Djikstras Algorithm - European Train System

Daniel Sherif

Fall 2024

Introduction

In the following report, we will create and walk through the implementation of a European 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. Then, using Djikstras Algorithm, being able to traverse and evaluate this data efficiently, regardless of its size.

Inner Workings

By keeping account of paths taken when searching, we are able to track already traversed cities, to avoid loops. This is done with a array we call "done". Additionally, this array can be used to track and remember already traversed cities. So the done array, keeps track of different kinds of paths, and the cities traversed within those paths.

When starting in a city, specified as a parameter, we go through all the city's connections and add them to a priority queue, smallest distance first. This priority queue keeps the smallest distance connections always up top, allowing us to expand our searches based on the smallest times for paths. As can be seen in Figure 1, we use the queue to iterate through a possible path. We look through all connections, smallest distance always first. See Figure 1 for code, best explained through visualization.

```
public Path findPath(String fromName, String
...
while (!queue.isEmpty()) {
    Path cur = queue.poll();
    ...
    done[cur.city.id] = cur;
```

```
            if (cur.city == to) {
                return cur;// stop if done.
            }

            for (Connection conn : cur.city.connections) {
                int newDist = cur.dist + conn.dst;
                if (done[conn.destination.id] == null
                || newDist < done[conn.destination.id].dist) {
                    queue.add(new Path(conn.destination,
                    cur.city, cur.dist + conn.dst));
                }
            }
        }
        return null; // find nothing return nothing
    }
```

Figure 1: *Process of finding and saving paths. Returning smallest valid path after iteration.*

While traversing, the "done" array notes down different cities which have been traversed, allowing for a later print of all cities traversed between a connection. Notice how in Figure 1, there's a condition explicitly ruling out distances that are larger than the ones previously searched, then adding them to the priority queue in the end. So the smallest sub-path, connection, per city is chosen. Also, in the start of the while loop, the smallest distance "path" is also taken and set to cur, later processed.

To know when to exit, there's a condition outside the while loop which checks if we ever get to the destination city, if so we return whatever is on top of the queue currently and eventually connected to the destination city, which is the shortest possible path to destination, see Figure 1.

Explanation can be very confusing, that's why it is very important to understand and process the code, but even more so drawing and pondering about the issue is very important before starting to program a task like this. Attempting to explain it with words alone to someone new to this type of algorithm is difficult and requires some alone time with the problem and concept.

All Results

Next up, we print our results for 12 paths all over Europe. Table 1 displays the results, which includes amount of cities traversed, execution time for

that path and distance. See Table 1.

From	To	Distance	Exec time	Cities
Malmö	Stockholm	273 min	1 µs	10
Göteborg	Helsingborg	127 min	49 µs	5
Umeå	Luleå	212 min	17 µs	3
Köpenhamn	Hamburg	276 min	83 µs	2
Berlin	Paris	477 min	89 µs	7
Amsterdam	Bryssel	123 min	7 µs	2
Rom	Milano	207 min	12 µs	4
Wien	Prag	241 min	9 µs	2
Barcelona	Madrid	158 min	9 µs	3
London	Manchester	231 min	29 µs	3
Geneve	Zürich	214 min	32 µs	3
Budapest	Bukarest	900 min	109 µs	2

Table 1:
Table displaying 12 paths through Europe. Note that "Cities" signifies cities traversed for the path.

Next up we compare results in Table 1 to another Train System that only is able to search through Sweden. Table 2 represents the Swedish train system results. Notice how Table 2 has execution times over 0 ms.

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:
Swedish Train System. Received times.

In Table 1, we see that execution times are consistently far under the 1 millisecond mark, While Table 2 has some marks that are above 0 milliseconds. This makes the efficiency of using Dijkstra's algorithm apparent, considering the greater distance between cities and the much bigger map that Table 1 is accounting for. Note that Table 2 is measuring milliseconds, and Figure 1

measures microseconds. While not the most accurate comparison, considering the efficiency and scale that Dijkstra's algorithm reaches, its advantages and superiority is clear.

Complexity Analysis - Dijkstra Algorithm

This section is going to reflect and analyse why we get the different execution times, what higher execution time is caused by. We will see how and why the complexity depends on amount of "paths" that have to be considered. If we start with the path Malmö-Stockholm, top of the list, notice how it is has a decently long distance and traverses the most cities, but has the lowest execution time. This result alone tells us that amount of cities and distance itself do not have a direct correlation, is not a determining factor for higher execution times. This is completely logical too, as the finding of paths, like mentioned before, bases itself by iteratively going through every possible path and picking out the fastest one.

If we look at the path Köpenhamn-Hamburg, third highest execution time, it traverses only two cities and has very similar path length to previous path example. The reason it takes so much more time is because there are more paths to consider. In the previous example, the paths were "isolated" within Sweden, as all connections from Stockholm that can be taken by train are Swedish, so there are only a handful. If we go from Denmark though, Köpenhamn, there are connections, possible paths, to multiple cities of both Sweden and Germany, which causes some complexity. Although there are only two cities traversed in the results, the search may have 30 times the resulting cities actually considered and evaluated, imagine how many appends that is on the queue, and evaluations inside the priority queue sorting it.

Another great and similar example would be the longest path, Budapest to Bukarest. Although Budapest and Bukarest have a direct connection between each other, the program needs to consider many countries that border Budapest before ending up, finally evaluating Bukarest. This example further proves the Köpenhamn-Hamburg example, and concludes this analysis.

Summarized, the complexity of Dijkstras Algorithm depends on **amount of paths that need to be considered** rather than the amount of cities traversed or distance traveled.