Graphs

Algorithms and data structures ID1021

Johan Montelius

Fall term 2023

Introduction

In this assignment you will explore a more general data structure called *a graph*. A graph is simply a set of *nodes* (or *vertices*) connected through *edges* (or *links*). The edges can be linked together to form *paths* that connect nodes that are not directly connected. Edges can be directed so even though there is a connection from node A to B there might not be a connection from B to A.

In one way we have already worked with graphs even though the graphs that we have seen have been very simple. A tree is a special graph where we have *a root*, directed edges and where each node is only the destination of one edge (apart from the root that has none). This means that if we follow the left branch from a node we will find a set of nodes that are completely separated from the nodes found in the right branch. Moreover, and more important, there are no circular paths in a tree. Since there are no circular structures in a tree life becomes simple.

A linked list is also a graph and this is of course an even simpler structure. What we will explore in this assignment is what happens when we let go of any restrictions and allow nodes to have links to any other link in the graph - a small step for rules, a giant leap for complexity.

A train from Malmö

To have something realistic to work with we will use the railroad network of Sweden. The map that we will use has 52 cities and 75 connections. A connection is bidirectional so when we have a connection between Stockholm and Uppsala this can be used in either direction. Your first task is to take a description of all the connections and turn this into a graph that we will later use to find the shortest paths between cities in Sweden using trains.

the graph

The map that we will use is a *csv file* listing connections on the form "from,to,minutes". We should read the file and for each line first make sure that the two cities are known and then add the bidirectional connection to the graph. The first question we need to answer is how to represent the graph. Our choice depends on how we are going to use the graph and since we are going to search for paths we need to quickly retrieve the directly connected cities given a city.

One solution is to represent a city as a structure with a *name* and the *neighbors* represented as an array holding the direct *connections*. A connection is simply a *city*, and a *distance* in minutes.

Create two public classes, `City` and `Connection`. Apart from constructors the city class should have a public method to add a new connection to the city. The method should take the destination city and distance as arguments `connect(City nxt, int dst)`.

When you have the city class defined it's time to implement the map. The map will be a collection of cities and we need a way of quickly lookup the object that represents a city given the name of a city. The `lookup` procedure is only used when we add new cities to the map and when we start to traverse the map for example to find the node that represents "Malmö" and the node that represents "Mjölby". Once the nodes are found all information we need are given by the city objects.

hashing to our rescue

Since you now know how a hash table works this is a golden opportunity to use one. If you create a hash function that takes a string as argument you can produce a hash value by being creative in how to add the characters together we get something that works surprisingly well. Let's do something like follows:

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

The `mod` value is selected to be a large enough value to allow our cites to spread out in an array yet small enough to not waste space. A prime is preferred and why not choose 541 since it will allow our 52 cites ample of room in an array. We will have some collisions, so you need to be able to handle them.

the map

The following skeleton code should give you an idea of what the map looks like. Start by implementing the methods: `hash`, `lookup`. The lookup method should return the city given a name if present in the array of cities or create the city if it is a name we have not seen before.

```
public class Map {

  private City[] cities;
  private final int mod = 541;

  public Map(String file) {

      cities = new City[mod];

      try (BufferedReader br = new BufferedReader(new FileReader(file))) {
          String line;
          while ((line = br.readLine()) != null) {
            :
          }
      } catch (Exception e) {
          System.out.println(" file " + file + " not found or corrupt");
      }
  }
  :
}
```

For each row in the map file you should lookup the two cites and then add a connection to each of them. When you have handled all rows in the file your map is ready to be used. How many collisions do we have among our 52 cities?

Shortest path from A to B

Your task is now to implement a program that finds the shortest path between two cities. We will, at least to start with, ignore what the path looks like and be satisfied if we can find the shortest path and present how many minutes the trip will take.

Your first solution will be a very simple solution. It will turn out that this simple solution has severe limitations and your task is to identify the problem.

depth-first

The strategy that you will use is to do a *depth-first search* for the shortest path. Create a file `Naive.java` and add a main method that looks something like this:

```
public class Naive {

    :

    public static void main(String[] args) {

        Map map = new Map("trains.csv");

        String from = args[0];
        String to = args[1];
        Integer max = Integer.valueOf(args[2]);

        long  t0 = System.nanoTime();
        Integer dist = shortest(map.lookup(from), map.lookup(to), max);
        long time = (System.nanoTime() - t0)/1_000_000;

        System.out.println("shortest: " + dist + " min (" + time + " ms)");
    }
}
```

a max depth

The `max` value is set to prevent the search for a path to end up in an endless loop. Since our search strategy will be depth-first we will consider going from ”Malmö” to ”Lund” and from there to ”Malmö” to take the train to ”Lund” etc. We either need to identify loops as we go or set a maximum allowed path. We can for example say that we want to find the shortest path form ”Malmö” to ”Göteborg” but it must be less than 300 minutes (the quickest is 153 min). Even if our algorithm will consider going back and forth to Lund several times it will loose 26 minutes in each round trip. After 11 trips it is running out of time and must consider some other alternative.

This skeleton code should get you started. When searching for the shortest path from `from` to `to` then either you do not have more time, you are actually there or you need to find the shortest path using one of the connections.

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

```
    if (from == to)
       return 0;

    Integer shrt = null;

    for (int i = 0; i < from.neighbors.length; i++) {
      if (from.neighbors[i] != null) {
        Connection conn = from.neighbors[i];
        :
        :
      }
    }
    return shrt;
  }
```

When you try a connection you first determine how long time you have left. If you should find a path from A to B in less than 300 minutes and A is connected to C with a direct line that takes 52 minutes then you need to find a path from C to B in less than 248 minutes. If you try all connections of A and keep the shortest one you have solved the problem.

some benchmarks

Let's try to find the shortest paths for some trips, try these (you have to give a high enough max value or a path is not found). Present the minimum path found and how long time it took to find the path (or if you gave up).

- Malmö to Göteborg
- Göteborg to Stockholm
- Malmö to Stockholm
- Stockholm to Sundsvall
- Stockholm Umeå
- Göteborg to Sundsvall
- Sundsvall to Umeå
- Umeå to Göteborg
- Göteborg to Umeå

At first you might start looking for bugs in you code but the problem is not the code, the problem is complexity. Why can you find a path from

Umeå to Göteborg but not (maybe you had enough patience, how long time did it take) from Göteborg to Umeå? How is it that we can find a path from Göteborg to Umeå but have so much problems finding the path in the opposite direction?

detect loops

We know that loops are part of the problem so let's find a way to avoid loops. Create a new program `Paths.java` that is a copy of your current program. Let this version hold an array called `path` that is large enough to hold any path (no path will be longer than the number of cites). Then as you go down in the search, first check that the city that you're visiting is not in the path already. If it is, we abort the search (return null).

```
public class Paths {

    City[] path;
    int sp;

    public Paths() {
        path = new City[54];
        sp = 0;
    }

    private Integer shortest(City from, City to) {

      :
      for (int i = 0; i < sp; i++) {
        if (path[i] == from)
          return null;
      }
      path[sp++] = from;
      :
      :
      path[sp--] = null;
      return shrt;
    }
    :
}
```

In this version we do not need the max value since we will not end up in an infinite loop anyway. Re-run the tests you did before, any improvements? We have a solution that does not require us to set a maximum path (that might be too short) but the performance might not be the best.

To improve things we can work with a dynamic maximum path. We start off with a max value set to `null` i.e. no restrictions. As soon as we find a path to the destination we know that the shortest path must be short so we have our maximum value.

This works recursively so in each step we can refine the maximum depth. Let's say you're searching for a path from A to E and have the max value set to `null`. Let's now say that A is directly connected to B, C and D and when you try B you find a path from B to E that is 460 minutes. If the distance from A to B is 60 minutes, then any other path from A to E must be shorter than 520 minutes. If you do these updates of the `max` value when you try the different direct connections you do not need an initial max value (it can be `null` until we actually find a path).

Implement this improvement and again try to find the shortest path from Malmö to Kiruna. Any improvement?

Things to ponder

All though you now have a solution that works for this example it is still very inefficient for larger maps. Make a table and see how long time it takes to find a path from Malmö to different cites further and further away. The graph that we are working with only has 52 nodes, what would happen if we increased the map to have a hundred nodes? Would your implementation be able to find the shortest paths given a map of Europe?

The problem with the current solution is that it does not remember anything. When searching for the best path from Malmö to Kiruna at one point it the algorithm finds it self in Stockholm. It then finds the shortest path from Stockholm to Kiruna (889) and uses this to calculate the shortest path from Södertälje to Kiruna (910).

If the search came from Norrköping we then know the shortest path from Norrköping (969) if going over Södertälje. Before we leave Norrköping we want to explore the connection going to Katrineholm. When we do this it would be very valuable to remember what the best path from Stockholm to Kiruna was. If we do not remember we will have to explore all paths from Stockholm again.