DAproject

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 ()
- Graph getLines () const
- int T2_1maxflow (string station_A, string station_B)
- int T2_2maxflowAllStations ()
- int T2_3municipality (string municipality)
- int T2_3district (string district)
- int T2_4maxArrive (string destination)
- int T3_1MinCost (string source, string destination)
- int T4_1ReducedConectivity (vector< string > unwantedEdges, string s, string t)
- int T4_2Top_K_ReducedConectivity (vector< string > unwantedEdges)
- 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
```

Definition at line 80 of file CPheadquarters.cpp.

```
00080                                       {
00081     return this->lines;
00082 }
```

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 14 of file CPheadquarters.cpp.

```
00014                                {
00015
00016     //-----------------------------------------Read
      network.csv-----------------------------------------
00017     std::ifstream inputFile1(R"(../network.csv)");
00018     string line1;
00019     std::getline(inputFile1, line1); // ignore first line
00020     while (getline(inputFile1, line1, '\n')) {
00021
00022         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00023             line1.pop_back(); // Remove the '\r' character
00024         }
00025
00026         string station_A;
00027         string station_B;
00028         string temp;
00029         int capacity;
00030         string service;
00031
00032         stringstream inputString(line1);
00033
00034         getline(inputString, station_A, ',');
00035         getline(inputString, station_B, ',');
00036         getline(inputString, temp, ',');
00037         getline(inputString, service, ',');
00038
00039         capacity = stoi(temp);
00040         lines.addVertex(station_A);
00041         lines.addVertex(station_B);
00042
00043         lines.addEdge(station_A, station_B, capacity, service);
00044     }
00045
00046
00047     //-----------------------------------------Read
      stations.csv-----------------------------------------
00048     std::ifstream inputFile2(R"(../stations.csv)");
00049     string line2;
00050     std::getline(inputFile2, line2); // ignore first line
00051
00052     while (getline(inputFile2, line2, '\n')) {
00053
00054         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00055             line1.pop_back(); // Remove the '\r' character
00056         }
00057
00058         string nome;
00059         string distrito;
00060         string municipality;
00061         string township;
00062         string line;
00063
00064         stringstream inputString(line2);
00065
00066         getline(inputString, nome, ',');
00067         getline(inputString, distrito, ',');
00068         getline(inputString, municipality, ',');
00069         getline(inputString, township, ',');
00070         getline(inputString, line, ',');
00071
00072         Station station(nome, distrito, municipality, township, line);
00073         stations[nome] = station;
00074
00075         // print information about the station, to make sure it was imported correctly
00076         //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « "
      township: " « township « " line: " « line « endl;
00077     }
00078 }
```

4.1.2.3 T2_1maxflow()

```
int CPheadquarters::T2_1maxflow (
            string stationA,
            string stationB )
```

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

Parameters

stationA	
stationB	

Returns

Definition at line 92 of file CPheadquarters.cpp.

```
00092                                                                {
00093     Vertex *source = lines.findVertex(stationA); // set source vertex
00094     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00095
00096     // Check if these stations even exist
00097     if (source == nullptr || sink == nullptr) {
00098         std::cerr « "Source or sink vertex not found." « std::endl;
00099         return 1;
00100     }
00101     int maxFlow = lines.edmondsKarp(stationA, stationB);
00102
00103     if (maxFlow == 0) {
00104         cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " «
      stationA
00105              « endl;
00106     } else {
00107         cout « "maxFlow:\t" « maxFlow « endl;
00108     }
00109
00110     return 1;
00111 }
```

4.1.2.4 T2_2maxflowAllStations()

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

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;

Returns

Definition at line 123 of file CPheadquarters.cpp.

```
00123                                           {
00124     vector<string> stations;
00125     int maxFlow = 0;
00126     auto length = lines.getVertexSet().size();
00127     for (int i = 0; i < length; ++i) {
00128         for (int j = i + 1; j < length; ++j) {
00129             string stationA = lines.getVertexSet()[i]->getId();
00130             string stationB = lines.getVertexSet()[j]->getId();
00131             int flow = lines.edmondsKarp(stationA, stationB);
00132             if (flow == maxFlow) {
00133                 stations.push_back(stationB);
00134                 stations.push_back(stationA);
00135             } else if (flow > maxFlow) {
00136                 stations.clear();
00137                 stations.push_back(stationB);
00138                 stations.push_back(stationA);
00139                 maxFlow = flow;
00140             }
00141         }
00142     }
00143     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00144     for (int i = 0; i < stations.size(); i = i + 2) {
00145         cout « "------------------------\n";
00146         cout « "Source:" « stations[i + 1] « '\n';
00147         cout « "Target:" « stations[i] « '\n';
00148         cout « "------------------------\n";
00149     }
00150     return 0;
00151 }
```

4.1.2.5 T2_3district()

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

Definition at line 172 of file CPheadquarters.cpp.

```
00172                                                {
00173     vector<string> desired_stations;
00174     for (auto p: stations) {
00175         if (p.second.get_district() == district) {
00176             desired_stations.push_back(p.second.get_name());
00177         }
00178     }
00179     return lines.mul_edmondsKarp(lines.find_sources(desired_stations),
      lines.find_targets(desired_stations));
00180 }
```

4.1.2.6 T2_3municipality()

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

- 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

Parameters

municipality	

Returns

Definition at line 160 of file CPheadquarters.cpp.

```
00160                                                    {
00161     vector<string> desired_stations;
00162     for (auto p: stations) {
00163         if (p.second.get_municipality() == municipality) {
00164             desired_stations.push_back(p.second.get_name());
00165         }
00166     }
00167     vector<string> souces = lines.find_sources(desired_stations);
00168     vector<string> targets = lines.find_targets(desired_stations);
00169     return lines.mul_edmondsKarp(souces, targets);
00170 }
```

4.1.2.7 T2_4maxArrive()

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

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

Parameters

destination	

Returns

maxFlow

Definition at line 188 of file CPheadquarters.cpp.

```
00188                                                   {
00189     Vertex *dest = lines.findVertex(destination);
00190     int maxFlow = 0;
00191 
00192     // iterate over all vertices to find incoming and outgoing vertices
00193     for (auto &v: lines.getVertexSet()) {
00194         if (v != dest) {
00195 
00196             int flow = lines.edmondsKarp(v->getId(), destination);
00197 
00198             // Update the maximum flow if this vertex contributes to a higher maximum
00199             if (flow > maxFlow) {
00200                 maxFlow = flow;
00201             }
00202         }
00203 
00204     }
00205 
00206     cout << endl;
00207     for (auto &e: dest->getIncoming()) {
00208         cout << e->getOrig()->getId() << " -> " << e->getDest()->getId() << " : " << e->getWeight() << endl;
00209 
00210     }
00211 
00212     cout << "Max number of trains that can simultaneously arrive at " << destination << ": " << maxFlow <<
      endl;
00213     return maxFlow;
00214 
00215 }
```

4.1.2.8 T3_1MinCost()

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

- Calculate the maximum amount of trains that can simultaneously travel between two specific stations with minimum cost for the company constraints:

Minimize cost

1. 'Maintain the same level of service'

steps: 1 - find all possible paths between source and destination 2 - define the optimal path

Parameters

source	
destination	

Returns

Definition at line 232 of file CPheadquarters.cpp.

```
00232                                                   {
```

```
00233     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00234     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00235     if (sourceVertex == nullptr || destVertex == nullptr) {
00236         cerr « "Source or destination vertex not found. Try again" « endl;
00237         return 1;
00238     }
00239
00240     Graph graph = lines;
00241
00242     std::vector<Vertex *> path;
00243     std::vector<std::vector<Vertex *» allPaths;
00244
00245
00246     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00247
00248     vector<int> maxFlows;
00249     vector<int> totalCosts;
00250
00251     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00252     for (auto path: allPaths) {
00253         int minWeight = 10;
00254         int totalCost = 0; // total cost of this path
00255         for (int i = 0; i + 1 < path.size(); i++) {
00256             std::cout « path[i]->getId() « " -> ";
00257             Edge *e = graph.findEdge(path[i], path[i + 1]);
00258             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00259             if (e->getWeight() < minWeight) {
00260                 minWeight = e->getWeight();
00261             }
00262
00263             // according to the problem's specification, the cost of STANDARD service is 2 euros and
    ALFA PENDULAR is 4
00264             if (e->getService() == "STANDARD") {
00265                 totalCost += 2;
00266             } else if (e->getService() == "ALFA PENDULAR") {
00267                 totalCost += 4;
00268             }
00269         }
00270         maxFlows.push_back(minWeight);
00271         totalCosts.push_back(totalCost);
00272         cout « " -> " « path[path.size() - 1]->getId() « endl;
00273         cout « "Max flow for this path: " « minWeight « " trains. ";
00274         cout « "Total cost: " « totalCost « " euros." « endl;
00275         std::cout « std::endl;
00276     }
00277
00278     // find the path with the minimum cost per train
00279     int maxTrains = 0;
00280     int resCost;
00281     double max_value = 10000;
00282     for (int i = 0; i < maxFlows.size(); ++i) {
00283         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00284         if (costPerTrain < max_value) {
00285             max_value = costPerTrain;
00286             maxTrains = maxFlows[i];
00287             resCost = totalCosts[i];
00288         }
00289     }
00290
00291     cout « "Max number of trains that can travel between " « source « " and " « destination
00292          « " with minimum cost"
00293          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00294     return maxTrains;
00295 }
```

4.1.2.9 T4_1ReducedConectivity()

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

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;

Parameters

unwantedEdges	
s	
t	

Returns

Definition at line 306 of file CPheadquarters.cpp.

```
00306
      {
00307     Graph graph;
00308     ifstream inputFile1;
00309     inputFile1.open(R"(../network.csv)");
00310     string line1;
00311
00312     getline(inputFile1, line1);
00313     line1 = "";
00314
00315     while (getline(inputFile1, line1)) {
00316         string station_A;
00317         string station_B;
00318         string temp;
00319         int capacity;
00320         string service;
00321         bool flag = true;
00322
00323         stringstream inputString(line1);
00324
00325         getline(inputString, station_A, ';');
00326         getline(inputString, station_B, ';');
00327         getline(inputString, temp, ';');
00328         capacity = stoi(temp);
00329         getline(inputString, service, ';');
00330
00331         graph.addVertex(station_A);
00332         graph.addVertex(station_B);
00333         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00334             if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00335                 flag=false;
00336
00337             }
00338         }
00339         if(flag) {
00340             graph.addEdge(station_A, station_B, capacity, service);
00341         }
00342         line1 = "";
00343     }
00344
00345     Vertex *source = graph.findVertex(s); // set source vertex
00346     Vertex *sink = graph.findVertex(t); // set sink vertex
00347
00348     // Check if these stations even exist
00349     if (source == nullptr || sink == nullptr) {
00350         std::cerr « "Source or sink vertex not found." « std::endl;
00351         return 1;
00352     }
00353     int maxFlow = graph.edmondsKarp(s, t);
00354
00355     if (maxFlow == 0) {
00356         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00357              « endl;
00358     }
00359     cout « "maxFlow:\t" « maxFlow « endl;
00360
00361
00362     return 1;
00363 }
```

4.1.2.10 T4_2Top_K_ReducedConectivity()

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

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

Parameters

unwantedEdges	

Returns

Definition at line 371 of file CPheadquarters.cpp.

```
00371                                                                                {
00372     Graph graph;
00373     ifstream inputFile1;
00374     inputFile1.open(R"(../network.csv)");
00375     string line1;
00376 
00377     getline(inputFile1, line1);
00378     line1 = "";
00379 
00380     while (getline(inputFile1, line1)) {
00381         string station_A;
00382         string station_B;
00383         string temp;
00384         int capacity;
00385         string service;
00386         bool flag = true;
00387 
00388         stringstream inputString(line1);
00389 
00390         getline(inputString, station_A, ';');
00391         getline(inputString, station_B, ';');
00392         getline(inputString, temp, ';');
00393         capacity = stoi(temp);
00394         getline(inputString, service, ';');
00395 
00396         graph.addVertex(station_A);
00397         graph.addVertex(station_B);
00398         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00399             if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00400                 flag=false;
00401                 break;
00402             }
00403         }
00404         if(flag) {
00405             graph.addEdge(station_A, station_B, capacity, service);
00406         }
00407         line1 = "";
00408     }
00409 
00410 
00411     vector<pair<int, int» top_k;
00412     auto length = lines.getVertexSet().size();
00413     for (int i = 0; i < length; ++i) {
00414         string destination = lines.getVertexSet()[i]->getId();
00415         Vertex *dest = lines.findVertex(destination);
00416 
00417         int maxFlow1 = 0;
00418         int maxFlow2 = 0;
00419 
00420         for (auto &v: lines.getVertexSet()) {
00421             if (v != dest) {
00422                 int flow = lines.edmondsKarp(v->getId(), destination);
00423                 if (flow > maxFlow1) {
00424                     maxFlow1 = flow;
00425                 }
00426                 flow = graph.edmondsKarp(v->getId(), destination);
00427                 if (flow > maxFlow2) {
00428                     maxFlow2 = flow;
00429                 }
00430             }
00431         }
00432 
00433         int diff = maxFlow1 - maxFlow2;
00434         auto p = pair(i,diff);
00435         top_k.push_back(p);
00436         cout « "a";
00437     }
00438     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00439         return left.second > right.second;
00440     });
00441     for(int i = 0; i < 10; i++){
```

```
00442         cout « i+1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second «
      '\n';
00443     }
00444     return 1;
00445 }
```

4.1.2.11 test()

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

Definition at line 113 of file CPheadquarters.cpp.

```
00113                         {
00114     int flow = lines.edmondsKarp(lines.getVertexSet()[324]->getId(),
      lines.getVertexSet()[507]->getId());
00115 }
```

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
- int capacity
- 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 capacity

```
int Edge::capacity  [protected]
```

Definition at line 107 of file VertexEdge.h.

4.2.4.2 dest

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

Definition at line 105 of file VertexEdge.h.

4.2.4.3 flow

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

Definition at line 116 of file VertexEdge.h.

4.2.4.4 orig

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

Definition at line 113 of file VertexEdge.h.

4.2.4.5 reverse

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

Definition at line 114 of file VertexEdge.h.

4.2.4.6 selected

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

Definition at line 110 of file VertexEdge.h.

4.2.4.7 service

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

Definition at line 108 of file VertexEdge.h.

4.2.4.8 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
- bool addVertex (const std::string &id)
- bool addEdge (const std::string &sourc, const std::string &dest, int w, const std::string &service)
- 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
- int edmondsKarp (const std::string &s, const std::string &t)
- int mul_edmondsKarp (std::vector< std::string > souces, std::vector< std::string > 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)
- Edge * findEdge (Vertex *source, Vertex *destination)

Protected Member Functions

- int findVertexIdx (const std::string &id) const
- void updateFlow (Vertex *s, Vertex *t, int bottleneck)
- int findMinResidual (Vertex *s, Vertex *t)
- bool findAugmentingPath (const std::string &s, const std::string &t)
- void testAndVisit (std::queue< Vertex * > &q, Edge *e, Vertex *w, double residual)
- bool isIn (std::string n, std::vector< std::string > vec)

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 93 of file Graph.cpp.

```
00093              {
00094     deleteMatrix(distMatrix, vertexSet.size());
00095     deleteMatrix(pathMatrix, vertexSet.size());
00096 }
```

4.3.3 Member Function Documentation

4.3.3.1 addBidirectionalEdge()

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

Definition at line 62 of file Graph.cpp.

```
00062
    {
00063     auto v1 = findVertex(sourc);
00064     auto v2 = findVertex(dest);
00065     if (v1 == nullptr || v2 == nullptr)
00066         return false;
00067     auto e1 = v1->addEdge(v2, w, service);
00068     auto e2 = v2->addEdge(v1, w, service);
00069     e1->setReverse(e2);
00070     e2->setReverse(e1);
00071     return true;
00072 }
```

4.3.3.2 addEdge()

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

Definition at line 52 of file Graph.cpp.

```
00052                                                                                                {
00053     auto v1 = findVertex(sourc);
00054     auto v2 = findVertex(dest);
00055     if (v1 == nullptr || v2 == nullptr)
00056         return false;
00057     v1->addEdge(v2, w, service);
00058
00059     return true;
00060 }
```

4.3.3.3 addVertex()

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

Definition at line 40 of file Graph.cpp.

```
00040                                                {
00041     if (findVertex(id) != nullptr)
00042         return false;
00043     vertexSet.push_back(new Vertex(id));
00044     return true;
00045 }
```

4.3.3.4 edmondsKarp()

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

Definition at line 192 of file Graph.cpp.

```
00192                                                                {
00193     for (auto e: vertexSet) {
00194         for (auto i: e->getAdj()) {
00195             i->setFlow(0);
00196         }
00197     }
00198     int maxFlow = 0;
00199     while (findAugmentingPath(s, t)) {
00200         int bottleneck = findMinResidual(findVertex(s), findVertex(t));
00201         updateFlow(findVertex(s), findVertex(t), bottleneck);
00202         maxFlow += bottleneck;
00203     }
00204     return maxFlow;
00205 }
```

4.3.3.5 find_sources()

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

Definition at line 207 of file Graph.cpp.

```
00207                                                                                                {
00208     std::vector<std::string> res;
00209     for (std::string s: desired_stations) {
00210         auto v = findVertex(s);
00211         if (v == nullptr) {
00212             std::cout « "Trouble finding source " « s « '\n';
00213             return res;
00214         }
00215         for (auto e: v->getIncoming()) {
00216             if (!isIn(e->getOrig()->getId(), desired_stations)) {
00217                 res.push_back(s);
00218             }
00219         }
00220     }
00221     return res;
00222 }
```

4.3.3.6 find_targets()

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

Definition at line 224 of file Graph.cpp.

```
00224                                                                                {
00225     std::vector<std::string> res;
00226     for (std::string s: desired_stations) {
00227         auto v = findVertex(s);
00228         if (v == nullptr) {
00229             std::cout « "Trouble finding target " « s « '\n';
00230             return res;
00231         }
00232         for (auto e: v->getAdj()) {
00233             if (!isIn(e->getDest()->getId(), desired_stations)) {
00234                 res.push_back(s);
00235             }
00236         }
00237     }
00238     return res;
00239 }
```

4.3.3.7 findAllPaths()

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

Definition at line 290 of file Graph.cpp.

```
00291                                                                                                {
00292     path.push_back(source);
00293     source->setVisited(true);
00294
00295     if (source == destination) {
00296         allPaths.push_back(path);
00297     } else {
00298         for (auto edge: source->getAdj()) {
00299             Vertex *adjacent = edge->getDest();
00300             if (!adjacent->isVisited()) {
00301                 findAllPaths(adjacent, destination, path, allPaths);
00302             }
00303         }
00304     }
00305
00306     path.pop_back();
00307     source->setVisited(false);
00308 }
```

4.3.3.8 findAugmentingPath()

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

Definition at line 130 of file Graph.cpp.

```
00130                                                                                {
00131     Vertex *source = findVertex(s);
00132     Vertex *target = findVertex(t);
00133     if (source == nullptr || target == nullptr) {
00134         return false;
00135     }
00136     for (auto v: vertexSet) {
00137         v->setVisited(false);
00138         v->setPath(nullptr);
00139     }
00140     source->setVisited(true);
00141     std::queue<Vertex *> q;
```

```
00142     q.push(source);
00143     while (!q.empty()) {
00144         auto v = q.front();
00145         q.pop();
00146         for (auto e: v->getAdj()) {
00147             auto w = e->getDest();
00148             double residual = e->getWeight() - e->getFlow();
00149             testAndVisit(q, e, w, residual);
00150         }
00151         for (auto e: v->getIncoming()) {
00152             auto w = e->getDest();
00153             double residual = e->getFlow();
00154             testAndVisit(q, e->getReverse(), w, residual);
00155         }
00156         if (target->isVisited()) {
00157             return true;
00158         }
00159     }
00160     return false;
00161 }
```

4.3.3.9 findEdge()

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

Definition at line 314 of file Graph.cpp.

```
00314                                                       {
00315
00316     for (auto edge: source->getAdj()) {
00317         if (edge->getDest() == destination) {
00318             return edge;
00319         }
00320     }
00321     return nullptr;
00322 }
```

4.3.3.10 findMinResidual()

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

Definition at line 163 of file Graph.cpp.

```
00163                                             {
00164     double minResidual = INT_MAX;
00165     for (auto v = t; v != s;) {
00166         auto e = v->getPath();
00167         if (e->getDest() == v) {
00168             minResidual = std::min(minResidual, e->getWeight() - e->getFlow());
00169             v = e->getOrig();
00170         } else {
00171             minResidual = std::min(minResidual, e->getFlow());
00172             v = e->getDest();
00173         }
00174     }
00175     return minResidual;
00176 }
```

4.3.3.11 findVertex()

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

Definition at line 18 of file Graph.cpp.

```
00018                                                 {
00019     for (auto v: vertexSet) {
00020         if (v->getId() == id)
00021             return v;
00022     }
00023     return nullptr;
00024 }
```

4.3.3.12 findVertexIdx()

```
int Graph::findVertexIdx (
            const std::string & id ) const  [protected]
```

Definition at line 29 of file Graph.cpp.

```
00029                                                {
00030     for (unsigned i = 0; i < vertexSet.size(); i++)
00031         if (vertexSet[i]->getId() == id)
00032             return i;
00033     return -1;
00034 }
```

4.3.3.13 getNumVertex()

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

Definition at line 7 of file Graph.cpp.

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

4.3.3.14 getVertexSet()

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

Definition at line 11 of file Graph.cpp.

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

4.3.3.15 isIn()

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

Definition at line 242 of file Graph.cpp.

```
00242                                                    {
00243     for (std::string s: vec) {
00244         if (s == n) return true;
00245     }
00246     return false;
00247 }
```

4.3.3.16 mul_edmondsKarp()

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

Definition at line 250 of file Graph.cpp.

```
00250                                                                                {
00251     auto it1 = souces.begin();
00252     while (it1 != souces.end()) {
00253         if (isIn(*it1, targets)) {
00254             it1 = souces.erase(it1);
00255         } else it1++;
00256     }
00257
```

```
00258     auto it2 = targets.begin();
00259     while (it2 != targets.end()) {
00260         if (isIn(*it2, souces)) {
00261             it2 = souces.erase(it2);
00262         } else it2++;
00263     }
00264
00265     addVertex("temp_source");
00266     for (std::string s: souces) {
00267         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00268     }
00269
00270     addVertex("temp_targets");
00271     for (std::string s: targets) {
00272         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00273     }
00274     for (auto e: vertexSet) {
00275         for (auto i: e->getAdj()) {
00276             i->setFlow(0);
00277         }
00278     }
00279     int maxFlow = 0;
00280     while (findAugmentingPath("temp_source", "temp_targets")) {
00281         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00282         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00283         maxFlow += bottleneck;
00284     }
00285     return maxFlow;
00286 }
```

4.3.3.17 print()

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

Definition at line 102 of file Graph.cpp.

```
00102                         {
00103     std::cout « "---------------- Graph----------------\n";
00104     std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
00105     std::cout « "Vertices:\n";
00106     for (const auto &vertex: vertexSet) {
00107         std::cout « vertex->getId() « " ";
00108     }
00109     std::cout « "\nEdges:\n";
00110     for (const auto &vertex: vertexSet) {
00111         for (const auto &edge: vertex->getAdj()) {
00112             std::cout « vertex->getId() « " -> " « edge->getDest()->getId() « " (weight: " «
      edge->getWeight() « ", service: " « edge->getService() « ")" « std::endl;
00113         }
00114     }
00115 }
```

4.3.3.18 testAndVisit()

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

Definition at line 119 of file Graph.cpp.

```
00119                                                                                   {
00120     if (!w->isVisited() && residual > 0) {
00121         w->setVisited(true);
00122         w->setPath(e);
00123         q.push(w);
00124     }
00125 }
```

4.3.3.19 updateFlow()

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

Definition at line 178 of file Graph.cpp.

```
00178                                                       {
00179     for (auto v = t; v != s;) {
00180         auto e = v->getPath();
00181         double flow = e->getFlow();
00182         if (e->getDest() == v) {
00183             e->setFlow(flow + bottleneck);
00184             v = e->getOrig();
00185         } else {
00186             e->setFlow(flow - bottleneck);
00187             v = e->getDest();
00188         }
00189     }
00190 }
```

4.3.4 Member Data Documentation

4.3.4.1 distMatrix

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

Definition at line 61 of file Graph.h.

4.3.4.2 pathMatrix

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

Definition at line 62 of file Graph.h.

4.3.4.3 vertexSet

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

Definition at line 59 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
00009 using namespace std;
00010
00014 void CPheadquarters::read_files() {
00015
00016     //-------------------------------------------Read
      network.csv-------------------------------------------
00017     std::ifstream inputFile1(R"(../network.csv)");
00018     string line1;
00019     std::getline(inputFile1, line1); // ignore first line
00020     while (getline(inputFile1, line1, '\n')) {
00021
00022         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00023             line1.pop_back(); // Remove the '\r' character
00024         }
00025
00026         string station_A;
00027         string station_B;
00028         string temp;
00029         int capacity;
00030         string service;
00031
00032         stringstream inputString(line1);
00033
00034         getline(inputString, station_A, ',');
00035         getline(inputString, station_B, ',');
00036         getline(inputString, temp, ',');
00037         getline(inputString, service, ',');
00038
00039         capacity = stoi(temp);
00040         lines.addVertex(station_A);
00041         lines.addVertex(station_B);
00042
00043         lines.addEdge(station_A, station_B, capacity, service);
00044     }
00045
00046
00047     //-------------------------------------------Read
      stations.csv-------------------------------------------
00048     std::ifstream inputFile2(R"(../stations.csv)");
00049     string line2;
00050     std::getline(inputFile2, line2); // ignore first line
00051
00052     while (getline(inputFile2, line2, '\n')) {
00053
00054         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00055             line1.pop_back(); // Remove the '\r' character
00056         }
00057
00058         string nome;
00059         string distrito;
```

```
00060         string municipality;
00061         string township;
00062         string line;
00063 
00064         stringstream inputString(line2);
00065 
00066         getline(inputString, nome, ',');
00067         getline(inputString, distrito, ',');
00068         getline(inputString, municipality, ',');
00069         getline(inputString, township, ',');
00070         getline(inputString, line, ',');
00071 
00072         Station station(nome, distrito, municipality, township, line);
00073         stations[nome] = station;
00074 
00075         // print information about the station, to make sure it was imported correctly
00076         //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « "
      township: " « township « " line: " « line « endl;
00077     }
00078 }
00079 
00080 Graph CPheadquarters::getLines() const {
00081     return this->lines;
00082 }
00083 
00092 int CPheadquarters::T2_1maxflow(string stationA, string stationB) {
00093     Vertex *source = lines.findVertex(stationA); // set source vertex
00094     Vertex *sink = lines.findVertex(stationB); // set sink vertex
00095 
00096     // Check if these stations even exist
00097     if (source == nullptr || sink == nullptr) {
00098         std::cerr « "Source or sink vertex not found." « std::endl;
00099         return 1;
00100     }
00101     int maxFlow = lines.edmondsKarp(stationA, stationB);
00102 
00103     if (maxFlow == 0) {
00104         cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " «
      stationA
00105              « endl;
00106     } else {
00107         cout « "maxFlow:\t" « maxFlow « endl;
00108     }
00109 
00110     return 1;
00111 }
00112 
00113 void CPheadquarters::test() {
00114     int flow = lines.edmondsKarp(lines.getVertexSet()[324]->getId(),
      lines.getVertexSet()[507]->getId());
00115 }
00116 
00117 
00123 int CPheadquarters::T2_2maxflowAllStations() {
00124     vector<string> stations;
00125     int maxFlow = 0;
00126     auto length = lines.getVertexSet().size();
00127     for (int i = 0; i < length; ++i) {
00128         for (int j = i + 1; j < length; ++j) {
00129             string stationA = lines.getVertexSet()[i]->getId();
00130             string stationB = lines.getVertexSet()[j]->getId();
00131             int flow = lines.edmondsKarp(stationA, stationB);
00132             if (flow == maxFlow) {
00133                 stations.push_back(stationB);
00134                 stations.push_back(stationA);
00135             } else if (flow > maxFlow) {
00136                 stations.clear();
00137                 stations.push_back(stationB);
00138                 stations.push_back(stationA);
00139                 maxFlow = flow;
00140             }
00141         }
00142     }
00143     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00144     for (int i = 0; i < stations.size(); i = i + 2) {
00145         cout « "------------------------\n";
00146         cout « "Source:" « stations[i + 1] « '\n';
00147         cout « "Target:" « stations[i] « '\n';
00148         cout « "------------------------\n";
00149     }
00150     return 0;
00151 }
00152 
00160 int CPheadquarters::T2_3municipality(string municipality) {
00161     vector<string> desired_stations;
00162     for (auto p: stations) {
00163         if (p.second.get_municipality() == municipality) {
```

```
00164             desired_stations.push_back(p.second.get_name());
00165         }
00166     }
00167     vector<string> souces = lines.find_sources(desired_stations);
00168     vector<string> targets = lines.find_targets(desired_stations);
00169     return lines.mul_edmondsKarp(souces, targets);
00170 }
00171
00172 int CPheadquarters::T2_3district(string district) {
00173     vector<string> desired_stations;
00174     for (auto p: stations) {
00175         if (p.second.get_district() == district) {
00176             desired_stations.push_back(p.second.get_name());
00177         }
00178     }
00179     return lines.mul_edmondsKarp(lines.find_sources(desired_stations),
      lines.find_targets(desired_stations));
00180 }
00181
00188 int CPheadquarters::T2_4maxArrive(string destination) {
00189     Vertex *dest = lines.findVertex(destination);
00190     int maxFlow = 0;
00191
00192     // iterate over all vertices to find incoming and outgoing vertices
00193     for (auto &v: lines.getVertexSet()) {
00194         if (v != dest) {
00195
00196             int flow = lines.edmondsKarp(v->getId(), destination);
00197
00198             // Update the maximum flow if this vertex contributes to a higher maximum
00199             if (flow > maxFlow) {
00200                 maxFlow = flow;
00201             }
00202         }
00203
00204     }
00205
00206     cout « endl;
00207     for (auto &e: dest->getIncoming()) {
00208         cout « e->getOrig()->getId() « " -> " « e->getDest()->getId() « " : " « e->getWeight() « endl;
00209
00210     }
00211
00212     cout « "Max number of trains that can simultaneously arrive at " « destination « ": " « maxFlow «
      endl;
00213     return maxFlow;
00214
00215 }
00216
00217
00232 int CPheadquarters::T3_1MinCost(string source, string destination) {
00233     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00234     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00235     if (sourceVertex == nullptr || destVertex == nullptr) {
00236         cerr « "Source or destination vertex not found. Try again" « endl;
00237         return 1;
00238     }
00239
00240     Graph graph = lines;
00241
00242     std::vector<Vertex *> path;
00243     std::vector<std::vector<Vertex *» allPaths;
00244
00245
00246     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00247
00248     vector<int> maxFlows;
00249     vector<int> totalCosts;
00250
00251     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00252     for (auto path: allPaths) {
00253         int minWeight = 10;
00254         int totalCost = 0; // total cost of this path
00255         for (int i = 0; i + 1 < path.size(); i++) {
00256             std::cout « path[i]->getId() « " -> ";
00257             Edge *e = graph.findEdge(path[i], path[i + 1]);
00258             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00259             if (e->getWeight() < minWeight) {
00260                 minWeight = e->getWeight();
00261             }
00262
00263             // according to the problem's specification, the cost of STANDARD service is 2 euros and
      ALFA PENDULAR is 4
00264             if (e->getService() == "STANDARD") {
00265                 totalCost += 2;
00266             } else if (e->getService() == "ALFA PENDULAR") {
00267                 totalCost += 4;
```

```
00268             }
00269         }
00270         maxFlows.push_back(minWeight);
00271         totalCosts.push_back(totalCost);
00272         cout « " -> " « path[path.size() - 1]->getId() « endl;
00273         cout « "Max flow for this path: " « minWeight « " trains. ";
00274         cout « "Total cost: " « totalCost « " euros." « endl;
00275         std::cout « std::endl;
00276     }
00277
00278     // find the path with the minimum cost per train
00279     int maxTrains = 0;
00280     int resCost;
00281     double max_value = 10000;
00282     for (int i = 0; i < maxFlows.size(); ++i) {
00283         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00284         if (costPerTrain < max_value) {
00285             max_value = costPerTrain;
00286             maxTrains = maxFlows[i];
00287             resCost = totalCosts[i];
00288         }
00289     }
00290
00291     cout « "Max number of trains that can travel between " « source « " and " « destination
00292          « " with minimum cost"
00293          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00294     return maxTrains;
00295 }
00306 int CPheadquarters::T4_1ReducedConectivity(std::vector<std::string> unwantedEdges, std::string s,
    std::string t) {
00307     Graph graph;
00308     ifstream inputFile1;
00309     inputFile1.open(R"(../network.csv)");
00310     string line1;
00311
00312     getline(inputFile1, line1);
00313     line1 = "";
00314
00315     while (getline(inputFile1, line1)) {
00316         string station_A;
00317         string station_B;
00318         string temp;
00319         int capacity;
00320         string service;
00321         bool flag = true;
00322
00323         stringstream inputString(line1);
00324
00325         getline(inputString, station_A, ';');
00326         getline(inputString, station_B, ';');
00327         getline(inputString, temp, ';');
00328         capacity = stoi(temp);
00329         getline(inputString, service, ';');
00330
00331         graph.addVertex(station_A);
00332         graph.addVertex(station_B);
00333         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00334             if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00335                 flag=false;
00336
00337             }
00338         }
00339         if(flag) {
00340             graph.addEdge(station_A, station_B, capacity, service);
00341         }
00342         line1 = "";
00343     }
00344
00345     Vertex *source = graph.findVertex(s); // set source vertex
00346     Vertex *sink = graph.findVertex(t); // set sink vertex
00347
00348     // Check if these stations even exist
00349     if (source == nullptr || sink == nullptr) {
00350         std::cerr « "Source or sink vertex not found." « std::endl;
00351         return 1;
00352     }
00353     int maxFlow = graph.edmondsKarp(s, t);
00354
00355     if (maxFlow == 0) {
00356         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00357              « endl;
00358     }
00359     cout « "maxFlow:\t" « maxFlow « endl;
00360
00361
00362     return 1;
00363 }
```

```
00364
00371 int CPheadquarters::T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges) {
00372     Graph graph;
00373     ifstream inputFile1;
00374     inputFile1.open(R"(../network.csv)");
00375     string line1;
00376
00377     getline(inputFile1, line1);
00378     line1 = "";
00379
00380     while (getline(inputFile1, line1)) {
00381         string station_A;
00382         string station_B;
00383         string temp;
00384         int capacity;
00385         string service;
00386         bool flag = true;
00387
00388         stringstream inputString(line1);
00389
00390         getline(inputString, station_A, ';');
00391         getline(inputString, station_B, ';');
00392         getline(inputString, temp, ';');
00393         capacity = stoi(temp);
00394         getline(inputString, service, ';');
00395
00396         graph.addVertex(station_A);
00397         graph.addVertex(station_B);
00398         for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
00399             if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
00400                 flag=false;
00401                 break;
00402             }
00403         }
00404         if(flag) {
00405             graph.addEdge(station_A, station_B, capacity, service);
00406         }
00407         line1 = "";
00408     }
00409
00410
00411     vector<pair<int, int» top_k;
00412     auto length = lines.getVertexSet().size();
00413     for (int i = 0; i < length; ++i) {
00414         string destination = lines.getVertexSet()[i]->getId();
00415         Vertex *dest = lines.findVertex(destination);
00416
00417         int maxFlow1 = 0;
00418         int maxFlow2 = 0;
00419
00420         for (auto &v: lines.getVertexSet()) {
00421             if (v != dest) {
00422                 int flow = lines.edmondsKarp(v->getId(), destination);
00423                 if (flow > maxFlow1) {
00424                     maxFlow1 = flow;
00425                 }
00426                 flow = graph.edmondsKarp(v->getId(), destination);
00427                 if (flow > maxFlow2) {
00428                     maxFlow2 = flow;
00429                 }
00430             }
00431         }
00432
00433         int diff = maxFlow1 - maxFlow2;
00434         auto p = pair(i,diff);
00435         top_k.push_back(p);
00436         cout « "a";
00437     }
00438     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00439         return left.second > right.second;
00440     });
00441     for(int i = 0; i < 10; i++){
00442         cout « i+1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second «
    '\n';
00443     }
00444     return 1;
00445 }
```

5.2 CPheadquarters.h

```
00001 //
00002 // Created by Pedro on 23/03/2023.
00003 //
```

```
00004
00005 #ifndef DAPROJECT_CPHEADQUARTERS_H
00006 #define DAPROJECT_CPHEADQUARTERS_H
00007
00008
00009 #include "Graph.h"
00010 #include "Station.h"
00011
00012 using namespace std;
00013
00014 class CPheadquarters {
00015     Graph lines;
00016     unordered_map<string, Station> stations;
00017 public:
00018     void read_files();
00019
00020     Graph getLines() const;
00021
00022     int T2_1maxflow(string station_A, string station_B);
00023
00024     int T2_2maxflowAllStations();
00025
00026     int T2_3municipality(string municipality);
00027
00028     int T2_3district(string district);
00029
00030     int T2_4maxArrive(string destination);
00031
00032     int T3_1MinCost(string source, string destination);
00033
00034     int T4_1ReducedConectivity(vector<string> unwantedEdges, string s, string t);
00035
00036     int T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges);
00037
00038     void test();
00039
00040
00041
00042 };
00043
00044
00045 #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  * Auxiliary function to find a vertex with a given content.
00017  */
00018 Vertex *Graph::findVertex(const std::string &id) const {
00019     for (auto v: vertexSet) {
00020         if (v->getId() == id)
00021             return v;
00022     }
00023     return nullptr;
00024 }
00025
00026 /*
00027  * Finds the index of the vertex with a given content.
00028  */
00029 int Graph::findVertexIdx(const std::string &id) const {
00030     for (unsigned i = 0; i < vertexSet.size(); i++)
00031         if (vertexSet[i]->getId() == id)
00032             return i;
00033     return -1;
00034 }
00035
00036 /*
00037  *  Adds a vertex with a given content or info (in) to a graph (this).
00038  *  Returns true if successful, and false if a vertex with that content already exists.
```

```
00039  */
00040 bool Graph::addVertex(const std::string &id) {
00041     if (findVertex(id) != nullptr)
00042         return false;
00043     vertexSet.push_back(new Vertex(id));
00044     return true;
00045 }
00046
00047 /*
00048  * Adds an edge to a graph (this), given the contents of the source and
00049  * destination vertices and the edge weight (w).
00050  * Returns true if successful, and false if the source or destination vertex does not exist.
00051  */
00052 bool Graph::addEdge(const std::string &sourc, const std::string &dest, int w, const std::string
      &service) {
00053     auto v1 = findVertex(sourc);
00054     auto v2 = findVertex(dest);
00055     if (v1 == nullptr || v2 == nullptr)
00056         return false;
00057     v1->addEdge(v2, w, service);
00058
00059     return true;
00060 }
00061
00062 bool Graph::addBidirectionalEdge(const std::string &sourc, const std::string &dest, int w, std::string
      service) {
00063     auto v1 = findVertex(sourc);
00064     auto v2 = findVertex(dest);
00065     if (v1 == nullptr || v2 == nullptr)
00066         return false;
00067     auto e1 = v1->addEdge(v2, w, service);
00068     auto e2 = v2->addEdge(v1, w, service);
00069     e1->setReverse(e2);
00070     e2->setReverse(e1);
00071     return true;
00072 }
00073
00074
00075 void deleteMatrix(int **m, int n) {
00076     if (m != nullptr) {
00077         for (int i = 0; i < n; i++)
00078             if (m[i] != nullptr)
00079                 delete[] m[i];
00080         delete[] m;
00081     }
00082 }
00083
00084 void deleteMatrix(double **m, int n) {
00085     if (m != nullptr) {
00086         for (int i = 0; i < n; i++)
00087             if (m[i] != nullptr)
00088                 delete[] m[i];
00089         delete[] m;
00090     }
00091 }
00092
00093 Graph::~Graph() {
00094     deleteMatrix(distMatrix, vertexSet.size());
00095     deleteMatrix(pathMatrix, vertexSet.size());
00096 }
00097
00098
00099 /*
00100  *  print graph content
00101  */
00102 void Graph::print() const {
00103     std::cout « "---------------- Graph----------------\n";
00104     std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
00105     std::cout « "Vertices:\n";
00106     for (const auto &vertex: vertexSet) {
00107         std::cout « vertex->getId() « " ";
00108     }
00109     std::cout « "\nEdges:\n";
00110     for (const auto &vertex: vertexSet) {
00111         for (const auto &edge: vertex->getAdj()) {
00112             std::cout « vertex->getId() « " -> " « edge->getDest()->getId() « " (weight: " «
      edge->getWeight() « ", service: " « edge->getService() « ")" « std::endl;
00113         }
00114     }
00115 }
00116
00117 // ---------------------------------- Edmonds-Karp ---------------------------------
00118
00119 void Graph::testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual) {
00120     if (!w->isVisited() && residual > 0) {
00121         w->setVisited(true);
00122         w->setPath(e);
```

```
00123         q.push(w);
00124     }
00125 }
00126
00127 /*
00128  * An augmenting path is a simple path - a path that does not contain cycles
00129  */
00130 bool Graph::findAugmentingPath(const std::string &s, const std::string &t) {
00131     Vertex *source = findVertex(s);
00132     Vertex *target = findVertex(t);
00133     if (source == nullptr || target == nullptr) {
00134         return false;
00135     }
00136     for (auto v: vertexSet) {
00137         v->setVisited(false);
00138         v->setPath(nullptr);
00139     }
00140     source->setVisited(true);
00141     std::queue<Vertex *> q;
00142     q.push(source);
00143     while (!q.empty()) {
00144         auto v = q.front();
00145         q.pop();
00146         for (auto e: v->getAdj()) {
00147             auto w = e->getDest();
00148             double residual = e->getWeight() - e->getFlow();
00149             testAndVisit(q, e, w, residual);
00150         }
00151         for (auto e: v->getIncoming()) {
00152             auto w = e->getDest();
00153             double residual = e->getFlow();
00154             testAndVisit(q, e->getReverse(), w, residual);
00155         }
00156         if (target->isVisited()) {
00157             return true;
00158         }
00159     }
00160     return false;
00161 }
00162
00163 int Graph::findMinResidual(Vertex *s, Vertex *t) {
00164     double minResidual = INT_MAX;
00165     for (auto v = t; v != s;) {
00166         auto e = v->getPath();
00167         if (e->getDest() == v) {
00168             minResidual = std::min(minResidual, e->getWeight() - e->getFlow());
00169             v = e->getOrig();
00170         } else {
00171             minResidual = std::min(minResidual, e->getFlow());
00172             v = e->getDest();
00173         }
00174     }
00175     return minResidual;
00176 }
00177
00178 void Graph::updateFlow(Vertex *s, Vertex *t, int bottleneck) {
00179     for (auto v = t; v != s;) {
00180         auto e = v->getPath();
00181         double flow = e->getFlow();
00182         if (e->getDest() == v) {
00183             e->setFlow(flow + bottleneck);
00184             v = e->getOrig();
00185         } else {
00186             e->setFlow(flow - bottleneck);
00187             v = e->getDest();
00188         }
00189     }
00190 }
00191
00192 int Graph::edmondsKarp(const std::string &s, const std::string &t) {
00193     for (auto e: vertexSet) {
00194         for (auto i: e->getAdj()) {
00195             i->setFlow(0);
00196         }
00197     }
00198     int maxFlow = 0;
00199     while (findAugmentingPath(s, t)) {
00200         int bottleneck = findMinResidual(findVertex(s), findVertex(t));
00201         updateFlow(findVertex(s), findVertex(t), bottleneck);
00202         maxFlow += bottleneck;
00203     }
00204     return maxFlow;
00205 }
00206
00207 std::vector<std::string> Graph::find_sources(std::vector<std::string> desired_stations) {
00208     std::vector<std::string> res;
00209     for (std::string s: desired_stations) {
```

```
00210         auto v = findVertex(s);
00211         if (v == nullptr) {
00212             std::cout « "Trouble finding source " « s « '\n';
00213             return res;
00214         }
00215         for (auto e: v->getIncoming()) {
00216             if (!isIn(e->getOrig()->getId(), desired_stations)) {
00217                 res.push_back(s);
00218             }
00219         }
00220     }
00221     return res;
00222 }
00223
00224 std::vector<std::string> Graph::find_targets(std::vector<std::string> desired_stations) {
00225     std::vector<std::string> res;
00226     for (std::string s: desired_stations) {
00227         auto v = findVertex(s);
00228         if (v == nullptr) {
00229             std::cout « "Trouble finding target " « s « '\n';
00230             return res;
00231         }
00232         for (auto e: v->getAdj()) {
00233             if (!isIn(e->getDest()->getId(), desired_stations)) {
00234                 res.push_back(s);
00235             }
00236         }
00237     }
00238     return res;
00239 }
00240
00241
00242 bool Graph::isIn(std::string n, std::vector<std::string> vec) {
00243     for (std::string s: vec) {
00244         if (s == n) return true;
00245     }
00246     return false;
00247 }
00248
00249
00250 int Graph::mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets) {
00251     auto it1 = souces.begin();
00252     while (it1 != souces.end()) {
00253         if (isIn(*it1, targets)) {
00254             it1 = souces.erase(it1);
00255         } else it1++;
00256     }
00257
00258     auto it2 = targets.begin();
00259     while (it2 != targets.end()) {
00260         if (isIn(*it2, souces)) {
00261             it2 = souces.erase(it2);
00262         } else it2++;
00263     }
00264
00265     addVertex("temp_source");
00266     for (std::string s: souces) {
00267         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00268     }
00269
00270     addVertex("temp_targets");
00271     for (std::string s: targets) {
00272         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00273     }
00274     for (auto e: vertexSet) {
00275         for (auto i: e->getAdj()) {
00276             i->setFlow(0);
00277         }
00278     }
00279     int maxFlow = 0;
00280     while (findAugmentingPath("temp_source", "temp_targets")) {
00281         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00282         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00283         maxFlow += bottleneck;
00284     }
00285     return maxFlow;
00286 }
00287
00288 // ---------------------------------- Find ALL existing augmenting paths
    ----------------------------------
00289
00290 void Graph::findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00291                          std::vector<std::vector<Vertex *» &allPaths) {
00292     path.push_back(source);
00293     source->setVisited(true);
00294
00295     if (source == destination) {
```

```
00296         allPaths.push_back(path);
00297     } else {
00298         for (auto edge: source->getAdj()) {
00299             Vertex *adjacent = edge->getDest();
00300             if (!adjacent->isVisited()) {
00301                 findAllPaths(adjacent, destination, path, allPaths);
00302             }
00303         }
00304     }
00305
00306     path.pop_back();
00307     source->setVisited(false);
00308 }
00309
00310
00311 /*
00312  * find edge based on source and destination
00313  */
00314 Edge *Graph::findEdge(Vertex *source, Vertex *destination) {
00315
00316     for (auto edge: source->getAdj()) {
00317         if (edge->getDest() == destination) {
00318             return edge;
00319         }
00320     }
00321     return nullptr;
00322 }
00323
00324
00325
00326 // ------------------------------------------- find all stations that have more than one path to
    the destination ---------------------------------------------
```

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
00019     /*
00020     * Auxiliary function to find a vertex with a given ID.
00021     */
00022     Vertex *findVertex(const std::string &id) const;
00023
00024     /*
00025      *  Adds a vertex with a given content or info (in) to a graph (this).
00026      *  Returns true if successful, and false if a vertex with that content already exists.
00027      */
00028     bool addVertex(const std::string &id);
00029
00030     /*
00031      * Adds an edge to a graph (this), given the contents of the source and
00032      * destination vertices and the edge weight (w).
00033      * Returns true if successful, and false if the source or destination vertex does not exist.
00034      */
00035     bool addEdge(const std::string &sourc, const std::string &dest, int w, const std::string
    &service);
00036
00037     bool addBidirectionalEdge(const std::string &sourc, const std::string &dest, int w, std::string
    service);
00038
00039     [[nodiscard]] int getNumVertex() const;
00040
00041     [[nodiscard]] std::vector<Vertex *> getVertexSet() const;
00042
00043     void print() const;
00044
00045     int edmondsKarp(const std::string &s, const std::string &t);
00046
```

```
00047     int mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets);
00048
00049     std::vector<std::string> find_sources(std::vector<std::string> desired_stations);
00050
00051     std::vector<std::string> find_targets(std::vector<std::string> desired_stations);
00052
00053     void findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
00054                       std::vector<std::vector<Vertex *>> &allPaths);
00055
00056     Edge *findEdge(Vertex *source, Vertex *destination);
00057
00058 protected:
00059     std::vector<Vertex *> vertexSet;    // vertex set
00060
00061     double **distMatrix = nullptr;   // dist matrix for Floyd-Warshall
00062     int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall
00063
00064     /*
00065      * Finds the index of the vertex with a given content.
00066      */
00067     int findVertexIdx(const std::string &id) const;
00068
00069
00070     void updateFlow(Vertex *s, Vertex *t, int bottleneck);
00071
00072     int findMinResidual(Vertex *s, Vertex *t);
00073
00074     bool findAugmentingPath(const std::string &s, const std::string &t);
00075
00076     void testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual);
00077
00078     bool isIn(std::string n, std::vector<std::string> vec);
00079
00080
00081 };
00082
00083 void deleteMatrix(int **m, int n);
00084
00085 void deleteMatrix(double **m, int n);
00086
00087 #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 « "Enter station A: ";
00043                 getline(cin, a);
00044 
00045                 cout « "Enter station B: ";
00046                 getline(cin, b);
00047 
00048                 if (a.empty() || b.empty()) {
00049                     cerr « "Error: Station names cannot be empty." « endl;
00050                     break;
00051                 }
00052 
00053                 // call function to calculate max flow between stations A and B
00054                 CP.T2_1maxflow(a, b);
00055                 break;
00056             }
00057 
00058             case 2: {
00059                 CP.T2_2maxflowAllStations();
00060                 break;
00061             }
00062 
00063             case 3: {
00064                 cin.ignore();
00065                 string c;
00066                 cout « "Enter municipality: " « endl;
00067                 cout « "For example, PENAFIEL: ";
00068                 getline(cin, c);
00069                 cout « "The maximum flow im Municipality " « c « " is " « CP.T2_3municipality(c) «
    endl;
00070                 break;
00071             }
00072 
00073             case 4: {
00074                 cin.ignore();
00075                 string destination;
00076                 cout « "Enter destination: ";
00077                 getline(cin, destination);
00078                 CP.T2_4maxArrive(destination);
00079                 break;
00080             }
00081 
00082             case 5: {
00083                 cin.ignore();
00084                 string a, b;
00085                 cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
00086                 cout « "Enter station A: ";
00087                 getline(cin, a);
00088                 cout « endl;
00089                 cout « "Enter station B: ";
00090                 getline(cin, b);
00091 
00092                 if (a.empty() || b.empty()) {
00093                     cerr « "Error: Station names cannot be empty." « endl;
00094                     break;
00095                 }
00096 
00097                 CP.T3_1MinCost(a, b);
00098                 break;
00099             }
00100 
00101             case 6: {
00102                 cin.ignore();
00103                 vector<string> unwantedEdges;
00104                 string edgesource;
00105                 string edgetarget;
00106                 string b;
00107                 string a;
00108                 cout « "Enter station A: ";
00109                 getline(cin, a);
00110                 cout « "Enter station B: ";
00111                 getline(cin, b);
00112                 cout « '\n';
00113                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
    Type '.' to end listing: \n";
00114                 while (1){
00115                     cout « "Enter edge source or '.' to finish: ";
00116                     getline(cin, edgesource);
00117                     if(edgesource==".") break;
00118                     unwantedEdges.push_back(edgesource);
00119                     cout « "Enter edge target: ";
00120                     getline(cin, edgetarget);
00121                     unwantedEdges.push_back(edgetarget);
00122                 }
00123                 CP.T4_1ReducedConectivity(unwantedEdges,a,b);
```

```
00124             }
00125
00126             case 7: {
00127                 cin.ignore();
00128                 vector<string> unwantedEdges;
00129                 string edgesource;
00130                 string edgetarget;
00131                 cout « "List unwanted edges. Start by typing the edge source an then the edge destine.
      Type '.' to end listing: \n";
00132                 while (1){
00133                     cout « "Enter edge source or '.' to finish: ";
00134                     getline(cin, edgesource);
00135                     if(edgesource==".") break;
00136                     unwantedEdges.push_back(edgesource);
00137                     cout « "Enter edge target: ";
00138                     getline(cin, edgetarget);
00139                     unwantedEdges.push_back(edgetarget);
00140                 }
00141                 CP.T4_2Top_K_ReducedConectivity(unwantedEdges);
00142
00143                 break;
00144             }
00145
00146             case 8: {
00147                 cout « "Exiting program..." « endl;
00148                 break;
00149             }
00150
00151             default: {
00152                 cerr « "Error: Invalid option selected." « endl;
00153                 break;
00154             }
00155         }
00156     } while (n != 8);
00157
00158     return 0;
00159 }
```

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     int capacity;
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