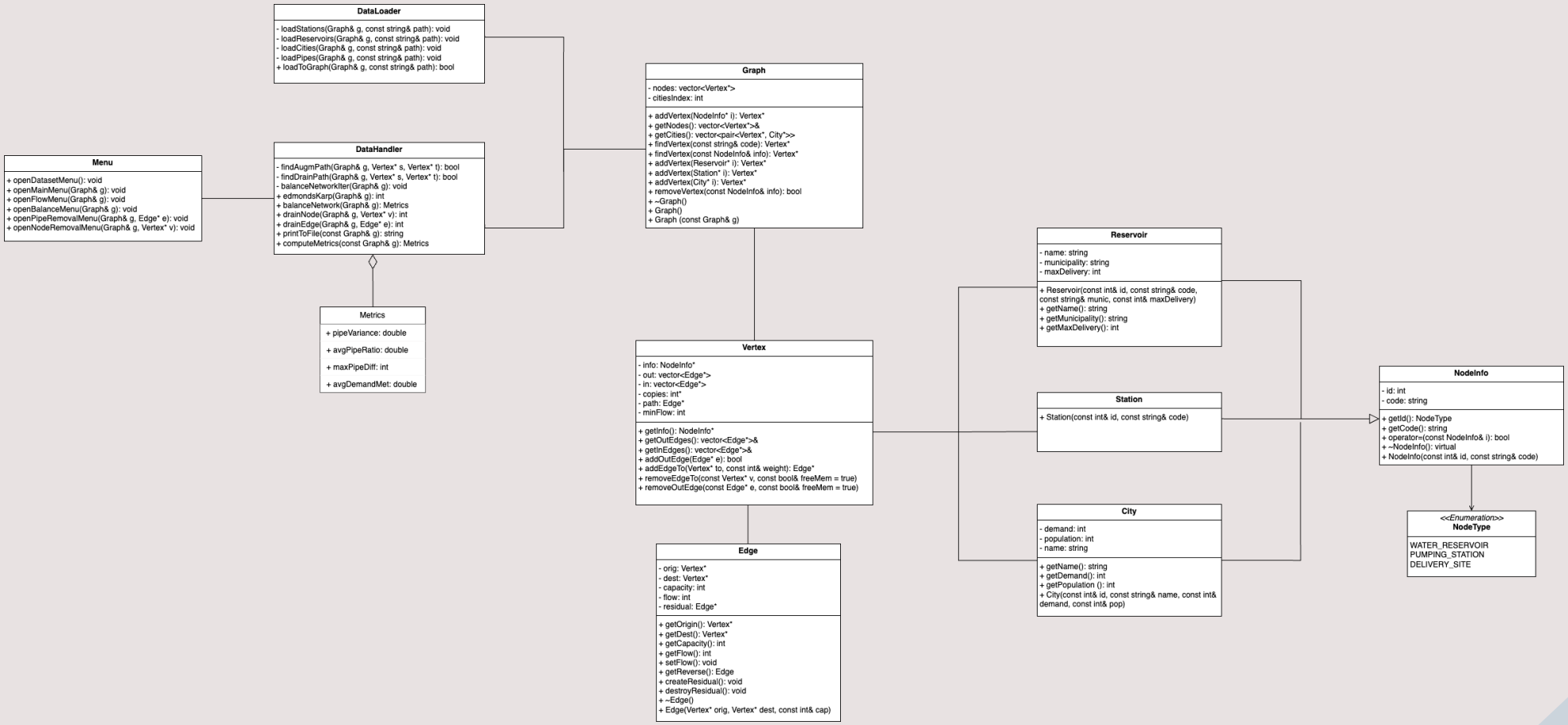
Project 1 -An Analysis Tool for Water Supply Management

Design of Algorithms (DA) - 23/24

The Classes used and their respective Attributes:

- City (const int& id, const string& code, const string& name, const int& demand, const int& pop);
- DataHandler ();
- DataLoader ();
- Edge (Vertex* orig, Vertex* dest, const int& cap);
- NodeInfo (const int &id, const string &code);
- Graph (const Graph& g);
- Reservoir (const int& id, const string& code, const string& name, const string& munic, const int& maxDelivery);
- Station (const int& id, const string& code);
- Vertex (NodeInfo* info);
- Vertex (const Vertex* v)
- Menu….

The Classes used and their respective Attributes:



Description of reading the given dataset:

```cpp
void DataLoader::loadReservoirs(Graph& g, const std::string& path) {

    std::string line;
    std::ifstream stream( s: path+"/Reservoir.csv");
    std::getline( &: stream , &: line);

    while (std::getline( &: stream, &: line)) {

        if (line.back()=='\r') line.pop_back();
        std::istringstream iss( s: line);
        std::string id_, code, name, munic, max_;

        std::getline( &: iss, &: name, dlm: ',');
        std::getline( &: iss, &: munic, dlm: ',');
        std::getline( &: iss, &: id_, dlm: ',');
        std::getline( &: iss, &: code, dlm: ',');
        std::getline( &: iss, &: max_, dlm: ',');

        int id = stoi( str: id_);
        int max = stoi( str: max_);

        // create obj and add it to graph
        auto* res = new Reservoir(id, code, name, munic, maxDelivery: max);
        g.addVertex( i: res);
    }
    stream.close();
}
```

```cpp
void DataLoader::loadCities(Graph& g, const std::string& path) {

    std::string line;
    std::ifstream stream( s: path+"/Cities.csv");
    std::getline( &: stream , &: line);

    while (std::getline( &: stream, &: line)) {

        if (line.back()=='\r') line.pop_back();
        std::istringstream iss( s: line);
        std::string id_, code, name, demand_, pop_;

        std::getline( &: iss, &: name, dlm: ',');
        std::getline( &: iss, &: id_, dlm: ',');
        std::getline( &: iss, &: code, dlm: ',');
        std::getline( &: iss, &: demand_, dlm: ',');
        std::getline( &: iss, &: pop_, dlm: ',');

        int id = stoi( str: id_);
        int demand = stoi( str: demand_);
        int pop = stoi( str: pop_);

        // create object
        auto* city = new City(id, code, name, demand, pop);
        g.addVertex( i: city);
    }
    stream.close();
}
```

```cpp
void DataLoader::loadStations(Graph& g, const std::string& path) {

    std::string line;
    std::ifstream stream( s: path+"/Stations.csv");
    std::getline( &: stream , &: line);

    while (std::getline( &: stream, &: line)) {

        if (line.back()=='\r') line.pop_back();
        std::istringstream iss( s: line);
        std::string id, code;

        // extract data
        std::getline( &: iss, &: id, dlm: ',');
        std::getline( &: iss, &: code, dlm: ',');

        // create object
        auto* station = new Station( id: stoi( str: id), code);
        g.addVertex( i: station);
    }
    stream.close();
}
```

```cpp
void DataLoader::loadPipes(Graph &g, const std::string &path) {

    std::string line;
    std::ifstream stream( s: path+"/Pipes.csv");
    std::getline( &: stream , &: line);

    while (std::getline( &: stream, &: line)) {

        if (line.back()=='\r') line.pop_back();
        std::istringstream iss( s: line);
        std::string c1, c2, cap, dir;

        std::getline( &: iss, &: c1, dlm: ',');
        std::getline( &: iss, &: c2, dlm: ',');
        std::getline( &: iss, &: cap, dlm: ',');
        std::getline( &: iss, &: dir, dlm: ',');

        // create edge
        Vertex* v1 = g.findVertex( code: c1);
        Vertex* v2 = g.findVertex( code: c2);
        v1->addEdgeTo( to: v2, weight: stoi( str: cap));
        if (dir == "0") { // bidirectional edge
            v2->addEdgeTo( to: v1, weight: stoi( str: cap));
        }
    }
    stream.close();
}
```

Description of reading the given dataset:

We read CSV files from this three function, implemented in the class DataLoader.

All this functions follow the same methods:

-loadStations() -> this function reads data from a CSV file named "Stations.csv" located in the specified path. Each line in the CSV file represents a station. The function parses each line, extracts the station ID and code, creates a Station object with this information, and adds it as a vertex to the graph g.

-loadPipes() -> similar to the previous function, this one reads data from a CSV file named "Pipes.csv" in the specified path. Each line represents a pipe connection between two stations, with fields for the IDs of the connected stations, capacity, and direction. It creates edges between the corresponding vertices in the graph g, representing the connections between stations.

-loadReservoirs() -> reads data from a CSV file named "Reservoir.csv" in the specified path. Each line represents a reservoir, with fields for ID, code, name, municipality, and maximum capacity. It creates Reservoir objects and adds them as vertices to the graph g.

-loadCities() -> reads data from a CSV file named "Cities.csv" in the specified path. Each line represents a city, with fields for ID, code, name, demand, and population. It creates City objects and adds them as vertices to the graph g.

Description of the graph used to represent the dataset:

This class provides essential functionalities for managing vertices in the graph, including adding, finding, and removing vertices, as well as accessing information about the graph's nodes.

- **Graph:**

```
#ifndef GRAPH_H
#define GRAPH_H

#include "Edge.h"

class Graph {
    std::vector<Vertex*> nodes;
    Vertex* addVertex(NodeInfo* i);
    // starting index of city nodes
    int citiesIndex = -1;
public:
    ~Graph();
    Graph();
    Graph(const Graph& g);
    const std::vector<Vertex*>& getNodes() const;
    std::vector<std::pair<Vertex*, City*>> getCities() const;
    Vertex* findVertex(const std::string& code) const;
    Vertex* findVertex(const NodeInfo& info) const;
    Vertex* addVertex(Reservoir* i);
    Vertex* addVertex(Station* i);
    Vertex* addVertex(City* i);
    bool removeVertex(const NodeInfo& info);
};

#endif
```

Description of the graph used to represent the dataset:

This class encapsulates functionalities related to vertices in a graph, including managing edges, accessing vertex information, and performing operations like adding and removing edges.

- **Vertex:**

```
#ifndef VERTEX_H
#define VERTEX_H

#include "Reservoir.h"
#include "Station.h"
#include "City.h"

class Edge;

class Vertex {
    NodeInfo* info;
    std::vector<Edge*> out;
    std::vector<Edge*> in;
    // keep track of copies of this Vertex to know
    // when to dealloc. memory for NodeInfo
    int* copies;
    Edge* path;
    int minFlow;
public:
    ~Vertex();
    Vertex(NodeInfo* info);
    Vertex(const Vertex* v);
    NodeInfo* getInfo() const;
    const std::vector<Edge*>& getOutEdges() const;
    const std::vector<Edge*>& getInEdges() const;
    // utility methods
    Edge* addEdgeTo(Vertex* to, const int& weight);
    bool removeEdgeTo(const Vertex* v, const bool& freeMem = true);
    bool removeOutEdge(const Edge* e, const bool& freeMem = true);
    friend class DataHandler;
};

#endif
```

Description of the graph used to represent the dataset:

This class encapsulates functionalities related to edges in a graph, including accessing edge properties, handling flow, and managing residual edges. It's designed specifically to represent pipelines in a graph structure.

- **Edge:**

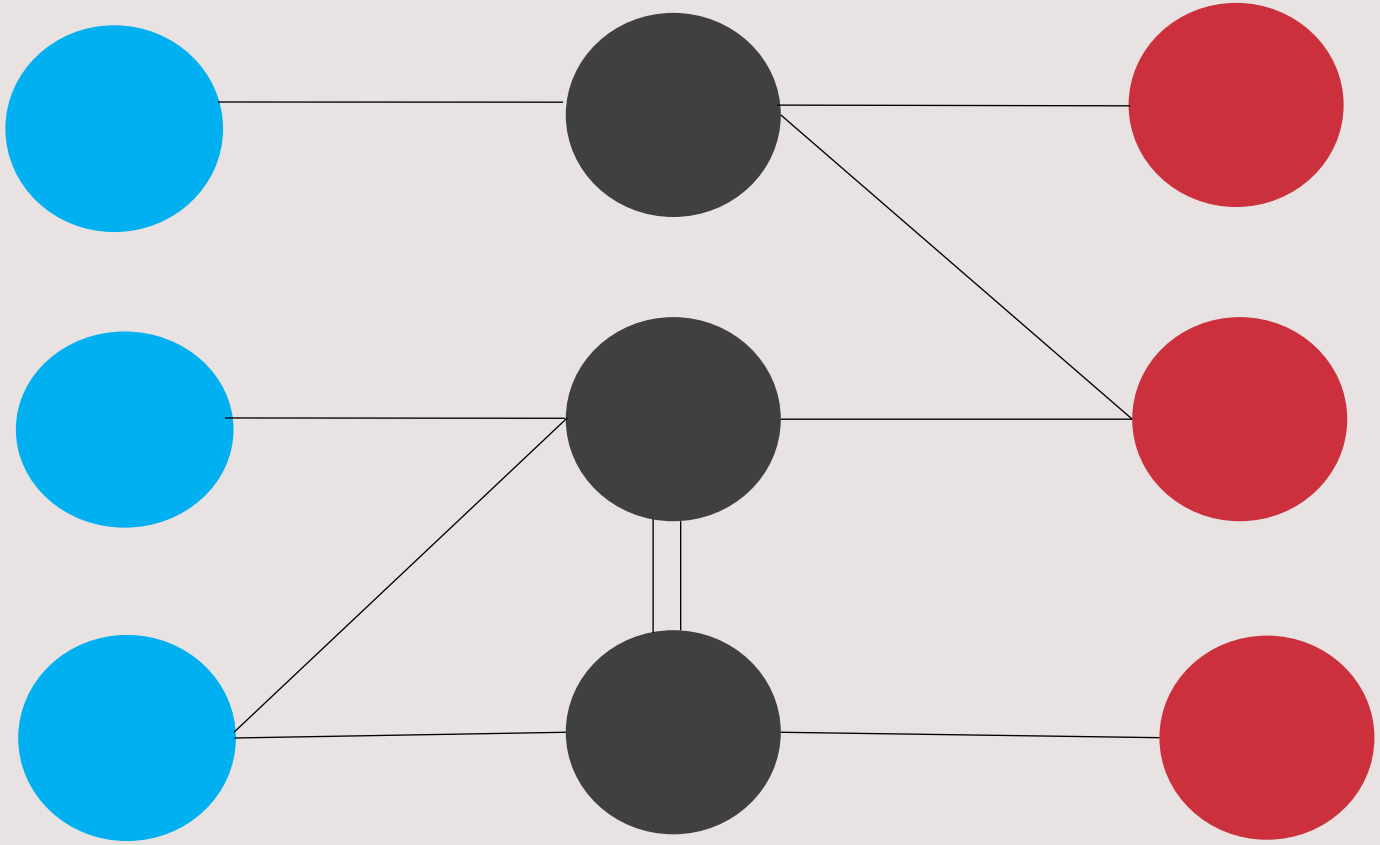
```
#ifndef EDGE_H
#define EDGE_H

#include "Vertex.h"

// Edges represent pipelines
class Edge {
    Vertex* orig;
    Vertex* dest;
    int capacity;
    int flow = 0;
    Edge* residual = nullptr;
public:
    ~Edge();
    Edge(Vertex* orig, Vertex* dest, const int& cap);
    Vertex* getOrigin() const;
    Vertex* getDest() const;
    int getCapacity() const;
    int getFlow() const;
    void setFlow(const int& flow);
    // handle residual counterpart
    Edge* getReverse() const;
    void createResidual();
    void destroyResidual();
};

#endif
```

Graph Representation



Description of implemented functionalities and associated algorithms:

We implemented the following functions, that are in the class DataHandler, to solve the questions :

-static bool findAugmPath(Graph& g, Vertex* s, Vertex* t) -> performs a breadth-first search to find an augmenting path in a graph. It initializes a queue for traversal, resets properties of each vertex, and enqueues the source vertex. During the search, it explores outgoing edges from each vertex, avoiding residual edges leading to the source, and updating vertices along the path. If the target vertex is reached, it returns true; otherwise, it returns false if no augmenting path is found. Complexity: O(E);

-static int edmondsKarp(Graph& g) -> Complexity: O(V * E^2) ;

-static bool findDrainPath(Graph& g, Vertex* s, Vertex* t)-> locates a path within a graph where flow can be drained from a source to a sink vertex. By employing a breadth-first search algorithm, it systematically explores the graph's edges, prioritizing non-residual and non-empty paths to ensure only viable routes are considered. Complexity: O(E);

-static int drainNode(Graph& g, Vertex* v)-> manages flow drainage within a graph. It begins by validating the input vertex v, returning 0 if it's invalid. Then, it iterates through drainable paths from v to the sink and from the source to v, decrementing flow along these paths accordingly. The function accumulates the total drained flow and returns it. Complexity: O(2*E*V);

-static int drainEdge(Graph& g, Edge* e)->manages flow drainage along a specified edge within a graph. It begins by validating the input edge e, returning 0 if it's invalid. Then, it iterates through two drainage phases: first, draining from the destination vertex of the edge to the sink, and then draining from the source to the origin vertex of the edge. During each phase, it iterates through drainable paths, decrementing flow along these paths according to the minimum flow encountered. The function accumulates the total drained flow and sets the flow of the edge to 0 before returning the drained amount. Complexity: O(2*E*V);

Description of implemented functionalities and associated algorithms:

-static string printToFile(const Graph& g)-> Print water network supply status to a file. Complexity: O(c), c -> no. of cities in the graph;

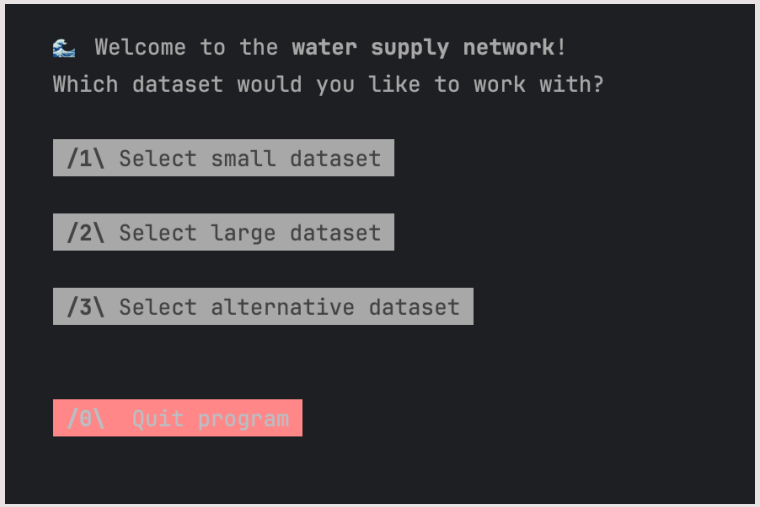
-static Metrics computeMetrics(const Graph& g)-> evaluates water supply network performance through metrics computation, including average demand met, basic pipe metrics, and pipe variance, aiding in network optimization and resource allocation decisions. Complexity: O(2*c + 2*E), c -> no. of cities in the graph;

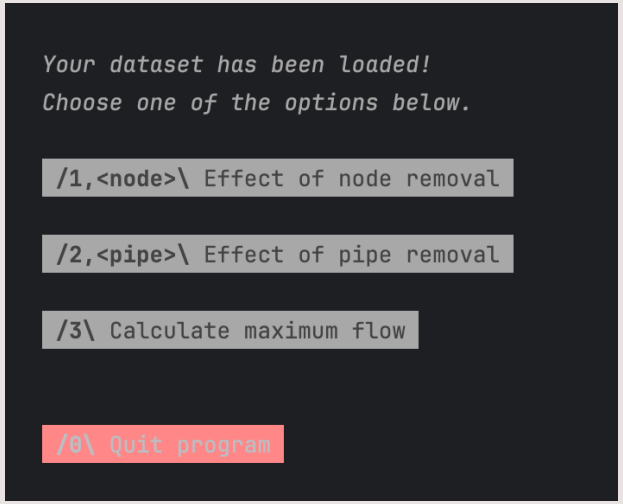
-static void balanceNetworkIter(Graph& g)-> conducts an iteration of a heuristics-based algorithm aimed at enhancing the performance of a water supply network graph. By iteratively identifying high flow pipes and redirecting flow to underutilized paths, the function strives to achieve a more balanced flow distribution within the network. This iterative approach helps optimize resource utilization and mitigate flow imbalances, contributing to the overall efficiency and effectiveness of the water supply system. Complexity: O(E^2 * 2*V);

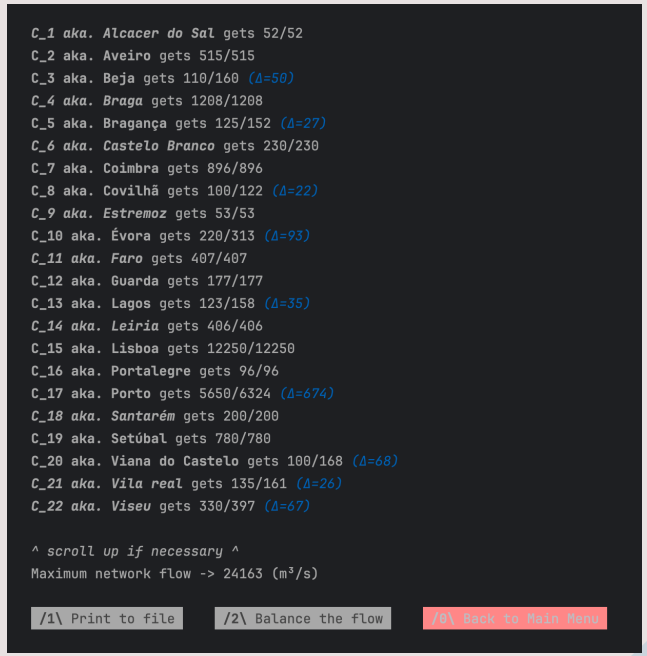
-static Metrics balanceNetwork(Graph& g)-> teratively optimizes the flow distribution within a network by rerouting flow through different pipes using heuristics, aiming to minimize variance in pipe flow ratios and enhance overall network performance. Complexity: best case: O(E^2 * 2*V), worst case: impossible to predict;

Description of the user interface:

The Menu class is responsible for handling user interaction and displaying menus related to the Water Supply Management







One or two functionality(s) to highlight:

Our program offers numerous features that allow users to have a comprehensive, objective, and informative experience.

The highlight in our work is the way that we implemented the questions T2.3, T3.1, T3.2 and T3.3.

Main difficulties :

During the execution of this project, we encountered some difficulties and errors along the way, which fortunately we managed to overcome.

Participation of each group member :

- Beatriz Sonnemberg (up202206098) –> 33,(3)%

- Gonçalo Sousa (up202207320) –> 33,(3)%

- Nuno Rios (up202206272) -> 33,(3)%