Dijkstra to our rescue

in Java

Algorithms and data structures ID1021

Johan Montelius

Spring 2025

Introduction

This assignment is a continuation on the graph assignment where you searched for the quickest train ride in Sweden. In this assignment you are going to improve the implementation by using Dijkstra's algorithm. The problem with the solution you had was that it did not remember where it had been and had to do the same thing over and over. Dijkstra's algorithm not only tackle this problem but will easily compute not only the quickest path from A to B but from A to all other stations in the network.

The idea behind Dijkstra's algorithm is to keep track of the shortest path to nodes that we have visited and then expand the search given the shortest path that we have so far. If we select the shortest path so far explored, and notice that it ends in our destination node then we are done; all other paths are longer and it does not matter if they will eventually lead to the destination node. Before we implement the algorithm we need some supporting data structures.

Some data structures

The data structures that you will need are partly the same as you have used in the graph assignment and partly data structures that you have implemented in previous assignments.

a city

A city will as before hold the name of the city and a structure that keeps its immediate neighbours. Go back a few weeks and use the dynamic array that you implemented or implement it as a linked list.

```
public class City {

    public String name;
    public Integer id;
    public Connections neighbours;

    :
```

When building the map we might as well number the cities (we still need the hash table). If we number the cities (0..n) we can use this identifier to index an array, more on this later.

a path

A path is a: `city`, the `previous` stop, and the total `distance` in minutes from our origin city.

```
    private class Path {

        private City city;
        private City prev;
        private Integer dist;

       :
```

A path entry simply state that a city is part of a path that might lead to our destination city. It only holds the city that is the previous step in this path so we don't actually have the path in our hands i.e. a complete path would be a sequence of path entries.

done

As the algorithm proceed we will find the shortest paths to cities in the network. We keep track of these paths in an array called `done`. We will have one entry per city and this is where we will use the integer identifier of the city as index.

When we are done, having a found the shortest path from Malmö to Stockholm, we will have a path entry in the array for Stockholm. The entry will tell us the time it takes to reach Stockholm and also that the previous city was Södertälje. By recursively examining the entries in the array we will be able to construct the whole path from Malmö to Stockholm.

When we perform our search we should start in our source city and expand the found paths as *slowly as possible*. To do this we need a priority queue of potential paths to expand.

a priority queue

Since we are always going to expand our search from the shortest path that we have so far we will use a priority queue to order the paths. You have already the code for a priority queue so if you only adapt it to hold paths - and make sure it can order paths - it should not require much coding.

You now have all the pieces to the puzzle so let's implement the algorithm.

Dijkstra's algorithm

The idea of the algorithm is to explore the graph slowly and only expand the search from the city that is closest to the origin city. If we start in a city A we will choose its closest neighbour to expand the search. In the next step we should choose the closest city that is either the immediate neighbour of A or immediate neighbour of the city that we just entered.

The cities that we should consider for the next expansions for a ring around the origin city. As we slowly progress we will eventually reach the destination city and will then have found the shortest path.

The cities, or rather paths, that we should consider in each round are of course kept in the priority queue. When we select the shortest path from this queue it might be that we have already found the shortest path to this city in which case we can simply ignore the path. If we have not yet a recorded shortest path to the city we take the path and add it to the `done` array. The algorithm thus proceed as follows.

- Remove the first entry from the queue, it is a path to a city:
- if the city is is the destination, we are done,
- otherwise if this is the first time we remove the city, update the `done` array and for each of the direct connections from the city add a new path to the queue.

The search is initialized by adding a first entry to the queue that describes the shortest path to our source city. If we search for the shortest path from Malmö to Stockholm, the first entry will simply be for example: city of Malmö, 0 and null (since we don't have a previous city). After the first round we will add the path to the `done` array and add the immediate neighbours to the priority queue.

white paper

Before implementing this algorithm take a white paper and draw a picture of what is in the `done` array, in the priority queue and what will happen in

each iteration. If you can not draw the picture it is unlikely that you will manage to implement the algorithm.

Benchmarks

When you have all the pieces of the puzzle you should be able to find the path from Malmö to Kiruna in much less time compared to your previous solution. Do some benchmarks and show how much you managed to improve the performance.

Now find the shortest paths to twelve cities in Europe starting from some specific city. List the time it takes to find the shortest path and how many entries you have in the `done` array when the path is found.

The number of elements in the `done` array is a measurement of how many cites that have been involved in the search. If you list this number and the time it took to find the shortest path you could estimate the run-time complexity of the implementation you have.

You can also reason about the complexity and see if your understanding of the complexity match the execution time that your benchmarks show. Assume we have a map of n cities and we are looking for the shortest distance from a given city to all other cities. In each iteration you will select a path from the queue with a city. If this is the first time we select it from the queue we place it in the done set and add its immediate neighbors to the queue, but this is only done once per city in the map. This will definitely give us a factor n in complexity but there is of course more work. How many immediate neighbors do we have and what do we have to do for each of the neighbors? How many paths will we have in the queue? If we can estimate this and know the complexity of the queue operations we might find a reasonable estimation - do your own calculation.