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.

So let's first define what a path is and then see how we keep track of the shortest and expand the search. A path is a: `city` the `previous` stop, the total `distance` in minutes from our destination, and a very handy queue `index`.

```
private class Path {

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

   :
```

A path entry simply state that a city is part of a shortest path (so far found) 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.

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.

Since all cities have been given unique identifiers, `0..n`, we start our search with a array called `done` that will hold all the shortest paths found so far i.e. a path is found and recorded in this array if we have reached the city. We record the path for a city a soon as we reach it and then also recorded from which city we came.

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. 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. 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).

The algorithm now proceed as follows.

- Remove the first entry from the queue:
- if is is the destination, we are done,
- otherwise place the city in the processed set (this is the shortest path to that city found so far). For each of the direct connections from that city, do the following:
  - If a path city is found in the queue (if its index is not null), update if possible the shortest path to that city and update the queue.
  - If the city is not found in the queue, add a new path 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 array implementation).

Also note that when we examine a neighbour city we can look in the array of existing paths (`done`) and - if there is a path entry - we will get the index that we can use to find the path in the priority queue. How these identifiers and indices are used might require some explanation.

a unique 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. But, 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: 0,1,2... and use the number as identifier.

When building the map we might as well number the cities (we still need the hash table). If the number the cities (0..n) we can use this identifier to index the array `done` that holds the shortest paths found so far.

A city will as before hold the name of the city and a structure that keeps its immediate neighbours. You could of course implement this as a array directly and keep track of how many neigbours there are, but you could also implement is as dynamic array that is `Iterable` (maybe something to try when everything is working).

```
public class City {

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

    :
```

A `Connection` is as in the previous assignment a structure is a small structure holding the neighbour `city` and the `distance` to this city.

update the queue

When we explore the neighbours of a city in a path that we are extending, we might find a neighbouring city that can be reached in a shorter distance than from what is recorded in the previously shortest paths found (`done`). How is this you might wonder but take for example that we have explored the neighbours of A and added paths leading to B, C and D to both the shortest paths found so far and to the queue. Now is the distance to B was 10, the distance to C was 4 and the distance to D was 9, we will dequeue the path of C in our next pass. If we now have a connection from C to B with a distance to B of 4 then the shortest path to B is of course 8. When we

update the path found in the `done` array, we are informed about the index of the path to B in the priority queue. This will help us move this entry towards the front of the queue (bubble).

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 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.