Dijkstra

Fatima Mohammad Ali

Fall 28/10/2023

Introduction

Dijkstra's algorithm is an algorithm that is used to efficiently find the shortest path in a graph by *remembering* tentative paths to all nodes. The purpose of this assignment was to improve the previous solutions by using Dijkstra's algorithm to find the shortest path between European cities.

The algorithm

Generally, the main approach of Dijkstra's algorithm is to keep track of two sets; the *processed set* that includes the *shortest path so far* and a set that holds adjacent nodes yet to be added to the shortest path. As a start, all nodes are set to null/positive infinity and the source node is set to zero. From the source, the shortest distance to its *adjacent* neighbours has to be found. When the shortest edge (distance) is found, its vertex (neighbour) is added to the processed set (shortest path so far). At this neighbour now, all its adjacent neighbours are updated to being the value of the neighbour plus the distance between them. For instance, if there is a node with summed value 13 so far, and the distances to its adjacent nodes are 30, 35, and 36, then those nodes would now respectively hold values 43, 48, and 49. The vertex with the lowest value is chosen next (in this example, 43 is chosen). This continues until the destination is reached or all reachable vertices are explored, which would result in the shortest path being in the processed array.

For this assignment, the processed set was an *array* while the unprocessed set was a *priority queue*. Since the priority queue was a *min-heap* (the root held the smallest value in the heap), a remove operation would extract the smallest value in the queue (since it dequeues the root). The nodes would also bubble up to the root after removals or when adding/updating nodes to maintain the min-heap property.

When implementing Dijkstra's algorithm, the removing and adding nodes procedure would continue as long as the queue was not empty or the destination node was reached. That is how all tentative nodes got visited and the shortest path was saved in the processed array.

Implementation

To start constructing Dijkstra's algorithm, several relevant objects had to be defined. As implemented in the previous assignment, the objects `City` and `Connection` were initialized. A `Connection` held the *neighbouring city* and the *distance* to it, and a `City` held a *name*, *integer id*, and *neighbours* (`Connection`).

A `Path` object that defined a *path* was initialized inside a `Dijkstra` class. A `Path` included a *city* (`City city`), its *previous stop* (`City prev`), and the total distance (`Integer dist` (in time)) to the destination. A path entry basically indicated that the given city was part of the shortest path. The class had two constructors, one that was given a single city as parameter (to add the source city), and one that was given a city, its previous stop, and the distance between as parameters. This was to record a path from a city and where it came from as a path.

The `Dijkstra` class also kept track of the processed and unprocessed sets. The processed set was an array of type `Path` (`Path[] done`) and the unprocessed set was the built-in priority queue `PriorityQueue<Path> queue`. The priority queue could have been implemented from scratch (since it was done in a previous assignment), but since it was permitted and suggested to use the built-in queue for this assignment, the imported queue was preferred.

The purpose of this task was to determine paths between given European cities. All cities and the distances between them were given in a csv file. When reading the file and constructing the map, each country was given an integer identifier 0,..,n. The identifier was later used to index the `done` array and also to facilitate navigating the *queue*.

Dijkstra using a priority queue

When starting the search from a *source city* to the *destination city*, all potential paths had to be saved/remembered somewhere. This is where the priority queue came in handy. The queue was initialized with the size of the map and each entry (as a path) held a city, where the path entered the city, and the distance to the city.

The first entry to the queue described the source city, so its previous stop was null. To start the enqueuing process, a two lined method was created. The method simply added the source city as a path to the queue (`queue.add(new Path(from))`, and then called a method `shortest` to enqueue the rest of the paths given the destination city.

The `shortest` method that added paths to the queue was implemented as follows:

```
//saves all paths to the queue
public void shortest(City dest){
    while(!queue.isEmpty()){ //while the queue is not empty
        Path entry = queue.remove(); //dequeue the first entry (head)
        City city = entry.city; //check the city of the entry
```

```
        //place the city in the processed set if it's not there already
        if(done[city.id] == null)
            done[city.id] = entry;

        //if the entry is the destination city, the search is done
        if(city == dest)
            break;

        //save the summed distance so far
        Integer sofar = entry.dist;
        //for each of the neighbouring cities of the entry
        for(Connection con : city.neighbours){
            City to = con.city;
            //if the neighbouring city is not in the processed set,
            //add it as a new path to the queue
            if(done[to.id] == null){
            //add the distance to the sum of the already saved path
                Path newpath = new Path(to, city, sofar + con.distance);
                queue.add(newpath);
            }
        }
    }
}
```

The method above enqueued paths to the queue and added "shortest paths so far found" in the `done` array.

Now, the `done` array had to return the shortest path (in time) given a city. This was done in an `Integer shortestDistance(City city)` method. The method checked two conditions. If the city was valid (not null) and if it was present in the `done` array, then its distance saved in the `done` array was returned (`return done[city.id].dist`). Otherwise, null was returned. This returned the summed up distance value along the cities in the shortest path to the destination city in question. Therefore, the shortest path was found using Dijkstra's algortihm.

Results

The benchmark results of finding: the shortest path (in minutes), the time to find the path, and number of entires to the `done` array for each destination city is given in table 1. All destination cities had *Stockholm* as a source city.

To compare with the previous assignment, shortest path from Malmö to Kiruna was benched and resulted in: 1162 min, given in 56 µs and with 115 entries to the done array.

Destination	**shortest path [min]**	**complexity [µs]**	**entries in `done`**
Köpenhamn	316	10	47
Oslo	374	10	53
Berlin	697	14	64
Helsingfors	730	16	68
Köln	826	20	75
Kiruna	889	24	79
Amsterdam	896	17	80
Warsawa	1004	24	88
Paris	1048	20	92
Manchester	1299	40	107
Milano	1403	56	113
Madrid	1620	86	125

Table 1: *E*ach timestamp was the minimum value of 1000 tries for each search to a destiantion city. The entries to the `done` array was calculated by incrementing a global counter for each addition to the array (excluding the addition of the source city).

Discussion

Adding and removing an entry from the priority queue is done in $O(log(n))$ time where n is the number of cities in the map. As the search expands, the path extends which makes the search time complexity also grow proportionally to the number cities (neighbours and the distances). This makes Dijkstra's algotihm having the big-O notation $O(n * log(n))$ where n is the number of cities in the graph (could generally be $O((V + E) * log(V))$, where V is the number of nodes in the graph and E is edges). Hence, the time complexity is proportional to the number of entries in the done array, and a *little more*.

In theory, this should be seen in the values of table 1. For instance, the search to Oslo (with 53 entries) gave 10µs, while the search to Manchester (double the entries, 107) gave 40 µs. Another example is that the search to Berlin (64 entries) gave 14 µs, while the one to Madrid (125 entries) gave 86µs. The increase in time complexity when doubling size n *could* be argued being proportional to $O(log(n))$, but the increase is perhaps too large. In fact, the doubling of size n resulted in almost *four* times bigger time complexity. This in itself could indicate a quadratic behaviour somewhere, which makes the algorithm move towards its worst case behaviour. Alternatively, this could raise the question whether outside factors, such as lack of warmup or cache behaviour, affected the results.

Additionally, since Dijkstra's algorithm remembers shortest paths through cities in the done array, it is without doubt superior compared to the previous solutions of the previous assignment. The path from Malmö to Kurna was found in 56 µs, while it was found in 1300 ms in the previous `Path` search.