Graphs - Swedish Train System

Daniel Sherif

Fall 2024

Introduction

In the following report, we will create and walk through the implementation of a Swedish train system, that bases itself of a file. The train system works by creating a graph with weighted connections, where the "weight" of the connections accounts for the time it takes to traverse those connections.

Inner Workings

In this section, we will just go through the most relevant parts of the code, more explanations will come further down the line and reflect on and compare different ways to implement methods.

In general, two objects are implemented, cities and connections. Cities include a ArrayList of connections, so connections get accessed by proxy of cities.

Hashing and Lookup

To find cities, lookup, we will use hashing. We will hash the city names into code, a key. To ensure the hash codes are different, we take each city, convert it to a code, then take modulus a certain number. We chose the number 70, as it is double the amount of cities, 35. After hashing the first city, for the next city, we multiply the old hash by a factor of 35, and add the "key" received from the next city string to it, then again take the modulus of the sum. See Figure 1.

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

```
        return hash;
    }
```

Figure 1: *Hashing function.*

For lookup, it relies on the hashing. Inside lookup, we create a new City of no city is found, additionally, we track collisions with hashing by printing out. This is more for debug and understanding of procedures while working with the project. Here we use linear probing to ensure collisions get handled properly. More on this later.

Mapping

As for the mapping, it works by reading every line of a file, assigning hash numbers through the lookup method, and then allocating the connections to them. Every line basically has a from, to, distance structure. More on lookup in next section.

Collisions with Hashing

The following are the results we get from the hash number 70. Inside the lookup method, if a city is not found on a certain hash code, it will add one to the hash code and take the modulus of that sum, giving it a new unique key. Every time a key needs to increase or change, we add to a variable, and every time that variable is more than 0, we print out the collision and the amount of time we probed. With the mod number 150, it has 5 collisions, see Figure 2, which all probe once other than one of the collision occurrences. This is sufficient.

```
Mod:  150 Probed:  1
Mod:  150 Probed:  1
Mod:  150 Probed:  1
Mod:  150 Probed:  1
Mod:  150 Probed:  2
```

Figure 2: *Collision results for Mapping.*

Searching for Paths

The Naive Solution

To find fastest path from and to one city to another, however far away, we traverse all cities and their connections and differentiate between the fastest paths. We do this using a recursive method, where we keep track of the

remaining time and the time taken, the total time, and return the shortest time of them all by having a comparing condition in the end of every loop iteration. In the end, we just return the shortest path. See Figure 3 for a visual representation.

```
            ...
            int curTime = max - dst;
            Integer path = shortest(conn.destination, to, curTime);
            // Go through all cities/conn, path
            if (path != null) {
                int totalDst = dst + path;
                if (shrt == null || totalDst < shrt) {
                    shrt = totalDst;
                } // null accounts for first ever assignment
            }
        }
    }
    return shrt; // return the shortest distance!
            ...
```

Figure 3: *Relevant code inside naive solution's loop.*

Testing some cities and combinations, we get following times, see Table 1.

Path	**Distance**	**TimeOfExec**
MLMÖ-GBG	153 min	0 ms
GBG-STKLM	211 min	2 ms
MLMÖ-STKLM	273 min	5 ms
STKLM-SNDVL	327 min	26 ms
STKLM-UMEÅ	null	651 ms
GBG-SNDVL	null	1056 ms
SNDVL-UMEÅ	190 ms	0 ms
UMEÅ-GBG	null	? ms
GBG-UMEÅ	null	1000 ms

Table 1:
Times received for different paths. Null signals no time found and ? means execution time is unknown or unrelaible.

Table 1 shows that some paths do not return any time. This is ofcourse heavily dependant on a set "max" value, that basically tells the method to stop looping after a certain travel time has been exceeded. Derived from this observation, we understand that the problem lies somewhere in the looping.

We see that some of the searches are instantaneous, or at least so fast that it is a decimal value when looking at milliseconds. We generalize these values to 0 as accuracy is not the greatest concern yet. Next section we implement a better solution and see how much more efficient it is time wise.

The Better Solution

This section is going to explore a better implementation that solves the looping issue in the "naive" solution. It is also going to provide some benchmarks.
The better solution works by tracking all traversed cities and adding them to an array. At the start of every recursive iteration of the method, we check all traversed cities and compare it to the current city we are traveling from, thus avoiding any loops from before. All else stays roughly the same as in Figure 3.
Running benchmarks on the explained function goes as following. See Table 2.

Path	**Distance**	**TimeOfExec**
MLMÖ-GBG	153 min	1 ms
GBG-STKLM	215 min	2 ms
MLMÖ-STKLM	273 min	1 ms
STKLM-SNDVL	327 min	0 ms
STKLM-UMEÅ	517 min	0 ms
GBG-SNDVL	643 min	0 ms
SNDVL-UMEÅ	190 ms	0 ms
UMEÅ-GBG	732 min	0 ms
GBG-UMEÅ	732 min	0 ms

Table 2:
Times received for different paths, using better solution.

As seen in Table 2, comparing to Table 1, the time of execution is substantially faster. Also we actually receive times for every path. Paths are also consistent if from and to are interchanged, see last two rows on the table.

This is faster because there is no overhead for the program to check, there is no guessed max value so the program does not need to do redundant searches, leading to much faster search times.