Dijkstra to our rescue

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

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.

Shortest path first

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. 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, the resulting path will be even longer.

There is some bookkeeping that we need to do but it turns out that we only need to keep track of the shortest path so far found to each node. If we also want to know what the path looks like we also make a note of this but all in all we only need one array with one entry per city.

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. The first entry that we add to the queue is an entry that describes the source city. An entry will hold: the city, the distance to the city and from where the path entered the city. So the first entry will simply be for example: city of Malmö, 0 and null.

In addition to the priority queue we will keep a set of processed paths that we have explored. This is a simple set that we want to index by the name of a city so we might as well use a similar hash table as we used when constructing the map.

The algorithm now proceed as follows.

- If first entry in the queue terminates in the destination, we are done.
- If not, remove the shortest path from the priority queue and place it in the processed set. For each of the direct connections from that city, do the following:
  - If the city connects to a city that is found in the queue update if possible the shortest path to that city and update the queue,
  - If the city is not found in the queue add it to the queue.

Note that we should update the entries in the queue given a city. If we do not want to search through the whole queue we want to be able to find the relevant path entry given the city. Also note that when we do the update we will possibly move an entry towards the root i.e. we have found a shorter path. So the queue should be organized in a way that allows us to do this as efficiently as possible.

an identity sequence

So far we have only used then name of a city to do the lookup procedure. We of course need to have this when we do the initial construction of the map since the only thing that identifies a city is its name.

Since we in this assignment need a way of quickly finding a queue entry given a city, it would be good to instead number the cities: 1,2,3... and use the number as identifier.

When building the map we might as well number the cities (we still need the hash table). We can then easily construct a table of path entries that can be accesses given the index of the city.

the path entry

When we extend a path we take the neighbors of the city and see if we have a shorter paths to any of the neighbors. This using the identifier of a city we can find the path entry of the city, so far so good.

Now if we find that we have a shorter path to the city we should update the priority queue. Since we do not want to search through the whole queue to locate the entry we want to have enough information in the entry itself to position it in the priority queue. When we have located the entry we should let it *bubble* towards the root.

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.

You might also try one more thing; Dijkstra's algorithm is normally used to find the shortest paths to all other nodes in the network not just a single destination node. It turns out that this is what it does and the only change in your implementation is to not stop when you find the destination node. If you continue until the priority queue is empty you have the shortest paths to all nodes in the network. Give it a try and present the execution time for finding the shortest paths to all cities from Malmö.