An Analysis Tool for Railway Network Management

1.0

Generated by Doxygen 1.9.7

Chapter 1

DAproject

1.1 Deadline is April 7, 2023 at midnight

1.1.1 Checklist

- [T1.1: 1.0 point] Obviously, a first task will be to create a simple interface menu exposing all the functionalities implemented in the most user-friendly way possible. This menu will also be instrumental for you to showcase the work you have developed in a short demo to be held at the end of the project.
- [T1.2: 1.0 point] Similarly, you will also have to develop some basic functionality (accessible through your menu) to read and parse the provided data set files. This functionality will enable you (and the eventual user) to select alternative railway networks for analysis. With the extracted information, you are to create one (or more) appropriate graphs upon which you will carry out the requested tasks. The modelling of the graph is entirely up to you, so long as it is a sensible representation of the railway network and enables the correct application of the required algorithms.
- [T1.3: 2.0 points] In addition, you should also include documentation of all the implemented code, using Doxygen, indicating for each implemented algorithm the corresponding time complexity
- [T2.1: 3.5 points] :heavy_check_mark: Calculate the maximum number of trains that can simultaneously travel between two specific stations. Note that your implementation should take any valid source and destination stations as input;
- [T2.2: 2.0 points] :heavy_check_mark: Determine, from all pairs of stations, which ones (if more than one) require the most amount of trains when taking full advantage of the existing network capacity;
- [T2.3: 1.5 points] Indicate where management should assign larger budgets for the purchasing and maintenance of trains. That is, your implementation should be able to report the top-k municipalities and districts, regarding their transportation needs;
- [T2.4: 1 point] :heavy_check_mark: Report the maximum number of trains that can simultaneously arrive at a given station, taking into consideration the entire railway grid
- [T3.1: 2.0 points] Calculate the maximum amount of trains that can simultaneously travel between two specific stations with minimum cost for the company. Note that your system should also take any valid source and destination stations as input;
- [T4.1: 2.5 points] Calculate the maximum number of trains that can simultaneously travel between two specific stations in a network of reduced connectivity. Reduced connectivity is understood as being a subgraph (generated by your system) of the original railway network. Note that your system should also take any valid source and destination stations as input;

- [T4.2: 1.5 points] Provide a report on the stations that are the most affected by each segment failure, i.e., the top-k most affected stations for each segment to be considered
- [T5.1: 2.0 points] Use the (hopefully) user-friendly interface you have developed to illustrate the various algorithm results for a sample set of railway grids which you should develop specifically for the purposes of this demo. For instance, you can develop a small set of very modest railway networks for contrived capacities so that you can highlight the "correctness" of your solution. For instance, a grid that has a "constricted" segment where all traffic must go through, will clearly have a segment very "sensitive" to failures.

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Class Documentation

4.1 CPheadquarters Class Reference

Public Member Functions

- void read_network (string path)

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

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

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

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

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

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

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

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

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

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

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

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

4.1.1 Detailed Description

Definition at line 14 of file CPheadquarters.h.

4.1.2 Member Function Documentation

4.1.2.1 getLines()

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

Returns the Graph object.

Returns

Graph

Definition at line 144 of file CPheadquarters.cpp.

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

4.1.2.2 read_files()

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

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

Definition at line 77 of file CPheadquarters.cpp.

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

4.1.2.3 read_network()

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

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

Parameters

path	

Definition at line 13 of file CPheadquarters.cpp.

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

4.1.2.4 read_stations()

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

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

Parameters

path	

Definition at line 44 of file CPheadquarters.cpp.

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

4.1.2.5 T2_1maxflow()

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

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

Takes any valid source and destination stations as input

Parameters

stationA	
stationB	

Returns

maxFlow

Definition at line 149 of file CPheadquarters.cpp.

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

4.1.2.6 T2_2maxflowAllStations()

`int CPheadquarters::T2_2maxflowAllStations ( )`

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

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

See also

this function uses Graph::edmondsKarp() function

Returns

maxFlow

Definition at line 171 of file CPheadquarters.cpp.

```
00171                                          {
00172     vector<string> stations;
00173     int maxFlow = 0;
00174     auto length = lines.getVertexSet().size();
00175     // Start the timer
00176     auto start_time = std::chrono::high_resolution_clock::now();
00177     cout « "Calculating max flow for all pairs of stations..." « endl;
00178     cout « "Please stand by..." « endl;
00179     for (int i = 0; i < length; ++i) {
00180         for (int j = i + 1; j < length; ++j) {
00181             string stationA = lines.getVertexSet()[i]->getId();
00182             string stationB = lines.getVertexSet()[j]->getId();
00183             int flow = lines.edmondsKarp(stationA, stationB);
00184             if (flow == maxFlow) {
00185                 stations.push_back(stationB);
00186                 stations.push_back(stationA);
00187             } else if (flow > maxFlow) {
00188                 stations.clear();
00189                 stations.push_back(stationB);
00190                 stations.push_back(stationA);
00191                 maxFlow = flow;
00192             }
00193         }
00194     }
00195     // End the timer
00196     auto end_time = std::chrono::high_resolution_clock::now();
00197 
00198     // Compute the duration
00199     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00200 
00201     // Print the duration
00202     std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
00203 
00204     cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
00205     for (int i = 0; i < stations.size(); i = i + 2) {
00206         cout « "-------------------------\n";
00207         cout « "Source: " « stations[i + 1] « '\n';
00208         cout « "Target: " « stations[i] « '\n';
00209         cout « "-------------------------\n";
00210     }
00211     return maxFlow;
00212 }
```

4.1.2.7 T2_3district()

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

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

Reports the top-k districts, regarding their transportation needs

Definition at line 244 of file CPheadquarters.cpp.

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

4.1.2.8 T2_3municipality()

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

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

Reports the top-k municipalities, regarding their transportation needs

Definition at line 215 of file CPheadquarters.cpp.

```
00215                                            {
00216     vector<pair<string , int» top_k;
00217     set<string> sett;
00218     for (auto m : stations) {
00219         sett.insert(m.second.get_district());
00220     }
00221     for (auto m : sett) {
00222         vector<string> desired_stations;
00223         for (auto p: stations) {
00224             if (p.second.get_municipality() == m) {
00225                 desired_stations.push_back(p.second.get_name());
00226             }
00227         }
00228 
00229
00230         vector<string> souces = lines.find_sources(desired_stations);
00231         vector<string> targets = lines.find_targets(desired_stations);
00232         int diff=lines.mul_edmondsKarp(souces, targets);
00233         auto p = pair(m, diff);
00234         top_k.push_back(p);
00235     }
00236     std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
00237         return left.second > right.second;
00238     });
00239     for (int i = 0; i < 10; i++) {
00240         cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
00241     }
00242 }
```

4.1.2.9 T2_4maxArrive()

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

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

Parameters

destination	

Returns

maximum flow in a given station

Note

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

Definition at line 272 of file CPheadquarters.cpp.

```
00272                                                  {
00273     Vertex *dest = lines.findVertex(destination);
00274     int maxFlow = 0;
00275
00276     // iterate over all vertices to find incoming and outgoing vertices
00277     for (auto &v: lines.getVertexSet()) {
00278         if (v != dest) {
00279
00280             int flow = lines.edmondsKarp(v->getId(), destination);
00281
00282             // Update the maximum flow if this vertex contributes to a higher maximum
00283             if (flow > maxFlow) {
00284                 maxFlow = flow;
00285             }
00286         }
00287
00288     }
00289
00290     cout « endl;
00291     for (auto &e: dest->getIncoming()) {
00292         cout « e->getOrig()->getId() « " -> " « e->getDest()->getId() « " : " « e->getWeight() « endl;
00293
00294     }
00295
00296     cout « "Max number of trains that can simultaneously arrive at " « destination « ": " « maxFlow «
      endl;
00297     return maxFlow;
00298
00299 }
```

4.1.2.10 T3_1MinCost()

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

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

Parameters

source	
destination	

Returns

maximum flow between two specific stations

Definition at line 303 of file CPheadquarters.cpp.

```
00303                                                                      {
00304     Vertex *sourceVertex = lines.findVertex(source); // set source vertex
00305     Vertex *destVertex = lines.findVertex(destination); // set sink vertex
00306     if (sourceVertex == nullptr || destVertex == nullptr) {
00307         cerr « "Source or destination vertex not found. Try again" « endl;
00308         return 1;
00309     }
00310 
00311     Graph graph = lines;
00312 
00313     std::vector<Vertex *> path;
00314     std::vector<std::vector<Vertex *» allPaths;
00315 
00316 
00317     graph.findAllPaths(sourceVertex, destVertex, path, allPaths);
00318 
00319     vector<int> maxFlows;
00320     vector<int> totalCosts;
00321 
00322     cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
00323     for (auto path: allPaths) {
00324         int minWeight = 10;
00325         int totalCost = 0; // total cost of this path
00326         for (int i = 0; i + 1 < path.size(); i++) {
00327             std::cout « path[i]->getId() « " -> ";
00328             Edge *e = graph.findEdge(path[i], path[i + 1]);
00329             cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
00330             if (e->getWeight() < minWeight) {
00331                 minWeight = e->getWeight();
00332             }
00333 
00334             // according to the problem's specification, the cost of STANDARD service is 2 euros and
      ALFA PENDULAR is 4
00335             if (e->getService() == "STANDARD") {
00336                 totalCost += 2;
00337             } else if (e->getService() == "ALFA PENDULAR") {
00338                 totalCost += 4;
00339             }
00340         }
00341         maxFlows.push_back(minWeight);
00342         totalCosts.push_back(totalCost);
00343         cout « " -> " « path[path.size() - 1]->getId() « endl;
00344         cout « "Max flow for this path: " « minWeight « " trains. ";
00345         cout « "Total cost: " « totalCost « " euros." « endl;
00346         std::cout « std::endl;
00347     }
00348 
00349     // find the path with the minimum cost per train
00350     int maxTrains = 0;
00351     int resCost;
00352     double max_value = 10000;
00353     for (int i = 0; i < maxFlows.size(); ++i) {
00354         double costPerTrain = (double) totalCosts[i] / maxFlows[i];
00355         if (costPerTrain < max_value) {
```

```
00356             max_value = costPerTrain;
00357             maxTrains = maxFlows[i];
00358             resCost = totalCosts[i];
00359         }
00360     }
00361
00362     cout « "Max number of trains that can travel between " « source « " and " « destination
00363          « " with minimum cost"
00364          « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
00365     return maxTrains;
00366 }
```

4.1.2.11 T4_1ReducedConectivity()

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

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

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

Note

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

Parameters

unwantedEdges	
s	
t	

Returns

maximum flow between two specific stations

Definition at line 369 of file CPheadquarters.cpp.

```
00369
      {
00370     Graph graph;
00371     std::ifstream inputFile1(R"(../network.csv)");
00372     string line1;
00373     std::getline(inputFile1, line1); // ignore first line
00374     while (getline(inputFile1, line1, '\n')) {
00375
00376         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00377             line1.pop_back(); // Remove the '\r' character
00378         }
00379
00380         string station_A;
00381         string station_B;
00382         string temp;
00383         int capacity;
00384         string service;
00385         bool flag=true;
00386
00387         stringstream inputString(line1);
00388
00389         getline(inputString, station_A, ',');
00390         getline(inputString, station_B, ',');
00391         getline(inputString, temp, ',');
00392         getline(inputString, service, ',');
00393
00394         capacity = stoi(temp);
00395         graph.addVertex(station_A);
00396         graph.addVertex(station_B);
00397
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                 }
00402         }
00403         if (flag) {
00404             graph.addEdge(station_A, station_B, capacity, service);
00405         }
00406         line1 = "";
00407     }
00408
00409     Vertex *source = graph.findVertex(s); // set source vertex
00410     Vertex *sink = graph.findVertex(t); // set sink vertex
00411
00412     // Check if these stations even exist
00413     if (source == nullptr || sink == nullptr) {
00414         std::cerr « "Source or sink vertex not found." « std::endl;
00415         return 1;
00416     }
00417     int maxFlow = graph.edmondsKarp(s, t);
00418
00419     if (maxFlow == 0) {
00420         cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
00421              « endl;
00422     }
00423     cout « "maxFlow:\t" « maxFlow « endl;
00424
00425
00426     return 1;
00427 }
```

4.1.2.12 T4_2Top_K_ReducedConectivity()

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

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

Parameters

unwantedEdges	

Returns

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

Definition at line 430 of file CPheadquarters.cpp.

```
00430                                                                              {
00431     Graph graph;
00432     std::ifstream inputFile1(R"(../network.csv)");
00433     string line1;
00434     std::getline(inputFile1, line1); // ignore first line
00435     while (getline(inputFile1, line1, '\n')) {
00436
00437         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00438             line1.pop_back(); // Remove the '\r' character
00439         }
00440
00441         string station_A;
00442         string station_B;
00443         string temp;
00444         int capacity;
00445         string service;
00446         bool flag=true;
00447
00448         stringstream inputString(line1);
```

```

        getline(inputString, station_A, ',');
        getline(inputString, station_B, ',');
        getline(inputString, temp, ',');
        getline(inputString, service, ',');

        capacity = stoi(temp);
        graph.addVertex(station_A);
        graph.addVertex(station_B);

        for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
            if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            graph.addEdge(station_A, station_B, capacity, service);
        }
        line1 = "";
    }
    vector<string> org = lines.getSources();
    vector<string> targ = lines.getTargets();

    lines.mul_edmondsKarp(org,targ);
    graph.mul_edmondsKarp(org,targ);
    vector<pair<int, int» top_k;

    auto length = lines.getVertexSet().size();
    for (int i = 0; i < length; ++i) {
        string destination = lines.getVertexSet()[i]->getId();
        auto v1 = lines.findVertex(destination);
        auto v2 = graph.findVertex(destination);
        int maxFlow1 = 0;
        int maxFlow2 = 0;
        for(auto e : v1->getIncoming()){
            maxFlow1+=e->getFlow();
        }
        for(auto e : v2->getIncoming()){
            maxFlow2+=e->getFlow();
        }
        int diff = maxFlow1 - maxFlow2;
        auto p = pair(i, diff);
        top_k.push_back(p);
    }
    std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
        return left.second > right.second;
    });
    for (int i = 0; i < 10; i++) {
        cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second
« '\n';
    }
    return 1;
}
```

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

- CPheadquarters.h
- CPheadquarters.cpp

4.2 Edge Class Reference

Public Member Functions

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

Protected Attributes

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

4.2.1 Detailed Description

Definition at line 78 of file VertexEdge.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Edge()

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

Definition at line 128 of file VertexEdge.cpp.

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

4.2.3 Member Function Documentation

4.2.3.1 getDest()

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

Definition at line 131 of file VertexEdge.cpp.

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

4.2.3.2 getFlow()

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

Definition at line 151 of file VertexEdge.cpp.

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

4.2.3.3 getOrig()

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

Definition at line 139 of file VertexEdge.cpp.

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

4.2.3.4 getReverse()

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

Definition at line 143 of file VertexEdge.cpp.

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

4.2.3.5 getService()

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

Definition at line 171 of file VertexEdge.cpp.

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

4.2.3.6 getWeight()

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

Definition at line 135 of file VertexEdge.cpp.

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

4.2.3.7 isSelected()

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

Definition at line 147 of file VertexEdge.cpp.

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

4.2.3.8 setFlow()

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

Definition at line 163 of file VertexEdge.cpp.

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

4.2.3.9 setReverse()

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

Definition at line 159 of file VertexEdge.cpp.

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

4.2.3.10 setSelected()

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

Definition at line 155 of file VertexEdge.cpp.

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

4.2.3.11 setService()

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

Definition at line 167 of file VertexEdge.cpp.

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

4.2.4 Member Data Documentation

4.2.4.1 dest

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

Definition at line 105 of file VertexEdge.h.

4.2.4.2 flow

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

Definition at line 116 of file VertexEdge.h.

4.2.4.3 orig

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

Definition at line 113 of file VertexEdge.h.

4.2.4.4 reverse

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

Definition at line 114 of file VertexEdge.h.

4.2.4.5 selected

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

Definition at line 110 of file VertexEdge.h.

4.2.4.6 service

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

Definition at line 108 of file VertexEdge.h.

4.2.4.7 weight

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

Definition at line 106 of file VertexEdge.h.

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

- VertexEdge.h
- VertexEdge.cpp

4.3 Graph Class Reference

Public Member Functions

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

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

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

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

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

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

  finds all the source vertexes of the entire graph

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

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

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

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

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

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

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

Protected Member Functions

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

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

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

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

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

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

Protected Attributes

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

4.3.1 Detailed Description

Definition at line 15 of file Graph.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 ∼Graph()

```
Graph::∼Graph ( )
```

Definition at line 63 of file Graph.cpp.

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

4.3.3 Member Function Documentation

4.3.3.1 addEdge()

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

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

Parameters

sourc	
dest	
w	
service	

Returns

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

Definition at line 34 of file Graph.cpp.

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

4.3.3.2 addVertex()

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

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

Parameters

id	

Returns

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

Definition at line 26 of file Graph.cpp.

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

4.3.3.3 deleteVertex()

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

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

Parameters

name	

Definition at line 323 of file Graph.cpp.

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

4.3.3.4 edmondsKarp()

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

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

Parameters

s	
t	

Returns

maximum flow

Note

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

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

Attention

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

Definition at line 163 of file Graph.cpp.

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

4.3.3.5 find_sources()

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

finds all the source vertexes of a sub_graph

Parameters

desired_stations	

Returns

> vector with the id's of the target vertexes

Definition at line 178 of file Graph.cpp.

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

4.3.3.6 find_targets()

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

finds all the target vertexes of a sub_graph

Parameters

desired_stations	

Returns

vector with the id's of the target vertexes

Definition at line 198 of file Graph.cpp.

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

4.3.3.7 findAllPaths()

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

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

Parameters

source	
destination	
path	
allPaths	

Definition at line 269 of file Graph.cpp.

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

4.3.3.8 findAugmentingPath()

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

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

Parameters

s	
t	

Returns

true if an augmenting path was found, and false otherwise

Note

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

Attention

This function uses the BFS algorithm.

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

Definition at line 98 of file Graph.cpp.

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

4.3.3.9 findEdge()

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

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

Parameters

source	
destination	

Returns

edge

Definition at line 291 of file Graph.cpp.

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

4.3.3.10 findMinResidual()

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

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

Parameters

s	
t	

Returns

the minimum residual capacity of an augmenting path

Definition at line 132 of file Graph.cpp.

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

4.3.3.11 findVertex()

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

Auxiliary function to find a vertex with a given ID.

Parameters

id	

Returns

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

Definition at line 16 of file Graph.cpp.

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

4.3.3.12 getNumVertex()

`int Graph::getNumVertex ( ) const`

Definition at line 7 of file Graph.cpp.

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

4.3.3.13 getSources()

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

finds all the source vertexes of the entire graph

Returns

vector with the id's of the source vertexes

Definition at line 302 of file Graph.cpp.

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

4.3.3.14 getTargets()

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

finds all the target vertexes of the entire graph

Returns

vector with the id's of the target vertexes

Definition at line 312 of file Graph.cpp.

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

4.3.3.15 getVertexSet()

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

Definition at line 11 of file Graph.cpp.

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

4.3.3.16 isIn()

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

Definition at line 218 of file Graph.cpp.

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

4.3.3.17 mul_edmondsKarp()

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

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

Parameters

souces	
targets	

Returns

maximum flow

Definition at line 226 of file Graph.cpp.

```
00226                                                                                            {
00227     auto it1 = souces.begin();
00228     while (it1 != souces.end()) {
00229         if (isIn(*it1, targets)) {
00230             it1 = souces.erase(it1);
00231         } else it1++;
00232     }
00233
00234     auto it2 = targets.begin();
00235     while (it2 != targets.end()) {
00236         if (isIn(*it2, souces)) {
00237             it2 = souces.erase(it2);
00238         } else it2++;
00239     }
00240
00241     addVertex("temp_source");
00242     for (std::string s: souces) {
00243         addEdge("temp_source", s, INT32_MAX, "STANDARD");
00244     }
00245
00246     addVertex("temp_targets");
00247     for (std::string s: targets) {
00248         addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
00249     }
00250     for (auto e: vertexSet) {
00251         for (auto i: e->getAdj()) {
00252             i->setFlow(0);
00253         }
00254     }
00255     int maxFlow = 0;
00256     while (findAugmentingPath("temp_source", "temp_targets")) {
00257         int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
00258         updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
00259         maxFlow += bottleneck;
00260     }
00261     deleteVertex("temp_targets");
00262     deleteVertex("temp_source");
00263     return maxFlow;
00264 }
```

4.3.3.18 print()

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

prints the graph

Definition at line 70 of file Graph.cpp.

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

4.3.3.19 testAndVisit()

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

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

Parameters

q	
e	
w	
residual	

Definition at line 88 of file Graph.cpp.

```
00088                                                                                     {
00089     if (!w->isVisited() && residual > 0) {
00090         w->setVisited(true);
00091         w->setPath(e);
00092         q.push(w);
00093     }
00094 }
```

4.3.3.20 updateFlow()

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

auxiliary function to update the flow of an augmenting path

Parameters

s	
t	
bottleneck	

Note

The bottleneck is the minimum residual capacity of an augmenting path

Definition at line 148 of file Graph.cpp.

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

4.3.4 Member Data Documentation

4.3.4.1 distMatrix

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

Definition at line 121 of file Graph.h.

4.3.4.2 pathMatrix

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

Definition at line 122 of file Graph.h.

4.3.4.3 vertexSet

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

Definition at line 119 of file Graph.h.

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

- Graph.h
- Graph.cpp

4.4 Station Class Reference

Public Member Functions

- Station ()

  Default constructor.
- Station (string name_, string district_, string municipality_, string township_, string line_)

  Constructor.
- string get_name ()

  Returns the station's name.
- string get_district ()

  Returns the station's district.
- string get_municipality ()

  Returns the station's municipality.
- string get_township ()

  Returns the station's township.
- string get_line ()

  Returns the station's line.

4.4.1 Detailed Description

Definition at line 12 of file Station.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Station() [1/2]

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

Default constructor.

Definition at line 35 of file Station.cpp.

```
00035                  {
00036
00037 }
```

4.4.2.2 Station() [2/2]

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

Constructor.

Parameters

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

Definition at line 7 of file Station.cpp.

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

4.4.3 Member Function Documentation

4.4.3.1 get_district()

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

Returns the station's district.

Returns

district

Definition at line 19 of file Station.cpp.

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

4.4.3.2 get_line()

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

Returns the station's line.

Returns

line

Definition at line 31 of file Station.cpp.

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

4.4.3.3 get_municipality()

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

Returns the station's municipality.

Returns

municipality

Definition at line 23 of file Station.cpp.

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

4.4.3.4 get_name()

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

Returns the station's name.

Returns

name

Definition at line 15 of file Station.cpp.

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

4.4.3.5 get_township()

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

Returns the station's township.

Returns

township

Definition at line 27 of file Station.cpp.

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

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

- Station.h
- Station.cpp

4.5 Vertex Class Reference

Public Member Functions

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

Protected Member Functions

- void print () const

Protected Attributes

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

4.5.1 Detailed Description

Definition at line 19 of file VertexEdge.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 Vertex()

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

Definition at line 7 of file VertexEdge.cpp.

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

4.5.3 Member Function Documentation

4.5.3.1 addEdge()

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

Definition at line 13 of file VertexEdge.cpp.

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

4.5.3.2 getAdj()

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

Definition at line 59 of file VertexEdge.cpp.

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

4.5.3.3 getDist()

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

Definition at line 75 of file VertexEdge.cpp.

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

4.5.3.4 getId()

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

Definition at line 55 of file VertexEdge.cpp.

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

4.5.3.5 getIncoming()

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

Definition at line 83 of file VertexEdge.cpp.

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

4.5.3.6 getIndegree()

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

Definition at line 71 of file VertexEdge.cpp.

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

4.5.3.7 getPath()

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

Definition at line 79 of file VertexEdge.cpp.

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

4.5.3.8 isProcessing()

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

Definition at line 67 of file VertexEdge.cpp.

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

4.5.3.9 isVisited()

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

Definition at line 63 of file VertexEdge.cpp.

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

4.5.3.10 operator<()

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

Definition at line 51 of file VertexEdge.cpp.

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

4.5.3.11 print()

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

Definition at line 112 of file VertexEdge.cpp.

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

4.5.3.12 removeEdge()

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

Definition at line 25 of file VertexEdge.cpp.

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

4.5.3.13 setDist()

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

Definition at line 103 of file VertexEdge.cpp.

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

4.5.3.14 setId()

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

Definition at line 87 of file VertexEdge.cpp.

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

4.5.3.15 setIndegree()

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

Definition at line 99 of file VertexEdge.cpp.

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

4.5.3.16 setPath()

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

Definition at line 107 of file VertexEdge.cpp.

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

4.5.3.17 setProcesssing()

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

Definition at line 95 of file VertexEdge.cpp.

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

4.5.3.18 setVisited()

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

Definition at line 91 of file VertexEdge.cpp.

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

4.5.4 Member Data Documentation

4.5.4.1 adj

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

Definition at line 60 of file VertexEdge.h.

4.5.4.2 dist

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

Definition at line 66 of file VertexEdge.h.

4.5.4.3 id

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

Definition at line 59 of file VertexEdge.h.

4.5.4.4 incoming

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

Definition at line 69 of file VertexEdge.h.

4.5.4.5 indegree

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

Definition at line 65 of file VertexEdge.h.

4.5.4.6 path

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

Definition at line 67 of file VertexEdge.h.

4.5.4.7 processing

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

Definition at line 64 of file VertexEdge.h.

4.5.4.8 queueIndex

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

Definition at line 71 of file VertexEdge.h.

4.5.4.9 visited

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

Definition at line 63 of file VertexEdge.h.

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

- VertexEdge.h
- VertexEdge.cpp

Chapter 5

File Documentation

5.1 CPheadquarters.cpp

```
//
// Created by Pedro on 23/03/2023.
//

#include <fstream>
#include <sstream>
#include "CPheadquarters.h"
#include <chrono>
#include <set>

using namespace std;

void CPheadquarters::read_network(string path){
    std::ifstream inputFile1(path);
    string line1;
    std::getline(inputFile1, line1); // ignore first line
    while (getline(inputFile1, line1, ’\n’)) {

        if (!line1.empty() && line1.back() == ’\r’) { // Check if the last character is ’\r’
            line1.pop_back(); // Remove the ’\r’ character
        }

        string station_A;
        string station_B;
        string temp;
        int capacity;
        string service;

        stringstream inputString(line1);

        getline(inputString, station_A, ’,’);
        getline(inputString, station_B, ’,’);
        getline(inputString, temp, ’,’);
        getline(inputString, service, ’,’);

        capacity = stoi(temp);
        lines.addVertex(station_A);
        lines.addVertex(station_B);

        lines.addEdge(station_A, station_B, capacity, service);
    }
}

void CPheadquarters::read_stations(string path){
    std::ifstream inputFile2(R"(../stations.csv)");
    string line2;
    std::getline(inputFile2, line2); // ignore first line

    while (getline(inputFile2, line2, ’\n’)) {

        if (!line2.empty() && line2.back() == ’\r’) { // Check if the last character is ’\r’
            line2.pop_back(); // Remove the ’\r’ character
        }

        string nome;
        string distrito;
        string municipality;
        string township;
        string line;

        stringstream inputString(line2);

        getline(inputString, nome, ',');
        getline(inputString, distrito, ',');
        getline(inputString, municipality, ',');
        getline(inputString, township, ',');
        getline(inputString, line, ',');

        Station station(nome, distrito, municipality, township, line);
        stations[nome] = station;

        // print information about the station, to make sure it was imported correctly
        //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « " township: " « township « " line: " « line « endl;
    }
}

void CPheadquarters::read_files() {

    //--------------------------------------------Read network.csv--------------------------------------------
    std::ifstream inputFile1(R"(../network.csv)");
    string line1;
    std::getline(inputFile1, line1); // ignore first line
    while (getline(inputFile1, line1, '\n')) {

        if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
            line1.pop_back(); // Remove the '\r' character
        }

        string station_A;
        string station_B;
        string temp;
        int capacity;
        string service;

        stringstream inputString(line1);

        getline(inputString, station_A, ',');
        getline(inputString, station_B, ',');
        getline(inputString, temp, ',');
        getline(inputString, service, ',');

        capacity = stoi(temp);
        lines.addVertex(station_A);
        lines.addVertex(station_B);

        lines.addEdge(station_A, station_B, capacity, service);
    }


    //--------------------------------------------Read stations.csv--------------------------------------------
    std::ifstream inputFile2(R"(../stations.csv)");
    string line2;
    std::getline(inputFile2, line2); // ignore first line

    while (getline(inputFile2, line2, '\n')) {

        if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
            line1.pop_back(); // Remove the '\r' character
        }

        string nome;
        string distrito;
        string municipality;
        string township;
        string line;

        stringstream inputString(line2);

        getline(inputString, nome, ',');
        getline(inputString, distrito, ',');
        getline(inputString, municipality, ',');
        getline(inputString, township, ',');
        getline(inputString, line, ',');

        Station station(nome, distrito, municipality, township, line);
        stations[nome] = station;

        // print information about the station, to make sure it was imported correctly
        //cout « "station: " « nome « " distrito: " « distrito « " municipality: " « municipality « " township: " « township « " line: " « line « endl;
    }
}
```

```


Graph CPheadquarters::getLines() const {
    return this->lines;
}


int CPheadquarters::T2_1maxflow(string stationA, string stationB) {
    Vertex *source = lines.findVertex(stationA); // set source vertex
    Vertex *sink = lines.findVertex(stationB); // set sink vertex

    // Check if these stations even exist
    if (source == nullptr || sink == nullptr) {
        std::cerr « "Source or sink vertex not found." « std::endl;
        return 0;
    }
    int maxFlow = lines.edmondsKarp(stationA, stationB);

    if (maxFlow == 0) {
        cerr « "Stations are not connected. Try stationB to stationA instead. " « stationB « " -> " « stationA
                « endl;
    } else {
        cout « "maxFlow:\t" « maxFlow « endl;
    }

    return maxFlow;
}


int CPheadquarters::T2_2maxflowAllStations() {
    vector<string> stations;
    int maxFlow = 0;
    auto length = lines.getVertexSet().size();
    // Start the timer
    auto start_time = std::chrono::high_resolution_clock::now();
    cout « "Calculating max flow for all pairs of stations..." « endl;
    cout « "Please stand by..." « endl;
    for (int i = 0; i < length; ++i) {
        for (int j = i + 1; j < length; ++j) {
            string stationA = lines.getVertexSet()[i]->getId();
            string stationB = lines.getVertexSet()[j]->getId();
            int flow = lines.edmondsKarp(stationA, stationB);
            if (flow == maxFlow) {
                stations.push_back(stationB);
                stations.push_back(stationA);
            } else if (flow > maxFlow) {
                stations.clear();
                stations.push_back(stationB);
                stations.push_back(stationA);
                maxFlow = flow;
            }
        }
    }
    // End the timer
    auto end_time = std::chrono::high_resolution_clock::now();

    // Compute the duration
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

    // Print the duration
    std::cout « "Time taken: " « duration.count() « " ms" « std::endl;

    cout « "Pairs of stations with the most flow [" « maxFlow « "]:\n";
    for (int i = 0; i < stations.size(); i = i + 2) {
        cout « "------------------------\n";
        cout « "Source: " « stations[i + 1] « '\n';
        cout « "Target: " « stations[i] « '\n';
        cout « "------------------------\n";
    }
    return maxFlow;
}


void CPheadquarters::T2_3municipality() {
    vector<pair<string , int» top_k;
    set<string> sett;
    for (auto m : stations) {
        sett.insert(m.second.get_district());
    }
    for (auto m : sett) {
        vector<string> desired_stations;
        for (auto p: stations) {
            if (p.second.get_municipality() == m) {
                desired_stations.push_back(p.second.get_name());
            }
        }


        vector<string> souces = lines.find_sources(desired_stations);
        vector<string> targets = lines.find_targets(desired_stations);
        int diff=lines.mul_edmondsKarp(souces, targets);
        auto p = pair(m, diff);
        top_k.push_back(p);
    }
    std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
        return left.second > right.second;
    });
    for (int i = 0; i < 10; i++) {
        cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
    }
}

void CPheadquarters::T2_3district() {
    vector<pair<string , int» top_k;
    set<string> sett;
    for (auto m : stations) {
        sett.insert(m.second.get_district());
    }
    for (auto m : sett) {
        vector<string> desired_stations;
        for (auto p: stations) {
            if (p.second.get_district() == m) {
                desired_stations.push_back(p.second.get_name());
            }
        }
        vector<string> souces = lines.find_sources(desired_stations);
        vector<string> targets = lines.find_targets(desired_stations);
        int diff=lines.mul_edmondsKarp(souces, targets);
        auto p = pair(m, diff);
        top_k.push_back(p);
    }
    std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
        return left.second > right.second;
    });
    for (int i = 0; i < 10; i++) {
        cout « i + 1 « "-" « top_k[i].first « " -> " « top_k[i].second « '\n';
    }
}


int CPheadquarters::T2_4maxArrive(string destination) {
    Vertex *dest = lines.findVertex(destination);
    int maxFlow = 0;

    // iterate over all vertices to find incoming and outgoing vertices
    for (auto &v: lines.getVertexSet()) {
        if (v != dest) {

            int flow = lines.edmondsKarp(v->getId(), destination);

            // Update the maximum flow if this vertex contributes to a higher maximum
            if (flow > maxFlow) {
                maxFlow = flow;
            }
        }

    }

    cout « endl;
    for (auto &e: dest->getIncoming()) {
        cout « e->getOrig()->getId() « " -> " « e->getDest()->getId() « " : " « e->getWeight() « endl;

    }

    cout « "Max number of trains that can simultaneously arrive at " « destination « ": " « maxFlow « endl;
    return maxFlow;

}



int CPheadquarters::T3_1MinCost(string source, string destination) {
    Vertex *sourceVertex = lines.findVertex(source); // set source vertex
    Vertex *destVertex = lines.findVertex(destination); // set sink vertex
    if (sourceVertex == nullptr || destVertex == nullptr) {
        cerr « "Source or destination vertex not found. Try again" « endl;
        return 1;
    }

    Graph graph = lines;

    std::vector<Vertex *> path;
    std::vector<std::vector<Vertex *» allPaths;


    graph.findAllPaths(sourceVertex, destVertex, path, allPaths);

    vector<int> maxFlows;
    vector<int> totalCosts;

    cout « "All possible paths between " « source « " and " « destination « ":\n" « endl;
    for (auto path: allPaths) {
        int minWeight = 10;
        int totalCost = 0; // total cost of this path
        for (int i = 0; i + 1 < path.size(); i++) {
            std::cout « path[i]->getId() « " -> ";
            Edge *e = graph.findEdge(path[i], path[i + 1]);
            cout « " (" « e->getWeight() « " trains, " « e->getService() « " service) ";
            if (e->getWeight() < minWeight) {
                minWeight = e->getWeight();
            }

            // according to the problem's specification, the cost of STANDARD service is 2 euros and ALFA PENDULAR is 4
            if (e->getService() == "STANDARD") {
                totalCost += 2;
            } else if (e->getService() == "ALFA PENDULAR") {
                totalCost += 4;
            }
        }
        maxFlows.push_back(minWeight);
        totalCosts.push_back(totalCost);
        cout « " -> " « path[path.size() - 1]->getId() « endl;
        cout « "Max flow for this path: " « minWeight « " trains. ";
        cout « "Total cost: " « totalCost « " euros." « endl;
        std::cout « std::endl;
    }

    // find the path with the minimum cost per train
    int maxTrains = 0;
    int resCost;
    double max_value = 10000;
    for (int i = 0; i < maxFlows.size(); ++i) {
        double costPerTrain = (double) totalCosts[i] / maxFlows[i];
        if (costPerTrain < max_value) {
            max_value = costPerTrain;
            maxTrains = maxFlows[i];
            resCost = totalCosts[i];
        }
    }

    cout « "Max number of trains that can travel between " « source « " and " « destination
         « " with minimum cost"
         « "(" « resCost « " euros): " « maxTrains « " trains\n" « endl;
    return maxTrains;
}


int CPheadquarters::T4_1ReducedConectivity(std::vector<std::string> unwantedEdges, std::string s, std::string t) {
    Graph graph;
    std::ifstream inputFile1(R"(../network.csv)");
    string line1;
    std::getline(inputFile1, line1); // ignore first line
    while (getline(inputFile1, line1, '\n')) {

        if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
            line1.pop_back(); // Remove the '\r' character
        }

        string station_A;
        string station_B;
        string temp;
        int capacity;
        string service;
        bool flag=true;

        stringstream inputString(line1);

        getline(inputString, station_A, ',');
        getline(inputString, station_B, ',');
        getline(inputString, temp, ',');
        getline(inputString, service, ',');

        capacity = stoi(temp);
        graph.addVertex(station_A);
        graph.addVertex(station_B);

        for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
                if(station_A==unwantedEdges[i] && station_B==unwantedEdges[i+1]){
                    flag=false;
                }
        }
        if (flag) {
            graph.addEdge(station_A, station_B, capacity, service);
        }
        line1 = "";
    }

    Vertex *source = graph.findVertex(s); // set source vertex
    Vertex *sink = graph.findVertex(t); // set sink vertex

    // Check if these stations even exist
    if (source == nullptr || sink == nullptr) {
        std::cerr « "Source or sink vertex not found." « std::endl;
        return 1;
    }
    int maxFlow = graph.edmondsKarp(s, t);

    if (maxFlow == 0) {
        cerr « "Stations are not connected. Try stationB to stationA instead. " « t « " -> " « s
             « endl;
    }
    cout « "maxFlow:\t" « maxFlow « endl;


    return 1;
}


int CPheadquarters::T4_2Top_K_ReducedConectivity(vector<string> unwantedEdges) {
    Graph graph;
    std::ifstream inputFile1(R"(../network.csv)");
    string line1;
    std::getline(inputFile1, line1); // ignore first line
    while (getline(inputFile1, line1, '\n')) {

        if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
            line1.pop_back(); // Remove the '\r' character
        }

        string station_A;
        string station_B;
        string temp;
        int capacity;
        string service;
        bool flag=true;

        stringstream inputString(line1);

        getline(inputString, station_A, ',');
        getline(inputString, station_B, ',');
        getline(inputString, temp, ',');
        getline(inputString, service, ',');

        capacity = stoi(temp);
        graph.addVertex(station_A);
        graph.addVertex(station_B);

        for (int i = 0; i < unwantedEdges.size(); i = i + 2) {
            if (station_A == unwantedEdges[i] && station_B == unwantedEdges[i + 1]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            graph.addEdge(station_A, station_B, capacity, service);
        }
        line1 = "";
    }
    vector<string> org = lines.getSources();
    vector<string> targ = lines.getTargets();

    lines.mul_edmondsKarp(org,targ);
    graph.mul_edmondsKarp(org,targ);
    vector<pair<int, int» top_k;

    auto length = lines.getVertexSet().size();
    for (int i = 0; i < length; ++i) {
        string destination = lines.getVertexSet()[i]->getId();
        auto v1 = lines.findVertex(destination);
        auto v2 = graph.findVertex(destination);
        int maxFlow1 = 0;
        int maxFlow2 = 0;
        for(auto e : v1->getIncoming()){
            maxFlow1+=e->getFlow();
        }
        for(auto e : v2->getIncoming()){
            maxFlow2+=e->getFlow();
        }
        int diff = maxFlow1 - maxFlow2;
        auto p = pair(i, diff);
        top_k.push_back(p);
    }
    std::sort(top_k.begin(), top_k.end(), [](auto &left, auto &right) {
        return left.second > right.second;
    });
    for (int i = 0; i < 10; i++) {
        cout « i + 1 « "-" « lines.getVertexSet()[top_k[i].first]->getId() « " -> " « top_k[i].second « '\n';
    }
    return 1;
}
```

5.2 CPheadquarters.h

```
//
// Created by Pedro on 23/03/2023.
//

#ifndef DAPROJECT_CPHEADQUARTERS_H
#define DAPROJECT_CPHEADQUARTERS_H


#include "Graph.h"
#include "Station.h"

using namespace std;

class CPheadquarters {
    Graph lines;
    unordered_map<string, Station> stations;
public:

    void read_network(string path);

    void read_stations(string path);

    void read_files();

    Graph getLines() const;

    int T2_1maxflow(string station_A, string station_B);

    int T2_2maxflowAllStations();

    void T2_3municipality();

    void T2_3district();

    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);

};


#endif //DAPROJECT_CPHEADQUARTERS_H
```

5.3 Graph.cpp

```
// By: Gonçalo Leão

#include <climits>
#include <queue>
#include "Graph.h"

int Graph::getNumVertex() const {
    return vertexSet.size();
}

```

```
std::vector<Vertex *> Graph::getVertexSet() const {
    return vertexSet;
}


Vertex *Graph::findVertex(const std::string &id) const {
    for (auto v: vertexSet) {
        if (v->getId() == id)
            return v;
    }
    return nullptr;
}



bool Graph::addVertex(const std::string &id) {
    if (findVertex(id) != nullptr)
        return false;
    vertexSet.push_back(new Vertex(id));
    return true;
}


bool Graph::addEdge(const std::string &sourc, const std::string &dest, int w, const std::string &service) {
    auto v1 = findVertex(sourc);
    auto v2 = findVertex(dest);
    if (v1 == nullptr || v2 == nullptr)
        return false;
    v1->addEdge(v2, w, service);

    return true;
}


void deleteMatrix(int **m, int n) {
    if (m != nullptr) {
        for (int i = 0; i < n; i++)
            if (m[i] != nullptr)
                delete[] m[i];
        delete[] m;
    }
}

void deleteMatrix(double **m, int n) {
    if (m != nullptr) {
        for (int i = 0; i < n; i++)
            if (m[i] != nullptr)
                delete[] m[i];
        delete[] m;
    }
}

Graph::~Graph() {
    deleteMatrix(distMatrix, vertexSet.size());
    deleteMatrix(pathMatrix, vertexSet.size());
}



void Graph::print() const {
    std::cout « "---------------- Graph----------------\n";
    std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
    std::cout « "Vertices:\n";
    for (const auto &vertex: vertexSet) {
        std::cout « vertex->getId() « " ";
    }
    std::cout « "\nEdges:\n";
    for (const auto &vertex: vertexSet) {
        for (const auto &edge: vertex->getAdj()) {
            std::cout « vertex->getId() « " -> " « edge->getDest()->getId() « " (weight: " « edge->getWeight() « ", service: " « edge->getService() « ")" « std::endl;
        }
    }
}

// --------------------------------- Edmonds-Karp ---------------------------------


void Graph::testAndVisit(std::queue<Vertex *> &q, Edge *e, Vertex *w, double residual) {
    if (!w->isVisited() && residual > 0) {
        w->setVisited(true);
        w->setPath(e);
        q.push(w);
    }
}

```

```cpp


bool Graph::findAugmentingPath(const std::string &s, const std::string &t) {
    Vertex *source = findVertex(s);
    Vertex *target = findVertex(t);
    if (source == nullptr || target == nullptr) {
        return false;
    }
    for (auto v: vertexSet) {
        v->setVisited(false);
        v->setPath(nullptr);
    }
    source->setVisited(true);
    std::queue<Vertex *> q;
    q.push(source);
    while (!q.empty()) {
        auto v = q.front();
        q.pop();
        for (auto e: v->getAdj()) {
            auto w = e->getDest();
            double residual = e->getWeight() - e->getFlow();
            testAndVisit(q, e, w, residual);
        }
        for (auto e: v->getIncoming()) {
            auto w = e->getDest();
            double residual = e->getFlow();
            testAndVisit(q, e->getReverse(), w, residual);
        }
        if (target->isVisited()) {
            return true;
        }
    }
    return false;
}


int Graph::findMinResidual(Vertex *s, Vertex *t) {
    double minResidual = INT_MAX;
    for (auto v = t; v != s;) {
        auto e = v->getPath();
        if (e->getDest() == v) {
            minResidual = std::min(minResidual, e->getWeight() - e->getFlow());
            v = e->getOrig();
        } else {
            minResidual = std::min(minResidual, e->getFlow());
            v = e->getDest();
        }
    }
    return minResidual;
}


void Graph::updateFlow(Vertex *s, Vertex *t, int bottleneck) {
    for (auto v = t; v != s;) {
        auto e = v->getPath();
        double flow = e->getFlow();
        if (e->getDest() == v) {
            e->setFlow(flow + bottleneck);
            v = e->getOrig();
        } else {
            e->setFlow(flow - bottleneck);
            v = e->getDest();
        }
    }
}


int Graph::edmondsKarp(const std::string &s, const std::string &t) {
    for (auto e: vertexSet) {
        for (auto i: e->getAdj()) {
            i->setFlow(0);
        }
    }
    int maxFlow = 0;
    while (findAugmentingPath(s, t)) {
        int bottleneck = findMinResidual(findVertex(s), findVertex(t));
        updateFlow(findVertex(s), findVertex(t), bottleneck);
        maxFlow += bottleneck;
    }
    return maxFlow;
}

std::vector<std::string> Graph::find_sources(std::vector<std::string> desired_stations) {
    std::vector<std::string> res;

    for (std::string s: desired_stations) {
        bool flag = true;
        auto v = findVertex(s);
        if (v == nullptr) {
            std::cout « "Trouble finding source " « s « ’\n’;
            continue;
        }
        for (auto e: v->getIncoming()) {
            if (isIn(e->getOrig()->getId(), desired_stations)) {
                flag=false;
            }
        }
        if (flag) res.push_back(s);
    }
    return res;
}

std::vector<std::string> Graph::find_targets(std::vector<std::string> desired_stations) {
    std::vector<std::string> res;
    for (std::string s: desired_stations) {
        bool flag = true;
        auto v = findVertex(s);
        if (v == nullptr) {
            std::cout « "Trouble finding target " « s « ’\n’;
            continue;
        }
        for (auto e: v->getAdj()) {
            if (isIn(e->getDest()->getId(), desired_stations)) {
                flag=false;
            }
        }
        if (flag) res.push_back(s);
    }
    return res;
}


bool Graph::isIn(std::string n, std::vector<std::string> vec) {
    for (std::string s: vec) {
        if (s == n) return true;
    }
    return false;
}


int Graph::mul_edmondsKarp(std::vector<std::string> souces, std::vector<std::string> targets) {
    auto it1 = souces.begin();
    while (it1 != souces.end()) {
        if (isIn(*it1, targets)) {
            it1 = souces.erase(it1);
        } else it1++;
    }

    auto it2 = targets.begin();
    while (it2 != targets.end()) {
        if (isIn(*it2, souces)) {
            it2 = souces.erase(it2);
        } else it2++;
    }

    addVertex("temp_source");
    for (std::string s: souces) {
        addEdge("temp_source", s, INT32_MAX, "STANDARD");
    }

    addVertex("temp_targets");
    for (std::string s: targets) {
        addEdge(s, "temp_targets", INT32_MAX, "STANDARD");
    }
    for (auto e: vertexSet) {
        for (auto i: e->getAdj()) {
            i->setFlow(0);
        }
    }
    int maxFlow = 0;
    while (findAugmentingPath("temp_source", "temp_targets")) {
        int bottleneck = findMinResidual(findVertex("temp_source"), findVertex("temp_targets"));
        updateFlow(findVertex("temp_source"), findVertex("temp_targets"), bottleneck);
        maxFlow += bottleneck;
    }
    deleteVertex("temp_targets");
    deleteVertex("temp_source");
    return maxFlow;
}

// --------------------------------- Find ALL existing augmenting paths ---------------------------------


```

```
void Graph::findAllPaths(Vertex *source, Vertex *destination, std::vector<Vertex *> &path,
                         std::vector<std::vector<Vertex *» &allPaths) {
    path.push_back(source);
    source->setVisited(true);

    if (source == destination) {
        allPaths.push_back(path);
    } else {
        for (auto edge: source->getAdj()) {
            Vertex *adjacent = edge->getDest();
            if (!adjacent->isVisited()) {
                findAllPaths(adjacent, destination, path, allPaths);
            }
        }
    }

    path.pop_back();
    source->setVisited(false);
}



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

    for (auto edge: source->getAdj()) {
        if (edge->getDest() == destination) {
            return edge;
        }
    }
    return nullptr;
}


std::vector<std::string> Graph::getSources() {
    std::vector<std::string> res;
    for (auto v : vertexSet) {
        if(v->getIncoming().empty()){
            res.push_back(v->getId());
        }
    }
    return res;
}

std::vector<std::string> Graph::getTargets() {
    std::vector<std::string> res;
    for (auto v : vertexSet) {
        if(v->getAdj().empty()){
            res.push_back(v->getId());
        }
    }
    return res;
}


void Graph::deleteVertex(std::string name) {
    auto v = findVertex(name);
    for(auto e : v->getAdj()){
        auto s = e->getDest()->getId();
        v->removeEdge(s);
    }
    for(auto e : v->getIncoming()){
        e->getOrig()->removeEdge(name);
    }
    auto it = vertexSet.begin();
    while (it!=vertexSet.end()){
        Vertex* currentVertex = *it;
        if(currentVertex->getId()==name){
            it=vertexSet.erase(it);
        }
        else{
            it++;
        }
    }
}

```

5.4 Graph.h

```
// By: Gonçalo Leão

#ifndef DA_TP_CLASSES_GRAPH
#define DA_TP_CLASSES_GRAPH

```

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>


#include "VertexEdge.h"

class Graph {
public:
    ~Graph();

    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);


    [[nodiscard]] int getNumVertex() const;

    [[nodiscard]] std::vector<Vertex *> getVertexSet() const;

    void print() const;

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

    std::vector<std::string> getSources();

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

    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:
    std::vector<Vertex *> vertexSet;    // vertex set

    double **distMatrix = nullptr;   // dist matrix for Floyd-Warshall
    int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall


    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);

    void deleteVertex(std::string name);
};

void deleteMatrix(int **m, int n);

void deleteMatrix(double **m, int n);

#endif /* DA_TP_CLASSES_GRAPH */
```

5.5 main.cpp

```
#include <iostream>
#include "CPheadquarters.h"

using namespace std;

int main() {
    CPheadquarters CP;
    string path;
    cout«"Insert path to file to consruct graph: ";
    getline(cin, path);
    CP.read_network(path);
    cout«"Insert path to file regarding stations: ";
    getline(cin, path);
    cout«endl;
    CP.read_stations(path);
    CP.getLines().print();
    int n;
    cout « "\n-------------- An Analysis Tool for Railway Network Management --------------\n" « endl;
    do {
        cout « "1 - T2.1 Max number of trains between stations\n";
        cout « "2 - T2.2 Stations that require the Max num of trains among all pairs of stations\n";
        cout « "3 - T2.3 Indicate where management should assign larger budgets for the purchasing and maintenance of trains\n";
        cout « "4 - T2.4 Max number of trains that can simultaneously arrive at a given station\n";
        cout « "5 - T3.1 Max number of trains that can simultaneously travel with minimum cost\n";
        cout « "6 - T4.1 Max number of trains between stations in a network of reduced connectivity\n";
        cout « "7 - T4.2 Top-10 most affected stations in a network of reduced connectivity\n";
        cout « "8 - Exit\n";


        bool validInput = false;

        while (!validInput) {
            cout « "Insert your option:\n";
            cin » n;

            if (cin.fail() || n < 1 || n > 8) {
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), ’\n’);
                cout « "Invalid input. Please enter a number between 1 and 8." « endl;
            } else {
                validInput = true;
            }
        }

        switch (n) {
            case 1: {
                cin.ignore(); // ignore newline character left in the input stream
                string a, b;
                cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
                cout « "Enter station A: ";
                getline(cin, a);

                cout « "Enter station B: ";
                getline(cin, b);

                if (a.empty() || b.empty()) {
                    cerr « "Error: Station names cannot be empty." « endl;
                    break;
                }

                // call function to calculate max flow between stations A and B
                CP.T2_1maxflow(a, b);
                break;
            }

            case 2: {
                CP.T2_2maxflowAllStations();
                break;
            }

            case 3: {
                cin.ignore();

                int c;
                cout « "Type 1 for Top-10 districts regarding flow" « ’\n’;
                cout « "Type 2 for Top-10 municipalities regarding flow" « ’\n’;
                cin » c;
                switch (c) {
                    case 1:
                        CP.T2_3district();
                        break;
                    case 2:
                        CP.T2_3municipality();
                        break;
                    default:
                        cout « "Invalid input";
                        break;
                }
                cout « endl;
                break;
            }

            case 4: {
                cin.ignore();
                string destination;
                cout « "Enter destination: ";
                getline(cin, destination);
                CP.T2_4maxArrive(destination);
                break;
            }

            case 5: {
                cin.ignore();
                string a, b;
                cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
                cout « "Enter station A: ";
                getline(cin, a);
                cout « endl;
                cout « "Enter station B: ";
                getline(cin, b);

                if (a.empty() || b.empty()) {
                    cerr « "Error: Station names cannot be empty." « endl;
                    break;
                }

                CP.T3_1MinCost(a, b);
                break;
            }

            case 6: {
                cin.ignore();
                vector<string> unwantedEdges;
                string edgesource;
                string edgetarget;
                string b;
                string a;
                cout « R"(Example: "Entroncamento" "Lisboa Oriente")" « endl;
                cout « "Enter station A: ";
                getline(cin, a);
                cout « "Enter station B: ";
                getline(cin, b);
                cout « ’\n’;
                cout « "List unwanted edges. Start by typing the edge source an then the edge destine. Type ’.’ to end listing: \n";
                cout « R"(Example: "Bustelo" "Meinedo" would delete the edge "Bustelo->Meinedo")" « endl;
                while (1){
                    cout « "Enter edge source or ’.’ to finish: ";
                    getline(cin, edgesource);
                    if(edgesource==".") break;
                    unwantedEdges.push_back(edgesource);
                    cout « "Enter edge target: ";
                    getline(cin, edgetarget);
                    unwantedEdges.push_back(edgetarget);
                }
                CP.T4_1ReducedConectivity(unwantedEdges,a,b);
                break;
            }

            case 7: {
                cin.ignore();
                vector<string> unwantedEdges;
                string edgesource;
                string edgetarget;
                cout « "List unwanted edges. Start by typing the edge source an then the edge destine. Type ’.’ to end listing: \n";
                cout « R"(Example: "Bustelo" "Meinedo" would delete the edge "Bustelo->Meinedo")" « endl;
                while (1){
                    cout « "Enter edge source or ’.’ to finish: ";
                    getline(cin, edgesource);
                    if(edgesource==".") break;
                    unwantedEdges.push_back(edgesource);
                    cout « "Enter edge target: ";
                    getline(cin, edgetarget);
                    unwantedEdges.push_back(edgetarget);
                }
                CP.T4_2Top_K_ReducedConectivity(unwantedEdges);

                break;
            }

            case 8: {
                cout « "Exiting program..." « endl;
                break;
            }

            default: {
                cerr « "Error: Invalid option selected." « endl;
                break;
            }
        }
```

```
    } while (n != 8);

    return 0;
}
```

5.6 Station.cpp

```
//
// Created by Pedro on 23/03/2023.
//

#include "Station.h"

Station::Station(string name_, string district_, string municipality_, string township_, string line_)
{
    name=name_;
    municipality=municipality_;
    district=district_;
    township=township_;
    line=line_;
}

string Station::get_name() {
    return name;
}

string Station::get_district() {
    return district;
}

string Station::get_municipality() {
    return municipality;
}

string Station::get_township() {
    return township;
}

string Station::get_line() {
    return line;
}

Station::Station() {

}
```

5.7 Station.h

```
//
// Created by Pedro on 23/03/2023.
//

#ifndef DAPROJECT_STATION_H
#define DAPROJECT_STATION_H

#include <string>

using namespace std;

class Station {
    string name;
    string district;
    string municipality;
    string township;
    string line;
public:
    Station();

    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();
};


#endif //DAPROJECT_STATION_H
```

5.8 VertexEdge.cpp

```
// By: Gonçalo Leão

#include "VertexEdge.h"

/************************* Vertex  **************************/

Vertex::Vertex(std::string id) : id(id) {}

/*
 * Auxiliary function to add an outgoing edge to a vertex (this),
 * with a given destination vertex (d) and edge weight (w).
 */
Edge *Vertex::addEdge(Vertex *d, int w, const std::string &service) {
    auto newEdge = new Edge(this, d, w, service);
    adj.push_back(newEdge);
    d->incoming.push_back(newEdge);
    return newEdge;
}

/*
 * Auxiliary function to remove an outgoing edge (with a given destination (d))
 * from a vertex (this).
 * Returns true if successful, and false if such edge does not exist.
 */
bool Vertex::removeEdge(std::string destID) {
    bool removedEdge = false;
    auto it = adj.begin();
    while (it != adj.end()) {
        Edge *edge = *it;
        Vertex *dest = edge->getDest();
        if (dest->getId() == destID) {
            it = adj.erase(it);
            // Also remove the corresponding edge from the incoming list
            auto it2 = dest->incoming.begin();
            while (it2 != dest->incoming.end()) {
                if ((*it2)->getOrig()->getId() == id) {
                    it2 = dest->incoming.erase(it2);
                } else {
                    it2++;
                }
            }
            delete edge;
            removedEdge = true; // allows for multiple edges to connect the same pair of vertices (multigraph)
        } else {
            it++;
        }
    }
    return removedEdge;
}

bool Vertex::operator<(Vertex &vertex) const {
    return this->dist < vertex.dist;
}

std::string Vertex::getId() const {
    return this->id;
}

std::vector<Edge *> Vertex::getAdj() const {
    return this->adj;
}

bool Vertex::isVisited() const {
    return this->visited;
}

bool Vertex::isProcessing() const {
    return this->processing;
}

unsigned int Vertex::getIndegree() const {
    return this->indegree;
}

double Vertex::getDist() const {
    return this->dist;
}

Edge *Vertex::getPath() const {
    return this->path;
}

std::vector<Edge *> Vertex::getIncoming() const {
    return this->incoming;
}

void Vertex::setId(int id) {
    this->id = id;
}

void Vertex::setVisited(bool visited) {
    this->visited = visited;
}

void Vertex::setProcesssing(bool processing) {
    this->processing = processing;
}

void Vertex::setIndegree(unsigned int indegree) {
    this->indegree = indegree;
}

void Vertex::setDist(double dist) {
    this->dist = dist;
}

void Vertex::setPath(Edge *path) {
    this->path = path;
}


void Vertex::print() const {
    std::cout « "Vertex: " « id « std::endl;
    std::cout « "Adjacent to: ";
    for (const Edge *e: adj) {
        std::cout « e->getDest()->getId() « " ";
    }
    std::cout « std::endl;
    std::cout « "Visited: " « visited « std::endl;
    std::cout « "Indegree: " « indegree « std::endl;
    std::cout « "Distance: " « dist « std::endl;
    std::cout « "Path: " « path « std::endl;
}


/********************** Edge  ****************************/

Edge::Edge(Vertex *orig, Vertex *dest, int w, const std::string &service) : orig(orig), dest(dest),
weight(w),
                                                                            service(service), flow(0)
{}

Vertex *Edge::getDest() const {
    return this->dest;
}

int Edge::getWeight() const {
    return this->weight;
}

Vertex *Edge::getOrig() const {
    return this->orig;
}

Edge *Edge::getReverse() const {
    return this->reverse;
}

bool Edge::isSelected() const {
    return this->selected;
}

double Edge::getFlow() const {
    return flow;
}

void Edge::setSelected(bool selected) {
    this->selected = selected;
}

void Edge::setReverse(Edge *reverse) {
    this->reverse = reverse;
```

```
}

void Edge::setFlow(double flow) {
    this->flow = flow;
}

void Edge::setService(const std::string &service) {
    this->service = service;
}

std::string Edge::getService() const {
    return this->service;
}
```

5.9 VertexEdge.h

```
// By: Gonçalo Leão

#ifndef DA_TP_CLASSES_VERTEX_EDGE
#define DA_TP_CLASSES_VERTEX_EDGE

#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>


class Edge;

#define INF std::numeric_limits<double>::max()

/************************* Vertex  **************************/

class Vertex {
public:
    Vertex(std::string id);

    bool operator<(Vertex &vertex) const; // // required by MutablePriorityQueue

    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:
    std::string id;           // identifier
    std::vector<Edge *> adj;  // outgoing edges

    // auxiliary fields
    bool visited = false; // used by DFS, BFS, Prim ...
    bool processing = false; // used by isDAG (in addition to the visited attribute)
    unsigned int indegree; // used by topsort
    double dist = 0;
    Edge *path = nullptr;

    std::vector<Edge *> incoming; // incoming edges

    int queueIndex = 0;         // required by MutablePriorityQueue and UFDS
    void print() const;
};


/************************ Edge  ****************************/

class Edge {
public:
    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);

    [[nodiscard]] std::string getService() const;

    void setService(const std::string &service);

protected:
    Vertex *dest; // destination vertex
    int weight; // edge weight, can also be used for capacity

    std::string service;
    // auxiliary fields
    bool selected = false;

    // used for bidirectional edges
    Vertex *orig;
    Edge *reverse = nullptr;

    double flow; // for flow-related problems
};

#endif /* DA_TP_CLASSES_VERTEX_EDGE */
```

Index