different connections. The city class will also have a method to add new connections to the city. The connection class will of course hold its constructor but it will also contain the from city, to city and the distance between them. These classes will also contain some method to find the connection array, the length between two cities and more since these methods will be useful later on.

The next part is to implement the map. The map will contain the different cities and there connections. It will be essential to implement a method which will be able to lookup a city and find its connections if it exists in our map. Storing the nodes in a good way is also essential. Hashing will be used in this case to store the nodes, the name of the current city will be hashed with the prime 541. This means that we will have to create a hash function as well as buckets that can handle collisions, the buckets will in this case store cities.

The next thing we will have to create is the lookup method. It will work as follows, it will check the hash value of the city and determine whether that spot is empty or not. If it is empty it will store the city at that position otherwise it will check the collided values if they are the city which is being looked for. If it goes to the end of the bucket and the city is not found it will store the city there.

```
:
if(cities[index] == null){
    cities[index] = new Buckets(name);
    return cities[index].getCity();
}

if(!cities[index].getCity().Name().equals(name)){
    Buckets pointer = cities[index];
while(!pointer.getCity().Name().equals(name) && pointer.getNext() != null)
        pointer = pointer.getNext();

    if (!pointer.getCity().Name().equals(name)){
        pointer.setNext(new Buckets(name));
        return pointer.getNext().getCity();
    } else{
        return pointer.getCity();
    }
:
```

In our map constructor we will then divide the from city, to city and distance in an array and use our lookup method on the two cities as well as create a connection between the two. This will be stored in a hash table by our lookup method. In our hash table with the prime 541 we will only have 4 collisions with our 52 cities.

The reason that the code above looks so clunky is because it was the easiest way for me to find all the necessary things (name, city, next) so I implemented all these methods to find these things.

Shortest path from A to B

Now that the map has been created the next part is to be able to find the shortest path between two cities. This first attempt is relatively simply but it has some issues. The shortest path is found by going through all the connections from a city and keeping the shortest distance. The max value will also be decreased by the distance in our recursive step so that method does not get stuck in an infinite loop.

```
for (int i = 0; i < from.Connections().length; i++){
    if(from.Connections()[i] != null){
        City.Connection connection = from.Connections()[i];

        Integer addShort = connection.Length();
        Integer distance = ShortestPath(connection.EndCity(),
        to, max-connection.Length());

        if(distance != null)
            addShort += distance;
        else
            continue;

        if(shortest == null)
            shortest = addShort;


        if(addShort < shortest)
            shortest = addShort;
    }
}
return shortest;
```

With this naive implementation we see in table 1 that the time to find the shortest distance is quite long and that some paths takes so long that I just give up. The reason that this takes so long because the max value has to be the correct length and that it will sometime get stuck in loops. When visiting a node we will explore all its connections and find the shortest distance. If the max is high enough and allows us to visit all nodes it becomes clear the time complexity will be a nightmare since every node is visited and revisited. As an example the reason that Göteborg to Umeå and vice versa has such a big difference in time is because it could be that Umeå is linked to Göteborg in a straight line so it is easy to find the shortest distance. Göteborg however might be connected to lets say 6 other cities which in turn create has a huge network of other nodes. The method will then look through these big networks and search for a shortest distance to Umeå even though they might not be connected to Umeå at all. This is not how the nodes are connected but it shows an example as to why the time difference is so great.

Detecting loops

The next approach to finding this shortest distance is by storing cities we have visited in our path in an array and if we have been in this city before we will continue on with another path.

This version of the code will check for loops by saving cities we have visited in an array, everytime we exit the recursive call we will remove the city if that connection was not a part of the shortest path. By implementing it this way we will not get stuck in loops obviously but we will also be to find the shortest path faster since we will not be searching through cities that have already been explored. Lets say node A, B and C are all connected to eachother if we go from A to B to C we will then not go from C to A or B looking for a new shorter path since we have already been to those cities.

As seen in table 2 this method is way faster than the previous implementation and it is actually possible to find all the different shortest distances. The longest travel which can be done in this is example is from Malmö to Kiruna, this is done in 600 ms in this implementation and the distance is 1162 minutes.

The next and final implementation of finding the shortest path is by allowing the code to have a dynamic maximum path. This means that there will be a max value which will be changing depending on the first shortest path we find. Say we have nodes A, B, C, D and E. If A connected to all but E and we are trying to find a path from A to E this implementation will work as follows. We find a path from A to B and then a path from B to E, this distance will be our current shortest distance and our max value will be set to this. All our paths from A to E must then be shorter than that max value. If we find a path that is shorter that distance will be the new max value. This will continue on until the shortest distance is found.

The implementation of this algorithm was surprisingly easy. Just copy the code from the previous solution and check if max is null or not and then do the same steps as in the previous solution. The other thing to be added is that the max value should be updated when the shortest path is found.

```
if (max == null)
    :
    if(shortest == null){
        shortest = addShort;
        max = addShort;
    }

    if(addShort < shortest){
        shortest = addShort;
        max = addShort;
    }
```

```
if (max != null){
    :
    if(shortest == null){
        shortest = addShort;
        max = addShort;
    }

    if(addShort < shortest){
        shortest = addShort;
        max = addShort;
    }
    :
```

With this solution we can see in table 3 that the time to find the shortest distance is now incredibly faster than the two previous attempts. With this solution the time to find Malmö to Kiruna was also tested and in this implementation the time it took to find it was 250 ms.

Benchmarks

Shortest time	Time taken ms	From city	To city
153	1100	Malmö	Göteborg
211	1200	Göteborg	Stockholm
273	1200	Malmö	Stockholm
327	900	Stockholm	Sundsvall
gave up	-	Stockholm	Umeå
gave up	-	Göteborg	Sundsvall
190	2200	Sundsvall	Umeå
705	6	Umeå	Göteborg
gave up	-	Göteborg	Umeå

Table 1: Table to see the shortest time between some cities and the time measured to find this path in ms

Shortest time	Time taken ms	From city	To city
153	170	Malmö	Göteborg
211	130	Göteborg	Stockholm
273	170	Malmö	Stockholm
327	120	Stockholm	Sundsvall
517	170	Stockholm	Umeå
515	130	Göteborg	Sundsvall
190	360	Sundsvall	Umeå
705	220	Umeå	Göteborg
705	150	Göteborg	Umeå

Table 2: Table to see the shortest time between some cities with a better solution. Time measured to find this path in ms

Shortest time	Time taken ms	From city	To city
153	1	Malmö	Göteborg
211	1	Göteborg	Stockholm
273	1	Malmö	Stockholm
327	5	Stockholm	Sundsvall
517	25	Stockholm	Umeå
515	13	Göteborg	Sundsvall
190	400	Sundsvall	Umeå
705	2	Umeå	Göteborg
705	60	Göteborg	Umeå

Table 3: Table to see the shortest time between some cities with an even better solution. Time measured to find this path in ms