Assignment 13 Report - Dijkstra's Algorithm in Java

Dean Tsankov

October 26, 2024

Introduction

In this final assignment I will present a much nicer solution to the proposed problem of the previous assignment on graphs. The conclusion we reached there was that in order to traverse a graph of nodes and find the shortest path between two specific ones, as efficiently as possible, is to keep track of all previous efforts we have done over the graph to reach a solution in optimal execution time. This will be done here using Dijkstra's algorithm. This algorithm has the benefit of not only solving the mentioned problem, but it also records additional information regarding the shortest path to each of the nodes along the path to the one we are searching for.

This is in essence how Dijkstra's works: When we keep track of the shortest path to the nodes we have visited when we have to make a choice along which connection to go next, we will chose the one which gives the shortest distance. When this choice leads us to the destination node we will be certain that that was the shortest way to reach it. Thinking further about it this is almost the same as a breadth-first search algorithm we have implemented before, but this time in order to be able to quickly select which of the traversed nodes had the shortest path we will execute the BFS using a priority queue instead of the simple FIFO queue.

Here again in order to test the functionality we will be using a CSV file with the train lines connecting different cities. Now we are given not only the connections in Sweden, but also a larger set of cities throughout Europe so that we can gauge how much better Dijkstra's algorithm performs compared to the previous semi naive way.

Auxilary data structures

The structures we will need here are partially similar to the ones in the previous task. The `City` and the `Connection` classes are almost entirely identical with the simple exception that we will now keep track of an `id` value for each city, more on that later.

Cities and Connections

Generally a city now holds a name, an id and list of connections to other cities. A connection itself is the structure which holds a destination city and the value of the time to reach it.

Paths

Next there is another class similar to the previous implementation - a path. The path will hold information for the distance between a current city and a destination city. One such object represents a city that is part of a path to our destination city. It therefore points to the city that is the previous step of the complete path so there is no representation of the whole path to the destination. but instead it can be described by a sequence of path entries.

Since I decide to make the `Path` class a seperate file I had to also implement getter and setter functions for its properties in order to not expose them explicitly.

```
(...)
public class Path implements Comparable<Path>{
    private Map.City city;
    private Map.City prev;
    private Integer dist;
    private Integer index;
    public Path(Map.City currCity, Map.City prevCity, Integer dist) {
        this.index = null;
        this.city = currCity;
        this.prev = prevCity;
        this.dist = dist;
    }
    public Map.City getCity() {
        return this.city;
    }
    public Integer getIndex() {
        return this.index;
    }
    public void setIndex(Integer index) {
        this.index = index;
    }
    public void updatePath(Map.City prevCity, Integer dist) {
        this.prev = prevCity;
```

```
            this.dist = dist;
        }
        public Integer getDistance() {
            return this.dist;
        }
        @Override
        public int compareTo(Path s) {
            return Integer.compare(this.dist, s.dist);
        }
    }
    (...)
```

A done array

In order for us to be able to keep track of the progress of the algorithm there is a need to include a simple path array where we will record every node which has already been examined. This is where we will need to use the integer identifier for each city to allow us to quickly check if we had the corresponding index in the done array. If it includes a path to city, that would mean that we had already examined that one and there would be no need to duplicate our work. Not only this, but this array will allow us to follow the shortest sequence of cities we took in order to reach our destination as each path entry has a reference to the previous city. So if we traverse them we get the complete shortest path.

A priority queue

As mentioned previously for the optimal execution of this algorithm we would like to, on each iteration, select the city node which yields the shortest distance. This can ultimately be done best by utilizing a priority queue, which by its structure always keeps elements in a way where it is convenient to dequeue or poll the smallest one by some criteria, in this case - distance. We have discussed the implementation of a priority queue in class and as such I will not go into details about it here, but there was minimal modification required in order to allow it to work for our purposes in this algorithm.

The Dijkstra algorithm

The procedure goes as follows - we want to expand our search slowly so we choose to proceed by either going to the closest immediate neighbor of our current node or to one of the neighbors of the node we came from. In the first iteration we have a single option, but on every next one this choice is left up to the priority queue which makes sure the choice is the smallest.

On selecting from the queue we also need to cross reference the received path with the done array since if we had already examined the case we can ignore it. Otherwise if it has not been we add it to the done array after enqueue-ing the appropriate new path to the priority queue. In a general sense the algorithm goes through the steps:

- Poll the queue and receive a path to a city
- If this city is the one we are searching for, we can finish execution
- Otherwise if the city is not in the done array, for each of its direct connections enqueue a new path and record it as done.

In code the implementation is quite simple and should approach something like the following:

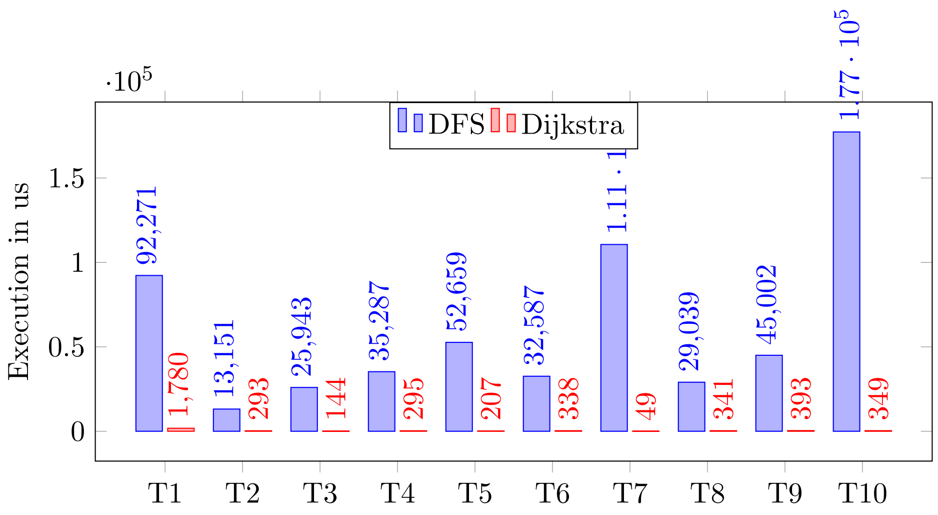
```
(...)
public static Result shortest(Map.City from, Map.City to) {
    PriorityQueue queue = new PriorityQueue();

    Path[] done = new Path[Map.mod];

    queue.enqueue(new Path(from, null, 0));

    while (!queue.isEmpty()) {
        Path fromCurrCity = queue.dequeue();
        Map.City toCity = fromCurrCity.getCity();
        Integer totalDist = fromCurrCity.getDistance();
        if (toCity.equals(to)) {
            int doneEntries = 0;
            for (Path p : done) {
                if (p != null) {
                    doneEntries++;
                }
            }
            return new Result(totalDist, doneEntries);
        } else {
            for (Map.Connection c : toCity.conections) {
                if (done[c.destination.id]==null) {
                    done[c.destination.id] = new Path(c.destination,
                            toCity, c.timeToReach + totalDist);
                    queue.enqueue(done[c.destination.id]);
                }
            }
        }
    }
```

```
        return null;
    }
    (...)
```

The execution begins from a path to the origin city which has no previous city and a distance of zero. Next we enter a loop and perform the above described steps. Now here I am also returning the amount of entries in the done array along with the distance in a separate `Result` object. This will help with benchmarking in the next section.

Gauging performance

Let us first compare how this new algorithm fares compared to the DFS one, over the same ten city-to-city benchmarks in Sweden:



There really appears to be quite the difference. But this should not surprise us when we consider how much additional unnecessary work we are cutting out of the execution by simply remembering the work we did over previous nodes in the traversal.

Next we are also asked to do a benchmark where we go to measure how Dijkstra's performs on the shortest path from one specific city to a couple other random ones from the whole European data set. We do this in order to hopefully be able to somehow approximate a time complexity for the algorithm.

To be able to do this I retrieve the size of the done array for each `shortest()` execution and then order the results according to it. Then that would yield the following graph showing the relation between execution time and nodes traversed.

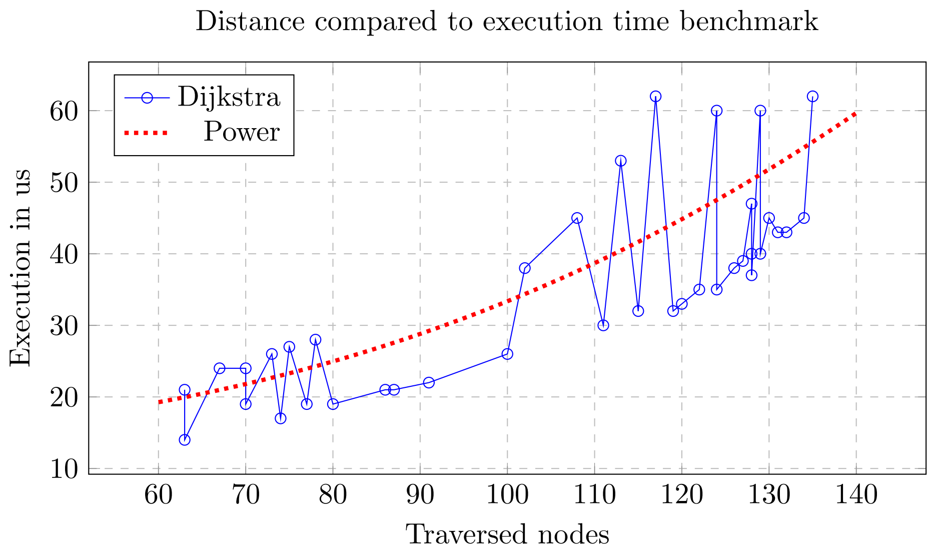


Figure 1: Distance compared to execution time benchmark

From here we could assume a general power function appearance with relation to the number of nodes. There is definitely also some influence from the amount of edges connecting said nodes, but here since we are dealing with train lines there is a mean value of edges emerging from each node (let us say about 4) and so its influence would probably average out and not be as apparent in the performance as the number or vertices themselves.

Closing

In conclusion, it is definitely evident that this approach to finding the shortest path over a graph, between two nodes is superior. I does not require too much additional effort to implement and the only consideration we had to make is to find out why the previous DFS algorithm performed as it did in order to negate its downfalls and improve it. Of course even in Dijkstra's there is some room for further improvement, for example if instead of using a priority queue we could use what is known as a Fibonacci heap and lower the time for operations in the queue. All in all with this we show the importance of careful consideration and methodical thinking which allows us to reach better solutions.

The code for the implementation and ran benchmarks is uploaded in the respective Github repository