Dijkstra

Ayda Ghalkhanbaz

Fall 2023

Introduction

This assignment is a continuation of the previous assignment. The aim of this assignment is also finding the shortest path between two cities but now using another approach, namely the Dijkstra algorithm. In later sections, we will discuss the algorithm's benchmark results and compare it with previous approaches.

Dijkstra Algorithm

The Dijkstra algorithm is a well-known method in computer science for finding the shortest path between two nodes in a graph. The general idea of this algorithm is to start from a source node and examine all its neighbours, while gradually expanding outwards. It also keeps track of the shortest path. In other words, the Dijkstra algorithm selects the nodes with smaller weights at each step, ultimately reaching the shortest path at the end.

For this assignment, the same graph structure as before with a slight modification was used. This modification involved the addition of a new attribute to the `City` class, named `id`, that indicates a unique number corresponding to each city.

Implementing the Dijkstra algorithm included creating a class named `Path` which represents a path with the city, its previous city and the distance between them. The Dijkstra class itself has two attributes; an array of `Path` and a priority queue of `Path`. The usage of these attributes will be explained along the implementation details.

Shortest Path

The logic behind the algorithm of the `shortest()` method is as follows:

1. Start from the source city and add it to the `queue` (this step is done in another method named `search()`, that calls `shortest()` after adding the source city to the `queue`)

2. Dequeue the city with highest priority (in this case, the shortest distance) from the `queue`
3. Add the city to the `done` array (indicating that we have visited a city with shortest distance)
4. Explore all its neighbours and for each neighbour not in the `done` array, create a new path and add it to the `queue`
5. Repeat steps 2 through 4 until there are no other candidates left and the `queue` becomes empty.

The implementation detail of the `shortest()` method is shown in the code snippet below.

```
public void shortest(City destination){
    while(!(queue.isEmpty())){
        // take the city with high priority
        Path p = queue.remove();
        City start = p.city;
        // if the city is not in done array, add it to done
        if(done[start.id] == null){
            done[start.id] = p;

            if(destination == start)
                break;
            // the current distance
            Integer soFar = p.distance;
            // exploring all neighbours
            for(Connection conn : start.neighbours){
                City to = conn.city;
            // keep searching only if city is not in done
                if(done[to.id] == null){
                // create a new path. start is the previous city
                    Path newPath =
                    new Path(to, start, conn.distance + soFar);
                    queue.add(newPath);
                }
            }
        }
    }
}
```

Result

This assignment involved two benchmarking processes. The first benchmark aimed to compare the Dijkstra algorithm with the one in the previous assignment (the results from the improved version), highlighting its improvements (see table 1). The second benchmark was done to determine the time complexity of the Dijkstra algorithm (see table 2).

Route	**Travel Time**	**Graph's result**	**Dijkstra**
Malmö to Göteborg	153	500	122
Göteborg to Stockholm	211	421	185
Malmö to Stockholm	273	199	231
Stockholm to Sundsvall	327	3200	260
Stockholm to Umeå	517	9500	288
Göteborg to Sundsvall	515	4900	300
Sundsvall to Umeå	190	267000	72
Göteborg to Umeå	705	36300	321
Malmö to Kiruna	1162	90000	483

Table 1: Execution time of finding the shortest path between two cities in μs [travel time is in minutes]

Destination	**Travel Time[min]**	**Run time[μs]**	**#n of stations**
Linköping	104	2.3	12
Malmö	273	5.6	40
Amsterdam	896	5.0	80
Bryssel	935	5.1	82
Paris	1048	5.7	92
Wien	1197	6.2	101
Birmingham	1249	6.3	104
Budapest	1354	6.7	110
Milano	1403	6.8	113
Glasgow	1483	7.1	120
Madrid	1620	7.5	125
Brindisi	1870	7.7	131

Table 2: Execution time of finding the shortest path between two cities in Europe. The source city is Stockholm.

Discussion

By looking at the results in table 1, it is evident that the Dijkstra algorithm outperforms the solutions from the previous assignment in terms of speed and efficiency. Although we improved our solution in the Graph assignment and got better results, the Dijkstra is designed explicitly to find the shortest path in a graph and there is no doubt that the Dijkstra outperforms naive solutions from the previous assignment.

The Dijkstra algorithm keeps track of the shortest path by saving it in an array and utilising a priority queue to always pick the city with the shortest distance. This method ensures that we won't waste the time by examining longer paths that might eventually lead to the same destination.

As demonstrated in table 2, we can see that the Dijkstra algorithm has a logarithmic behaviour. The runtime grows as the number of the cities between the source city and destination increases, but this growth occurs slowly, so that it resembles a logarithmic pattern.

The reason for Dijkstra's logarithmic time complexity lies behind its usage of priority queues. In each step, the algorithm extracts the minimum distance from the queue and updates the distance of its neighbours. In the worst case scenario, the algorithm has a time complexity of $\mathcal{O}((V + E) \cdot \log(V))$ where 'V' is the number of vertices (nodes) and 'E' is the number of edges in the graph.