An Analysis Tool for Railway Network Management

1.0

Generated by Doxygen 1.9.7

Chapter 1

An Analysis Tool for Railway Network Management

1.1 Project was delivered on April, 10th of 2023.

The purpose of the project is to develop an analysis tool for Railway Network. The tool will use a realistic dataset that resebles a real-life Railway Network of Portugal. The tool is to help to decide on how to best assign selected trains to lines, maximize revenue when multiple levels of service are offered, and identify sensitive sections of the network to failures to anticipate service disruption or mitigate its effects.

The project was developed by bachelor students of the *Faculty of Engineering of the University of Porto* as a Theoretical-Practical component of *Algorithm Design* course.

- David Burchakov (up202203777@edu.fe.up.pt)
- Pedro José Nunes Moutinho de Antas de Barros (up202108706@edu.fe.up.pt)

1.1.1 Checklist

- [T1.1: 1.0 point] :heavy_check_mark: Obviously, a first task will be to create a simple interface menu exposing all the functionalities implemented in the most user-friendly way possible. This menu will also be instrumental for you to showcase the work you have developed in a short demo to be held at the end of the project.
- [T1.2: 1.0 point] :heavy_check_mark: Similarly, you will also have to develop some basic functionality (accessible through your menu) to read and parse the provided data set files. This functionality will enable you (and the eventual user) to select alternative railway networks for analysis. With the extracted information, you are to create one (or more) appropriate graphs upon which you will carry out the requested tasks. The modelling of the graph is entirely up to you, so long as it is a sensible representation of the railway network and enables the correct application of the required algorithms.
- [T1.3: 2.0 points] :heavy_check_mark: In addition, you should also include documentation of all the implemented code, using Doxygen, indicating for each implemented algorithm the corresponding time complexity
- [T2.1: 3.5 points] :heavy_check_mark: Calculate the maximum number of trains that can simultaneously travel between two specific stations. Note that your implementation should take any valid source and destination stations as input;
- [T2.2: 2.0 points] :heavy_check_mark: Determine, from all pairs of stations, which ones (if more than one) require the most amount of trains when taking full advantage of the existing network capacity;

- [T2.3: 1.5 points] :heavy_check_mark: Indicate where management should assign larger budgets for the purchasing and maintenance of trains. That is, your implementation should be able to report the top-k municipalities and districts, regarding their transportation needs;
- [T2.4: 1 point] :heavy_check_mark: Report the maximum number of trains that can simultaneously arrive at a given station, taking into consideration the entire railway grid
- [T3.1: 2.0 points] :heavy_check_mark: Calculate the maximum amount of trains that can simultaneously travel between two specific stations with minimum cost for the company. Note that your system should also take any valid source and destination stations as input;
- [T4.1: 2.5 points] :heavy_check_mark: Calculate the maximum number of trains that can simultaneously travel between two specific stations in a network of reduced connectivity. Reduced connectivity is understood as being a subgraph (generated by your system) of the original railway network. Note that your system should also take any valid source and destination stations as input;
- [T4.2: 1.5 points] :heavy_check_mark: Provide a report on the stations that are the most affected by each segment failure, i.e., the top-k most affected stations for each segment to be considered
- [T5.1: 2.0 points] :heavy_check_mark: Use the (hopefully) user-friendly interface you have developed to illustrate the various algorithm results for a sample set of railway grids which you should develop specifically for the purposes of this demo. For instance, you can develop a small set of very modest railway networks for contrived capacities so that you can highlight the “correctness” of your solution. For instance, a grid that has a “constricted” segment where all traffic must go through, will clearly have a segment very “sensitive” to failures.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 CPheadquarters Class Reference

Public Member Functions

- void read_network (string path)

  Reads the file network.csv when given the path to the file and stores the information in a Graph.
- void read_stations (string path)

  Reads the files stations.csv when given the path to the file and stores the information in an unordered_map.
- void read_files ()

  Reads the files network.csv and stations.csv and stores the information in the Graph and unordered_map.
- Graph getLines () const

  Returns the Graph object.
- int T2_1maxflow (string station_A, string station_B)

  Calculates the maximum number of trains that can simultaneously travel between two specific stations.
- int T2_2maxflowAllStations ()

  Determines from all pairs of stations, which ones (if more than one) require the most amount of trains when taking full advantage of the existing network capacity.
- void T2_3municipality ()

  Indicates where management should assign larger budgets for the purchasing and maintenance of trains.
- void T2_3district ()

  Indicates where management should assign larger budgets for the purchasing and maintenance of trains.
- int T2_4maxArrive (string destination)

  Reports the maximum number of trains that can simultaneously arrive at a given station, taking into consideration the entire railway grid.
- int T3_1MinCost (string source, string destination)

  Calculates the maximum amount of trains that can simultaneously travel between two specific stations with minimum cost for the company steps: ∗1 - find all possible paths between source and destination ∗2 - define the optimal path, that is, has minimum cost per train.
- int T4_1ReducedConectivity (vector< string > unwantedEdges, string s, string t)

  Calculates the maximum number of trains that can simultaneously travel between two specific stations in a network of reduced connectivity.
- int T4_2Top_K_ReducedConectivity (vector< string > unwantedEdges)

  Provides a report on the stations that are the most affected by each segment failure, i.e., the top-k most affected stations for each segment to be considered.

4.1.1 Detailed Description

Definition at line 14 of file CPheadquarters.h.

4.1.2 Member Function Documentation

4.1.2.1 getLines()

```
Graph CPheadquarters::getLines ( ) const
```

Returns the Graph object.

Returns

Graph

Definition at line 144 of file CPheadquarters.cpp.

```
00144                                   {
00145     return this->lines;
00146 }
```

4.1.2.2 read_files()

```
void CPheadquarters::read_files ( )
```

Reads the files network.csv and stations.csv and stores the information in the Graph and unordered_map.

Definition at line 77 of file CPheadquarters.cpp.

```
00077                                   {
00078
00079     //-------------------------------------------Read
      network.csv-----------------------------------------
00080     std::ifstream inputFile1(R"(../network.csv)");
00081     string line1;
00082     std::getline(inputFile1, line1); // ignore first line
00083     while (getline(inputFile1, line1, '\n')) {
00084
00085         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00086             line1.pop_back(); // Remove the '\r' character
00087         }
00088
00089         string station_A;
00090         string station_B;
00091         string temp;
00092         int capacity;
00093         string service;
00094
00095         stringstream inputString(line1);
00096
00097         getline(inputString, station_A, ',');
00098         getline(inputString, station_B, ',');
00099         getline(inputString, temp, ',');
00100         getline(inputString, service, ',');
00101
00102         capacity = stoi(temp);
00103         lines.addVertex(station_A);
00104         lines.addVertex(station_B);
00105
00106         lines.addEdge(station_A, station_B, capacity, service);
00107     }
00108
00109
00110     //-------------------------------------------Read
      stations.csv-----------------------------------------
00111     std::ifstream inputFile2(R"(../stations.csv)");
00112     string line2;
00113     std::getline(inputFile2, line2); // ignore first line
```

```
00114
00115     while (getline(inputFile2, line2, '\n')) {
00116
00117         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00118             line1.pop_back(); // Remove the '\r' character
00119         }
00120
00121         string nome;
00122         string distrito;
00123         string municipality;
00124         string township;
00125         string line;
00126
00127         stringstream inputString(line2);
00128
00129         getline(inputString, nome, ',');
00130         getline(inputString, distrito, ',');
00131         getline(inputString, municipality, ',');
00132         getline(inputString, township, ',');
00133         getline(inputString, line, ',');
00134
00135         Station station(nome, distrito, municipality, township, line);
00136         stations[nome] = station;
00137
00138         // print information about the station, to make sure it was imported correctly
00139         //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « "
      township: " « township « " line: " « line « endl;
00140     }
00141 }
```

4.1.2.3 read_network()

```
void CPheadquarters::read_network (
              string path )
```

Reads the file network.csv when given the path to the file and stores the information in a Graph.

Parameters

path	

Definition at line 13 of file CPheadquarters.cpp.

```
00013                                                 {
00014     std::ifstream inputFile1(path);
00015     string line1;
00016     std::getline(inputFile1, line1); // ignore first line
00017     while (getline(inputFile1, line1, '\n')) {
00018
00019         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00020             line1.pop_back(); // Remove the '\r' character
00021         }
00022
00023         string station_A;
00024         string station_B;
00025         string temp;
00026         int capacity;
00027         string service;
00028
00029         stringstream inputString(line1);
00030
00031         getline(inputString, station_A, ',');
00032         getline(inputString, station_B, ',');
00033         getline(inputString, temp, ',');
00034         getline(inputString, service, ',');
00035
00036         capacity = stoi(temp);
00037         lines.addVertex(station_A);
00038         lines.addVertex(station_B);
00039
00040         lines.addEdge(station_A, station_B, capacity, service);
00041     }
00042 }
```

4.1.2.4 read_stations()

```
void CPheadquarters::read_stations (
            string path )
```

Reads the files stations.csv when given the path to the file and stores the information in an unordered_map.

Parameters

path	

Definition at line 44 of file CPheadquarters.cpp.

```
00044                                                {
00045     std::ifstream inputFile2(R"(../stations.csv)");
00046     string line2;
00047     std::getline(inputFile2, line2); // ignore first line
00048 
00049     while (getline(inputFile2, line2, '\n')) {
00050 
00051         if (!line2.empty() && line2.back() == '\r') { // Check if the last character is '\r'
00052             line2.pop_back(); // Remove the '\r' character
00053         }
00054 
00055         string nome;
00056         string distrito;
00057         string municipality;
00058         string township;
00059         string line;
00060 
00061         stringstream inputString(line2);
00062 
00063         getline(inputString, nome, ',');
00064         getline(inputString, distrito, ',');
00065         getline(inputString, municipality, ',');
00066         getline(inputString, township, ',');
00067         getline(inputString, line, ',');
00068 
00069         Station station(nome, distrito, municipality, township, line);
00070         stations[nome] = station;
00071 
00072         // print information about the station, to make sure it was imported correctly
00073         //cout << "station: " << nome << " distrito: " << distrito << " municipality: " << municipality << "
      township: " << township << " line: " << line << endl;
00074     }
00075 }
```

4.1.2.5 T2_1maxflow()

```
int CPheadquarters::T2_1maxflow (
            string station_A,
            string station_B )
```

Calculates the maximum number of trains that can simultaneously travel between two specific stations.

Takes any valid source and destination stations as input

Parameters

stationA	
stationB	

Returns

maxFlow

Definition at line 149 of file CPheadquarters.cpp.

```
00149                                                                    {
00150     Vertex *source = lines.findVertex(stationA); // set source vertex
00151     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00152 
00153     // Check if these stations even exist
00154     if (source == nullptr || sink == nullptr) {
00155         std::cerr « "Source or sink vertex not found." « std::endl;
00156         return 0;
00157     }
00158     int maxFlow = lines.edmondsKarp(stationA, stationB);
00159 
00160     if (maxFlow == 0) {
00161         cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " « 
      stationA
00162              « endl;
00163     } else {
00164         cout « "maxFlow:\t" « maxFlow « endl;
00165     }
00166 
00167     return maxFlow;
00168 }
```

4.1.2.6 T2_2maxflowAllStations()

```
int CPheadquarters::T2_2maxflowAllStations ( )
```

Determines from all pairs of stations, which ones (if more than one) require the most amount of trains when taking full advantage of the existing network capacity.

Print to the terminal all pairs of stations that require the most amount of trains (if more than one). Count the time it takes to run the algorithm and print it to the terminal.

See also

this function uses Graph::edmondsKarp() function

Returns

maxFlow

Definition at line 171 of file CPheadquarters.cpp.

```
00171                                                {
00172     vector<string> stations;
00173     int maxFlow = 0;
00174     auto length = lines.getVertexSet().size();
00175     // Start the timer
00176     auto start_time = std::chrono::high_resolution_clock::now();
00177     cout « "Calculating max flow for all pairs of stations..." « endl;
00178     cout « "Please stand by..." « endl;
00179     for (int i = 0; i < length; ++i) {
00180         for (int j = i + 1; j < length; ++j) {
00181             string stationA = lines.getVertexSet()[i]->getId();
00182             string stationB = lines.getVertexSet()[j]->getId();
00183             int flow = lines.edmondsKarp(stationA, stationB);
00184             if (flow == maxFlow) {
00185                 stations.push_back(stationB);
00186                 stations.push_back(stationA);
00187             } else if (flow > maxFlow) {
00188                 stations.clear();
00189                 stations.push_back(stationB);
00190                 stations.push_back(stationA);
00191                 maxFlow = flow;
00192             }
00193         }
00194     }
00195     // End the timer
00196     auto end_time = std::chrono::high_resolution_clock::now();
00197 
00198     // Compute the duration
00199     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00200 
```

```
00201     // Print the duration
00202     std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
00203
00204     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00205     for (int i = 0; i < stations.size(); i = i + 2) {
00206         cout « "------------------------\n";
00207         cout « "Source: " « stations[i + 1] « '\n';
00208         cout « "Target: " « stations[i] « '\n';
00209         cout « "------------------------\n";
00210     }
00211     return maxFlow;
00212 }
```

4.1.2.7 T2_3district()

```
void CPheadquarters::T2_3district ( )
```

Indicates where management should assign larger budgets for the purchasing and maintenance of trains.

Reports the top-k districts, regarding their transportation needs

Definition at line 244 of file CPheadquarters.cpp.

```
00244                                   {
00245     vector<pair<string , int» top_k;
00246     set<string> sett;
00247     for (auto m : stations) {
00248         sett.insert(m.second.get_district());
00249     }
00250     for (auto m : sett) {
00251         vector<string> desired_stations;
00252         for (auto p: stations) {
00253             if (p.second.get_district() == m) {
00254                 desired_stations.push_back(p.second.get_name());
00255             }
00256         }
00257         vector<string> souces = lines.find_sources(desired_stations);
00258         vector<string> targets = lines.find_targets(desired_stations);
00259         int diff=lines.mul_edmondsKarp(souces, targets);
00260         auto p = pair(m, diff);
00261         top_k.push_back(p);
00262     }
00263     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00264         return left.second > right.second;
00265     });
00266     for (int i = 0; i < 10; i++) {
00267         cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
00268     }
00269 }
```

4.1.2.8 T2_3municipality()

```
void CPheadquarters::T2_3municipality ( )
```

Indicates where management should assign larger budgets for the purchasing and maintenance of trains.

Reports the top-k municipalities, regarding their transportation needs

Definition at line 215 of file CPheadquarters.cpp.

```
00215                                       {
00216     vector<pair<string , int» top_k;
00217     set<string> sett;
00218     for (auto m : stations) {
00219         sett.insert(m.second.get_district());
00220     }
00221     for (auto m : sett) {
00222         vector<string> desired_stations;
00223         for (auto p: stations) {
00224             if (p.second.get_municipality() == m) {
00225                 desired_stations.push_back(p.second.get_name());
00226             }
00227         }
00228
```

```
00229
00230         vector<string> souces = lines.find_sources(desired_stations);
00231         vector<string> targets = lines.find_targets(desired_stations);
00232         int diff=lines.mul_edmondsKarp(souces, targets);
00233         auto p = pair(m, diff);
00234         top_k.push_back(p);
00235     }
00236     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00237         return left.second > right.second;
00238     });
00239     for (int i = 0; i < 10; i++) {
00240         cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
00241     }
00242 }
```

4.1.2.9 T2_4maxArrive()

```
int CPheadquarters::T2_4maxArrive (
            string destination )
```

Reports the maximum number of trains that can simultaneously arrive at a given station, taking into consideration the entire railway grid.

Parameters

destination	

Returns

maximum flow in a given station

Note

we consider the source station as the station that does not have any incoming edges

Definition at line 272 of file CPheadquarters.cpp.

```
00272                                                    {
00273     int maxFlow = 0;
00274     vector<string> org = lines.getSources();
00275     vector<string> targ = lines.getTargets();
00276     auto v = lines.findVertex(destination);
00277     lines.mul_edmondsKarp(org,targ);
00278     for(auto e : v->getIncoming()){
00279         maxFlow+=e->getFlow();
00280     }
00281     cout « "Max Flow: " « maxFlow « '\n';
00282     return maxFlow;
00283 }
```

4.1.2.10 T3_1MinCost()

```
int CPheadquarters::T3_1MinCost (
            string source,
            string destination )
```

Calculates the maximum amount of trains that can simultaneously travel between two specific stations with minimum cost for the company steps: ∗1 - find all possible paths between source and destination ∗2 - define the optimal path, that is, has minimum cost per train.

Parameters

source	
destination	

Returns

maximum flow between two specific stations

Definition at line 287 of file CPheadquarters.cpp.

```
00287                                                                {
00288     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00289     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00290     if (sourceVertex == nullptr || destVertex == nullptr) {
00291         cerr « "Source or destination vertex not found. Try again" « endl;
00292         return 1;
00293     }
00294
00295     Graph graph = lines;
00296
00297     std::vector<Vertex *> path;
00298     std::vector<std::vector<Vertex *» allPaths;
00299
00300
00301     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00302
00303     vector<int> maxFlows;
00304     vector<int> totalCosts;
00305
00306     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00307     for (auto path: allPaths) {
00308         int minWeight = 10;
00309         int totalCost = 0; // total cost of this path
00310         for (int i = 0; i + 1 < path.size(); i++) {
00311             std::cout « path[i]->getId() « " -> ";
00312             Edge *e = graph.findEdge(path[i], path[i + 1]);
00313             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00314             if (e->getWeight() < minWeight) {
00315                 minWeight = e->getWeight();
00316             }
00317
00318             // according to the problem's specification, the cost of STANDARD service is 2 euros and
      ALFA PENDULAR is 4
00319             if (e->getService() == "STANDARD") {
00320                 totalCost += 2;
00321             } else if (e->getService() == "ALFA PENDULAR") {
00322                 totalCost += 4;
00323             }
00324         }
00325         maxFlows.push_back(minWeight);
00326         totalCosts.push_back(totalCost);
00327         cout « " -> " « path[path.size() - 1]->getId() « endl;
00328         cout « "Max flow for this path: " « minWeight « " trains. ";
00329         cout « "Total cost: " « totalCost « " euros." « endl;
00330         std::cout « std::endl;
00331     }
00332
00333     // find the path with the minimum cost per train
00334     int maxTrains = 0;
00335     int resCost;
00336     double max_value = 10000;
00337     for (int i = 0; i < maxFlows.size(); ++i) {
00338         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00339         if (costPerTrain < max_value) {
00340             max_value = costPerTrain;
00341             maxTrains = maxFlows[i];
00342             resCost = totalCosts[i];
00343         }
00344     }
00345
00346     cout « "Max number of trains that can travel between " « source « " and " « destination
00347          « " with minimum cost"
00348          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00349     return maxTrains;
00350 }
```

4.1.2.11 T4_1ReducedConectivity()

```
int CPheadquarters::T4_1ReducedConectivity (
            vector< string > unwantedEdges,
            string s,
            string t )
```

Calculates the maximum number of trains that can simultaneously travel between two specific stations in a network of reduced connectivity.

Reduced connectivity is a subgraph of the original railway network. Takes any valid source and destination stations as input.

Note

it allows a user to remove edges from the railway network.

Parameters

unwantedEdges	
s	
t	

Returns

maximum flow between two specific stations

Definition at line 353 of file CPheadquarters.cpp.

```
00353
      {
00354     Graph graph;
00355     std::ifstream inputFile1(R"(../network.csv)");
00356     string line1;
00357     std::getline(inputFile1, line1); // ignore first line
00358     while (getline(inputFile1, line1, '\n')) {
00359
00360         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00361             line1.pop_back(); // Remove the '\r' character
00362         }
00363
00364         string station_A;
00365         string station_B;
00366         string temp;
00367         int capacity;
00368         string service;
00369         bool flag=true;
00370
00371         stringstream inputString(line1);
00372
00373         getline(inputString, station_A, ',');
00374         getline(inputString, station_B, ',');
00375         getline(inputString, temp, ',');
00376         getline(inputString, service, ',');
00377
00378         capacity = stoi(temp);
00379         graph.addVertex(station_A);
00380         graph.addVertex(station_B);
00381
00382         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00383                 if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00384                     flag=false;
00385                 }
00386         }
00387         if (flag) {
00388             graph.addEdge(station_A, station_B, capacity, service);
00389         }
00390         line1 = "";
00391     }
```

```
00392
00393     Vertex *source = graph.findVertex(s); // set source vertex
00394     Vertex *sink = graph.findVertex(t); // set sink vertex
00395
00396     // Check if these stations even exist
00397     if (source == nullptr || sink == nullptr) {
00398         std::cerr « "Source or sink vertex not found." « std::endl;
00399         return 1;
00400     }
00401     int maxFlow = graph.edmondsKarp(s, t);
00402
00403     if (maxFlow == 0) {
00404         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00405              « endl;
00406     }
00407     cout « "maxFlow:\t" « maxFlow « endl;
00408
00409
00410     return 1;
00411 }
```

4.1.2.12 T4_2Top_K_ReducedConectivity()

```
int CPheadquarters::T4_2Top_K_ReducedConectivity (
            vector< string > unwantedEdges )
```

Provides a report on the stations that are the most affected by each segment failure, i.e., the top-k most affected stations for each segment to be considered.

Parameters

unwantedEdges	

Returns

top-k most affected stations for each segment to be considered

Definition at line 414 of file CPheadquarters.cpp.

```
00414                                                                                {
00415     Graph graph;
00416     std::ifstream inputFile1(R"(../network.csv)");
00417     string line1;
00418     std::getline(inputFile1, line1); // ignore first line
00419     while (getline(inputFile1, line1, '\n')) {
00420
00421         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00422             line1.pop_back(); // Remove the '\r' character
00423         }
00424
00425         string station_A;
00426         string station_B;
00427         string temp;
00428         int capacity;
00429         string service;
00430         bool flag=true;
00431
00432         stringstream inputString(line1);
00433
00434         getline(inputString, station_A, ',');
00435         getline(inputString, station_B, ',');
00436         getline(inputString, temp, ',');
00437         getline(inputString, service, ',');
00438
00439         capacity = stoi(temp);
00440         graph.addVertex(station_A);
00441         graph.addVertex(station_B);
00442
00443         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00444             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00445                 flag = false;
00446                 break;
00447             }
```

```
00448         }
00449         if (flag) {
00450             graph.addEdge(station_A, station_B, capacity, service);
00451         }
00452         line1 = "";
00453     }
00454     vector<string> org = lines.getSources();
00455     vector<string> targ = lines.getTargets();
00456 
00457     lines.mul_edmondsKarp(org,targ);
00458     graph.mul_edmondsKarp(org,targ);
00459     vector<pair<int, int» top_k;
00460 
00461     auto length = lines.getVertexSet().size();
00462     for (int i = 0; i < length; ++i) {
00463         string destination = lines.getVertexSet()[i]->getId();
00464         auto v1 = lines.findVertex(destination);
00465         auto v2 = graph.findVertex(destination);
00466         int maxFlow1 = 0;
00467         int maxFlow2 = 0;
00468         for(auto e : v1->getIncoming()){
00469             maxFlow1+=e->getFlow();
00470         }
00471         for(auto e : v2->getIncoming()){
00472             maxFlow2+=e->getFlow();
00473         }
00474         int diff = maxFlow1 - maxFlow2;
00475         auto p = pair(i, diff);
00476         top_k.push_back(p);
00477     }
00478     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00479         return left.second > right.second;
00480     });
00481     for (int i = 0; i < 10; i++) {
00482         cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second
    « '\n';
00483     }
00484     return 1;
00485 }
```

The documentation for this class was generated from the following files:

- CPheadquarters.h
- CPheadquarters.cpp

4.2 Edge Class Reference

Public Member Functions

- Edge (Vertex ∗orig, Vertex ∗dest, int w, const std::string &service)
- Vertex ∗ getDest () const
- int getWeight () const
- bool isSelected () const
- Vertex ∗ getOrig () const
- Edge ∗ getReverse () const
- double getFlow () const
- void setSelected (bool selected)
- void setReverse (Edge ∗reverse)
- void setFlow (double flow)
- std::string getService () const
- void setService (const std::string &service)

Protected Attributes

- Vertex * dest
- int weight
- std::string service
- bool selected = false
- Vertex * orig
- Edge * reverse = nullptr
- double flow

4.2.1 Detailed Description

Definition at line 78 of file VertexEdge.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Edge()

```
Edge::Edge (
            Vertex * orig,
            Vertex * dest,
            int w,
            const std::string & service )
```

Definition at line 128 of file VertexEdge.cpp.

```
00128                                                   : orig(orig), dest(dest),
      weight(w),
00129                                                         service(service), flow(0)
      {}
```

4.2.3 Member Function Documentation

4.2.3.1 getDest()

```
Vertex * Edge::getDest ( ) const
```

Definition at line 131 of file VertexEdge.cpp.

```
00131                                {
00132     return this->dest;
00133 }
```

4.2.3.2 getFlow()

```
double Edge::getFlow ( ) const
```

Definition at line 151 of file VertexEdge.cpp.

```
00151                               {
00152     return flow;
00153 }
```

4.2.3.3 getOrig()

```
Vertex * Edge::getOrig ( ) const
```

Definition at line 139 of file VertexEdge.cpp.

```
00139                           {
00140     return this->orig;
00141 }
```

4.2.3.4 getReverse()

```
Edge * Edge::getReverse ( ) const
```

Definition at line 143 of file VertexEdge.cpp.

```
00143                            {
00144     return this->reverse;
00145 }
```

4.2.3.5 getService()

```
std::string Edge::getService ( ) const
```

Definition at line 171 of file VertexEdge.cpp.

```
00171                                {
00172     return this->service;
00173 }
```

4.2.3.6 getWeight()

```
int Edge::getWeight ( ) const
```

Definition at line 135 of file VertexEdge.cpp.

```
00135                          {
00136     return this->weight;
00137 }
```

4.2.3.7 isSelected()

```
bool Edge::isSelected ( ) const
```

Definition at line 147 of file VertexEdge.cpp.

```
00147                            {
00148     return this->selected;
00149 }
```

4.2.3.8 setFlow()

```
void Edge::setFlow (
            double flow )
```

Definition at line 163 of file VertexEdge.cpp.

```
00163                             {
00164     this->flow = flow;
00165 }
```

4.2.3.9 setReverse()

```
void Edge::setReverse (
            Edge * reverse )
```

Definition at line 159 of file VertexEdge.cpp.

```
00159                                    {
00160     this->reverse = reverse;
00161 }
```

4.2.3.10 setSelected()

```
void Edge::setSelected (
            bool selected )
```

Definition at line 155 of file VertexEdge.cpp.

```
00155                                   {
00156     this->selected = selected;
00157 }
```

4.2.3.11 setService()

```
void Edge::setService (
            const std::string & service )
```

Definition at line 167 of file VertexEdge.cpp.

```
00167                                              {
00168     this->service = service;
00169 }
```

4.2.4 Member Data Documentation

4.2.4.1 dest

```
Vertex* Edge::dest  [protected]
```

Definition at line 105 of file VertexEdge.h.

4.2.4.2 flow

```
double Edge::flow  [protected]
```

Definition at line 116 of file VertexEdge.h.

4.2.4.3 orig

```
Vertex* Edge::orig  [protected]
```

Definition at line 113 of file VertexEdge.h.

4.2.4.4 reverse

```
Edge* Edge::reverse = nullptr  [protected]
```

Definition at line 114 of file VertexEdge.h.

4.2.4.5 selected

```
bool Edge::selected = false  [protected]
```

Definition at line 110 of file VertexEdge.h.

4.2.4.6 service

```
std::string Edge::service  [protected]
```

Definition at line 108 of file VertexEdge.h.

4.2.4.7 weight

```
int Edge::weight  [protected]
```

Definition at line 106 of file VertexEdge.h.

The documentation for this class was generated from the following files:

- VertexEdge.h
- VertexEdge.cpp

4.3 Graph Class Reference

Public Member Functions

- Vertex * findVertex (const std::string &id) const

  Auxiliary function to find a vertex with a given ID.
- bool addVertex (const std::string &id)

  Adds a vertex with a given content or info (in) to a graph (this).
- bool addEdge (const std::string &sourc, const std::string &dest, int w, const std::string &service)

  Adds an edge to a graph (this), given the contents of the source and destination vertices and the edge weight (w).
- int getNumVertex () const
- std::vector< Vertex * > getVertexSet () const
- void print () const

  prints the graph
- int edmondsKarp (const std::string &s, const std::string &t)

  finds the maximum flow in the graph, given a source and a target
- std::vector< std::string > getSources ()

  finds all the source vertexes of the entire graph

- std::vector< std::string > getTargets ()

  finds all the target vertexes of the entire graph
- int mul_edmondsKarp (std::vector< std::string > souces, std::vector< std::string > targets)

  finds the maximum flow in the graph, given a set of sources and a set of targets
- std::vector< std::string > find_sources (std::vector< std::string > desired_stations)

  finds all the source vertexes of a sub_graph
- std::vector< std::string > find_targets (std::vector< std::string > desired_stations)

  finds all the target vertexes of a sub_graph
- void findAllPaths (Vertex *source, Vertex *destination, std::vector< Vertex * > &path, std::vector< std::vector< Vertex * > > &allPaths)

  finds all existing paths for a given source and destination return a vector of paths as an out parameter
- Edge * findEdge (Vertex *source, Vertex *destination)

  find an edge in the graph, based on a a source and a destination vertices

Protected Member Functions

- void updateFlow (Vertex *s, Vertex *t, int bottleneck)

  auxiliary function to update the flow of an augmenting path
- int findMinResidual (Vertex *s, Vertex *t)

  auxiliary function to find the minimum residual capacity of an augmenting path
- bool findAugmentingPath (const std::string &s, const std::string &t)

  auxiliary function to find an augmenting path, given a source and a target
- void testAndVisit (std::queue< Vertex * > &q, Edge *e, Vertex *w, double residual)

  auxiliary function to test and visit a vertex, given a queue, an edge, a vertex and a residual
- bool isIn (std::string n, std::vector< std::string > vec)
- void deleteVertex (std::string name)

  delete a vertex from the graph, making a subgraph from a graph

Protected Attributes

- std::vector< Vertex * > vertexSet
- double ** distMatrix = nullptr
- int ** pathMatrix = nullptr

4.3.1 Detailed Description

Definition at line 15 of file Graph.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 ~Graph()

```
Graph::~Graph ( )
```

Definition at line 63 of file Graph.cpp.

```
00063                {
00064     deleteMatrix(distMatrix, vertexSet.size());
00065     deleteMatrix(pathMatrix, vertexSet.size());
00066 }
```

4.3.3 Member Function Documentation

4.3.3.1 addEdge()

```
bool Graph::addEdge (
            const std::string & sourc,
            const std::string & dest,
            int w,
            const std::string & service )
```

Adds an edge to a graph (this), given the contents of the source and destination vertices and the edge weight (w).

Parameters

sourc	
dest	
w	
service	

Returns

true if successful, and false if the source or destination vertex does not exist.

Definition at line 34 of file Graph.cpp.

```
00034                                                                                                {
00035     auto v1 = findVertex(sourc);
00036     auto v2 = findVertex(dest);
00037     if (v1 == nullptr || v2 == nullptr)
00038         return false;
00039     v1->addEdge(v2, w, service);
00040
00041     return true;
00042 }
```

4.3.3.2 addVertex()

```
bool Graph::addVertex (
            const std::string & id )
```

Adds a vertex with a given content or info (in) to a graph (this).

Parameters

id	

Returns

true if successful, and false if a vertex with that content already exists.

Definition at line 26 of file Graph.cpp.

```
00026                                          {
00027     if (findVertex(id) != nullptr)
00028         return false;
00029     vertexSet.push_back(new Vertex(id));
00030     return true;
00031 }
```

4.3.3.3 deleteVertex()

```
void Graph::deleteVertex (
            std::string name )  [protected]
```

delete a vertex from the graph, making a subgraph from a graph

Parameters

name	

Definition at line 323 of file Graph.cpp.

```
00323                                       {
00324     auto v = findVertex(name);
00325     for(auto e : v->getAdj()){
00326         auto s = e->getDest()->getId();
00327         v->removeEdge(s);
00328     }
00329     for(auto e : v->getIncoming()){
00330         e->getOrig()->removeEdge(name);
00331     }
00332     auto it = vertexSet.begin();
00333     while (it!=vertexSet.end()){
00334         Vertex* currentVertex = *it;
00335         if(currentVertex->getId()==name){
00336             it=vertexSet.erase(it);
00337         }
00338         else{
00339             it++;
00340         }
00341     }
00342 }
```

4.3.3.4 edmondsKarp()

```
int Graph::edmondsKarp (
            const std::string & s,
            const std::string & t )
```

finds the maximum flow in the graph, given a source and a target

Parameters

s	
t	

Returns

maximum flow

Note

The Edmonds-Karp algorithm is a special case of the Ford-Fulkerson algorithm.

It uses Breadth-First Search to find the augmenting paths with the minimum number of edges

Attention

The time complexity of the Edmonds-Karp algorithm is O(V∗E^2), where V is the number of vertices and E is the number of edges in the graph.

Definition at line 163 of file Graph.cpp.

```
00163                                                   {
00164     for (auto e: vertexSet) {
00165         for (auto i: e->getAdj()) {
00166             i->setFlow(0);
00167         }
00168     }
00169     int maxFlow = 0;
00170     while (findAugmentingPath(s, t)) {
00171         int bottleneck = findMinResidual(findVertex(s), findVertex(t));
00172         updateFlow(findVertex(s), findVertex(t), bottleneck);
00173         maxFlow += bottleneck;
00174     }
00175     return maxFlow;
00176 }
```

4.3.3.5 find_sources()

```
std::vector< std::string > Graph::find_sources (
            std::vector< std::string > desired_stations )
```

finds all the source vertexes of a sub_graph

Parameters

desired_stations	

Returns

vector with the id's of the target vertexes

Definition at line 178 of file Graph.cpp.

```
00178                                                                                   {
00179     std::vector<std::string> res;
00180
00181     for (std::string s: desired_stations) {
00182         bool flag = true;
00183         auto v = findVertex(s);
00184         if (v == nullptr) {
00185             std::cout « "Trouble finding source " « s « '\n';
00186             continue;
00187         }
00188         for (auto e: v->getIncoming()) {
00189             if (isIn(e->getOrig()->getId(), desired_stations)) {
00190                 flag=false;
00191             }
00192         }
00193         if (flag) res.push_back(s);
00194     }
00195     return res;
00196 }
```

4.3.3.6 find_targets()

```
std::vector< std::string > Graph::find_targets (
            std::vector< std::string > desired_stations )
```

finds all the target vertexes of a sub_graph

Parameters

desired_stations	

Returns

vector with the id's of the target vertexes

Definition at line 198 of file Graph.cpp.

```
00198                                                                                {
00199     std::vector<std::string> res;
00200     for (std::string s: desired_stations) {
00201         bool flag = true;
00202         auto v = findVertex(s);
00203         if (v == nullptr) {
00204             std::cout « "Trouble finding target " « s « '\n';
00205             continue;
00206         }
00207         for (auto e: v->getAdj()) {
00208             if (isIn(e->getDest()->getId(), desired_stations)) {
00209                 flag=false;
00210             }
00211         }
00212         if (flag) res.push_back(s);
00213     }
00214     return res;
00215 }
```

4.3.3.7 findAllPaths()

```
void Graph::findAllPaths (
            Vertex * source,
            Vertex * destination,
            std::vector< Vertex * > & path,
            std::vector< std::vector< Vertex * > > & allPaths )
```

finds all existing paths for a given source and destination return a vector of paths as an out parameter

Parameters

source	
destination	
path	
allPaths	

Definition at line 269 of file Graph.cpp.

```
00270                                                                                {
00271     path.push_back(source);
00272     source->setVisited(true);
00273
00274     if (source == destination) {
00275         allPaths.push_back(path);
00276     } else {
00277         for (auto edge: source->getAdj()) {
00278             Vertex *adjacent = edge->getDest();
00279             if (!adjacent->isVisited()) {
00280                 findAllPaths(adjacent, destination, path, allPaths);
00281             }
00282         }
00283     }
00284
00285     path.pop_back();
00286     source->setVisited(false);
00287 }
```

4.3.3.8 findAugmentingPath()

```
bool Graph::findAugmentingPath (
            const std::string & s,
            const std::string & t )  [protected]
```

auxiliary function to find an augmenting path, given a source and a target

Parameters

s	
t	

Returns

true if an augmenting path was found, and false otherwise

Note

An augmenting path is a simple path - a path that does not contain cycles

Attention

This function uses the BFS algorithm.

The time complexity of the BFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph.

Definition at line 98 of file Graph.cpp.

```
00098                                                                    {
00099     Vertex *source = findVertex(s);
00100     Vertex *target = findVertex(t);
00101     if (source == nullptr || target == nullptr) {
00102         return false;
00103     }
00104     for (auto v: vertexSet) {
00105         v->setVisited(false);
00106         v->setPath(nullptr);
00107     }
00108     source->setVisited(true);
00109     std::queue<Vertex *> q;
00110     q.push(source);
00111     while (!q.empty()) {
00112         auto v = q.front();
00113         q.pop();
00114         for (auto e: v->getAdj()) {
00115             auto w = e->getDest();
00116             double residual = e->getWeight() - e->getFlow();
00117             testAndVisit(q, e, w, residual);
00118         }
00119         for (auto e: v->getIncoming()) {
00120             auto w = e->getDest();
00121             double residual = e->getFlow();
00122             testAndVisit(q, e->getReverse(), w, residual);
00123         }
00124         if (target->isVisited()) {
00125             return true;
00126         }
00127     }
00128     return false;
00129 }
```

4.3.3.9 findEdge()

```
Edge * Graph::findEdge (
            Vertex * source,
            Vertex * destination )
```

find an edge in the graph, based on a a source and a destination vertices

Parameters

source	
destination	

Returns

edge

Definition at line 291 of file Graph.cpp.

```
00291                                                       {
00292
00293     for (auto edge: source->getAdj()) {
00294         if (edge->getDest() == destination) {
00295             return edge;
00296         }
00297     }
00298     return nullptr;
00299 }
```

4.3.3.10 findMinResidual()

```
int Graph::findMinResidual (
            Vertex * s,
            Vertex * t )  [protected]
```

auxiliary function to find the minimum residual capacity of an augmenting path

Parameters

s	
t	

Returns

the minimum residual capacity of an augmenting path

Definition at line 132 of file Graph.cpp.

```
00132                                                {
00133     double minResidual = INT_MAX;
00134     for (auto v = t; v != s;) {
00135         auto e = v->getPath();
00136         if (e->getDest() == v) {
00137             minResidual = std::min(minResidual, e->getWeight() - e->getFlow());
00138             v = e->getOrig();
00139         } else {
00140             minResidual = std::min(minResidual, e->getFlow());
00141             v = e->getDest();
00142         }
00143     }
00144     return minResidual;
00145 }
```

4.3.3.11 findVertex()

```
Vertex * Graph::findVertex (
            const std::string & id ) const
```

Auxiliary function to find a vertex with a given ID.

Parameters

id	

Returns

vertex pointer to vertex with given content, or nullptr if not found

Definition at line 16 of file Graph.cpp.

```
00016                                                   {
00017     for (auto v: vertexSet) {
00018         if (v->getId() == id)
00019             return v;
00020     }
00021     return nullptr;
00022 }
```

4.3.3.12 getNumVertex()

```
int Graph::getNumVertex ( ) const
```

Definition at line 7 of file Graph.cpp.

```
00007                                {
00008     return vertexSet.size();
00009 }
```

4.3.3.13 getSources()

```
std::vector< std::string > Graph::getSources ( )
```

finds all the source vertexes of the entire graph

Returns

vector with the id's of the source vertexes

Definition at line 302 of file Graph.cpp.

```
00302                                       {
00303     std::vector<std::string> res;
00304     for (auto v : vertexSet) {
00305         if(v->getIncoming().empty()){
00306             res.push_back(v->getId());
00307         }
00308     }
00309     return res;
00310 }
```

4.3.3.14 getTargets()

```
std::vector< std::string > Graph::getTargets ( )
```

finds all the target vertexes of the entire graph

Returns

vector with the id's of the target vertexes

Definition at line 312 of file Graph.cpp.

```
00312                                       {
00313     std::vector<std::string> res;
00314     for (auto v : vertexSet) {
00315         if(v->getAdj().empty()){
00316             res.push_back(v->getId());
00317         }
00318     }
00319     return res;
00320 }
```

4.3.3.15 getVertexSet()

```
std::vector< Vertex * > Graph::getVertexSet ( ) const
```

Definition at line 11 of file Graph.cpp.

```
00011                                                {
00012     return vertexSet;
00013 }
```

4.3.3.16 isIn()

```
bool Graph::isIn (
            std::string n,
            std::vector< std::string > vec )  [protected]
```

Definition at line 218 of file Graph.cpp.

```
00218                                                          {
00219     for (std::string s: vec) {
00220         if (s == n) return true;
00221     }
00222     return false;
00223 }
```

4.3.3.17 mul_edmondsKarp()

```
int Graph::mul_edmondsKarp (
            std::vector< std::string > souces,
            std::vector< std::string > targets )
```

finds the maximum flow in the graph, given a set of sources and a set of targets

Parameters

souces	
targets	

Returns

maximum flow

Definition at line 226 of file Graph.cpp.

```
00226                                                                                {
00227     auto it1 = souces.begin();
00228     while (it1 != souces.end()) {
00229         if (isIn(*it1, targets)) {
00230             it1 = souces.erase(it1);
00231         } else it1++;
00232     }
00233
00234     auto it2 = targets.begin();
00235     while (it2 != targets.end()) {
00236         if (isIn(*it2, souces)) {
00237             it2 = souces.erase(it2);
00238         } else it2++;
00239     }
00240
00241     addVertex("temp_source");
00242     for (std::string s: souces) {
00243         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00244     }
00245
```

```
00246     addVertex("temp_targets");
00247     for (std::string s: targets) {
00248         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00249     }
00250     for (auto e: vertexSet) {
00251         for (auto i: e->getAdj()) {
00252             i->setFlow(0);
00253         }
00254     }
00255     int maxFlow = 0;
00256     while (findAugmentingPath("temp_source", "temp_targets")) {
00257         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00258         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00259         maxFlow += bottleneck;
00260     }
00261     deleteVertex("temp_targets");
00262     deleteVertex("temp_source");
00263     return maxFlow;
00264 }
```

4.3.3.18 print()

```
void Graph::print ( ) const
```

prints the graph

Definition at line 70 of file Graph.cpp.

```
00070                       {
00071     std::cout « "---------------- Graph----------------\n";
00072     std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
00073     std::cout « "Vertices:\n";
00074     for (const auto &vertex: vertexSet) {
00075         std::cout « vertex->getId() « " ";
00076     }
00077     std::cout « "\nEdges:\n";
00078     for (const auto &vertex: vertexSet) {
00079         for (const auto &edge: vertex->getAdj()) {
00080             std::cout « vertex->getId() « " -> " « edge->getDest()->getId() « " (weight: " «
      edge->getWeight() « ", service: " « edge->getService() « ")" « std::endl;
00081         }
00082     }
00083 }
```

4.3.3.19 testAndVisit()

```
void Graph::testAndVisit (
            std::queue< Vertex * > & q,
            Edge * e,
            Vertex * w,
            double residual )  [protected]
```

auxiliary function to test and visit a vertex, given a queue, an edge, a vertex and a residual

Parameters

q	
e	
w	
residual	

Definition at line 88 of file Graph.cpp.

```
00088                                                                                   {
00089     if (!w->isVisited() && residual > 0) {
00090         w->setVisited(true);
00091         w->setPath(e);
```

```
00092         q.push(w);
00093     }
00094 }
```

4.3.3.20 updateFlow()

```
void Graph::updateFlow (
            Vertex * s,
            Vertex * t,
            int bottleneck )  [protected]
```

auxiliary function to update the flow of an augmenting path

Parameters

s	
t	
bottleneck	

Note

The bottleneck is the minimum residual capacity of an augmenting path

Definition at line 148 of file Graph.cpp.

```
00148                                                                {
00149     for (auto v = t; v != s;) {
00150         auto e = v->getPath();
00151         double flow = e->getFlow();
00152         if (e->getDest() == v) {
00153             e->setFlow(flow + bottleneck);
00154             v = e->getOrig();
00155         } else {
00156             e->setFlow(flow - bottleneck);
00157             v = e->getDest();
00158         }
00159     }
00160 }
```

4.3.4 Member Data Documentation

4.3.4.1 distMatrix

```
double** Graph::distMatrix = nullptr  [protected]
```

Definition at line 121 of file Graph.h.

4.3.4.2 pathMatrix

```
int** Graph::pathMatrix = nullptr  [protected]
```

Definition at line 122 of file Graph.h.

4.3.4.3 vertexSet

```
std::vector<Vertex *> Graph::vertexSet  [protected]
```

Definition at line 119 of file Graph.h.

The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

4.4 Station Class Reference

Public Member Functions

- Station ()
  Default constructor.
- Station (string name_, string district_, string municipality_, string township_, string line_)
  Constructor.
- string get_name ()
  Returns the station's name.
- string get_district ()
  Returns the station's district.
- string get_municipality ()
  Returns the station's municipality.
- string get_township ()
  Returns the station's township.
- string get_line ()
  Returns the station's line.

4.4.1 Detailed Description

Definition at line 12 of file Station.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Station() [1/2]

```
Station::Station ( )
```

Default constructor.

Definition at line 35 of file Station.cpp.

```
00035                   {
00036
00037 }
```

4.4.2.2 Station() [2/2]

```
Station::Station (
            string name_,
            string district_,
            string municipality_,
            string township_,
            string line_ )
```

Constructor.

Parameters

*name_*	
*district_*	
*municipality_*	
*township_*	
*line_*	

Definition at line 7 of file Station.cpp.

```
00007                                                                                                      {
00008     name=name_;
00009     municipality=municipality_;
00010     district=district_;
00011     township=township_;
00012     line=line_;
00013 }
```

4.4.3 Member Function Documentation

4.4.3.1 get_district()

```
string Station::get_district ( )
```

Returns the station's district.

Returns

district

Definition at line 19 of file Station.cpp.

```
00019                                {
00020     return district;
00021 }
```

4.4.3.2 get_line()

```
string Station::get_line ( )
```

Returns the station's line.

Returns

line

Definition at line 31 of file Station.cpp.

```
00031                            {
00032     return line;
00033 }
```

4.4.3.3 get_municipality()

```
string Station::get_municipality ( )
```

Returns the station's municipality.

Returns

municipality

Definition at line 23 of file Station.cpp.

```
00023                                   {
00024     return municipality;
00025 }
```

4.4.3.4 get_name()

```
string Station::get_name ( )
```

Returns the station's name.

Returns

name

Definition at line 15 of file Station.cpp.

```
00015                                {
00016     return name;
00017 }
```

4.4.3.5 get_township()

```
string Station::get_township ( )
```

Returns the station's township.

Returns

township

Definition at line 27 of file Station.cpp.

```
00027                                  {
00028     return township;
00029 }
```

The documentation for this class was generated from the following files:

- Station.h
- Station.cpp

4.5 Vertex Class Reference

Public Member Functions

- Vertex (std::string id)
- bool operator< (Vertex &vertex) const
- std::string getId () const
- std::vector< Edge * > getAdj () const
- bool isVisited () const
- bool isProcessing () const
- unsigned int getIndegree () const
- double getDist () const
- Edge * getPath () const
- std::vector< Edge * > getIncoming () const
- void setId (int info)
- void setVisited (bool visited)
- void setProcesssing (bool processing)
- void setIndegree (unsigned int indegree)
- void setDist (double dist)
- void setPath (Edge *path)
- Edge * addEdge (Vertex *dest, int w, const std::string &service)
- bool removeEdge (std::string destID)

Protected Member Functions

- void print () const

Protected Attributes

- std::string id
- std::vector< Edge * > adj
- bool visited = false
- bool processing = false
- unsigned int indegree
- double dist = 0
- Edge * path = nullptr
- std::vector< Edge * > incoming
- int queueIndex = 0

4.5.1 Detailed Description

Definition at line 19 of file VertexEdge.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 Vertex()

```
Vertex::Vertex (
            std::string id )
```

Definition at line 7 of file VertexEdge.cpp.

```
00007 : id(id) {}
```

4.5.3 Member Function Documentation

4.5.3.1 addEdge()

```
Edge * Vertex::addEdge (
            Vertex * dest,
            int w,
            const std::string & service )
```

Definition at line 13 of file VertexEdge.cpp.

```
00013                                                                    {
00014     auto newEdge = new Edge(this, d, w, service);
00015     adj.push_back(newEdge);
00016     d->incoming.push_back(newEdge);
00017     return newEdge;
00018 }
```

4.5.3.2 getAdj()

```
std::vector< Edge * > Vertex::getAdj ( ) const
```

Definition at line 59 of file VertexEdge.cpp.

```
00059                                        {
00060     return this->adj;
00061 }
```

4.5.3.3 getDist()

```
double Vertex::getDist ( ) const
```

Definition at line 75 of file VertexEdge.cpp.

```
00075                              {
00076     return this->dist;
00077 }
```

4.5.3.4 getId()

```
std::string Vertex::getId ( ) const
```

Definition at line 55 of file VertexEdge.cpp.

```
00055                             {
00056     return this->id;
00057 }
```

4.5.3.5 getIncoming()

```
std::vector< Edge * > Vertex::getIncoming ( ) const
```

Definition at line 83 of file VertexEdge.cpp.

```
00083                                          {
00084     return this->incoming;
00085 }
```

4.5.3.6 getIndegree()

```
unsigned int Vertex::getIndegree ( ) const
```

Definition at line 71 of file VertexEdge.cpp.

```
00071                                       {
00072     return this->indegree;
00073 }
```

4.5.3.7 getPath()

```
Edge * Vertex::getPath ( ) const
```

Definition at line 79 of file VertexEdge.cpp.

```
00079                                  {
00080     return this->path;
00081 }
```

4.5.3.8 isProcessing()

```
bool Vertex::isProcessing ( ) const
```

Definition at line 67 of file VertexEdge.cpp.

```
00067                                  {
00068     return this->processing;
00069 }
```

4.5.3.9 isVisited()

```
bool Vertex::isVisited ( ) const
```

Definition at line 63 of file VertexEdge.cpp.

```
00063                                {
00064     return this->visited;
00065 }
```

4.5.3.10 operator<()

```
bool Vertex::operator< (
            Vertex & vertex ) const
```

Definition at line 51 of file VertexEdge.cpp.

```
00051                                                {
00052     return this->dist < vertex.dist;
00053 }
```

4.5.3.11 print()

```
void Vertex::print ( ) const  [protected]
```

Definition at line 112 of file VertexEdge.cpp.

```
00112                           {
00113     std::cout « "Vertex: " « id « std::endl;
00114     std::cout « "Adjacent to: ";
00115     for (const Edge *e: adj) {
00116         std::cout « e->getDest()->getId() « " ";
00117     }
00118     std::cout « std::endl;
00119     std::cout « "Visited: " « visited « std::endl;
00120     std::cout « "Indegree: " « indegree « std::endl;
00121     std::cout « "Distance: " « dist « std::endl;
00122     std::cout « "Path: " « path « std::endl;
00123 }
```

4.5.3.12 removeEdge()

```
bool Vertex::removeEdge (
            std::string destID )
```

Definition at line 25 of file VertexEdge.cpp.

```
00025                                           {
00026     bool removedEdge = false;
00027     auto it = adj.begin();
00028     while (it != adj.end()) {
00029         Edge *edge = *it;
00030         Vertex *dest = edge->getDest();
00031         if (dest->getId() == destID) {
00032             it = adj.erase(it);
00033             // Also remove the corresponding edge from the incoming list
00034             auto it2 = dest->incoming.begin();
00035             while (it2 != dest->incoming.end()) {
00036                 if ((*it2)->getOrig()->getId() == id) {
00037                     it2 = dest->incoming.erase(it2);
00038                 } else {
00039                     it2++;
00040                 }
00041             }
00042             delete edge;
00043             removedEdge = true; // allows for multiple edges to connect the same pair of vertices
      (multigraph)
00044         } else {
00045             it++;
00046         }
00047     }
00048     return removedEdge;
00049 }
```

4.5.3.13 setDist()

```
void Vertex::setDist (
            double dist )
```

Definition at line 103 of file VertexEdge.cpp.

```
00103                                       {
00104     this->dist = dist;
00105 }
```

4.5.3.14 setId()

```
void Vertex::setId (
            int info )
```

Definition at line 87 of file VertexEdge.cpp.

```
00087                                 {
00088     this->id = id;
00089 }
```

4.5.3.15 setIndegree()

```
void Vertex::setIndegree (
            unsigned int indegree )
```

Definition at line 99 of file VertexEdge.cpp.

```
00099                                                 {
00100     this->indegree = indegree;
00101 }
```

4.5.3.16 setPath()

```
void Vertex::setPath (
            Edge * path )
```

Definition at line 107 of file VertexEdge.cpp.

```
00107                                {
00108     this->path = path;
00109 }
```

4.5.3.17 setProcesssing()

```
void Vertex::setProcesssing (
            bool processing )
```

Definition at line 95 of file VertexEdge.cpp.

```
00095                                        {
00096     this->processing = processing;
00097 }
```

4.5.3.18 setVisited()

```
void Vertex::setVisited (
            bool visited )
```

Definition at line 91 of file VertexEdge.cpp.

```
00091                                    {
00092     this->visited = visited;
00093 }
```

4.5.4 Member Data Documentation

4.5.4.1 adj

```
std::vector<Edge *> Vertex::adj  [protected]
```

Definition at line 60 of file VertexEdge.h.

4.5.4.2 dist

```
double Vertex::dist = 0  [protected]
```

Definition at line 66 of file VertexEdge.h.

4.5.4.3 id

```
std::string Vertex::id  [protected]
```

Definition at line 59 of file VertexEdge.h.

4.5.4.4 incoming

```
std::vector<Edge *> Vertex::incoming  [protected]
```

Definition at line 69 of file VertexEdge.h.

4.5.4.5 indegree

```
unsigned int Vertex::indegree  [protected]
```

Definition at line 65 of file VertexEdge.h.

4.5.4.6 path

```
Edge* Vertex::path = nullptr  [protected]
```

Definition at line 67 of file VertexEdge.h.

4.5.4.7 processing

```
bool Vertex::processing = false  [protected]
```

Definition at line 64 of file VertexEdge.h.

4.5.4.8 queueIndex

```
int Vertex::queueIndex = 0  [protected]
```

Definition at line 71 of file VertexEdge.h.

4.5.4.9 visited

```
bool Vertex::visited = false  [protected]
```

Definition at line 63 of file VertexEdge.h.

The documentation for this class was generated from the following files:

- VertexEdge.h
- VertexEdge.cpp

Chapter 5

File Documentation

5.1 CPheadquarters.cpp

```
00001 //
00002 // Created by Pedro on 23/03/2023.
00003 //
00004
00005 #include <fstream>
00006 #include <sstream>
00007 #include "CPheadquarters.h"
00008 #include <chrono>
00009 #include <set>
00010
00011 using namespace std;
00012
00013 void CPheadquarters::read_network(string path){
00014     std::ifstream inputFile1(path);
00015     string line1;
00016     std::getline(inputFile1, line1); // ignore first line
00017     while (getline(inputFile1, line1, '\n')) {
00018
00019         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00020             line1.pop_back(); // Remove the '\r' character
00021         }
00022
00023         string station_A;
00024         string station_B;
00025         string temp;
00026         int capacity;
00027         string service;
00028
00029         stringstream inputString(line1);
00030
00031         getline(inputString, station_A, ',');
00032         getline(inputString, station_B, ',');
00033         getline(inputString, temp, ',');
00034         getline(inputString, service, ',');
00035
00036         capacity = stoi(temp);
00037         lines.addVertex(station_A);
00038         lines.addVertex(station_B);
00039
00040         lines.addEdge(station_A, station_B, capacity, service);
00041     }
00042 }
00043
00044 void CPheadquarters::read_stations(string path){
00045     std::ifstream inputFile2(R"(../stations.csv)");
00046     string line2;
00047     std::getline(inputFile2, line2); // ignore first line
00048
00049     while (getline(inputFile2, line2, '\n')) {
00050
00051         if (!line2.empty() && line2.back() == '\r') { // Check if the last character is '\r'
00052             line2.pop_back(); // Remove the '\r' character
00053         }
00054
00055         string nome;
00056         string distrito;
00057         string municipality;
00058         string township;
```

```
00059         string line;
00060
00061         stringstream inputString(line2);
00062
00063         getline(inputString, nome, ',');
00064         getline(inputString, distrito, ',');
00065         getline(inputString, municipality, ',');
00066         getline(inputString, township, ',');
00067         getline(inputString, line, ',');
00068
00069         Station station(nome, distrito, municipality, township, line);
00070         stations[nome] = station;
00071
00072         // print information about the station, to make sure it was imported correctly
00073         //cout << "station: " << nome << " distrito: " << distrito << " municipality: " << municipality << "
      township: " << township << " line: " << line << endl;
00074     }
00075 }
00076
00077 void CPheadquarters::read_files() {
00078
00079     //-------------------------------------------Read
      network.csv-------------------------------------------
00080     std::ifstream inputFile1(R"(../network.csv)");
00081     string line1;
00082     std::getline(inputFile1, line1); // ignore first line
00083     while (getline(inputFile1, line1, '\n')) {
00084
00085         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00086             line1.pop_back(); // Remove the '\r' character
00087         }
00088
00089         string station_A;
00090         string station_B;
00091         string temp;
00092         int capacity;
00093         string service;
00094
00095         stringstream inputString(line1);
00096
00097         getline(inputString, station_A, ',');
00098         getline(inputString, station_B, ',');
00099         getline(inputString, temp, ',');
00100         getline(inputString, service, ',');
00101
00102         capacity = stoi(temp);
00103         lines.addVertex(station_A);
00104         lines.addVertex(station_B);
00105
00106         lines.addEdge(station_A, station_B, capacity, service);
00107     }
00108
00109
00110     //-------------------------------------------Read
      stations.csv-------------------------------------------
00111     std::ifstream inputFile2(R"(../stations.csv)");
00112     string line2;
00113     std::getline(inputFile2, line2); // ignore first line
00114
00115     while (getline(inputFile2, line2, '\n')) {
00116
00117         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00118             line1.pop_back(); // Remove the '\r' character
00119         }
00120
00121         string nome;
00122         string distrito;
00123         string municipality;
00124         string township;
00125         string line;
00126
00127         stringstream inputString(line2);
00128
00129         getline(inputString, nome, ',');
00130         getline(inputString, distrito, ',');
00131         getline(inputString, municipality, ',');
00132         getline(inputString, township, ',');
00133         getline(inputString, line, ',');
00134
00135         Station station(nome, distrito, municipality, township, line);
00136         stations[nome] = station;
00137
00138         // print information about the station, to make sure it was imported correctly
00139         //cout << "station: " << nome << " distrito: " << distrito << " municipality: " << municipality << "
      township: " << township << " line: " << line << endl;
00140     }
00141 }
```

```
00142
00143
00144 Graph CPheadquarters::getLines() const {
00145     return this->lines;
00146 }
00147
00148
00149 int CPheadquarters::T2_1maxflow(string stationA, string stationB) {
00150     Vertex *source = lines.findVertex(stationA); // set source vertex
00151     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00152
00153     // Check if these stations even exist
00154     if (source == nullptr || sink == nullptr) {
00155         std::cerr « "Source or sink vertex not found." « std::endl;
00156         return 0;
00157     }
00158     int maxFlow = lines.edmondsKarp(stationA, stationB);
00159
00160     if (maxFlow == 0) {
00161         cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " «
      stationA
00162              « endl;
00163     } else {
00164         cout « "maxFlow:\t" « maxFlow « endl;
00165     }
00166
00167     return maxFlow;
00168 }
00169
00170
00171 int CPheadquarters::T2_2maxflowAllStations() {
00172     vector<string> stations;
00173     int maxFlow = 0;
00174     auto length = lines.getVertexSet().size();
00175     // Start the timer
00176     auto start_time = std::chrono::high_resolution_clock::now();
00177     cout « "Calculating max flow for all pairs of stations..." « endl;
00178     cout « "Please stand by..." « endl;
00179     for (int i = 0; i < length; ++i) {
00180         for (int j = i + 1; j < length; ++j) {
00181             string stationA = lines.getVertexSet()[i]->getId();
00182             string stationB = lines.getVertexSet()[j]->getId();
00183             int flow = lines.edmondsKarp(stationA, stationB);
00184             if (flow == maxFlow) {
00185                 stations.push_back(stationB);
00186                 stations.push_back(stationA);
00187             } else if (flow > maxFlow) {
00188                 stations.clear();
00189                 stations.push_back(stationB);
00190                 stations.push_back(stationA);
00191                 maxFlow = flow;
00192             }
00193         }
00194     }
00195     // End the timer
00196     auto end_time = std::chrono::high_resolution_clock::now();
00197
00198     // Compute the duration
00199     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00200
00201     // Print the duration
00202     std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
00203
00204     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00205     for (int i = 0; i < stations.size(); i = i + 2) {
00206         cout « "------------------------\n";
00207         cout « "Source: " « stations[i + 1] « '\n';
00208         cout « "Target: " « stations[i] « '\n';
00209         cout « "------------------------\n";
00210     }
00211     return maxFlow;
00212 }
00213
00214
00215 void CPheadquarters::T2_3municipality() {
00216     vector<pair<string , int» top_k;
00217     set<string> sett;
00218     for (auto m : stations) {
00219         sett.insert(m.second.get_district());
00220     }
00221     for (auto m : sett) {
00222         vector<string> desired_stations;
00223         for (auto p: stations) {
00224             if (p.second.get_municipality() == m) {
00225                 desired_stations.push_back(p.second.get_name());
00226             }
00227         }
```

```
00228
00229
00230         vector<string> souces = lines.find_sources(desired_stations);
00231         vector<string> targets = lines.find_targets(desired_stations);
00232         int diff=lines.mul_edmondsKarp(souces, targets);
00233         auto p = pair(m, diff);
00234         top_k.push_back(p);
00235     }
00236     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00237         return left.second > right.second;
00238     });
00239     for (int i = 0; i < 10; i++) {
00240         cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
00241     }
00242 }
00243
00244 void CPheadquarters::T2_3district() {
00245     vector<pair<string , int» top_k;
00246     set<string> sett;
00247     for (auto m : stations) {
00248         sett.insert(m.second.get_district());
00249     }
00250     for (auto m : sett) {
00251         vector<string> desired_stations;
00252         for (auto p: stations) {
00253             if (p.second.get_district() == m) {
00254                 desired_stations.push_back(p.second.get_name());
00255             }
00256         }
00257         vector<string> souces = lines.find_sources(desired_stations);
00258         vector<string> targets = lines.find_targets(desired_stations);
00259         int diff=lines.mul_edmondsKarp(souces, targets);
00260         auto p = pair(m, diff);
00261         top_k.push_back(p);
00262     }
00263     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00264         return left.second > right.second;
00265     });
00266     for (int i = 0; i < 10; i++) {
00267         cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
00268     }
00269 }
00270
00271
00272 int CPheadquarters::T2_4maxArrive(string destination) {
00273     int maxFlow = 0;
00274     vector<string> org = lines.getSources();
00275     vector<string> targ = lines.getTargets();
00276     auto v = lines.findVertex(destination);
00277     lines.mul_edmondsKarp(org,targ);
00278     for(auto e : v->getIncoming()){
00279         maxFlow+=e->getFlow();
00280     }
00281     cout « "Max Flow: " « maxFlow « '\n';
00282     return maxFlow;
00283 }
00284
00285
00286
00287 int CPheadquarters::T3_1MinCost(string source, string destination) {
00288     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00289     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00290     if (sourceVertex == nullptr || destVertex == nullptr) {
00291         cerr « "Source or destination vertex not found. Try again" « endl;
00292         return 1;
00293     }
00294
00295     Graph graph = lines;
00296
00297     std::vector<Vertex *> path;
00298     std::vector<std::vector<Vertex *» allPaths;
00299
00300
00301     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00302
00303     vector<int> maxFlows;
00304     vector<int> totalCosts;
00305
00306     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00307     for (auto path: allPaths) {
00308         int minWeight = 10;
00309         int totalCost = 0; // total cost of this path
00310         for (int i = 0; i + 1 < path.size(); i++) {
00311             std::cout « path[i]->getId() « " -> ";
00312             Edge *e = graph.findEdge(path[i], path[i + 1]);
00313             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00314             if (e->getWeight() < minWeight) {
```

```
00315                 minWeight = e->getWeight();
00316             }
00317
00318             // according to the problem's specification, the cost of STANDARD service is 2 euros and
      ALFA PENDULAR is 4
00319             if (e->getService() == "STANDARD") {
00320                 totalCost += 2;
00321             } else if (e->getService() == "ALFA PENDULAR") {
00322                 totalCost += 4;
00323             }
00324         }
00325         maxFlows.push_back(minWeight);
00326         totalCosts.push_back(totalCost);
00327         cout « " -> " « path[path.size() - 1]->getId() « endl;
00328         cout « "Max flow for this path: " « minWeight « " trains. ";
00329         cout « "Total cost: " « totalCost « " euros." « endl;
00330         std::cout « std::endl;
00331     }
00332
00333     // find the path with the minimum cost per train
00334     int maxTrains = 0;
00335     int resCost;
00336     double max_value = 10000;
00337     for (int i = 0; i < maxFlows.size(); ++i) {
00338         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00339         if (costPerTrain < max_value) {
00340             max_value = costPerTrain;
00341             maxTrains = maxFlows[i];
00342             resCost = totalCosts[i];
00343         }
00344     }
00345
00346     cout « "Max number of trains that can travel between " « source « " and " « destination
00347          « " with minimum cost"
00348          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00349     return maxTrains;
00350 }
00351
00352
00353 int CPheadquarters::T4_1ReducedConectivity(std::vector<std::string> unwantedEdges, std::string s,
      std::string t) {
00354     Graph graph;
00355     std::ifstream inputFile1(R"(../network.csv)");
00356     string line1;
00357     std::getline(inputFile1, line1); // ignore first line
00358     while (getline(inputFile1, line1, '\n')) {
00359
00360         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00361             line1.pop_back(); // Remove the '\r' character
00362         }
00363
00364         string station_A;
00365         string station_B;
00366         string temp;
00367         int capacity;
00368         string service;
00369         bool flag=true;
00370
00371         stringstream inputString(line1);
00372
00373         getline(inputString, station_A, ',');
00374         getline(inputString, station_B, ',');
00375         getline(inputString, temp, ',');
00376         getline(inputString, service, ',');
00377
00378         capacity = stoi(temp);
00379         graph.addVertex(station_A);
00380         graph.addVertex(station_B);
00381
00382         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00383                 if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00384                     flag=false;
00385                 }
00386         }
00387         if (flag) {
00388             graph.addEdge(station_A, station_B, capacity, service);
00389         }
00390         line1 = "";
00391     }
00392
00393     Vertex *source = graph.findVertex(s); // set source vertex
00394     Vertex *sink = graph.findVertex(t); // set sink vertex
00395
00396     // Check if these stations even exist
00397     if (source == nullptr || sink == nullptr) {
00398         std::cerr « "Source or sink vertex not found." « std::endl;
00399         return 1;
```

```
00400     }
00401     int maxFlow = graph.edmondsKarp(s, t);
00402
00403     if (maxFlow == 0) {
00404         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00405              « endl;
00406     }
00407     cout « "maxFlow:\t" « maxFlow « endl;
00408
00409
00410     return 1;
00411 }
00412
00413
00414 int CPheadquarters::T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges) {
00415     Graph graph;
00416     std::ifstream inputFile1(R"(../network.csv)");
00417     string line1;
00418     std::getline(inputFile1, line1); // ignore first line
00419     while (getline(inputFile1, line1, '\n')) {
00420
00421         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00422             line1.pop_back(); // Remove the '\r' character
00423         }
00424
00425         string station_A;
00426         string station_B;
00427         string temp;
00428         int capacity;
00429         string service;
00430         bool flag=true;
00431
00432         stringstream inputString(line1);
00433
00434         getline(inputString, station_A, ',');
00435         getline(inputString, station_B, ',');
00436         getline(inputString, temp, ',');
00437         getline(inputString, service, ',');
00438
00439         capacity = stoi(temp);
00440         graph.addVertex(station_A);
00441         graph.addVertex(station_B);
00442
00443         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00444             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00445                 flag = false;
00446                 break;
00447             }
00448         }
00449         if (flag) {
00450             graph.addEdge(station_A, station_B, capacity, service);
00451         }
00452         line1 = "";
00453     }
00454     vector<string> org = lines.getSources();
00455     vector<string> targ = lines.getTargets();
00456
00457     lines.mul_edmondsKarp(org,targ);
00458     graph.mul_edmondsKarp(org,targ);
00459     vector<pair<int, int» top_k;
00460
00461     auto length = lines.getVertexSet().size();
00462     for (int i = 0; i < length; ++i) {
00463         string destination = lines.getVertexSet()[i]->getId();
00464         auto v1 = lines.findVertex(destination);
00465         auto v2 = graph.findVertex(destination);
00466         int maxFlow1 = 0;
00467         int maxFlow2 = 0;
00468         for(auto e : v1->getIncoming()){
00469             maxFlow1+=e->getFlow();
00470         }
00471         for(auto e : v2->getIncoming()){
00472             maxFlow2+=e->getFlow();
00473         }
00474         int diff = maxFlow1 - maxFlow2;
00475         auto p = pair(i, diff);
00476         top_k.push_back(p);
00477     }
00478     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00479         return left.second > right.second;
00480     });
00481     for (int i = 0; i < 10; i++) {
00482         cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second
      « '\n';
00483     }
00484     return 1;
00485 }
```

5.2 CPheadquarters.h

```
00001 //
00002 // Created by Pedro on 23/03/2023.
00003 //
00004
00005 #ifndef DAPROJECT_CPHEADQUARTERS_H
00006 #define DAPROJECT_CPHEADQUARTERS_H
00007
00008
00009 #include "Graph.h"
00010 #include "Station.h"
00011
00012 using namespace std;
00013
00014 class CPheadquarters {
00015     Graph lines;
00016     unordered_map<string, Station> stations;
00017 public:
00018
00023     void read_network(string path);
00024
00029     void read_stations(string path);
00030
00034     void read_files();
00035
00040     Graph getLines() const;
00041
00050     int T2_1maxflow(string station_A, string station_B);
00051
00060     int T2_2maxflowAllStations();
00061
00067     void T2_3municipality();
00068
00074     void T2_3district();
00075
00083     int T2_4maxArrive(string destination);
00084
00095     int T3_1MinCost(string source, string destination);
00096
00108     int T4_1ReducedConectivity(vector<string> unwantedEdges, string s, string t);
00109
00116     int T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges);
00117
00118 };
00119
00120
00121 #endif //DAPROJECT_CPHEADQUARTERS_H
```

5.3 Graph.cpp

```
00001 // By: Gonçalo Leão
00002
00003 #include <climits>
00004 #include <queue>
00005 #include "Graph.h"
00006
00007 int Graph::getNumVertex() const {
00008     return vertexSet.size();
00009 }
00010
00011 std::vector<Vertex *> Graph::getVertexSet() const {
00012     return vertexSet;
00013 }
00014
00015
00016 Vertex *Graph::findVertex(const std::string &id) const {
00017     for (auto v: vertexSet) {
00018         if (v->getId() == id)
00019             return v;
00020     }
00021     return nullptr;
00022 }
00023
00024
00025
00026 bool Graph::addVertex(const std::string &id) {
00027     if (findVertex(id) != nullptr)
00028         return false;
00029     vertexSet.push_back(new Vertex(id));
00030     return true;
00031 }
00032
```

```
00033
00034 bool Graph::addEdge(const std::string &sourc, const std::string &dest, int w, const std::string
      &service) {
00035     auto v1 = findVertex(sourc);
00036     auto v2 = findVertex(dest);
00037     if (v1 == nullptr || v2 == nullptr)
00038         return false;
00039     v1->addEdge(v2, w, service);
00040
00041     return true;
00042 }
00043
00044
00045 void deleteMatrix(int **m, int n) {
00046     if (m != nullptr) {
00047         for (int i = 0; i < n; i++)
00048             if (m[i] != nullptr)
00049                 delete[] m[i];
00050         delete[] m;
00051     }
00052 }
00053
00054 void deleteMatrix(double **m, int n) {
00055     if (m != nullptr) {
00056         for (int i = 0; i < n; i++)
00057             if (m[i] != nullptr)
00058                 delete[] m[i];
00059         delete[] m;
00060     }
00061 }
00062
00063 Graph::~Graph() {
00064     deleteMatrix(distMatrix, vertexSet.size());
00065     deleteMatrix(pathMatrix, vertexSet.size());
00066 }
00067
00068
00069
00070 void Graph::print() const {
00071     std::cout « "---------------- Graph----------------\n";
00072     std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
00073     std::cout « "Vertices:\n";
00074     for (const auto &vertex: vertexSet) {
00075         std::cout « vertex->getId() « " ";
00076     }
00077     std::cout « "\nEdges:\n";
00078     for (const auto &vertex: vertexSet) {
00079         for (const auto &edge: vertex->getAdj()) {
00080             std::cout « vertex->getId() « " -> " « edge->getDest()->getId() « " (weight: " «
      edge->getWeight() « ", service: " « edge->getService() « ")" « std::endl;
00081         }
00082     }
00083 }
00084
00085 // ---------------------------------- Edmonds-Karp ---------------------------------
00086
00087
00088 void Graph::testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual) {
00089     if (!w->isVisited() && residual > 0) {
00090         w->setVisited(true);
00091         w->setPath(e);
00092         q.push(w);
00093     }
00094 }
00095
00096
00097
00098 bool Graph::findAugmentingPath(const std::string &s, const std::string &t) {
00099     Vertex *source = findVertex(s);
00100     Vertex *target = findVertex(t);
00101     if (source == nullptr || target == nullptr) {
00102         return false;
00103     }
00104     for (auto v: vertexSet) {
00105         v->setVisited(false);
00106         v->setPath(nullptr);
00107     }
00108     source->setVisited(true);
00109     std::queue<Vertex *> q;
00110     q.push(source);
00111     while (!q.empty()) {
00112         auto v = q.front();
00113         q.pop();
00114         for (auto e: v->getAdj()) {
00115             auto w = e->getDest();
00116             double residual = e->getWeight() - e->getFlow();
00117             testAndVisit(q, e, w, residual);
```

```
00118         }
00119         for (auto e: v->getIncoming()) {
00120             auto w = e->getDest();
00121             double residual = e->getFlow();
00122             testAndVisit(q, e->getReverse(), w, residual);
00123         }
00124         if (target->isVisited()) {
00125             return true;
00126         }
00127     }
00128     return false;
00129 }
00130
00131
00132 int Graph::findMinResidual(Vertex *s, Vertex *t) {
00133     double minResidual = INT_MAX;
00134     for (auto v = t; v != s;) {
00135         auto e = v->getPath();
00136         if (e->getDest() == v) {
00137             minResidual = std::min(minResidual, e->getWeight() - e->getFlow());
00138             v = e->getOrig();
00139         } else {
00140             minResidual = std::min(minResidual, e->getFlow());
00141             v = e->getDest();
00142         }
00143     }
00144     return minResidual;
00145 }
00146
00147
00148 void Graph::updateFlow(Vertex *s, Vertex *t, int bottleneck) {
00149     for (auto v = t; v != s;) {
00150         auto e = v->getPath();
00151         double flow = e->getFlow();
00152         if (e->getDest() == v) {
00153             e->setFlow(flow + bottleneck);
00154             v = e->getOrig();
00155         } else {
00156             e->setFlow(flow - bottleneck);
00157             v = e->getDest();
00158         }
00159     }
00160 }
00161
00162
00163 int Graph::edmondsKarp(const std::string &s, const std::string &t) {
00164     for (auto e: vertexSet) {
00165         for (auto i: e->getAdj()) {
00166             i->setFlow(0);
00167         }
00168     }
00169     int maxFlow = 0;
00170     while (findAugmentingPath(s, t)) {
00171         int bottleneck = findMinResidual(findVertex(s), findVertex(t));
00172         updateFlow(findVertex(s), findVertex(t), bottleneck);
00173         maxFlow += bottleneck;
00174     }
00175     return maxFlow;
00176 }
00177
00178 std::vector<std::string> Graph::find_sources(std::vector<std::string> desired_stations) {
00179     std::vector<std::string> res;
00180
00181     for (std::string s: desired_stations) {
00182         bool flag = true;
00183         auto v = findVertex(s);
00184         if (v == nullptr) {
00185             std::cout « "Trouble finding source " « s « '\n';
00186             continue;
00187         }
00188         for (auto e: v->getIncoming()) {
00189             if (isIn(e->getOrig()->getId(), desired_stations)) {
00190                 flag=false;
00191             }
00192         }
00193         if (flag) res.push_back(s);
00194     }
00195     return res;
00196 }
00197
00198 std::vector<std::string> Graph::find_targets(std::vector<std::string> desired_stations) {
00199     std::vector<std::string> res;
00200     for (std::string s: desired_stations) {
00201         bool flag = true;
00202         auto v = findVertex(s);
00203         if (v == nullptr) {
00204             std::cout « "Trouble finding target " « s « '\n';
```

```
00205             continue;
00206         }
00207         for (auto e: v->getAdj()) {
00208             if (isIn(e->getDest()->getId(), desired_stations)) {
00209                 flag=false;
00210             }
00211         }
00212         if (flag) res.push_back(s);
00213     }
00214     return res;
00215 }
00216
00217
00218 bool Graph::isIn(std::string n, std::vector<std::string> vec) {
00219     for (std::string s: vec) {
00220         if (s == n) return true;
00221     }
00222     return false;
00223 }
00224
00225
00226 int Graph::mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets) {
00227     auto it1 = souces.begin();
00228     while (it1 != souces.end()) {
00229         if (isIn(*it1, targets)) {
00230             it1 = souces.erase(it1);
00231         } else it1++;
00232     }
00233
00234     auto it2 = targets.begin();
00235     while (it2 != targets.end()) {
00236         if (isIn(*it2, souces)) {
00237             it2 = souces.erase(it2);
00238         } else it2++;
00239     }
00240
00241     addVertex("temp_source");
00242     for (std::string s: souces) {
00243         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00244     }
00245
00246     addVertex("temp_targets");
00247     for (std::string s: targets) {
00248         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00249     }
00250     for (auto e: vertexSet) {
00251         for (auto i: e->getAdj()) {
00252             i->setFlow(0);
00253         }
00254     }
00255     int maxFlow = 0;
00256     while (findAugmentingPath("temp_source", "temp_targets")) {
00257         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00258         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00259         maxFlow += bottleneck;
00260     }
00261     deleteVertex("temp_targets");
00262     deleteVertex("temp_source");
00263     return maxFlow;
00264 }
00265
00266 // ---------------------------------- Find ALL existing augmenting paths
    ----------------------------------
00267
00268
00269 void Graph::findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00270                          std::vector<std::vector<Vertex *>> &allPaths) {
00271     path.push_back(source);
00272     source->setVisited(true);
00273
00274     if (source == destination) {
00275         allPaths.push_back(path);
00276     } else {
00277         for (auto edge: source->getAdj()) {
00278             Vertex *adjacent = edge->getDest();
00279             if (!adjacent->isVisited()) {
00280                 findAllPaths(adjacent, destination, path, allPaths);
00281             }
00282         }
00283     }
00284
00285     path.pop_back();
00286     source->setVisited(false);
00287 }
00288
00289
00290
```

```
00291 Edge *Graph::findEdge(Vertex *source, Vertex *destination) {
00292
00293     for (auto edge: source->getAdj()) {
00294         if (edge->getDest() == destination) {
00295             return edge;
00296         }
00297     }
00298     return nullptr;
00299 }
00300
00301
00302 std::vector<std::string> Graph::getSources() {
00303     std::vector<std::string> res;
00304     for (auto v : vertexSet) {
00305         if(v->getIncoming().empty()){
00306             res.push_back(v->getId());
00307         }
00308     }
00309     return res;
00310 }
00311
00312 std::vector<std::string> Graph::getTargets() {
00313     std::vector<std::string> res;
00314     for (auto v : vertexSet) {
00315         if(v->getAdj().empty()){
00316             res.push_back(v->getId());
00317         }
00318     }
00319     return res;
00320 }
00321
00322
00323 void Graph::deleteVertex(std::string name) {
00324     auto v = findVertex(name);
00325     for(auto e : v->getAdj()){
00326         auto s = e->getDest()->getId();
00327         v->removeEdge(s);
00328     }
00329     for(auto e : v->getIncoming()){
00330         e->getOrig()->removeEdge(name);
00331     }
00332     auto it = vertexSet.begin();
00333     while (it!=vertexSet.end()){
00334         Vertex* currentVertex = *it;
00335         if(currentVertex->getId()==name){
00336             it=vertexSet.erase(it);
00337         }
00338         else{
00339             it++;
00340         }
00341     }
00342 }
00343
```

5.4 Graph.h

```
00001 // By: Gonçalo Leão
00002
00003 #ifndef DA_TP_CLASSES_GRAPH
00004 #define DA_TP_CLASSES_GRAPH
00005
00006 #include <iostream>
00007 #include <vector>
00008 #include <queue>
00009 #include <limits>
00010 #include <algorithm>
00011
00012
00013 #include "VertexEdge.h"
00014
00015 class Graph {
00016 public:
00017     ~Graph();
00018
00024     Vertex *findVertex(const std::string &id) const;
00025
00031     bool addVertex(const std::string &id);
00032
00042     bool addEdge(const std::string &sourc, const std::string &dest, int w, const std::string
      &service);
00043
00044
00045     [[nodiscard]] int getNumVertex() const;
```

```
00046
00047     [[nodiscard]] std::vector<Vertex *> getVertexSet() const;
00048
00052     void print() const;
00053
00063     int edmondsKarp(const std::string &s, const std::string &t);
00064
00069     std::vector<std::string> getSources();
00070
00075     std::vector<std::string> getTargets();
00076
00083     int mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets);
00084
00090     std::vector<std::string> find_sources(std::vector<std::string> desired_stations);
00091
00097     std::vector<std::string> find_targets(std::vector<std::string> desired_stations);
00098
00107     void findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00108                       std::vector<std::vector<Vertex *» &allPaths);
00109
00116     Edge *findEdge(Vertex *source, Vertex *destination);
00117
00118 protected:
00119     std::vector<Vertex *> vertexSet;    // vertex set
00120
00121     double **distMatrix = nullptr;   // dist matrix for Floyd-Warshall
00122     int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall
00123
00124
00132     void updateFlow(Vertex *s, Vertex *t, int bottleneck);
00133
00140     int findMinResidual(Vertex *s, Vertex *t);
00141
00151     bool findAugmentingPath(const std::string &s, const std::string &t);
00152
00160     void testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual);
00161
00162     bool isIn(std::string n, std::vector<std::string> vec);
00163
00168     void deleteVertex(std::string name);
00169 };
00170
00171 void deleteMatrix(int **m, int n);
00172
00173 void deleteMatrix(double **m, int n);
00174
00175 #endif /* DA_TP_CLASSES_GRAPH */
```

5.5 main.cpp

```
00001 #include <iostream>
00002 #include "CPheadquarters.h"
00003
00004 using namespace std;
00005
00006 int main() {
00007     CPheadquarters CP;
00008     string path;
00009     cout«"Insert path to file to consrtuct graph: ";
00010     getline(cin, path);
00011     CP.read_network(path);
00012     cout«"Insert path to file regarding stations: ";
00013     getline(cin, path);
00014     cout«endl;
00015     CP.read_stations(path);
00016     CP.getLines().print();
00017     int n;
00018     cout « "\n-------------- An Analysis Tool for Railway Network Management --------------\n" « endl;
00019     do {
00020         cout « "1 - T2.1 Max number of trains between stations\n";
00021         cout « "2 - T2.2 Stations that require the Max num of trains among all pairs of stations\n";
00022         cout « "3 - T2.3 Indicate where management should assign larger budgets for the purchasing and
      maintenance of trains\n";
00023         cout « "4 - T2.4 Max number of trains that can simultaneously arrive at a given station\n";
00024         cout « "5 - T3.1 Max number of trains that can simultaneously travel with minimum cost\n";
00025         cout « "6 - T4.1 Max number of trains between stations in a network of reduced
      connectivity\n";
00026         cout « "7 - T4.2 Top-10 most affected stations in a network of reduced connectivity\n";
00027         cout « "8 - Exit\n";
00028
00029
00030         bool validInput = false;
00031
```

```
00032         while (!validInput) {
00033             cout « "Insert your option:\n";
00034             cin » n;
00035 
00036             if (cin.fail() || n < 1 || n > 8) {
00037                 cin.clear();
00038                 cin.ignore(numeric_limits<streamsize>::max(), '\n');
00039                 cout « "Invalid input. Please enter a number between 1 and 8." « endl;
00040             } else {
00041                 validInput = true;
00042             }
00043         }
00044 
00045         switch (n) {
00046             case 1: {
00047                 cin.ignore(); // ignore newline character left in the input stream
00048                 string a, b;
00049                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00050                 cout « "Enter station A: ";
00051                 getline(cin, a);
00052 
00053                 cout « "Enter station B: ";
00054                 getline(cin, b);
00055 
00056                 if (a.empty() || b.empty()) {
00057                     cerr « "Error: Station names cannot be empty." « endl;
00058                     break;
00059                 }
00060 
00061                 // call function to calculate max flow between stations A and B
00062                 CP.T2_1maxflow(a, b);
00063                 break;
00064             }
00065 
00066             case 2: {
00067                 CP.T2_2maxflowAllStations();
00068                 break;
00069             }
00070 
00071             case 3: {
00072                 cin.ignore();
00073 
00074                 int c;
00075                 cout « "Type 1 for Top-10 districts regarding flow" « '\n';
00076                 cout « "Type 2 for Top-10 municipalities regarding flow" « '\n';
00077                 cin » c;
00078                 switch (c) {
00079                     case 1:
00080                         CP.T2_3district();
00081                         break;
00082                     case 2:
00083                         CP.T2_3municipality();
00084                         break;
00085                     default:
00086                         cout « "Invalid input";
00087                         break;
00088                 }
00089                 cout « endl;
00090                 break;
00091             }
00092 
00093             case 4: {
00094                 cin.ignore();
00095                 string destination;
00096                 cout « "Enter destination: ";
00097                 getline(cin, destination);
00098                 CP.T2_4maxArrive(destination);
00099                 break;
00100             }
00101 
00102             case 5: {
00103                 cin.ignore();
00104                 string a, b;
00105                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00106                 cout « "Enter station A: ";
00107                 getline(cin, a);
00108                 cout « endl;
00109                 cout « "Enter station B: ";
00110                 getline(cin, b);
00111 
00112                 if (a.empty() || b.empty()) {
00113                     cerr « "Error: Station names cannot be empty." « endl;
00114                     break;
00115                 }
00116 
00117                 CP.T3_1MinCost(a, b);
00118                 break;
```

```
00119             }
00120
00121             case 6: {
00122                 cin.ignore();
00123                 vector<string> unwantedEdges;
00124                 string edgesource;
00125                 string edgetarget;
00126                 string b;
00127                 string a;
00128                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00129                 cout « "Enter station A: ";
00130                 getline(cin, a);
00131                 cout « "Enter station B: ";
00132                 getline(cin, b);
00133                 cout « '\n';
00134                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
      Type '.' to end listing: \n";
00135                 cout « R"(Example: "Bustelo" "Meinedo" would delete the edge "Bustelo->Meinedo")" «
      endl;
00136                 while (1){
00137                     cout « "Enter edge source or '.' to finish: ";
00138                     getline(cin, edgesource);
00139                     if(edgesource==".") break;
00140                     unwantedEdges.push_back(edgesource);
00141                     cout « "Enter edge target: ";
00142                     getline(cin, edgetarget);
00143                     unwantedEdges.push_back(edgetarget);
00144                 }
00145                 CP.T4_1ReducedConectivity(unwantedEdges,a,b);
00146                 break;
00147             }
00148
00149             case 7: {
00150                 cin.ignore();
00151                 vector<string> unwantedEdges;
00152                 string edgesource;
00153                 string edgetarget;
00154                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
      Type '.' to end listing: \n";
00155                 cout « R"(Example: "Bustelo" "Meinedo" would delete the edge "Bustelo->Meinedo")" «
      endl;
00156                 while (1){
00157                     cout « "Enter edge source or '.' to finish: ";
00158                     getline(cin, edgesource);
00159                     if(edgesource==".") break;
00160                     unwantedEdges.push_back(edgesource);
00161                     cout « "Enter edge target: ";
00162                     getline(cin, edgetarget);
00163                     unwantedEdges.push_back(edgetarget);
00164                 }
00165                 CP.T4_2Top_K_ReducedConectivity(unwantedEdges);
00166
00167                 break;
00168             }
00169
00170             case 8: {
00171                 cout « "Exiting program..." « endl;
00172                 break;
00173             }
00174
00175             default: {
00176                 cerr « "Error: Invalid option selected." « endl;
00177                 break;
00178             }
00179         }
00180     } while (n != 8);
00181
00182     return 0;
00183 }
```

5.6 Station.cpp

```
00001 //
00002 // Created by Pedro on 23/03/2023.
00003 //
00004
00005 #include "Station.h"
00006
00007 Station::Station(string name_, string district_, string municipality_, string township_, string line_)
      {
00008     name=name_;
00009     municipality=municipality_;
00010     district=district_;
```

```
00011     township=township_;
00012     line=line_;
00013 }
00014
00015 string Station::get_name() {
00016     return name;
00017 }
00018
00019 string Station::get_district() {
00020     return district;
00021 }
00022
00023 string Station::get_municipality() {
00024     return municipality;
00025 }
00026
00027 string Station::get_township() {
00028     return township;
00029 }
00030
00031 string Station::get_line() {
00032     return line;
00033 }
00034
00035 Station::Station() {
00036
00037 }
```

5.7 Station.h

```
00001 //
00002 // Created by Pedro on 23/03/2023.
00003 //
00004
00005 #ifndef DAPROJECT_STATION_H
00006 #define DAPROJECT_STATION_H
00007
00008 #include <string>
00009
00010 using namespace std;
00011
00012 class Station {
00013     string name;
00014     string district;
00015     string municipality;
00016     string township;
00017     string line;
00018 public:
00022     Station();
00023
00032     Station(string name_, string district_, string municipality_, string township_, string line_);
00033
00038     string get_name();
00039
00044     string get_district();
00045
00050     string get_municipality();
00051
00056     string get_township();
00057
00062     string get_line();
00063 };
00064
00065
00066 #endif //DAPROJECT_STATION_H
```

5.8 VertexEdge.cpp

```
00001 // By: Gonçalo Leão
00002
00003 #include "VertexEdge.h"
00004
00005 /************************* Vertex  *************************/
00006
00007 Vertex::Vertex(std::string id) : id(id) {}
00008
00009 /*
00010  * Auxiliary function to add an outgoing edge to a vertex (this),
00011  * with a given destination vertex (d) and edge weight (w).
```

```
00012  */
00013 Edge *Vertex::addEdge(Vertex *d, int w, const std::string &service) {
00014     auto newEdge = new Edge(this, d, w, service);
00015     adj.push_back(newEdge);
00016     d->incoming.push_back(newEdge);
00017     return newEdge;
00018 }
00019
00020 /*
00021  * Auxiliary function to remove an outgoing edge (with a given destination (d))
00022  * from a vertex (this).
00023  * Returns true if successful, and false if such edge does not exist.
00024  */
00025 bool Vertex::removeEdge(std::string destID) {
00026     bool removedEdge = false;
00027     auto it = adj.begin();
00028     while (it != adj.end()) {
00029         Edge *edge = *it;
00030         Vertex *dest = edge->getDest();
00031         if (dest->getId() == destID) {
00032             it = adj.erase(it);
00033             // Also remove the corresponding edge from the incoming list
00034             auto it2 = dest->incoming.begin();
00035             while (it2 != dest->incoming.end()) {
00036                 if ((*it2)->getOrig()->getId() == id) {
00037                     it2 = dest->incoming.erase(it2);
00038                 } else {
00039                     it2++;
00040                 }
00041             }
00042             delete edge;
00043             removedEdge = true; // allows for multiple edges to connect the same pair of vertices
      (multigraph)
00044         } else {
00045             it++;
00046         }
00047     }
00048     return removedEdge;
00049 }
00050
00051 bool Vertex::operator<(Vertex &vertex) const {
00052     return this->dist < vertex.dist;
00053 }
00054
00055 std::string Vertex::getId() const {
00056     return this->id;
00057 }
00058
00059 std::vector<Edge *> Vertex::getAdj() const {
00060     return this->adj;
00061 }
00062
00063 bool Vertex::isVisited() const {
00064     return this->visited;
00065 }
00066
00067 bool Vertex::isProcessing() const {
00068     return this->processing;
00069 }
00070
00071 unsigned int Vertex::getIndegree() const {
00072     return this->indegree;
00073 }
00074
00075 double Vertex::getDist() const {
00076     return this->dist;
00077 }
00078
00079 Edge *Vertex::getPath() const {
00080     return this->path;
00081 }
00082
00083 std::vector<Edge *> Vertex::getIncoming() const {
00084     return this->incoming;
00085 }
00086
00087 void Vertex::setId(int id) {
00088     this->id = id;
00089 }
00090
00091 void Vertex::setVisited(bool visited) {
00092     this->visited = visited;
00093 }
00094
00095 void Vertex::setProcesssing(bool processing) {
00096     this->processing = processing;
00097 }
```

```
00098
00099 void Vertex::setIndegree(unsigned int indegree) {
00100     this->indegree = indegree;
00101 }
00102
00103 void Vertex::setDist(double dist) {
00104     this->dist = dist;
00105 }
00106
00107 void Vertex::setPath(Edge *path) {
00108     this->path = path;
00109 }
00110
00111
00112 void Vertex::print() const {
00113     std::cout « "Vertex: " « id « std::endl;
00114     std::cout « "Adjacent to: ";
00115     for (const Edge *e: adj) {
00116         std::cout « e->getDest()->getId() « " ";
00117     }
00118     std::cout « std::endl;
00119     std::cout « "Visited: " « visited « std::endl;
00120     std::cout « "Indegree: " « indegree « std::endl;
00121     std::cout « "Distance: " « dist « std::endl;
00122     std::cout « "Path: " « path « std::endl;
00123 }
00124
00125
00126 /********************** Edge  ****************************/
00127
00128 Edge::Edge(Vertex *orig, Vertex *dest, int w, const std::string &service) : orig(orig), dest(dest),
    weight(w),
00129                                                                             service(service), flow(0)
    {}
00130
00131 Vertex *Edge::getDest() const {
00132     return this->dest;
00133 }
00134
00135 int Edge::getWeight() const {
00136     return this->weight;
00137 }
00138
00139 Vertex *Edge::getOrig() const {
00140     return this->orig;
00141 }
00142
00143 Edge *Edge::getReverse() const {
00144     return this->reverse;
00145 }
00146
00147 bool Edge::isSelected() const {
00148     return this->selected;
00149 }
00150
00151 double Edge::getFlow() const {
00152     return flow;
00153 }
00154
00155 void Edge::setSelected(bool selected) {
00156     this->selected = selected;
00157 }
00158
00159 void Edge::setReverse(Edge *reverse) {
00160     this->reverse = reverse;
00161 }
00162
00163 void Edge::setFlow(double flow) {
00164     this->flow = flow;
00165 }
00166
00167 void Edge::setService(const std::string &service) {
00168     this->service = service;
00169 }
00170
00171 std::string Edge::getService() const {
00172     return this->service;
00173 }
```

5.9 VertexEdge.h

```
00001 // By: Gonçalo Leão
00002
```

```
00003 #ifndef DA_TP_CLASSES_VERTEX_EDGE
00004 #define DA_TP_CLASSES_VERTEX_EDGE
00005
00006 #include <iostream>
00007 #include <vector>
00008 #include <queue>
00009 #include <limits>
00010 #include <algorithm>
00011
00012
00013 class Edge;
00014
00015 #define INF std::numeric_limits<double>::max()
00016
00017 /************************* Vertex  *************************/
00018
00019 class Vertex {
00020 public:
00021     Vertex(std::string id);
00022
00023     bool operator<(Vertex &vertex) const; // // required by MutablePriorityQueue
00024
00025     std::string getId() const;
00026
00027     std::vector<Edge *> getAdj() const;
00028
00029     bool isVisited() const;
00030
00031     bool isProcessing() const;
00032
00033     unsigned int getIndegree() const;
00034
00035     double getDist() const;
00036
00037     Edge *getPath() const;
00038
00039     std::vector<Edge *> getIncoming() const;
00040
00041     void setId(int info);
00042
00043     void setVisited(bool visited);
00044
00045     void setProcesssing(bool processing);
00046
00047     void setIndegree(unsigned int indegree);
00048
00049     void setDist(double dist);
00050
00051     void setPath(Edge *path);
00052
00053     Edge *addEdge(Vertex *dest, int w, const std::string &service);
00054
00055     bool removeEdge(std::string destID);
00056
00057
00058 protected:
00059     std::string id;           // identifier
00060     std::vector<Edge *> adj;  // outgoing edges
00061
00062     // auxiliary fields
00063     bool visited = false; // used by DFS, BFS, Prim ...
00064     bool processing = false; // used by isDAG (in addition to the visited attribute)
00065     unsigned int indegree; // used by topsort
00066     double dist = 0;
00067     Edge *path = nullptr;
00068
00069     std::vector<Edge *> incoming; // incoming edges
00070
00071     int queueIndex = 0;        // required by MutablePriorityQueue and UFDS
00072     void print() const;
00073 };
00074
00075
00076 /********************** Edge  ****************************/
00077
00078 class Edge {
00079 public:
00080     Edge(Vertex *orig, Vertex *dest, int w, const std::string &service);
00081
00082     Vertex *getDest() const;
00083
00084     int getWeight() const;
00085
00086     bool isSelected() const;
00087
00088     Vertex *getOrig() const;
00089
```

```
00090     Edge *getReverse() const;
00091
00092     double getFlow() const;
00093
00094     void setSelected(bool selected);
00095
00096     void setReverse(Edge *reverse);
00097
00098     void setFlow(double flow);
00099
00100     [[nodiscard]] std::string getService() const;
00101
00102     void setService(const std::string &service);
00103
00104 protected:
00105     Vertex *dest; // destination vertex
00106     int weight; // edge weight, can also be used for capacity
00107
00108     std::string service;
00109     // auxiliary fields
00110     bool selected = false;
00111
00112     // used for bidirectional edges
00113     Vertex *orig;
00114     Edge *reverse = nullptr;
00115
00116     double flow; // for flow-related problems
00117 };
00118
00119 #endif /* DA_TP_CLASSES_VERTEX_EDGE */
```

Index