Graph

William Frid

Fall 2023

Introduction

Graphs used as data structures come in many forms and complexities. We've previously explored graphs of the types tree, trie, linked list, and so on. What these types have had in common is simplicity as they either had a maximum of two branches (connected nodes) or were one-dimensional (linked lists) that could however go both directions.

This report will be using a cut-down version of the Swedish railway system as an example while exploring a more complex form of graph alongside path-finding algorithms. The graph will consist of cities and connections (edges and nodes), with each city having a minimum of one connection up to many. The first part will focus on importing the data and constructing the graph (which we'll be called `Map` henceforth), and after that, explore three algorithms for path-finding which we'll be calling `Naive`, `Paths`, and `PathsLimit`.

Constructing The Map

The first step of building the map is to, inside the class `Map`, use the constructor and import a given CSV file consisting of the data. By using a loop, the code can fetch the three arguments from, to, and distance and pass those to a separate method called `addConnection()`. This method is key and clearly describes what needs to be done in order to build up the graph. Have a look at the snippet below.

```java
private void addConnection (String from, String to, int di) {
    City fromCity = lookup(from);
    City toCity = lookup(to);

    if (fromCity == null)
        fromCity = addCity(from);
    if (toCity == null)
        toCity = addCity(to);
    fromCity.addConnection(toCity, di);
    toCity.addConnection(fromCity, di);
}
```

The first step in `addConnection()` is to search for the target cities using a lookup algorithm. This can easily accomplished since the method `addCity()` doesn't just create new cities but also inserts them into a hash table with open addressing using a helper method called `hash()`. Now let's say we were to insert "Stockholm" and "Södertälje" (as those two are in fact connected). Then `addConnection()` first discovers that none of them exists. So it creates them and adds them to the hash table. Next, it calls upon each city's inner method `addConnection()` (mustn't be confused with the outer method which shares the name) with the other city as an argument, as the connections go both directions. The inner `addConnection()` simply receives a destination city and distance as arguments, generates an object of type `Connection`, and adds that to the main city's connection array.

Have a look at the snipped illustrating the `City` class below as this can help add understanding for this structure. Note that the class `City` is nested inside `Map` as it's a part of the map.

```
class City {
    String name;
    Connection[] connections;

    City (String n) { ... }
    void addConnection (City dest, int di) { ... }

    class Connection {
        City destination;
        int distance;
        Connection (City dest, int di) { ... }
    }
}
```

To analyze that the construction of the graph worked correctly, a method called `print()` was written that printed out each city with its connections. This is highly recommended if you wish to follow along in this report.

Naive

The First path-finding algorithm, called `Naive`, has its name for the reason that it is indeed naive. It doesn't take into consideration loops nor remember previous paths. This leads, in theory, to poor performance. But enough with the speculations, let's instead have a look at the code (inserted below). The first thing we notice is how it's recursive and works using a parameter called `max` which is an argument holding the max time the journey gets to take since there's no other way to prevent stack overflows in this case.

Assume we wish to find the shortest path from Malmö to Göteborg that takes a maximum of 300 minutes. Then we can upon the method `shortest()` with these arguments. The thing happening is that the two if-statements of the method perform checks (more on these two later). Then the iteration over the `from` city's connections starts with the for loop. This is where it explores the different possible paths. For each connection, it calls upon itself using recursion. If the city the connection leads to is the goal, then it returns the value 0, and if it exceeds the given max time, it returns null (the two if-statements at the beginning).

Next, process the returned result. If it is null, we know it was a bad path, and continue the interaction of the connections. Otherwise, we set a new shortest value equal to the distance between this city, and the city the connection leads to. Note that this is where it becomes more complex as this happens on multiple levels. Malmö, for instance, is only connected to Lund, which is connected to multiple other cities. If we assume we called upon this method using Lund as the argument `from`, then it would use the for loop, compare the times it takes to get to Göteborg via all connections it has and save the fastest one.

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

    Integer shrt = null;
    for (Map.City.Connection conn : from.connections) {

        Integer t = shortest(conn.destination, to, max - conn.distance);

        if (t == null) // Bad route.
            continue;

        Integer new_shrt = conn.distance;
        if (t != 0)
            new_shrt += t;

        if (shrt == null || shrt.compareTo(new_shrt) > 0)
            shrt = new_shrt;
    }
    return shrt;
}
```

Now if we benchmark this algorithm using 9 different paths, we can the results seen in Table 1 seen below. During the benchmarking, the maximum allowed time had to be increased to be able to find the longer paths. This, however, led to extremely slow searches in some cases as the path Göteborg to Umeå for instance was never even found (as time was of the essence when running this benchmark). Based on these results we can conclude that the algorithm `Naive` isn't suitable for this type of work since its time complexity increases rapidly and makes even a simple search from Stockholm to Umeå take 39 seconds.

From	To	Distance(min)	Search(ms)
Malmö	Göteborg	153	0
Göteborg	Stockholm	211	1
Malmö	Stockholm	273	1
Stockholm	Sundsvall	327	19
Stockholm	Umeå	517	39000
Göteborg	Sundsvall	515	62000
Sundsvall	Umeå	190	0
Umeå	Göteborg	705	7
Göteborg	Umeå	X	X

Table 1: Benchmark for algorithm Naive.

Before we continue, notice how the algorithm was able to find the path from Umeå to Göteborg, but not the other way around. This is because of how the graph is built and where the loops are located. This can be fixed by preventing the loops, which is was the next section will cover.

Paths

The next algorithm, which we call `Paths` is similar to `Naive`, but has a key difference. Instead of having the parameter `max`, it can keep track of where it has been, and by doing so, avoid loops to speed up the process of the search. This works by creating an object of type `Paths`. This object holds an array of a set size and a pointer. Then upon calling its method called `shortset()` (same name as in the class `Naive`, it'll use these to prevent the loops. Have a look at the snippet below before you continue to read and notice how its main logic is similar to what was used earlier.

```
private Integer shortest(Map.City from, Map.City to) {
    if (from == to)
        return 0;

    path[sp++] = from;

    Integer shrt = null;
    outerloop:
    for (Map.City.Connection conn : from.connections) {

        for (Map.City c : path)
            if (c == conn.destination)
            continue outerloop;

        Integer dist = shortest(conn.destination, to);

        if (dist != null && (shrt == null || shrt > dist + conn.distance))
            shrt = dist + conn.distance;
    }
    path[sp--] = null;
    return shrt;
}
```

If we focus on what's different, we see that we keep track of the cities using the array called `path` at the start and end of the method. On top of that, we use an inner loop to prevent the loops by using the "continue" functionality on the outer loop (iterating over the connections).

By preventing loops, we've improved the overall performance and enabled the algorithm to find some paths using the `Naive` algorithm that won't be found in a very long time. Looking at Table 2 we can clearly see these results and draw the conclusion that this is a better option than `Naive`. But does it scale well? More on that later.

From	To	Distance(min)	Search(ms)
Malmö	Göteborg	153	209
Göteborg	Stockholm	211	82
Malmö	Stockholm	273	163
Stockholm	Sundsvall	327	119
Stockholm	Umeå	517	182
Göteborg	Sundsvall	515	149
Sundsvall	Umeå	190	426
Umeå	Göteborg	705	156
Göteborg	Umeå	705	220

Table 2: Benchmark for algorithm Paths.

PathsLimit

The algorithm `Paths` did work better than `Naive`, but we will improve it even more by adding a dynamic maximum available that stops it from exploring paths it can early tell take a longer time. This algorithm's snippet font be included since it is so similar to the one used for `Paths`. Instead, a short text-based description is used: Right outside the loop which iterates the connections we create an integer called `max` set to its highest possible value. Then for each connection, we compare this value to the connection's distance, and if it's larger, then we follow that connection. If not, we continue to the next loop iteration and check the next connection instead. Now it follows the connection and returns a value, it will update the `max` value to the new shortest path time.

This approach should in theory lead to better performance. And indeed it does! Looking at Table 3 we see the results of a benchmark comparing the time for the algorithm `Paths` and `PathsLimit` to find a path from Malmö to Kiruna. The result tells us that by adding this form of limit, we can improve the results by preventing checking paths we know early are slower than others we've already found.

Algorithm	Search(ms)
Paths	858
pathsLimit	579

Table 3: Benchmark comparing the time to find a path from Malmö to Kiruna.

Larger Data Sets

Now let's assume we were to work with a larger data set. A larger railway system, let's say for the whole of Europe. Then how would the `PathsLimit` algorithm perform? To explore this in a simple way, a benchmark run testing how the time to find a path increases more rapidly the larger the distance between the two cities.

In Table 4 we can see how the results of this benchmark display a rapid increase in searching time when the distance between the cities increases. Based on this we can conclude that the `pathsLimit` algorithm, which was the best algorithm we've tested in this report still isn't good enough to handle larger data sets, and this set used is both cut down (with fewer cities than in reality), and only covers one country. Using this for Europe and not only Sweden isn't realistic.

From	**To**	**Distance (min)**	**Search(ms)**
Malmö	Lund	13	0
Malmö	Varberg	114	114
Malmö	Katrineholm	218	112
Malmö	Västerås	297	144
Malmö	Storvik	405	224
Malmö	Sundsvall	600	270
Malmö	Umeå	790	460
Malmö	Kiruna		1162

Table 4: Benchmark exploring how the search time increases with distance.

Discussion

The benchmarks run for this report were somewhat simple. For clearer results, each benchmark should've been run at least 1000 times to give more reliable data. However, the results displayed in this report go well together with other similar reports results and the behavior of the algorithms is similar to what some peers have experienced.

Furthermore, graphs representing small portions of the railroad system would be very helpful for the decision of how each algorithm worked. Instead, a tremendous amount of text (for somewhat simple algorithms) and snippets were inserted that can easily become overwhelming.

Now if we focus on the algorithms, it's worth mentioning how each different algorithm the report explored, performed somewhat better than the previous one. But as mentioned in the instructions given by the teacher for this assignment, these algorithms are good. They are slow and "stupid" as they either lack limits (as in can't handle loops) or simply forget paths they have already checked. There are better well-suited algorithms out there for work like this, such as Dijkstra's algorithm.

Conclusion

When working with graphs where finding the shortest path is required one should use a proper algorithm since the simple algorithms quickly become slow as soon as the graphs grow too large. This report handled a small data set with only 54 nodes, now imagine what would happen if we have 1000 nodes instead. The takeaway is that it's important to write proper algorithms that can perform the task even when the data set grows and that small adjustments, such as adding the limit support to the `Paths` algorithm to create `PathsLimit` isn't enough.