isep
École d'ingénieurs du numérique

II.2315 Algorithmics and Advanced Programming

Final Report

18th june 2020

Team Members
Andrew POURET (10083)
Nicolas WAGNER (10101)

Summary

Introduction

This report describes and analyses the work done for our final project of the ISEP “Algorithmie et Programmation Avancée” class. The project aims to put into practice the different design patterns and the graph theory seen throughout this semester’s algorithmics course.

The group choose to work on a dataset of the Subway System of the city of Prague, Czech Republic and use Java 11 (wrapped with Maven) without any other libraries except JUnit. The goal is to model the subway network of Prague with a unweighted and weighted graph, to perform search algorithms to find the shortest paths between stations and identify clusters in the network.

Class structure

Throughout this project we choose to partition our code in 4 packages (not counting the few test classes).

- *graph*
- *algorithms*
- *dto*
- *dtos*

Package “Graph”

This package contains the Object Oriented Programming description of the graph used throughout the project.

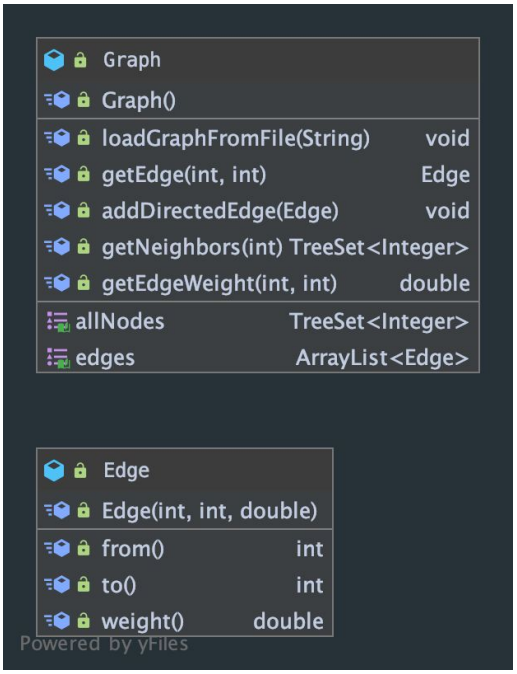

FIGURE 1 : Structure of classes within the “graph” package

Graph

Representation of the graph used throughout the project. Its attributes are the the list of nodes/vertices represented as integers and the edges, which a link between a nodes.

The Graph needs to initialized by providing a formatted file csv (*network_subway.csv* in our case).

Edge

Is the an object that contains a relation between two nodes (described as an int). The class has a from node, a to node and has a weight for the relation.

Package “algorithms”

Package containing all of the algorithms Graph Theory oriented used throughout the project.

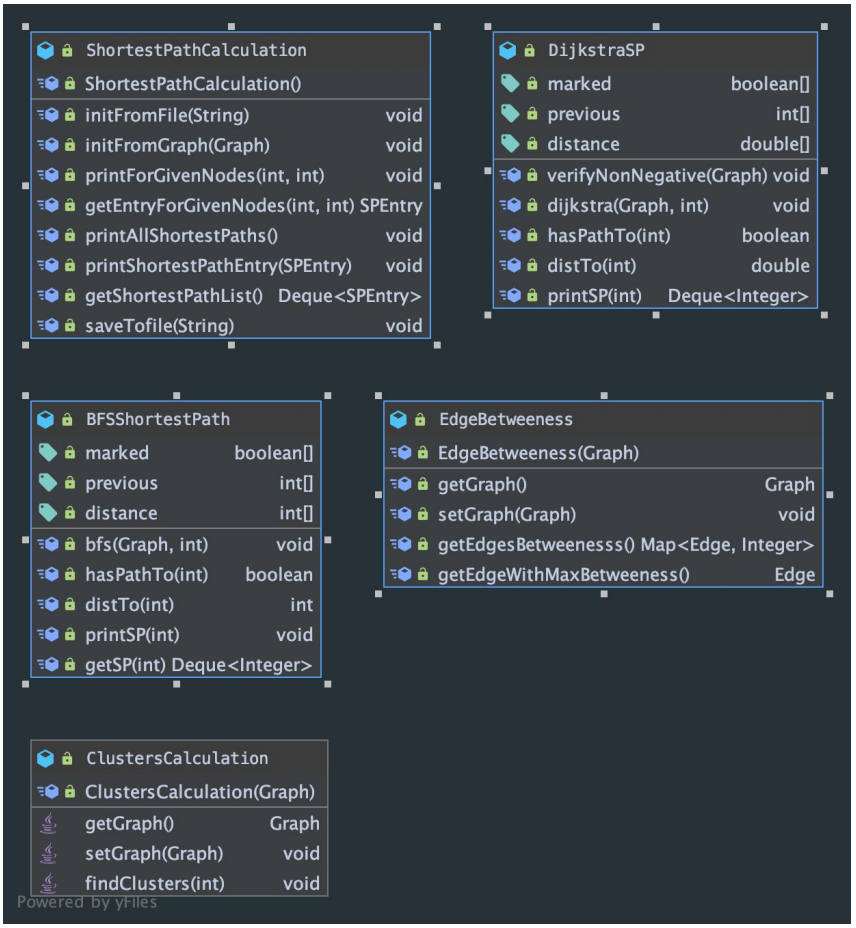

FIGURE 2: Class Structure of within the “algorithms” package

BFSShortestPath

The Breadth First Search algorithm starting from a start node with the possibility to compute the shortest path to nodes. This is used to find the shortest path without taking into account edge weight.

DijkstraSP

The Dijkstra algorithm with a computation of the shortest path from a start node chosen during initialization. This is used to find the shortest path by taking into account edge weight.

ShortestPathCalculations

Algorithm that can be initialized either by loading a Graph Object or loading a json file containing the results of this algorithm (to avoid intensive recomputing). Once the algorithm is initialized it computes and stores in its *shortestPathList* the results shortest paths between all nodes. The results can also be stored in a json file and it is possible to print or get the shortest path (BFS + Dijikstra) for two given nodes.

ClustersCalculation

Algorithm initialized using a graph that generates a given amount of sub graphs/clusters and modify the current graph accordingly.

EdgeBetweeness

Algorithm using in the Graph Clustering Algorithm initialized by passing graph that that computes the edge betweenness which means the number of the shortest paths that pass on an edge. It is possible to compute the edge betweenness of all edges and get the edge that has the maximum betweenness.

Package “dto”

Package containing objects that store data and calculation results in order to simplify the implementation of some the algorithms and facilitate interfacing algorithms throughout the application.

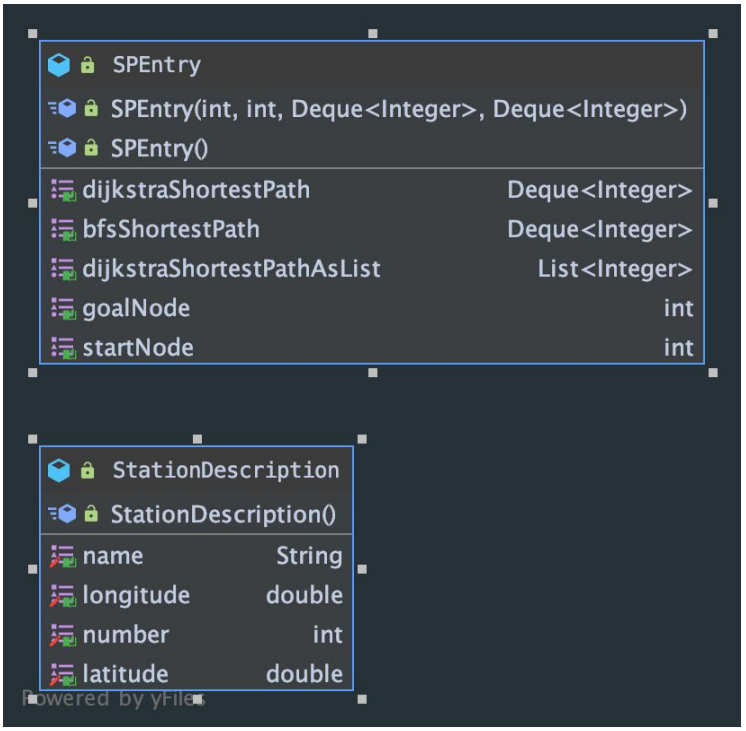

FIGURE 3 : Class Structure of the “dto” package

SPEntry

The Shortest Path Entry Data Transfer object is used in the ShortPathCalculation algorithm to store results. It stores the shortest path results of both the BFS Shortest Path Algorithm and the Dijsktra Shortest Path Algorithm between a start node and a goal node (represented as an integer).

StationDescription

The Station Descript Data Transfer Object is a java object representation of a row in *stop_prague.csv*. The class contains a name of the stop, the node number the station as well the longitude and the latitude to compute the weight.

Package “utils”

A set of utilitary classes used throughout the application.

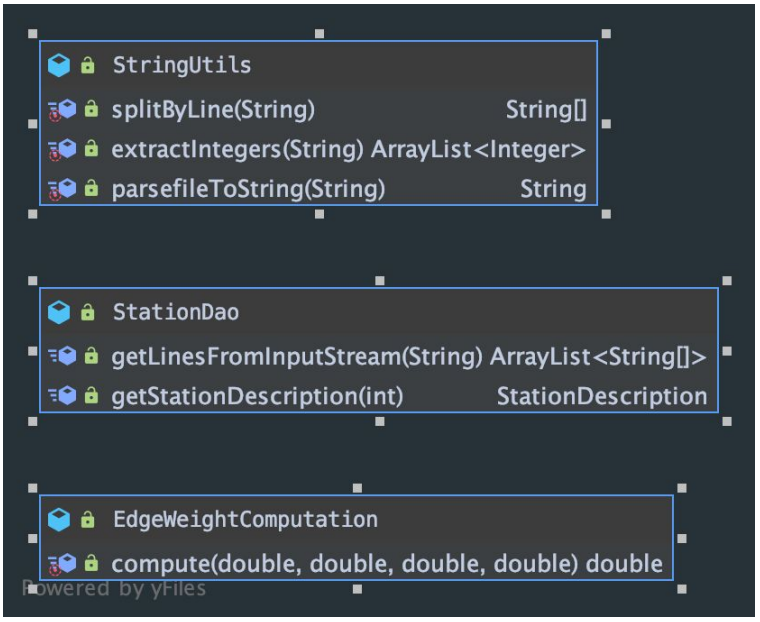

FIGURE 4 : Class Structure of the “utils” package

StringUtils

A utility class to a parse a given file to text and methods to simplify the use of the parsed text.

StationDao

Station Data Access Object, a class that provides a querying interface for the information on stations within the file *stops_prague.csv*. The querying returns StationDescripton DTOs that contains the name of the station, the lattitude and longitude for a given station number. This is very useful for the computation of weights and have a human friendly application (to display Stop Names instead of numbers for example).

EdgeWeightComputation

A utility class that provides a method that computes the edge weight from the latitude and longitude of both nodes..

Graph Import

We sourced the original data from an article in Nature : *A collection of public transport network data sets for 25 cities* , by Rainer Kujala, Christoffer Weckström, Richard K. Darst, Miloš N Mladenović & Jari Saramäki

In the given files, we selected the subway network description containing the following columns :

- from_stop_I;
- to_stop_I;
- duration_avg;
- n_vehicles;
- route_I_counts;

For our analysis through graph theory, we were only interested in the first 2 columns, describing the number of the station the subway was coming from and the number of the station it was travelling to. These two columns allow us to create the full graph, with all edges and nodes.

We also used another csv file containing for each station :

- latitude
- longitude
- name of the station
- number of the station

To import the full graph we used the following methods:

NB: methods are described by the classic java format :
[return type functionName(attributeType attributeName)] and the class they are in.

ArrayList<String[]> getLinesFromInputStream*(String* ***path****) in* ***StationDao*** *class:*

This parsing method is a classic use a the file reader function. It adds every line of the csv file as a String[] to an ArrayList, which is after used in the getStationDescription function.

StationDesciption getStationDescription*(int* ***stationNumber****) in* ***StationDao*** *class:*

This method is used to generate a StationDescription object from a station number. It finds the station number in the ArrayList and creates a StationDescription object with the latitude, longitude, name and number of the station.

void loadGraphFromFile*(String* ***path****) in* ***Graph*** *class:*

The main method to generate the graph. The required argument is the path of the dataset needed. This method iterates through the CSV file line after line and takes the first columns value(from_stops) as leftStation and rightStation.
It then uses the getStationDescription for each station to create a StationDescription for them.
It then adds the new edge from leftStation to rightStation to the edges collection, with its weight if they are provided.

Using these three methods, the graph can be created and imported in the Main class as :

```
Graph graph = new Graph();
graph.loadGraphFromFile( path: "network_subway.csv");
```

FIGURE 5: Code of the initialization of a graph using a CSV file in the main method.

Unweighted Graph

The unweighted graph is simply a list of edges that were previously imported from a file.

Shortest Path with BFS

Once the graph is initialized we can compute the compute the shortest path between a start node and a goal node.

This algorithm starts by doing Breadth First search:

1. Marks a given node as the start node.
2. Adds the node to an empty queue
3. Explores un-visited node adjacent to the starting node
4. Marks the node as visited.
5. Stores the distance to the un-visited node and stores the number (label) previously visited node.
6. Adds the node to the queue
7. Re-runs the algorithm until the queue is empty (which means every node is visited).

The algorithm then verifies if the a shortest path exists between the goal node and the start node (from the stored matrix of visited nodes). Then it computes the shortest path by retracing from the goal node to the start node the list of previously visited nodes.

Weighted Graph

Edge Weight Calculation Formula

The weighted graph for this project is defined by the geographical distance between the nodes, associated to each edge. We decided to compute the distance with the latitude and the longitude of each station given in the dataset.

To compute such distance, we implemented Harvesine formula:

$$\mathbf{a = \sin^2(\varphi_B - \varphi_A) + \cos(\varphi_A) \cdot \cos(\varphi_B) \cdot \sin^2(\lambda_B - \lambda_A)}$$

$$\mathbf{c = 2 \arctan\left(\frac{\sqrt{a}}{\sqrt{1-a}}\right)}$$

$$\mathbf{d = 6\,371 \times c}$$

with φ the latitude and λ the longitude.

The distance d here is in kilometers, so we converted it in meters.

Edge Weight Calculation Implementation

The implementation of the formula is in the compute method in EdgeComputation class which returns a *double*.

```java
public static double compute(double lat1, double lon1, double lat2, double lon2) {
    final int R = 6371; // Radius of the earth

    double latDistance = Math.toRadians(lat2 - lat1);
    double lonDistance = Math.toRadians(lon2 - lon1);
    double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
            + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
            * Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    return R * c * 1000;
}
```

FIGURE 6: Weight Calculation Algorithm from both of the node's latitude.

Shortest Path with the Dijkstra Algorithm

Once the weights added, we can compute the Dijkstra algorithm on our graph and find the shortest path.
Dijkstra basically works in 6 steps :

1. Mark all unvisited nodes. Create a set of all the unvisited nodes called unvisited.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. Otherwise, the current value will be kept.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Clustering

To find the clusters in our graph, the goal is to find the edges with the highest betweenness, remove them and compute the clusters.

To this end, we use the classes EdgeBetweeness and ClustersCalculation.

Edge Betweenness

Using the class *Edge Betweenness* we can get the betweenness of all edges in the graph, and find the edge with the maximum betweenness.

Our implementation of the *Edge Betweenness* computes for each the pair of nodes the number of times it occurs in the shortest paths between all nodes.

```
   public Map<Edge, Integer> getEdgesBetweenesss() {
        ShortestPathCalculation spCalculation = new
ShortestPathCalculation();
        spCalculation.initFromGraph(this.graph);

        Map<Edge, Integer> edgeBetweeness = new HashMap<>();

        for (Edge edge : this.graph.getEdges()) {
            edgeBetweeness.put(edge, 0);
        }

        // gets iterates through each shortest path between nodes and
increments the edge betweenness
        for (SPEntry entry : spCalculation.getShortestPathList()) {
            List<Integer> shortestPath =
entry.getDijkstraShortestPathAsList();

            if (shortestPath.size() > 0) {
                for (int i = 0; i < shortestPath.size() - 1; i++) {
                    Edge edge = this.graph.getEdge(shortestPath.get(i),
shortestPath.get(i + 1));
                    int betweenness = edgeBetweeness.get(edge);
                    edgeBetweeness.put(edge, betweenness + 1);
                }
            }
        }
        return edgeBetweeness;
    }
```

FIGURE 8: Implementation of the edge betweenness algorithm

We use this method in the ClustersCalculation method called findClusters, which returns an ArrayList of the clusters, which are themselves ArrayList<Integer> containing the nodes.

Calculating Clusters

The findClusters method creates a copy of the graph without the edge with the maximum betweenness, and then computes the clusters, until the number of maximum clusters is reached.

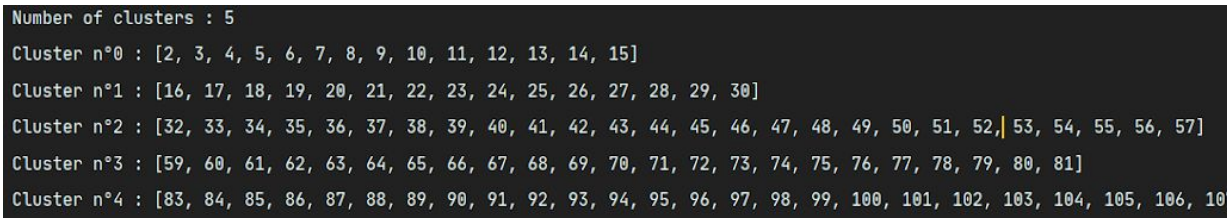

FIGURE 9: the output of the findClusters method with 5 clusters.

Conclusion

We successfully fulfilled the project and were able to model Prague subway into a graph as well computing shortest paths between nodes and clustering the graph in any given number of subgraphs.

This project helps us understand how real world problems such as finding the shortest path in a public transportation network can be solved using Graphs and different search algorithms.

The Graph Clustering here can help us find densely connected regions in order to reduce some of the calculation time (it's faster to a shortest path on smaller sub graphs).

If we were to put into production our algorithm we would need to work on reducing the calculation time by parallelizing calculations and caching in RAM memory results or intermediary results.

Even with some computing optimisations the graph here unfortunately doesn't take into account other means of transport, weather, strikes, public transportation schedules, delays etc and is very far from the algorithms used by Google Maps or City Mapper. A relevant improvement would also consists of implementing a more user-friendly graphical interface, to visualize the graph and the several algorithms.

Besides those possible improvements we are proud to present a complete project, and to have reached every objective we had set for it, within the deadlines. We also learned a lot on how to apply graph theory to a concrete and useful application.