Graph

Mahad Ahmed

Fall 2022

Introduction

In this assignment we explored a more general data structure called *graph*.

Graph

A graph is a data structure containing a set of vertices which are connected through lines called *edges*. Simply put a graph is a set of nodes connected to other nodes. A *path* is a sequences of nodes that allows you to go from node to another which are not directly connected.

We have worked with graphs in previous assignment, for example trees. Trees are a special type of graph that has a root node and directed edges. Linked list is also a type of graph which is even simpler than a tree.

A train from Malmö

In this task we implemented something realistic, namely the railroad network of Sweden using a graph data structure. We used a map containing 52 cites and 75 connections. A connection a bidirectional. Think of it as a two-way train ticket, to and from. A connection between Uppsala and Stockholm can be used in either direction.

The first task was to take the description of all the connections and turn this into a graph. Later on it will be used to find the shortest path between cites.

the graph

We were given a "map" which is a *cvs file* containing a list of all of the connections in the form of (from,to,minutes). So how have we designed our graph?

We have two classes, *City* and *Connections*. The object City which has a name as a string and neighbors which are connections as an array i.e an array with connections. The class *Connections* has an object called Connections, which takes in as parameters a city and a distance. The connections gives us the city as well as how much time it takes to get there. Easy to implement the two classes so far.

We also a add method where we add new connections to a city. Also easy to implement. We take as argument a destination which is a city and the distance it takes to get there. Then we put it inside of the array called neighbors with the other connections for the same city. We just put it at the first available spot in the array.

Code implementation of add

Here is the code for adding a connection to a city.

```
public void addConnection(City destination, Integer distance){
        Connection Connection = new Connection(destination,
            distance);
        for (int i = 0; i < neighbors.length; i++) {
            if(neighbors[i] == null){
                neighbors[i] = Connection;
                return;
            }
        }
    }
```

hashing to our rescue

In a previous task we have worked with hash functions and it's implementation. Now we have used here so we don't waste space and have a quick lookup. We were provided with hash function which looks something like this:

```
private int hash(String name) {
        int hash = 7;
        for (int i = 0; i < name.length(); i++)
            hash = (hash*31 % MOD) + name.charAt(i);
        return hash % MOD;
    }
```

The mod used was 541 which is a prime. As said in previous assignments using a prime number gives us better hashing without as many collisions.

the map

We received a skeleton code for a class called *Map*. It has city and mod as properties. We also received how to read our cvs file with the cites and distances.

For this task here we implemented a method called *lookup*. The hash function is also implemented in the Map. What the lookup method should do is given a name, return the city if it is present in the array. If it is not present then create the city.

Then for each row in the cvs file we do a lookup between two cites and add a connection to each of them. Now we can use this to find the shortest path between to cites.

```java
public City lookup(String name) {
      int index = hash(name);
      for (int i = index; i < cities.length; i++) {
          if (cities[i] == null) {
              cities[i] = new City(name);
              return cities[i];
          }
          else if (cities[i].name.equals(name))
              return cities[i];
      }
      for (int i = 0; i < index; i++) {
          if (cities[i] == null) {
              cities[i] = new City(name);
              return cities[i];
          }
          else if (cities[i].name.equals(name))
              return cities[i];
      }
      return null;
  }
```

Shortest path from A to B

Here come different implementation to find the shortest path.

Naive

In this task we did a depth-first search to find the shortest path. This implementation is naive as per it's name. Because when doing DFS i doesn't guarantee that a node let's call it node 1 is visited before another node 2 starting from the beginning vertex. Then this means node 1 is closer to the source. So when we get a "shortest path" it is no guarantee that this is indeed the shortest path.

Also when doing DFS we look after all possible paths before returning the shortest. Sometimes we can even get stuck at an endless loop. So how are we going to solve this?

a max depth

We use a max value to prevent when searching a path that it will end up in a endless loop. So for example when we want to now what the shortest path from Stockholm to Karlskrona (my hometown) we set a max value which is a the time it takes to get there. We set the max value to 400 minutes. So we want to find the shortest path but it must be less than 400 minutes. If it takes more than that then we will look for another path.

Benchmarks

Max	**From**	**To**	**timeToFind (ms)**	**Time(min)**
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	10	273
400	Stockholm	Sundsvall	269	327
600	Stockholm	Umeå	558636	517
600	Göteborg	Sundsvall	436585	515
200	Sundvall	Umeå	0	190
800	Umeå	Göteborg	27	705
800	Göteborg	Umeå	come bruh	705

Table 1: Benchmark for Naive

The naive implementation was not all that good. Because it could get stuck in an endless loop. Also it checks every single node before it returns a shortest path. This will inevitably take a long time. I even gave up doing some of the benchmarks.

Max	From	To	timeToFind (ms)	Time(min)
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	0	273
400	Stockholm	Sundsvall	3	327
600	Stockholm	Umeå	10	517
600	Göteborg	Sundsvall	4	515
200	Sundvall	Umeå	0	190
800	Umeå	Göteborg	1	705
10 000	Malmö	Kiruna	231	1162

Table 2: Benchmark for path

This path implementation is better because we can abort a search if we already know that it is not shorter than the other path saved in our array. This will save us a lot of time and the computer can get some well needed rest. In this implementation we don't need a max value since we never end up in a endless loop.

Max	From	To	timeToFind (ms)	Time(min)
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	0	273
400	Stockholm	Sundsvall	2	327
600	Stockholm	Umeå	12	517
600	Göteborg	Sundsvall	5	515
200	Sundvall	Umeå	0	190
800	Umeå	Göteborg	1	705
800	Göteborg	Umeå	50	705
10 000	Malmö	Kiruna	200	1162

Table 3: Benchmark for path slightly improved

The last implementation is a little bit faster than the previous one. In this implementation you set the max value after we find a path. So if we find a path from A to B takes 60 and B to Z takes 400 minutes. Then we know that if A is connected to multiple connections like C,D then it must be shorter than 460 minutes. We see small improvements from the previous one.