Graphs in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment, I will explore another more general data structure called a graph, using the Swedish railway network. A graph is a set of nodes that can be connected through edges, so e.g. a linked list or a binary tree that we previously saw are specific types of graphs. The map I am using has 52 Swedish cities and 75 (bidirectional) connection between them.

Creating the map

In my implementation I followed to given material closely. A have the `City` class storing the name of the city and an array of (up to) 4 connections. A `Connection` stores only the destination `City` and the length of the route in minutes.
My function `find` attempts to lookup a `City` in the graph and if it fails to find it, it creates a new `City` with the given name and return its reference. Since we mostly want to read the graph, it would make sense to opt for O(1) access time. As we learned last week, hashing is a great way to do so. I used the given hash function and decided to go with open addressing mainly because I find it easier, but also because it's faster and has better cache performance. First I used `mod` = 79 and got a lot of collisions, so by trial and error I managed to reduce this to 13 with `mod` = 223. My implementation of the `find` function is shown below.

```
public City find(String name){
    Integer index = hash(name, 223);
    if (cities[index]==null){
        cities[index] = new City(name);
        return cities[index];
    }
    while(cities[index%cities.length]!=null){
        if(cities[index%cities.length].name.equals(name)){
            return cities[index%cities.length];
```

```
        }
        index++;
    }
    cities[index%cities.length] = new City(name);
    return cities[index%cities.length];
}
```

A row in our given input file `trains.csv` contains one connection per row in a *from,to,minutes* format (with UTF-8 encoding). For populating the graph, I read the file row by row, find both City objects and create two new `Connection`-s (one in each directions) between the two cities. I think it would be neater to have one Connection object represent both directions (so that a Connection object would store both to and from) but I decided to follow the given aid. My `connect(City to, int length)` method of a City class adds a connection to the first empty slot in the City's connections array:

```
public void connect(City to, int length){
    for (int i = 0; i<4; i++){
        if(this.connections[i]==null){
            this.connections[i] = new Connection(to, length);
            return;
        }
    }
    System.out.println("More than 4 connection for: "+this.name);
}
```

As you can see, I take advantage of the fact that a `City` has at most 4 connections, but just to make sure, I print a message for myself in case I ever try to add a 5th connection.
I used these two methods to populate my graph, as presented in the following code:

```
public Graph(String file){
    this.mod = 223;
    this.cities = new City[mod];
    try (BufferedReader br = new BufferedReader(new
    InputStreamReader(new FileInputStream(file), "UTF-8"))){
        String line;
        while ((line = br.readLine()) != null) {
            String[] row = line.split(",");
            City one = this.find(row[0]);
            City two = this.find(row[1]);
            one.connect(two, Integer.parseInt(row[2]));
            two.connect(one,Integer.parseInt(row[2]));
```

```
        }
    } catch (Exception e) {
        System.out.println(" file " + file + " not found or
        corrupt");
    }
}
```

Now that my graph is populated, I am ready to move on to the bigger part of the assignment. The task is to find the shortest path from city A to city B.

Simple solution

For now, all we care about is the length of the path (in minutes) and not the cities we pass through. First we will use a depth-first search. The problem we will encounter, is that as opposed to the binary tree we saw before, our map can (and does) have loops in it, where we may get lost in it. One way to make sure that the search doesn't enter an infinite loop is to include a max value, which if reached forces the program to consider another path.

```
private static Integer shortest(Graph.City from, Graph.City to,
Integer max) {
    if (max < 0) return null;
    if (from == to) return 0;
    Integer shrt = null;
    for (int i = 0; i < from.connections.length; i++) {
        if (from.connections[i] != null) {
            Graph.Connection conn = from.connections[i];
            Integer curr_dist = shortest(conn.to, to,
            max-conn.length);
            if (curr_dist!=null){
                if (shrt==null || shrt>curr_dist+conn.length){
                    shrt = curr_dist+conn.length;
                }
            }
            //else we reached to, move on with next conn
        }
    }
    return shrt;
}
```

benchmarks

To see how this naive version performs, I was given specific cities to test how long it takes for the program to find the shortest path. I needed to adjust

the max value a couple of time to find the best results, but I decided to do increments of a hundred. My results are in the table below:

trip	shortest path [min]	search time [ms]	max value
Malmö to Göteborg	153	0	200
Göteborg to Stockholm	211	1	300
Malmö to Stockholm	273	2	300
Stockholm to Sundsvall	327	23	400
Stockholm to Umeå	517	35k	600
Göteborg to Sundsvall	515	50k	600
Sundsvall to Umeå	190	0	200
Umeå to Göteborg	705	3	800
Göteborg to Umeå	705	N/A	800

For Göteborg to Umeå I let the program run for 10+ minutes with no result. The problem is that it depends too heavily on the order of which the connections are listed. In one direction, you may quickly exceed the max limit in every connection before finding the shortest one, while in the other direction, you probably have to go through a lot of connections that doesn't exceed the max just yet, but will eventually.

Improved implementation

Another way to handle the problem with loops is to try and detect them before entering one. In order to do this, we need to keep track of the cities visited. Before adding a new one, we check if we've already been there.
I created a class that is responsible for finding the shortest path. It holds an array of length 52, which is arguably much longer than needed, as there is no two cities that has a shortest path that visits all 52 cities, but since this is what was given to us, I didn't change it. The array is used as a stack, so a stack pointer was also necessary to keep track of where the last city we visited was written to. My code is shown below:

```
private Integer shortest(Graph.City from, Graph.City to) {
    if (from==to) return 0;
    Integer shrt = null;
    for (int i = 0; i < sp; i++) {
        if (path[i] == from)
        return null;
    }
    path[sp++] = from;
    for (int i = 0; i < from.connections.length; i++) {
        if (from.connections[i] != null) {
            Graph.Connection conn = from.connections[i];
```

```
                Integer dist = shortest(conn.to, to);
                if (dist!=null){
                    if (shrt==null || shrt>dist+conn.length){
                        shrt = dist+conn.length;
                    }
                }
            }
        }
        path[sp--] = null;
        return shrt;
    }
```

benchmarks

To compare, I ran the same benchmarks as before:

trip	shortest path [min]	search time [ms]
Malmö to Göteborg	153	138
Göteborg to Stockholm	211	61
Malmö to Stockholm	273	111
Stockholm to Sundsvall	327	95
Stockholm to Umeå	517	141
Göteborg to Sundsvall	515	120
Sundsvall to Umeå	190	307
Umeå to Göteborg	705	122
Göteborg to Umeå	705	173

As you can see, it's a lot more balanced this time. It also doesn't require us to set a max value, which makes it easier to include in bigger applications.

An even better implementation

Even more improvement can be done, by using a dynamic value for max, meaning that we start with max = null, and once we find a path to the destination, we set that distance as max, so from then onward, we only look at shorter paths. This will save us time by not having to examine all paths from A to B but rather aborting them once we're certain it is not the shortest. Here's my code modified from the previous two implementations:

```
private Integer shortest(Graph.City from, Graph.City to,
Integer max) {
    if (max!=null && max<0) return null;
    if (from==to) return 0;
    Integer shrt = null;
```

```
    for (int i = 0; i < sp; i++) {
        if (path[i] == from)
        return null;
    }
    path[sp++] = from;
    for (int i = 0; i < from.connections.length; i++) {
        if (from.connections[i] != null) {
            Graph.Connection conn = from.connections[i];
            Integer curr_dist;
            if (max==null) curr_dist = shortest(conn.to,to,max);
            else curr_dist = shortest(conn.to, to, max-conn.length);
            if (curr_dist!=null){
                if (shrt==null || shrt>curr_dist+conn.length){
                    shrt = curr_dist+conn.length;
                    max = shrt;
                }
            }
        }
    }
    path[sp--] = null;
    return shrt;
}
```

benchmarks

I once again tested the same searches, my results are shown below: As

trip	shortest path [min]	search time [ms]
Malmö to Göteborg	153	0
Göteborg to Stockholm	211	0
Malmö to Stockholm	273	0
Stockholm to Sundsvall	327	1
Stockholm to Umeå	517	5
Göteborg to Sundsvall	515	3
Sundsvall to Umeå	190	200
Umeå to Göteborg	705	0
Göteborg to Umeå	705	18
Malmö Kiruna	1162	69

a comparison, Malmö to Kiruna (which is possibly the longest trip in this database) took 480 ms with the previous implementation, and about an eternity with the first program.

Summary

We have seen three different programs that perform the same task: finding the shortest path from A to B on a map of Swedish cities and found an implementation that works decent for this problem size. However, if we were to upgrade to a map of Europe, let alone a world map, it would quickly turn out to be rather inefficient. The next step would be to give the map some "memory" so that we don't have to calculate the same thing over and over again - but this is left for assignment A which I'm going to take a look at later, if time allows.