Graphs

Emanuel Paraschiv

Fall 2024

Introduction

In this assignment we are going to implement a new data structure, the graph. To make this experiment seem more practical we will take a real life example: our graph will be a map of the train system in Sweden, with its connections between different cities. After we implement this graph, we will try different algorithms for finding the shortest path.

Graph

Our graph will consist of 52 nodes (cities) and 75 edges (connections) and will be bidirectional.

We can view each city as a Node, having a 'name' propriety and a 'connections' propriety that only represents the Nodes it is connected to.

```
public City(String name)
    this.name= name;
    this.connections= new Connection[60];
    this.count = 0;
```

We also have another structure, a Connection. This will have 2 proprieties: the destination (or neighbor) and the time it takes to reach it:

```
public Connection(City neighbour, int time)
    this.neighbour= neighbour;
    this.time= time;
```

In order to connect 2 cities we have a method connect():

```
public void connect(City nxt, int dst)
    //find the first available space in the connections
    for(int i= 0; i < connections.length; i++)
        if(connections[i]==null) { //find empty space
            Connection newC=new Connection(nxt, dst);
```

```
            connections[i]=newC; //add it
            return;
        throw new IllegalStateException("No more space");
```

Once these 2 structures are done we can implement the `Map`, which is simply a collection of cities.

A very efficient way of building a Map is to use Hash Tables, which will provide fast access. To start with, we need a hash function that given a String input will return an index:

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

My choice was to implement the Hash Tables using the Buckets technique. This will provide us a faster execution time. Now we can implement a `lookup()` function that will check if a City exists already and it will return it. Otherwise, if it is not present, the method will add it to the Map.

Firstly it will obtain the index of the city:

```
int inx = hash(name, mod);
```

Then it will search for it in our map if it is present, we return it:

```
for (int i = 0; i < cities.length; i++)
        if (cities[inx] != null && cities[inx].getN().equals(name))
            return cities[inx]; //city found, return it
```

Otherwise, we move to the next index in case of collision and continue our search. If the City can't be found, then we add it:

```
 // If city not found
City newC = new City(name);
cities[inx] = newC;
return newC;
```

Firstly to test for collisions I have mistakenly used a bigger size of array than necessary (80). However, after noticing that there are only 52 cities I decreased the size to 60 entries and got the following results when testing for collision:

Key map to same index	1	2	3	4	5	6
Collisions	20	11	11	0	0	0

Table 1: Collision Results for Hash Table Size 60

Shortest path

Now that our Graph is ready to use, we can think of creating an algorithm for finding the shortest path between 2 Cities. To make things simple we will only be interested in the execution time and not in the actual path followed, at least for the beginning.

We can start by implementing a 'Depth First Search' Algorithm. It will only return the shortest distance. One idea to prevent it from running for an infinite amount of time is to set a maximum time. This way, if we enter a loop it will stop after it reaches the maximum time.

As we will see, this method makes use of multiple `getter()` methods. In the beginning we will check some exception cases:

```
private static Integer shortest(City from, City to, Integer max)
    if (max< 0) return null;
    if (from==to) return 0;   // if we are at destination
```

After that we store the neighbors of the starting city in an array. Then we access each of them individually and get the time it takes to perform it:

```
Integer shrt= null;  //track shortest distance
//loop through all neighbors of current city
Connection[] neighbors = from.getConnections();

for (int i=0; i<neighbors.length; i++)
    if (neighbors[i]!=null)
        Connection conn= neighbors[i];
        City neighbor= conn.getN();
        Integer distance= conn.getT();
        Integer remMax = max - distance;
```

After we take the time we subtract it from the maximum allowed time and then call the method recursively from that node on.

```
Integer dist = shortest(neighbor, to, remMax);
// If a valid path is found
if(dist!= null)
    //  total distance including the distance to neighbor
    Integer totalDist = dist + distance;
```

If it is needed(shrt is null or the totalDist is smaller than shrt), we also update the it, and in the end we return shrt:

```
        shrt = totalDist;
    return shrt;
```

After running a benchmark to test the efficiency we notice however some problems. I have tried to adapt the maximum time to match the execution time, but for 2 connections the time was too long, as we can see from the following table:

Path	Distance Time	Time ms
Malmö to Göteborg	247	5
Göteborg to Stockholm	426	3500
Malmö to Stockholm	407	270
Stockholm to Sundsvall	327	320
Stockholm to Umeå	*No valid path*	
Sundsvall to Umeå	190	0
Umeå to Göteborg	861	11000
Göteborg to Umeå	*No valid path*	

Table 2: Shortest Paths and Travel Times

The DFS algorithm works by agresively going in depth. If it is lucky, it will find the target city very fast. However, otherwise it would have to start searching though the other nodes in the graph, possibly resulting in looping or exploring sub-optimal paths. It can be the case that as we go from Umeå to Göteborg, the algorithm finds a direct path, but when starting from the other end can involve multiple branches or even loops.

Detecting loops

Now that we know what the problem can be, it would be a great idea to try and prevent it.

In order to do so, we can add a simple condition to our algorithm: in the beginning we create an array (path) of cities that we visit, and when we are trying to access a new city, we first check our array. If it it already there, it means we are going to loop and in order to avoid that, we abort the execution:

```
for (int i= 0;i < sp;i++)
        if (path[i].equals(from))
            return null;
```

With this change, our algorithm is now able to find all paths in the benchmark, and the execution time has, overall, decreased:

Path	**Distance in min)**	**Time ms**
Malmö to Göteborg	247	152
Göteborg to Stockholm	426	58
Malmö to Stockholm	407	29
Stockholm to Sundsvall	327	11
Stockholm to Umeå	517	21
Sundsvall to Umeå	190	49
Umeå to Göteborg	861	20
Göteborg to Umeå	861	28

Table 3: Shortest Paths and Travel Times

There is however still room for improvement. Even though now we are avoiding loops, we still have a limitation which we can address by introducing a concept named "dynamic maximum". For short, our algorithm will start as before, with no restrictions. When it finds its first connection, it will record it as the maximum time spent possible. When it will explore other paths it will disconsider any longer one than the present maximum, and will accept only the ones shorter than it, updating constantly. Now we can run our benchmark again, and obtain the following values:

Path	**Distance in min**	**Time ms**
Malmö to Göteborg	247	20
Göteborg to Stockholm	426	18
Malmö to Stockholm	407	30
Stockholm to Sundsvall	327	9
Stockholm to Umeå	517	12
Sundsvall to Umeå	190	20
Umeå to Göteborg	861	11
Göteborg to Umeå	861	12

Table 4: Shortest Paths and Travel Times

Also, the shortest path from Malmö to Kiruna is achieved in 1214 min (87 ms)

Even though our depth first search algorithm seems to work for our small set of cities, we can still notice a decrease in efficiency as the 'Nodes' go further away. We can take a visual example, starting from the south-most city in Sweden, Malmö, and advance step by step, further and further to the north.

Destination City	Distance in min	Time ms
Göteborg	247	17
Stockholm	407	30
Sundsvall	652	55
Umeå	842	65
Kiruna	1214	87

Table 5: Distances and Travel Times to Destination Cities

Conclusion

Therefore, Graphs represent a very comprehensive data structure. This experiment made it easier to grasp and test different methods and implementations on it, allowing us to test it and choose the best approach.