# An Analysis Tool for Railway Network Management

1.0

Generated by Doxygen 1.9.7

# Chapter 1

# DAproject

## 1.1 Deadline is April 7, 2023 at midnight

### 1.1.1 Checklist

- [T1.1: 1.0 point] 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] 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] 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] 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] 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] 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] 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] 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_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.*
- int T2_3municipality (string municipality)

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

### 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 79 of file CPheadquarters.cpp.

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

#### 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 12 of file CPheadquarters.cpp.

```
00012                                {
00013
00014     //-------------------------------------------Read
      network.csv-------------------------------------------
00015     std::ifstream inputFile1(R"(../network.csv)");
00016     string line1;
00017     std::getline(inputFile1, line1); // ignore first line
00018     while (getline(inputFile1, line1, '\n')) {
00019
00020         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00021             line1.pop_back(); // Remove the '\r' character
00022         }
00023
00024         string station_A;
00025         string station_B;
00026         string temp;
00027         int capacity;
00028         string service;
00029
00030         stringstream inputString(line1);
00031
00032         getline(inputString, station_A, ',');
00033         getline(inputString, station_B, ',');
00034         getline(inputString, temp, ',');
00035         getline(inputString, service, ',');
00036
00037         capacity = stoi(temp);
00038         lines.addVertex(station_A);
00039         lines.addVertex(station_B);
00040
00041         lines.addEdge(station_A, station_B, capacity, service);
00042     }
00043
00044
00045     //-------------------------------------------Read
      stations.csv-------------------------------------------
00046     std::ifstream inputFile2(R"(../stations.csv)");
00047     string line2;
00048     std::getline(inputFile2, line2); // ignore first line
00049
00050     while (getline(inputFile2, line2, '\n')) {
00051
00052         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00053             line1.pop_back(); // Remove the '\r' character
00054         }
00055
00056         string nome;
00057         string distrito;
```

```
00058         string municipality;
00059         string township;
00060         string line;
00061
00062         stringstream inputString(line2);
00063
00064         getline(inputString, nome, ',');
00065         getline(inputString, distrito, ',');
00066         getline(inputString, municipality, ',');
00067         getline(inputString, township, ',');
00068         getline(inputString, line, ',');
00069
00070         Station station(nome, distrito, municipality, township, line);
00071         stations[nome] = station;
00072
00073         // print information about the station, to make sure it was imported correctly
00074         //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « "
      township: " « township « " line: " « line « endl;
00075     }
00076 }
```

#### 4.1.2.3 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 84 of file CPheadquarters.cpp.

```
00084                                                                   {
00085     Vertex *source = lines.findVertex(stationA); // set source vertex
00086     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00087
00088     // Check if these stations even exist
00089     if (source == nullptr || sink == nullptr) {
00090         std::cerr « "Source or sink vertex not found." « std::endl;
00091         return 1;
00092     }
00093     int maxFlow = lines.edmondsKarp(stationA, stationB);
00094
00095     if (maxFlow == 0) {
00096         cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " «
      stationA
00097              « endl;
00098     } else {
00099         cout « "maxFlow:\t" « maxFlow « endl;
00100     }
00101
00102     return 1;
00103 }
```

#### 4.1.2.4 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 111 of file CPheadquarters.cpp.

```
00111                                                {
00112     vector<string> stations;
00113     int maxFlow = 0;
00114     auto length = lines.getVertexSet().size();
00115     // Start the timer
00116     auto start_time = std::chrono::high_resolution_clock::now();
00117     cout « "Calculating max flow for all pairs of stations..." « endl;
00118     cout « "Please stand by..." « endl;
00119     for (int i = 0; i < length; ++i) {
00120         for (int j = i + 1; j < length; ++j) {
00121             string stationA = lines.getVertexSet()[i]->getId();
00122             string stationB = lines.getVertexSet()[j]->getId();
00123             int flow = lines.edmondsKarp(stationA, stationB);
00124             if (flow == maxFlow) {
00125                 stations.push_back(stationB);
00126                 stations.push_back(stationA);
00127             } else if (flow > maxFlow) {
00128                 stations.clear();
00129                 stations.push_back(stationB);
00130                 stations.push_back(stationA);
00131                 maxFlow = flow;
00132             }
00133         }
00134     }
00135     // End the timer
00136     auto end_time = std::chrono::high_resolution_clock::now();
00137
00138     // Compute the duration
00139     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00140
00141     // Print the duration
00142     std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
00143
00144     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00145     for (int i = 0; i < stations.size(); i = i + 2) {
00146         cout « "------------------------\n";
00147         cout « "Source: " « stations[i + 1] « '\n';
00148         cout « "Target: " « stations[i] « '\n';
00149         cout « "------------------------\n";
00150     }
00151     return 0;
00152 }
```

#### 4.1.2.5 T2_3district()

```
int CPheadquarters::T2_3district (
            string district )
```

Definition at line 167 of file CPheadquarters.cpp.

```
00167                                                {
00168     vector<string> desired_stations;
00169     for (auto p: stations) {
00170         if (p.second.get_district() == district) {
00171             desired_stations.push_back(p.second.get_name());
00172         }
00173     }
00174     return lines.mul_edmondsKarp(lines.find_sources(desired_stations),
      lines.find_targets(desired_stations));
00175 }
```

#### 4.1.2.6 T2_3municipality()

```
int CPheadquarters::T2_3municipality (
            string municipality )
```

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

Reports the top-k municipalities and districts, regarding their transportation needs

**Parameters**

| *municipality* | |
|---|---|

**Returns**

maximum flow in the given municipality

Definition at line 155 of file CPheadquarters.cpp.

```
00155                                                                {
00156     vector<string> desired_stations;
00157     for (auto p: stations) {
00158         if (p.second.get_municipality() == municipality) {
00159             desired_stations.push_back(p.second.get_name());
00160         }
00161     }
00162     vector<string> souces = lines.find_sources(desired_stations);
00163     vector<string> targets = lines.find_targets(desired_stations);
00164     return lines.mul_edmondsKarp(souces, targets);
00165 }
```

#### 4.1.2.7 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 178 of file CPheadquarters.cpp.

```
00178                                                    {
00179     Vertex *dest = lines.findVertex(destination);
00180     int maxFlow = 0;
00181
00182     // iterate over all vertices to find incoming and outgoing vertices
```

```
00183     for (auto &v: lines.getVertexSet()) {
00184         if (v != dest) {
00185 
00186             int flow = lines.edmondsKarp(v->getId(), destination);
00187 
00188             // Update the maximum flow if this vertex contributes to a higher maximum
00189             if (flow > maxFlow) {
00190                 maxFlow = flow;
00191             }
00192         }
00193 
00194     }
00195 
00196     cout « endl;
00197     for (auto &e: dest->getIncoming()) {
00198         cout « e->getOrig()->getId() « " -> " « e->getDest()->getId() « " : " « e->getWeight() « endl;
00199 
00200     }
00201 
00202     cout « "Max number of trains that can simultaneously arrive at " « destination « ": " « maxFlow « endl;
00203     return maxFlow;
00204 
00205 }
```

#### 4.1.2.8 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 209 of file CPheadquarters.cpp.

```
00209                                                                   {
00210     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00211     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00212     if (sourceVertex == nullptr || destVertex == nullptr) {
00213         cerr « "Source or destination vertex not found. Try again" « endl;
00214         return 1;
00215     }
00216 
00217     Graph graph = lines;
00218 
00219     std::vector<Vertex *> path;
00220     std::vector<std::vector<Vertex *» allPaths;
00221 
00222 
00223     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00224 
00225     vector<int> maxFlows;
00226     vector<int> totalCosts;
00227 
00228     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00229     for (auto path: allPaths) {
00230         int minWeight = 10;
00231         int totalCost = 0; // total cost of this path
00232         for (int i = 0; i + 1 < path.size(); i++) {
00233             std::cout « path[i]->getId() « " -> ";
00234             Edge *e = graph.findEdge(path[i], path[i + 1]);
```

```
00235             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00236             if (e->getWeight() < minWeight) {
00237                 minWeight = e->getWeight();
00238             }
00239
00240             // according to the problem's specification, the cost of STANDARD service is 2 euros and
     ALFA PENDULAR is 4
00241             if (e->getService() == "STANDARD") {
00242                 totalCost += 2;
00243             } else if (e->getService() == "ALFA PENDULAR") {
00244                 totalCost += 4;
00245             }
00246         }
00247         maxFlows.push_back(minWeight);
00248         totalCosts.push_back(totalCost);
00249         cout « " -> " « path[path.size() - 1]->getId() « endl;
00250         cout « "Max flow for this path: " « minWeight « " trains. ";
00251         cout « "Total cost: " « totalCost « " euros." « endl;
00252         std::cout « std::endl;
00253     }
00254
00255     // find the path with the minimum cost per train
00256     int maxTrains = 0;
00257     int resCost;
00258     double max_value = 10000;
00259     for (int i = 0; i < maxFlows.size(); ++i) {
00260         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00261         if (costPerTrain < max_value) {
00262             max_value = costPerTrain;
00263             maxTrains = maxFlows[i];
00264             resCost = totalCosts[i];
00265         }
00266     }
00267
00268     cout « "Max number of trains that can travel between " « source « " and " « destination
00269          « " with minimum cost"
00270          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00271     return maxTrains;
00272 }
```

#### 4.1.2.9 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 275 of file CPheadquarters.cpp.

```
00275
    {
00276     Graph graph;
00277     ifstream inputFile1;
00278     inputFile1.open(R"(../network.csv)");
00279     string line1;
00280
00281     getline(inputFile1, line1);
00282     line1 = "";
00283
00284     while (getline(inputFile1, line1)) {
00285         string station_A;
00286         string station_B;
00287         string temp;
00288         int capacity;
00289         string service;
00290         bool flag = true;
00291
00292         stringstream inputString(line1);
00293
00294         getline(inputString, station_A, ';');
00295         getline(inputString, station_B, ';');
00296         getline(inputString, temp, ';');
00297         capacity = stoi(temp);
00298         getline(inputString, service, ';');
00299
00300         graph.addVertex(station_A);
00301         graph.addVertex(station_B);
00302         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00303             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00304                 flag = false;
00305
00306             }
00307         }
00308         if (flag) {
00309             graph.addEdge(station_A, station_B, capacity, service);
00310         }
00311         line1 = "";
00312     }
00313
00314     Vertex *source = graph.findVertex(s); // set source vertex
00315     Vertex *sink = graph.findVertex(t); // set sink vertex
00316
00317     // Check if these stations even exist
00318     if (source == nullptr || sink == nullptr) {
00319         std::cerr « "Source or sink vertex not found." « std::endl;
00320         return 1;
00321     }
00322     int maxFlow = graph.edmondsKarp(s, t);
00323
00324     if (maxFlow == 0) {
00325         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00326              « endl;
00327     }
00328     cout « "maxFlow:\t" « maxFlow « endl;
00329
00330
00331     return 1;
00332 }
```

#### 4.1.2.10 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 335 of file CPheadquarters.cpp.

```
00335                                                                                {
00336     Graph graph;
00337     ifstream inputFile1;
00338     inputFile1.open(R"(../network.csv)");
00339     string line1;
00340 
00341     getline(inputFile1, line1);
00342     line1 = "";
00343 
00344     while (getline(inputFile1, line1)) {
00345         string station_A;
00346         string station_B;
00347         string temp;
00348         int capacity;
00349         string service;
00350         bool flag = true;
00351 
00352         stringstream inputString(line1);
00353 
00354         getline(inputString, station_A, ';');
00355         getline(inputString, station_B, ';');
00356         getline(inputString, temp, ';');
00357         capacity = stoi(temp);
00358         getline(inputString, service, ';');
00359 
00360         graph.addVertex(station_A);
00361         graph.addVertex(station_B);
00362         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00363             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00364                 flag = false;
00365                 break;
00366             }
00367         }
00368         if (flag) {
00369             graph.addEdge(station_A, station_B, capacity, service);
00370         }
00371         line1 = "";
00372     }
00373     vector<string> org = lines.getSources();
00374     vector<string> targ = lines.getTargets();
00375 
00376     lines.mul_edmondsKarp(org,targ);
00377     graph.mul_edmondsKarp(org,targ);
00378     vector<pair<int, int» top_k;
00379 
00380     auto length = lines.getVertexSet().size();
00381     for (int i = 0; i < length; ++i) {
00382         string destination = lines.getVertexSet()[i]->getId();
00383         auto v1 = lines.findVertex(destination);
00384         auto v2 = graph.findVertex(destination);
00385         int maxFlow1 = 0;
00386         int maxFlow2 = 0;
00387         for(auto e : v1->getIncoming()){
00388             maxFlow1+=e->getFlow();
00389         }
00390         for(auto e : v2->getIncoming()){
00391             maxFlow2+=e->getFlow();
00392         }
00393 
00394         if(destination=="Contumil"){
00395             continue;
00396         }
00397         int diff = maxFlow1 - maxFlow2;
00398         auto p = pair(i, diff);
00399         top_k.push_back(p);
00400         cout « "a";
00401     }
00402     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00403         return left.second > right.second;
00404     });
00405     for (int i = 0; i < 10; i++) {
00406         cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second
      « '\n';
00407     }
00408     return 1;
00409 }
```

#### 4.1.2.11 test()

```
void CPheadquarters::test ( )
```

Definition at line 105 of file CPheadquarters.cpp.

```
00105                          {
00106     int flow = lines.edmondsKarp(lines.getVertexSet()[324]->getId(),
      lines.getVertexSet()[507]->getId());
00107 }
```

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).*
- bool **addBidirectionalEdge** (const std::string &sourc, const std::string &dest, int w, std::string service)
- 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 ()
- std::vector< std::string > getTargets ()
- 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)
- std::vector< std::string > find_targets (std::vector< std::string > desired_stations)
- 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

- int **findVertexIdx** (const std::string &id) const
- 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 318 of file Graph.cpp.

```
00318                                           {
00319     auto v = findVertex(name);
00320     for(auto e : v->getAdj()){
00321         auto s = e->getDest()->getId();
00322         v->removeEdge(s);
00323     }
00324     for(auto e : v->getIncoming()){
00325         e->getOrig()->removeEdge(name);
00326     }
00327     auto it = vertexSet.begin();
00328     while (it!=vertexSet.end()){
00329         Vertex* currentVertex = *it;
00330         if(currentVertex->getId()==name){
00331             it=vertexSet.erase(it);
00332         }
00333         else{
00334             it++;
00335         }
00336     }
00337 }
```

#### 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 )
```

Definition at line 178 of file Graph.cpp.

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

#### 4.3.3.6 find_targets()

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

Definition at line 195 of file Graph.cpp.

```
00195                                                                                {
00196     std::vector<std::string> res;
00197     for (std::string s: desired_stations) {
00198         auto v = findVertex(s);
00199         if (v == nullptr) {
00200             std::cout « "Trouble finding target " « s « '\n';
00201             return res;
00202         }
00203         for (auto e: v->getAdj()) {
00204             if (!isIn(e->getDest()->getId(), desired_stations)) {
00205                 res.push_back(s);
00206             }
00207         }
00208     }
00209     return res;
00210 }
```

#### 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 264 of file Graph.cpp.

```
00265                                                                                {
00266     path.push_back(source);
00267     source->setVisited(true);
00268
00269     if (source == destination) {
00270         allPaths.push_back(path);
00271     } else {
00272         for (auto edge: source->getAdj()) {
00273             Vertex *adjacent = edge->getDest();
00274             if (!adjacent->isVisited()) {
00275                 findAllPaths(adjacent, destination, path, allPaths);
00276             }
00277         }
00278     }
00279
00280     path.pop_back();
00281     source->setVisited(false);
00282 }
```

#### 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 286 of file Graph.cpp.

```
00286                                                       {
00287
00288     for (auto edge: source->getAdj()) {
00289         if (edge->getDest() == destination) {
00290             return edge;
00291         }
00292     }
00293     return nullptr;
00294 }
```

#### 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 ( )
```

Definition at line 297 of file Graph.cpp.

```
00297                                          {
00298     std::vector<std::string> res;
00299     for (auto v : vertexSet) {
00300         if(v->getIncoming().empty()){
00301             res.push_back(v->getId());
00302         }
00303     }
00304     return res;
00305 }
```

#### 4.3.3.14 getTargets()

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

Definition at line 307 of file Graph.cpp.

```
00307                                          {
00308     std::vector<std::string> res;
00309     for (auto v : vertexSet) {
00310         if(v->getAdj().empty()){
00311             res.push_back(v->getId());
00312         }
00313     }
00314     return res;
00315 }
```

#### 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 213 of file Graph.cpp.

```
00213                                                                {
00214     for (std::string s: vec) {
00215         if (s == n) return true;
00216     }
00217     return false;
00218 }
```

#### 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 221 of file Graph.cpp.

```
00221                                                                                                {
00222     auto it1 = souces.begin();
00223     while (it1 != souces.end()) {
00224         if (isIn(*it1, targets)) {
00225             it1 = souces.erase(it1);
00226         } else it1++;
00227     }
00228
00229     auto it2 = targets.begin();
00230     while (it2 != targets.end()) {
00231         if (isIn(*it2, souces)) {
00232             it2 = souces.erase(it2);
00233         } else it2++;
00234     }
00235
00236     addVertex("temp_source");
00237     for (std::string s: souces) {
00238         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00239     }
00240
```

```
00241     addVertex("temp_targets");
00242     for (std::string s: targets) {
00243         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00244     }
00245     for (auto e: vertexSet) {
00246         for (auto i: e->getAdj()) {
00247             i->setFlow(0);
00248         }
00249     }
00250     int maxFlow = 0;
00251     while (findAugmentingPath("temp_source", "temp_targets")) {
00252         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00253         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00254         maxFlow += bottleneck;
00255     }
00256     deleteVertex("temp_targets");
00257     deleteVertex("temp_source");
00258     return maxFlow;
00259 }
```

#### 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 104 of file Graph.h.

#### 4.3.4.2 pathMatrix

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

Definition at line 105 of file Graph.h.

#### 4.3.4.3 vertexSet

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

Definition at line 102 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 (string name_, string district_, string municipality_, string township_, string line_)
- string get_name ()
- string get_district ()
- string get_municipality ()
- string get_township ()
- string get_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 ( )
```

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_ )
```

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 ( )
```

Definition at line 19 of file Station.cpp.

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

#### 4.4.3.2 get_line()

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

Definition at line 31 of file Station.cpp.

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

#### 4.4.3.3 get_municipality()

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

Definition at line 23 of file Station.cpp.

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

#### 4.4.3.4 get_name()

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

Definition at line 15 of file Station.cpp.

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

#### 4.4.3.5 get_township()

```
string Station::get_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
00010 using namespace std;
00011
00012 void CPheadquarters::read_files() {
00013
00014     //-------------------------------------------Read
      network.csv-------------------------------------------
00015     std::ifstream inputFile1(R"(../network.csv)");
00016     string line1;
00017     std::getline(inputFile1, line1); // ignore first line
00018     while (getline(inputFile1, line1, '\n')) {
00019
00020         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00021             line1.pop_back(); // Remove the '\r' character
00022         }
00023
00024         string station_A;
00025         string station_B;
00026         string temp;
00027         int capacity;
00028         string service;
00029
00030         stringstream inputString(line1);
00031
00032         getline(inputString, station_A, ',');
00033         getline(inputString, station_B, ',');
00034         getline(inputString, temp, ',');
00035         getline(inputString, service, ',');
00036
00037         capacity = stoi(temp);
00038         lines.addVertex(station_A);
00039         lines.addVertex(station_B);
00040
00041         lines.addEdge(station_A, station_B, capacity, service);
00042     }
00043
00044
00045     //-------------------------------------------Read
      stations.csv-------------------------------------------
00046     std::ifstream inputFile2(R"(../stations.csv)");
00047     string line2;
00048     std::getline(inputFile2, line2); // ignore first line
00049
00050     while (getline(inputFile2, line2, '\n')) {
00051
00052         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00053             line1.pop_back(); // Remove the '\r' character
00054         }
00055
00056         string nome;
```

```
00057         string distrito;
00058         string municipality;
00059         string township;
00060         string line;
00061 
00062         stringstream inputString(line2);
00063 
00064         getline(inputString, nome, ',');
00065         getline(inputString, distrito, ',');
00066         getline(inputString, municipality, ',');
00067         getline(inputString, township, ',');
00068         getline(inputString, line, ',');
00069 
00070         Station station(nome, distrito, municipality, township, line);
00071         stations[nome] = station;
00072 
00073         // print information about the station, to make sure it was imported correctly
00074         //cout << "station: " << nome << " distrito: " << distrito << " municipality: " << municipality << "
      township: " << township << " line: " << line << endl;
00075     }
00076 }
00077 
00078 
00079 Graph CPheadquarters::getLines() const {
00080     return this->lines;
00081 }
00082 
00083 
00084 int CPheadquarters::T2_1maxflow(string stationA, string stationB) {
00085     Vertex *source = lines.findVertex(stationA); // set source vertex
00086     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00087 
00088     // Check if these stations even exist
00089     if (source == nullptr || sink == nullptr) {
00090         std::cerr << "Source or sink vertex not found." << std::endl;
00091         return 1;
00092     }
00093     int maxFlow = lines.edmondsKarp(stationA, stationB);
00094 
00095     if (maxFlow == 0) {
00096         cerr << "Stations are not connected. Try stationB to stationA instead. " << stationB << " -> " <<
      stationA
00097              << endl;
00098     } else {
00099         cout << "maxFlow:\t" << maxFlow << endl;
00100     }
00101 
00102     return 1;
00103 }
00104 
00105 void CPheadquarters::test() {
00106     int flow = lines.edmondsKarp(lines.getVertexSet()[324]->getId(),
      lines.getVertexSet()[507]->getId());
00107 }
00108 
00109 
00110 
00111 int CPheadquarters::T2_2maxflowAllStations() {
00112     vector<string> stations;
00113     int maxFlow = 0;
00114     auto length = lines.getVertexSet().size();
00115     // Start the timer
00116     auto start_time = std::chrono::high_resolution_clock::now();
00117     cout << "Calculating max flow for all pairs of stations..." << endl;
00118     cout << "Please stand by..." << endl;
00119     for (int i = 0; i < length; ++i) {
00120         for (int j = i + 1; j < length; ++j) {
00121             string stationA = lines.getVertexSet()[i]->getId();
00122             string stationB = lines.getVertexSet()[j]->getId();
00123             int flow = lines.edmondsKarp(stationA, stationB);
00124             if (flow == maxFlow) {
00125                 stations.push_back(stationB);
00126                 stations.push_back(stationA);
00127             } else if (flow > maxFlow) {
00128                 stations.clear();
00129                 stations.push_back(stationB);
00130                 stations.push_back(stationA);
00131                 maxFlow = flow;
00132             }
00133         }
00134     }
00135     // End the timer
00136     auto end_time = std::chrono::high_resolution_clock::now();
00137 
00138     // Compute the duration
00139     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00140 
```

```
00141     // Print the duration
00142     std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
00143 
00144     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00145     for (int i = 0; i < stations.size(); i = i + 2) {
00146         cout « "------------------------\n";
00147         cout « "Source: " « stations[i + 1] « '\n';
00148         cout « "Target: " « stations[i] « '\n';
00149         cout « "------------------------\n";
00150     }
00151     return 0;
00152 }
00153 
00154 
00155 int CPheadquarters::T2_3municipality(string municipality) {
00156     vector<string> desired_stations;
00157     for (auto p: stations) {
00158         if (p.second.get_municipality() == municipality) {
00159             desired_stations.push_back(p.second.get_name());
00160         }
00161     }
00162     vector<string> souces = lines.find_sources(desired_stations);
00163     vector<string> targets = lines.find_targets(desired_stations);
00164     return lines.mul_edmondsKarp(souces, targets);
00165 }
00166 
00167 int CPheadquarters::T2_3district(string district) {
00168     vector<string> desired_stations;
00169     for (auto p: stations) {
00170         if (p.second.get_district() == district) {
00171             desired_stations.push_back(p.second.get_name());
00172         }
00173     }
00174     return lines.mul_edmondsKarp(lines.find_sources(desired_stations),
      lines.find_targets(desired_stations));
00175 }
00176 
00177 
00178 int CPheadquarters::T2_4maxArrive(string destination) {
00179     Vertex *dest = lines.findVertex(destination);
00180     int maxFlow = 0;
00181 
00182     // iterate over all vertices to find incoming and outgoing vertices
00183     for (auto &v: lines.getVertexSet()) {
00184         if (v != dest) {
00185 
00186             int flow = lines.edmondsKarp(v->getId(), destination);
00187 
00188             // Update the maximum flow if this vertex contributes to a higher maximum
00189             if (flow > maxFlow) {
00190                 maxFlow = flow;
00191             }
00192         }
00193 
00194     }
00195 
00196     cout « endl;
00197     for (auto &e: dest->getIncoming()) {
00198         cout « e->getOrig()->getId() « " -> " « e->getDest()->getId() « " : " « e->getWeight() « endl;
00199 
00200     }
00201 
00202     cout « "Max number of trains that can simultaneously arrive at " « destination « ": " « maxFlow «
      endl;
00203     return maxFlow;
00204 
00205 }
00206 
00207 
00208 
00209 int CPheadquarters::T3_1MinCost(string source, string destination) {
00210     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00211     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00212     if (sourceVertex == nullptr || destVertex == nullptr) {
00213         cerr « "Source or destination vertex not found. Try again" « endl;
00214         return 1;
00215     }
00216 
00217     Graph graph = lines;
00218 
00219     std::vector<Vertex *> path;
00220     std::vector<std::vector<Vertex *» allPaths;
00221 
00222 
00223     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00224 
00225     vector<int> maxFlows;
```

```
00226     vector<int> totalCosts;
00227
00228     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00229     for (auto path: allPaths) {
00230         int minWeight = 10;
00231         int totalCost = 0; // total cost of this path
00232         for (int i = 0; i + 1 < path.size(); i++) {
00233             std::cout « path[i]->getId() « " -> ";
00234             Edge *e = graph.findEdge(path[i], path[i + 1]);
00235             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00236             if (e->getWeight() < minWeight) {
00237                 minWeight = e->getWeight();
00238             }
00239
00240             // according to the problem's specification, the cost of STANDARD service is 2 euros and
    ALFA PENDULAR is 4
00241             if (e->getService() == "STANDARD") {
00242                 totalCost += 2;
00243             } else if (e->getService() == "ALFA PENDULAR") {
00244                 totalCost += 4;
00245             }
00246         }
00247         maxFlows.push_back(minWeight);
00248         totalCosts.push_back(totalCost);
00249         cout « " -> " « path[path.size() - 1]->getId() « endl;
00250         cout « "Max flow for this path: " « minWeight « " trains. ";
00251         cout « "Total cost: " « totalCost « " euros." « endl;
00252         std::cout « std::endl;
00253     }
00254
00255     // find the path with the minimum cost per train
00256     int maxTrains = 0;
00257     int resCost;
00258     double max_value = 10000;
00259     for (int i = 0; i < maxFlows.size(); ++i) {
00260         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00261         if (costPerTrain < max_value) {
00262             max_value = costPerTrain;
00263             maxTrains = maxFlows[i];
00264             resCost = totalCosts[i];
00265         }
00266     }
00267
00268     cout « "Max number of trains that can travel between " « source « " and " « destination
00269          « " with minimum cost"
00270          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00271     return maxTrains;
00272 }
00273
00274
00275 int CPheadquarters::T4_1ReducedConectivity(std::vector<std::string> unwantedEdges, std::string s,
    std::string t) {
00276     Graph graph;
00277     ifstream inputFile1;
00278     inputFile1.open(R"(../network.csv)");
00279     string line1;
00280
00281     getline(inputFile1, line1);
00282     line1 = "";
00283
00284     while (getline(inputFile1, line1)) {
00285         string station_A;
00286         string station_B;
00287         string temp;
00288         int capacity;
00289         string service;
00290         bool flag = true;
00291
00292         stringstream inputString(line1);
00293
00294         getline(inputString, station_A, ';');
00295         getline(inputString, station_B, ';');
00296         getline(inputString, temp, ';');
00297         capacity = stoi(temp);
00298         getline(inputString, service, ';');
00299
00300         graph.addVertex(station_A);
00301         graph.addVertex(station_B);
00302         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00303             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00304                 flag = false;
00305
00306             }
00307         }
00308         if (flag) {
00309             graph.addEdge(station_A, station_B, capacity, service);
00310         }
```

```
00311         line1 = "";
00312     }
00313
00314     Vertex *source = graph.findVertex(s); // set source vertex
00315     Vertex *sink = graph.findVertex(t); // set sink vertex
00316
00317     // Check if these stations even exist
00318     if (source == nullptr || sink == nullptr) {
00319         std::cerr « "Source or sink vertex not found." « std::endl;
00320         return 1;
00321     }
00322     int maxFlow = graph.edmondsKarp(s, t);
00323
00324     if (maxFlow == 0) {
00325         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00326              « endl;
00327     }
00328     cout « "maxFlow:\t" « maxFlow « endl;
00329
00330
00331     return 1;
00332 }
00333
00334
00335 int CPheadquarters::T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges) {
00336     Graph graph;
00337     ifstream inputFile1;
00338     inputFile1.open(R"(../network.csv)");
00339     string line1;
00340
00341     getline(inputFile1, line1);
00342     line1 = "";
00343
00344     while (getline(inputFile1, line1)) {
00345         string station_A;
00346         string station_B;
00347         string temp;
00348         int capacity;
00349         string service;
00350         bool flag = true;
00351
00352         stringstream inputString(line1);
00353
00354         getline(inputString, station_A, ';');
00355         getline(inputString, station_B, ';');
00356         getline(inputString, temp, ';');
00357         capacity = stoi(temp);
00358         getline(inputString, service, ';');
00359
00360         graph.addVertex(station_A);
00361         graph.addVertex(station_B);
00362         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00363             if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
00364                 flag = false;
00365                 break;
00366             }
00367         }
00368         if (flag) {
00369             graph.addEdge(station_A, station_B, capacity, service);
00370         }
00371         line1 = "";
00372     }
00373     vector<string> org = lines.getSources();
00374     vector<string> targ = lines.getTargets();
00375
00376     lines.mul_edmondsKarp(org,targ);
00377     graph.mul_edmondsKarp(org,targ);
00378     vector<pair<int, int» top_k;
00379
00380     auto length = lines.getVertexSet().size();
00381     for (int i = 0; i < length; ++i) {
00382         string destination = lines.getVertexSet()[i]->getId();
00383         auto v1 = lines.findVertex(destination);
00384         auto v2 = graph.findVertex(destination);
00385         int maxFlow1 = 0;
00386         int maxFlow2 = 0;
00387         for(auto e : v1->getIncoming()){
00388             maxFlow1+=e->getFlow();
00389         }
00390         for(auto e : v2->getIncoming()){
00391             maxFlow2+=e->getFlow();
00392         }
00393
00394         if(destination=="Contumil"){
00395             continue;
00396         }
00397         int diff = maxFlow1 - maxFlow2;
```

```
00398         auto p = pair(i, diff);
00399         top_k.push_back(p);
00400         cout « "a";
00401     }
00402     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00403         return left.second > right.second;
00404     });
00405     for (int i = 0; i < 10; i++) {
00406         cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second
      « '\n';
00407     }
00408     return 1;
00409 }
```

## 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:
00021     void read_files();
00022
00027     Graph getLines() const;
00028
00037     int T2_1maxflow(string station_A, string station_B);
00038
00047     int T2_2maxflowAllStations();
00048
00056     int T2_3municipality(string municipality);
00057
00058     int T2_3district(string district);
00059
00067     int T2_4maxArrive(string destination);
00068
00079     int T3_1MinCost(string source, string destination);
00080
00092     int T4_1ReducedConectivity(vector<string> unwantedEdges, string s, string t);
00093
00100     int T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges);
00101
00102     void test();
00103
00104
00105
00106 };
00107
00108
00109 #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     for (std::string s: desired_stations) {
00181         auto v = findVertex(s);
00182         if (v == nullptr) {
00183             std::cout « "Trouble finding source " « s « '\n';
00184             return res;
00185         }
00186         for (auto e: v->getIncoming()) {
00187             if (!isIn(e->getOrig()->getId(), desired_stations)) {
```

```
00188                 res.push_back(s);
00189             }
00190         }
00191     }
00192     return res;
00193 }
00194
00195 std::vector<std::string> Graph::find_targets(std::vector<std::string> desired_stations) {
00196     std::vector<std::string> res;
00197     for (std::string s: desired_stations) {
00198         auto v = findVertex(s);
00199         if (v == nullptr) {
00200             std::cout « "Trouble finding target " « s « '\n';
00201             return res;
00202         }
00203         for (auto e: v->getAdj()) {
00204             if (!isIn(e->getDest()->getId(), desired_stations)) {
00205                 res.push_back(s);
00206             }
00207         }
00208     }
00209     return res;
00210 }
00211
00212
00213 bool Graph::isIn(std::string n, std::vector<std::string> vec) {
00214     for (std::string s: vec) {
00215         if (s == n) return true;
00216     }
00217     return false;
00218 }
00219
00220
00221 int Graph::mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets) {
00222     auto it1 = souces.begin();
00223     while (it1 != souces.end()) {
00224         if (isIn(*it1, targets)) {
00225             it1 = souces.erase(it1);
00226         } else it1++;
00227     }
00228
00229     auto it2 = targets.begin();
00230     while (it2 != targets.end()) {
00231         if (isIn(*it2, souces)) {
00232             it2 = souces.erase(it2);
00233         } else it2++;
00234     }
00235
00236     addVertex("temp_source");
00237     for (std::string s: souces) {
00238         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00239     }
00240
00241     addVertex("temp_targets");
00242     for (std::string s: targets) {
00243         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00244     }
00245     for (auto e: vertexSet) {
00246         for (auto i: e->getAdj()) {
00247             i->setFlow(0);
00248         }
00249     }
00250     int maxFlow = 0;
00251     while (findAugmentingPath("temp_source", "temp_targets")) {
00252         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00253         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00254         maxFlow += bottleneck;
00255     }
00256     deleteVertex("temp_targets");
00257     deleteVertex("temp_source");
00258     return maxFlow;
00259 }
00260
00261 // ---------------------------------- Find ALL existing augmenting paths
    ----------------------------------
00262
00263
00264 void Graph::findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00265                          std::vector<std::vector<Vertex *» &allPaths) {
00266     path.push_back(source);
00267     source->setVisited(true);
00268
00269     if (source == destination) {
00270         allPaths.push_back(path);
00271     } else {
00272         for (auto edge: source->getAdj()) {
00273             Vertex *adjacent = edge->getDest();
```

```
00274             if (!adjacent->isVisited()) {
00275                 findAllPaths(adjacent, destination, path, allPaths);
00276             }
00277         }
00278     }
00279
00280     path.pop_back();
00281     source->setVisited(false);
00282 }
00283
00284
00285
00286 Edge *Graph::findEdge(Vertex *source, Vertex *destination) {
00287
00288     for (auto edge: source->getAdj()) {
00289         if (edge->getDest() == destination) {
00290             return edge;
00291         }
00292     }
00293     return nullptr;
00294 }
00295
00296
00297 std::vector<std::string> Graph::getSources() {
00298     std::vector<std::string> res;
00299     for (auto v : vertexSet) {
00300         if(v->getIncoming().empty()){
00301             res.push_back(v->getId());
00302         }
00303     }
00304     return res;
00305 }
00306
00307 std::vector<std::string> Graph::getTargets() {
00308     std::vector<std::string> res;
00309     for (auto v : vertexSet) {
00310         if(v->getAdj().empty()){
00311             res.push_back(v->getId());
00312         }
00313     }
00314     return res;
00315 }
00316
00317
00318 void Graph::deleteVertex(std::string name) {
00319     auto v = findVertex(name);
00320     for(auto e : v->getAdj()){
00321         auto s = e->getDest()->getId();
00322         v->removeEdge(s);
00323     }
00324     for(auto e : v->getIncoming()){
00325         e->getOrig()->removeEdge(name);
00326     }
00327     auto it = vertexSet.begin();
00328     while (it!=vertexSet.end()){
00329         Vertex* currentVertex = *it;
00330         if(currentVertex->getId()==name){
00331             it=vertexSet.erase(it);
00332         }
00333         else{
00334             it++;
00335         }
00336     }
00337 }
00338
```

## 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     bool addBidirectionalEdge(const std::string &sourc, const std::string &dest, int w, std::string
      service);
00045
00046     [[nodiscard]] int getNumVertex() const;
00047
00048     [[nodiscard]] std::vector<Vertex *> getVertexSet() const;
00049
00053     void print() const;
00054
00064     int edmondsKarp(const std::string &s, const std::string &t);
00065
00066     std::vector<std::string> getSources();
00067
00068     std::vector<std::string> getTargets();
00069
00076     int mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets);
00077
00078     std::vector<std::string> find_sources(std::vector<std::string> desired_stations);
00079
00080     std::vector<std::string> find_targets(std::vector<std::string> desired_stations);
00081
00090     void findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00091                       std::vector<std::vector<Vertex *>> &allPaths);
00092
00099     Edge *findEdge(Vertex *source, Vertex *destination);
00100
00101 protected:
00102     std::vector<Vertex *> vertexSet;    // vertex set
00103
00104     double **distMatrix = nullptr;   // dist matrix for Floyd-Warshall
00105     int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall
00106
00107     /*
00108      * Finds the index of the vertex with a given content.
00109      */
00110     int findVertexIdx(const std::string &id) const;
00111
00119     void updateFlow(Vertex *s, Vertex *t, int bottleneck);
00120
00127     int findMinResidual(Vertex *s, Vertex *t);
00128
00138     bool findAugmentingPath(const std::string &s, const std::string &t);
00139
00147     void testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual);
00148
00149     bool isIn(std::string n, std::vector<std::string> vec);
00150
00155     void deleteVertex(std::string name);
00156 };
00157
00158 void deleteMatrix(int **m, int n);
00159
00160 void deleteMatrix(double **m, int n);
00161
00162 #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     CP.read_files();
00009     //CP.getLines().print();
00010     int n;
00011     cout << "-------------- An Analysis Tool for Railway Network Management --------------\n" << endl;
00012     do {
00013         cout << "1 - T2.1 Max number of trains between stations\n";
00014         cout << "2 - T2.2 Stations that require the Max num of trains among all pairs of stations\n";
00015         cout << "3 - T2.3 Indicate where management should assign larger budgets for the purchasing and
      maintenance of trains\n";
```

```
00016         cout « "4 - T2.4 Max number of trains that can simultaneously arrive at a given station\n";
00017         cout « "5 - T3.1 Max number of trains that can simultaneously travel with minimum cost\n";
00018         cout « "6 - T4.1\n";
00019         cout « "7 - T4.2\n";
00020         cout « "8 - Exit\n";
00021
00022
00023         bool validInput = false;
00024
00025         while (!validInput) {
00026             cout « "Insert your option:\n";
00027             cin » n;
00028
00029             if (cin.fail() || n < 1 || n > 8) {
00030                 cin.clear();
00031                 cin.ignore(numeric_limits<streamsize>::max(), '\n');
00032                 cout « "Invalid input. Please enter a number between 1 and 8." « endl;
00033             } else {
00034                 validInput = true;
00035             }
00036         }
00037
00038         switch (n) {
00039             case 1: {
00040                 cin.ignore(); // ignore newline character left in the input stream
00041                 string a, b;
00042                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00043                 cout « "Enter station A: ";
00044                 getline(cin, a);
00045
00046                 cout « "Enter station B: ";
00047                 getline(cin, b);
00048
00049                 if (a.empty() || b.empty()) {
00050                     cerr « "Error: Station names cannot be empty." « endl;
00051                     break;
00052                 }
00053
00054                 // call function to calculate max flow between stations A and B
00055                 CP.T2_1maxflow(a, b);
00056                 break;
00057             }
00058
00059             case 2: {
00060                 CP.T2_2maxflowAllStations();
00061                 break;
00062             }
00063
00064             case 3: {
00065                 cin.ignore();
00066                 string c;
00067                 cout « R"(Example: "PENAFIEL")" « endl;
00068                 cout « "Enter municipality: " « endl;
00069                 cout « "For example, PENAFIEL: ";
00070                 getline(cin, c);
00071                 cout « "The maximum flow in Municipality " « c « " is " « CP.T2_3municipality(c) «
    endl;
00072                 cout « endl;
00073                 break;
00074             }
00075
00076             case 4: {
00077                 cin.ignore();
00078                 string destination;
00079                 cout « "Enter destination: ";
00080                 getline(cin, destination);
00081                 CP.T2_4maxArrive(destination);
00082                 break;
00083             }
00084
00085             case 5: {
00086                 cin.ignore();
00087                 string a, b;
00088                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00089                 cout « "Enter station A: ";
00090                 getline(cin, a);
00091                 cout « endl;
00092                 cout « "Enter station B: ";
00093                 getline(cin, b);
00094
00095                 if (a.empty() || b.empty()) {
00096                     cerr « "Error: Station names cannot be empty." « endl;
00097                     break;
00098                 }
00099
00100                 CP.T3_1MinCost(a, b);
00101                 break;
```

```
00102             }
00103
00104             case 6: {
00105                 cin.ignore();
00106                 vector<string> unwantedEdges;
00107                 string edgesource;
00108                 string edgetarget;
00109                 string b;
00110                 string a;
00111                 cout « "Enter station A: ";
00112                 getline(cin, a);
00113                 cout « "Enter station B: ";
00114                 getline(cin, b);
00115                 cout « '\n';
00116                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
      Type '.' to end listing: \n";
00117                 while (1){
00118                     cout « "Enter edge source or '.' to finish: ";
00119                     getline(cin, edgesource);
00120                     if(edgesource==".") break;
00121                     unwantedEdges.push_back(edgesource);
00122                     cout « "Enter edge target: ";
00123                     getline(cin, edgetarget);
00124                     unwantedEdges.push_back(edgetarget);
00125                 }
00126                 CP.T4_1ReducedConectivity(unwantedEdges,a,b);
00127                 break;
00128             }
00129
00130             case 7: {
00131                 cin.ignore();
00132                 vector<string> unwantedEdges;
00133                 string edgesource;
00134                 string edgetarget;
00135                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
      Type '.' to end listing: \n";
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_2Top_K_ReducedConectivity(unwantedEdges);
00146
00147                 break;
00148             }
00149
00150             case 8: {
00151                 cout « "Exiting program..." « endl;
00152                 break;
00153             }
00154
00155             default: {
00156                 cerr « "Error: Invalid option selected." « endl;
00157                 break;
00158             }
00159         }
00160     } while (n != 8);
00161
00162     return 0;
00163 }
```

## 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:
00019     Station();
00020     Station(string name_, string district_, string municipality_, string township_, string line_);
00021     string get_name();
00022     string get_district();
00023     string get_municipality();
00024     string get_township();
00025     string get_line();
00026 };
00027
00028
00029 #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