Dijkstra's Algorithm

Emanuel Paraschiv

Fall 2024

Introduction

In this experiment we are going to talk about a more performant algorithm for searching in a graph. Dijkstra's algorithm is a great tool that returns the shortest path in a graph between 2 vertices in a very comprehensive manner, which clearly overtakes the 'Depth First Search' algorithm implemented earlier.

Starting point

In order to develop this algorithm we will start building on what we have already from one of our previous experiments. Then, to test our implementation we will use a document called 'europe.csv' which contains a list of some important cities in Europe along with the distance between them.

As before, we will need 2 structures: a City and a Connection. Each city will have 3 proprieties: a name, an index and a collection of Connections (or neighboring cities). We can choose to choose any structure for holding this collection, but a good idea is to make use of a Dynamic Array, that we implemented earlier:

```
public class City
    public String name;
    public Integer id;
    private DynamicArray<Connection> connections;
```

As before, a Connection will hold the neighbor of a City and the time it takes to reach it. However, we need a new structure, which we will call `Path`. This will hold 2 cities and the distance between them.

Of course, we will need `getter()` methods for all fields in this structures, as our algorithm will make use of them.

Now, our shortest-path algorithm will work by finding the smallest distance between 2 cities, which is achieved by exploring different paths.

It is therefore crucial to know at each moment which is the shortest path that we can take. We can store the paths in a collection and choose from there, what should this structure be? One idea is to use a queue.

However normal queue won't work, because it follows the FIFO principle which isn't helpful in our scenario. We want a queue which is able to dequeue only the shortest path from all present ones, and in order to do that we will use a `Priority Queue`. The mechanism of it is simple: it will make sure that the first item in it (path) is the shortest one available, so therefore every time we `dequeue` we will get the shortest path that is present.

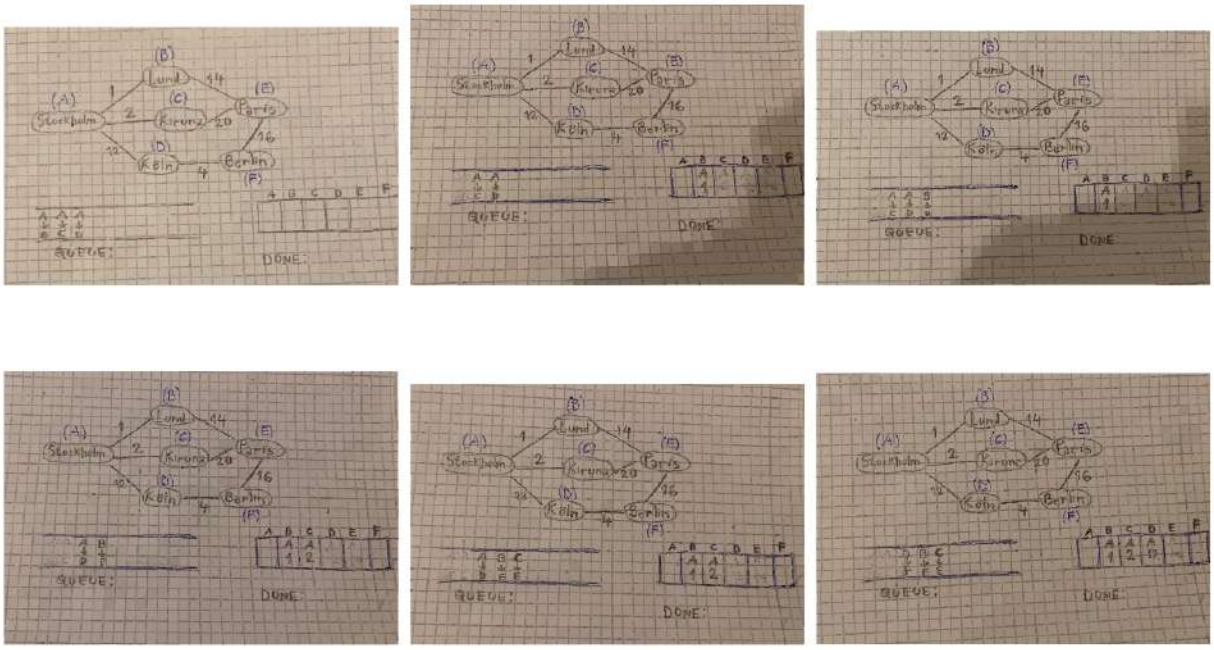
We will also like a way to know which paths did we already visit. We can create an array for this matter which we will call `'done'`, which will also act like a hash table: to each index we assign a path. We add paths to it only if that position is empty or if the path we want to add is shorter than the one present already.

Logic

The next example is meant to create intuition:

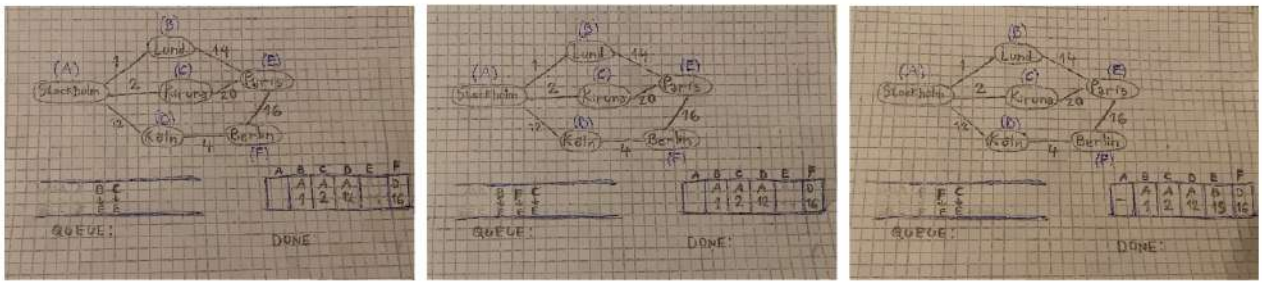
Now that we know what we need, let's take a practical example to understand the algorithm better. Let's assume that our Map has 6 cities (Stockholm, Lund, Kiruna, Köln, Paris and Berlin) and we want to find the shortest path from Stockholm to Paris.

To make things simpler, we assign a letter for each city, from A to F.



We start from vertex A and enqueue the paths to its Connections, B, C and D(from the shortest to longest). Then we dequeue the first path (shortest) and add it to our 'done' array (A to B having length 1), (*2nd image*).

Immediately after that we are located at vertex B, so we enqueue its connections (if they are not already). After we are done, we dequeue the



shortest path and add it to the 'done' array (A to C). This process continues with each vertex: enqueue its connections using priority principle, dequeue the shortest path and if it is the case, add it to the 'done' array. When we find the destination vertex, our execution stops (*Image 9*).

Implementation

The whole implementation can seem demanding, but once one understands the process described earlier, it becomes smoother. In order to keep it accessible, we will go through our algorithm step-by-step with explanations.

Firstly, we will have to create the 2 helper structures described above: the Priority Queue and the array of visited paths('done'). Then we will start from the first path, which will consists of the way from the first city to itself, and we will enqueue it to our queue. After that, we will make an instance of Path that we will use to iterate until we find the complete way to the destination city:

```
PQueue queue=new PQueue();// create queue
Path[] done= new Path[mod]; //done array
//Add initial path (src->src)
Path first= new Path(src,src,0);
queue.enqueue(first);
Path currentP;
```

The following procedure will continue until our queue is empty, and thus every path in the graph was visited.To begin the iteration, we will dequeue the first path (which is the shortest). We will check if this path contains the destination, case in which we stop the execution and return it directly:

```
while(!queue.isEmpty()) {
        currentP=queue.dequeue(); //shortest path
        City currentCity=currentP.getCity();
        if(currentCity==dst) {
            return currentP;
        }else{   //*If we still have to search for it*
```

If however the destination city is not found in the beginning of the loop, we will have to explore other Paths and Cities. In order to make the access

easier, we will use a hash function to give us the index in the 'done' array for that City. It is the same hash function used in the previous assignment. Once we get it, we check its presence in the array: if not present or shorter than the present one we add it, otherwise we just ignore it:

```
int cityInd = hash(currentCity.getN(), mod);
if (done[cityInd]==null || done[cityInd].getD() > currentP.getD()) {
    done[cityInd] = currentP;
}else{
    continue; //skip
}
```

Once we have this city in our hands, it is time to check its connections. We will make use of the Dynamic Array to store them, and will check each and every of them in the following manner. Notice how we can't directly access this array's elements using basic indexing, but need a getter method apart from an original array.

We get the first connection and compute the total distance to it (up to the previous city and from there to the neighbor).

```
DynamicArray<Connection> connections = currentCity.getConnections();
for (int i= 0; i< connections.getSize();i++) {
    City ngbh= connections.get(i).getN(); // The actual neighbour
    int newDist= currentP.getD() + connections.get(i).getT(); // Get the distan
    // ... next conetents below
}
```

Then we check if there is any shorter path in the 'done' array:

```
boolean shorter=false; // check if there is a shorter path already
for (int j= 0;j< done.length;j++) {
    if (done[j]!= null && done[j].getCity().getId() == ngbh.getId()) {
        if (done[j].getD()< newDist) {
            shorter= true;
            break;
        }
    }
}
```

If there are shorter paths than we simply start once again from the beginning and test the rest of the neighbors. If however this path is shorter, then we enqueue it.

```
if (!shorter) {
    Path newPath=new Path(ngbh,currentCity,newDist);
```

```
            queue.enqueue(newPath);
        }
}
```

If we finish the execution of the while loop (the queue gets empty) before we find the path to the destination City, than our Dijkstra method will simply return `null`. One interesting addition is a new field in the Path class. If we add a field 'prevPath' to refer to the previous path executed then each time we find a shorter path in the dijkstra's algorithm we can perform 'newPath.prevPath=currentP;', which will chain all paths. With one simple helper method in the Main class, we can therefore reconstruct the whole path followed.

If we try to run our algorithm for the path `Malmö` → `Kiruna`, then we get the following chaining:

Malmö → Lund → Hässleholm → Alvesta → Nässjö → Mjölby → Linköping → Norrköping → Södertälje → Stockholm → Uppsala → Gävle → Sundsvall → Umeå → Boden → Gällivare → Kiruna

The distance is `1162` and the execution time, compared to our last DFS algorithm has drastically improved. Be careful however with debugging (print) statements, because they increase the execution time considerably. If I am to remove all of them, the execution time for this experiment is only *9 ms*.

Testing

Now, the efficiency of Dijkstra's allows us to test it to a bigger scale, such as a map of cities in Europe. We can choose a city such as Turin, in the North of Italy, and find the shortest path to 12 other cities.

The results of the benchmark are presented in the table below:

Longest Path:

Turin → Milano → Verona → Innsbruck → Zürich → Basel → Mannheim → Frankfurt → Göttingen → Hannover → Hamburg → Köpenhamn → Landskrona → Lund → Hässleholm → Kristianstad → Karlskrona → Emmaboda → Kalmar

Shortest path:

Turin → Genova

Destination City	**Distance**	**Time in ms**
Paris	782	3
Luxenburg	873	8
Göttingen	1035	6
Warsawa	1506	29
Constanta	2100	17
Barcelona	780	9
Kalmar	1733	27
Hannover	1102	6
Birmingham	1053	12
Genova	149	3
Venedig	205	10
Świnoujście	1523	47

Table 1: Distances and Travel Times to Destination Cities

Notice however that the execution time is not necessarily linearly dependent on the traveled distance. It can be influenced by more other factors, such as the number of neighbors of a city, how many connections are there between source and destination, etc.

If we reason about the time complexity, we have to take into consideration multiple factors. To start with, in a graph with n Cities, we know that each city is explored once, so the complexity should be at least linearly dependent on the size of the graph, at least $\mathcal{O}(n)$. However, the searching

involves other processes too. For example, we have to work with the neighbors of each city. If 'city1' has 'x' neighbors, we have to process all of them. If 'city2' has 'y' then we do the same for them. So the complexity becomes dependent also on the number of Connections of each city.

Also, every path is explored (and added to the queue). It is not necessary that all nodes are connected, so the number of paths can't necessarily be `n-1`, but we can say that we have m number of paths. In the case of our priority queue, the complexity is $\mathcal{O}(\log n)$. Thus, the actual time complexity of Dijkstra's algorithm is dependent on: vertices, edges and queue operations, all at the same time.

Conclusion

Therefore, Dijkstra's algorithm is a very performant tool for finding the shortest path between 2 vertices, having an important role in daily life situations and applications.