WATER SUPPLY MANAGEMENT

Group 12_9

Rui Cruz up202208011

Miguel Guerrinha up20225038

Tiago Teixeira up202208511

PROJECT OBJECTIVE

- This project consists of developing a coded system that can track the water supply network in Portugal.
- From there, different information can be taken from the nodes, representing pumping stations, reservoirs and cities, and adaptations can be made to the network to simulate section failures, optimal allocations and service disruptions.

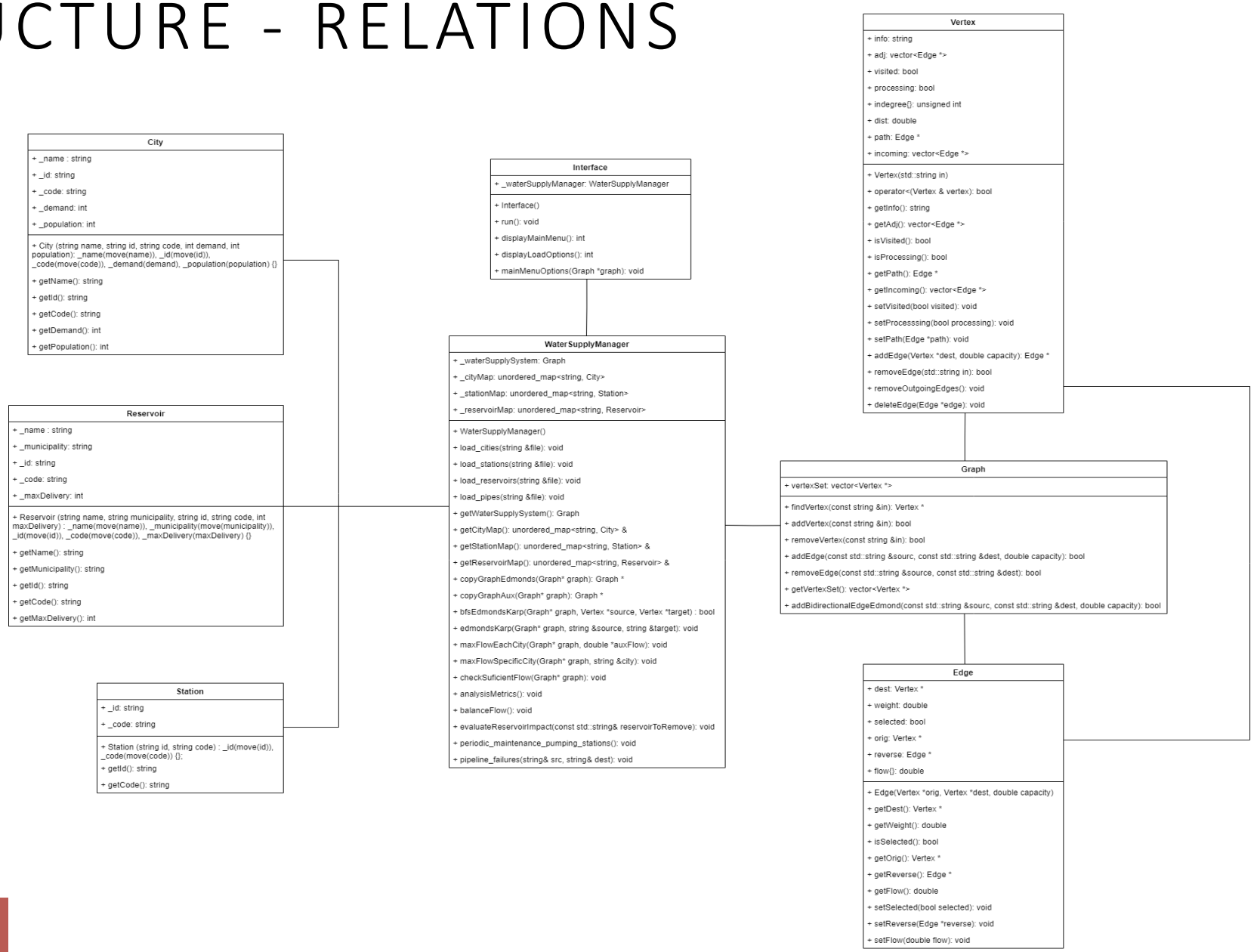
INDEX

0. SYSTEM STRUCTURE

0. SYSTEM STRUCTURE - RELATIONS

List of Classes:

- Inferface
- WaterSupplyManager
- Graph
- Vertex
- Edge
- City
- Reservoir
- Station



0. SYSTEM STRUCTURE - FILES

- City .h/.cpp :: class with Get functions -> stored in **unordered_map<string, City> _cityMap**
- Reservoir .h/.cpp :: class with Get functions -> stored in **unordered_map<string, Reservoir> _reservoirMap**
- Station .h/ .cpp :: class with Get functions -> stored in **unordered_map<string, Station> _stationMap**
- Pipeline .h/.cpp :: class with Get functions
- Interface .h/ .cpp :: Runs the System, draws the Menus and creates the Graph Instance
- WaterSupplyManager .h/.cpp :: Creates Graph by reading .csv files and contains functions to use in the system
- main.cpp :: Runs the Interface function “run()"

1. PARSING OF INPUT DATA

1. PARSING OF INPUT DATA - GRAPH

Our graph consists of a **vector<Vertex *>**, with each vertex saving the **code (string)** of its node type (reservoir, pumping station or city) and edges, with a **destination** node reference and the weight associated, which represents a pipe's **capacity**.

Because of this approach, we decided to change the Vertex Class to hold a string variable (in other words, its info) to represent each of the nodes, since the three different types mentioned above only coincide in this parameter.

```
class WaterSupplyManager {
private:
    /// Representa o sistema de abastecimento de água como um grafo
    Graph _waterSupplySystem;

    /// Hash table tendo o código de uma cidade associado a um objeto CIty
    std::unordered_map<std::string, City> _cityMap;

    /// Hash table tendo o código de uma estação de bombeamento associado a um objeto Station
    std::unordered_map<std::string, Station> _stationMap;

    /// Hash table tendo o código de um reservatório de água associado a um objeto Reservoir
    std::unordered_map<std::string, Reservoir> _reservoirMap;
```

The data structures used, from **WaterSupplyManager** class:

- *_waterSupplySystem* is the graph that represents the network
- three *unordered maps* are used to store information for each of the node types, that can be accessed by the respective get functions for each of those classes.

1. PARSING OF INPUT DATA - DATASET

The reading of the .csv files is done by the sequential call of 4 different loading functions from the WaterSupplyManager.cpp. While the first three are used to load the respective nodes into the graph (reservoirs, pumping stations and cities), the last one is used for its edges (pipelines). All the functions below have a time complexity of **O(n)**.

Node builders:

- void load_cities(const std::string &file) :: reads a city csv, adds nodes with city code strings to the graph and saves each of those on an unordered map for cities.
- void load_stations(const std::string &file) :: reads a station csv, adds nodes with station code strings to the graph and saves each of those on an unordered map for stations.
- void load_reservoirs(const std::string &file) :: reads a reservoir csv, adds nodes with reservoir code strings to the graph and saves each of those on an unordered map for reservoirs.

Edge builder:

- void load_pipes(const std::string &file) :: reads a pipe csv, adds edges according to existing nodes and deals with edge bidirectionality by adding the edge reversed.

2. BASIC SERVICE METRICS

2. BASIC SERVICE METRICS

T2.1.1.: Maximum Amount of Water Reaching each City

A Maximum Flow algorithm (with Edmonds-Karp) is used to determine the optimal flow through the network:

- void maxFlowEachCity(Graph* graph, double *auxFlow) > Time Complexity: O(VE + (V * (E^2)) + 2V)

A “main” source and target are created and added to the graph and connected to each “existing” source and sink. Then, the following function is called:

- void edmondsKarp(Graph* graph, const string &source, const string &target) > Time Complexity: O(V * (E^2))

Where the parameters *source* and *target* are the new “main” nodes added. Once finished, they are removed from the graph.

T2.1.2.: Maximum Amount of Water Reaching a Specific City

To calculate the Max Flow for one City, the function:

- void maxFlowSpecificCity(Graph* graph, const string &city) > Time Complexity: O(VE + (V * (E^2)) + 3V)

calls the first function from 2.1.1 and after collecting the maxflow value for the city passed as a parameter, it prints the result.

2. BASIC SERVICE METRICS

T2.2.: Checking for Desired Water Rate Level in all Cities

- void checkSuficientFlow(Graph* graph) > Time Complexity: O(VE + (V * (E^2)) + 3V)

Using a similar algorithm from the previous functions, a maxflow value is calculated for each city and, if that value does not meet that city's demand, the deficit is printed to the console.

T2.3.: Initial Metrics and Balancing Algorithm

This functionality makes use of the function: void balanceFlow() > Time Complexity: O(VE + V(E + E))

In the beginning and end of its body, void analysisMetrics(), with Time Complexity: O(2VE + 2N), is called to determine and print out the metrics (mean, variance and difference), so that a comparison can be made visually, on the console.

In the graph, there are a few reservoirs that aren't deducting the maximal flow that they provide to their destinations. So, to make a better use of this spare flow, we could select their outgoing pipes that have a ratio that is lower than the mean, and send a part of the unused flow to those, in order to balance them out to a value that is closer to the mean.

This would be possible because reservoirs are only connected to pumping stations and these, in some cases, end up receiving more than they deliver, which means: we can select pipes that send flow to pumping stations, that happen to be above the ratio, and redirect a part of that flow to other pipes, that have a ratio below the mean.

3. SYSTEM FAILURE SIMULATION

3. SYSTEM FAILURE SIMULATION

T3.1.: Water Reservoir out of Commission

To check the impact of a reservoir out of service on the various delivery sites, the function:

- void evaluateReservoirImpact(const std::string& reservoirToRemove)
  > Time Complexity: O(E + (VE + (V * (E^2)) + 3V) + N)

removes temporarily the reservoir passed as a parameter (saving its edges in the process) and calls checkSuficientFlow() to calculate the new changes. Once that's done, the reservoir and its edges are added to the graph once again.

The pseudo-code shown simulates a possible solution that would only need to calculate the baseline Max Flow a few times, if not once. It starts with the last function and terminates once all reservoirs, read from the _reservoirMap, have been removed. The idea would be to analyse the flow in pipes that were coming from the removed reservoir and decrement their value on the total Max Flow.

```
function updateFlowForReservoirRemoval(graph, reservoir):
    graph.removeVertex(reservoir)
    // Update flow calculations incrementally
    // Adjust flow paths and capacities affected by the removal
    // Logic for updating flow calculations

function evaluateImpact(graph):
    // Evaluate the impact on delivery capacity and identify affected cities
    // Return updated flow or impact metric

function compareResults(MaxFlowsByRemoval, baselineMaxFlow):
    // Compare results obtained from incremental updates with baseline Max-Flow
    // Show differences or changes in flow capacity after each reservoir removal

function incrementalMaxFlowUpdate():
    Graph newGraph = copyGraphAux(&graph);
    vector<double> MaxFlowsByRemoval;
    double baselineMaxFlow;
    maxFlowEachCity(newGraph, baselineMaxFlow); // Calculate baseline Max-Flow

    // For each reservoir to be removed
    for reservoir in _reservoirMap:

        // Update flow for reservoir removal
        updateFlowForReservoirRemoval(newGraph, reservoir);

        double impactAnalysis = evaluateImpact(newGraph); // Evaluate impact
        MaxFlowsByRemoval.push_back(impactAnalysis); // Save current flow

    // Compare results with baseline Max-Flow calculation
    compareResults(MaxFlowsByRemoval, baselineMaxFlow);
```

3. SYSTEM FAILURE SIMULATION

T3.2.: Pumping Station out of Service

- void periodic_maintenance_pumping_stations()

> Time Complexity: O((2+V)(VE + (V * (E^2)) + 2V) + 2VE + N^2 + N)

For this problem, the function above, starts by calculating the total Max Flow with maxFlowEachCity() and saving the city flows in a vector **flowBefore** (later used for comparisons). Then, by the removal of stations, incrementally, using maxFlowEachCity() once again, the new Max Flow value is compared to the baseline value. If these are the same, it means the station removed does not affect the network's max flow.

For the second part of the function, the user can check which cities were affected most, by the removal of a pumping station (on input). The auxiliar function mentioned above, is used once more to make the comparison with the baseline flow and cities affected are placed in a vector **flowAfter**. Finally, the results collected by each vector are placed in a new one, **flowRatio**, for the user to check by input.

3. SYSTEM FAILURE SIMULATION

T3.3.: Pipeline Failure

(This functionality uses maxFlowEachCity() like the previous one)

- void pipeline_failures(const std::string& src, const std::string& dest)

> Time Complexity: O(2(VE + (V * (E^2)) + 2V) + 2VE + N^2)

To simulate a pipeline's failure between two nodes, two string codes are passed as parameters to represent the source node and destination node, respectively.

The Max Flow is calculated twice: first time, as a baseline value, and after that, without the edge/pipe that was removed.

For each calculation, cities and their respective codes are placed on vectors: **flowCitiesBefore** and **flowCitiesAfter**, so that the results can be compared in the end, to show water supply improvements and losses for each city.

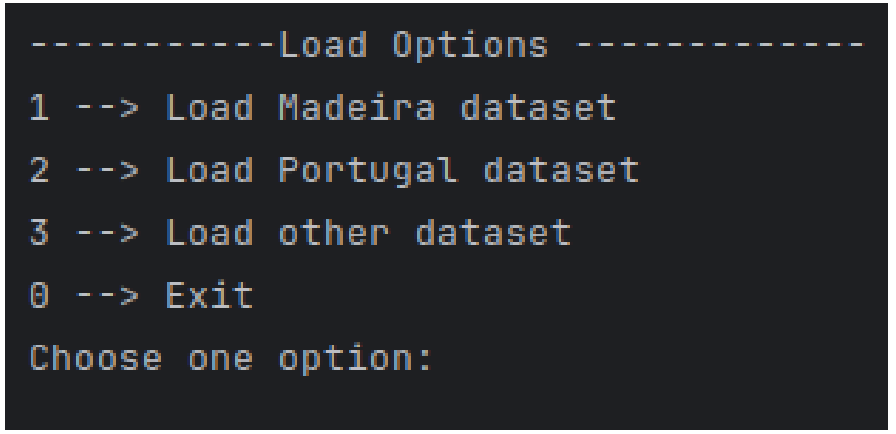
4. INTERFACE

4. INTERFACE

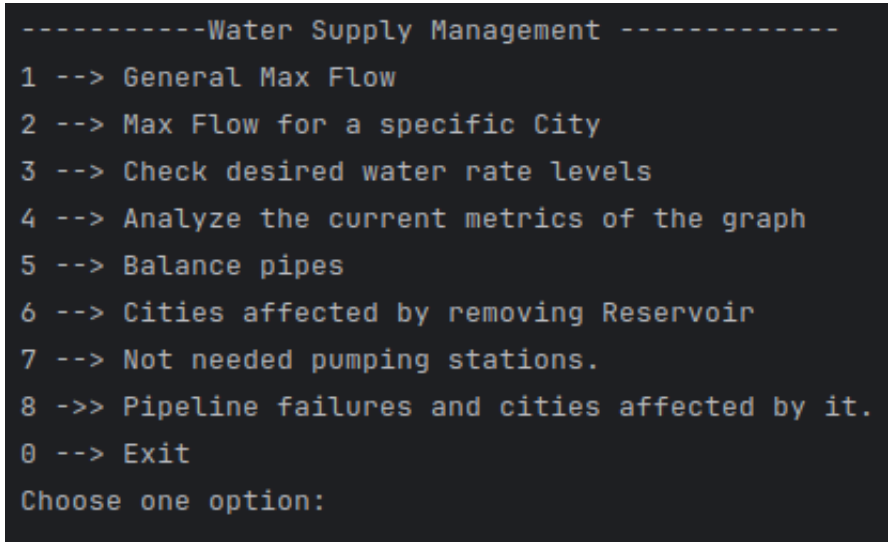
Our interface is based on two menus:

- The first one is presented when the code starts. It is responsible for loading one of the predefined datasets or inserting user-made datasets to the system. Option 3 will ask the user to input the file path for each of the nodes (reservoirs, pumping stations and cities) and edges (pipes).

- The second menu is shown right after the first menu's choice is selected. It provides 8 diferent options to access each of the functionalities we presented before, regarding basic service metrics and reliability and sensitivity to failures. Some of the functionalities will require the user to input some sort of value, such as node codes. Selecting 0 closes the menu and terminates the system.



Menu 1 – Dataset Loader



Menu 2 – Functionalities

4. INTERFACE – QUICK EXAMPLE

Suppose that for this example you had selected the large Dataset on Menu 1 and now you selected option 7 (**Pumping Station out of Service**). Upon selection, you are shown the list of results along with a new “y/n” choice:

```
Choose one option: 7
PS_2
PS_7
PS_25
PS_33
PS_66
PS_67
PS_78
There are 7 pumping stations pumping stations that can be temporarily deactivated without affecting the network's max flow.
Do you want to see the cities most affected if we disable a particular pumping station? (y/n) y
What is the code for the desired pumping station? PS_3
Were affected 2 cities, how many do you want to see? 2
The city of Braga saw a decrease of 33.77% in its total incoming flow.
The city of Bragança saw a decrease of 12% in its total incoming flow.
```

The user can then write the code for a pumping station to check how it impacts the network. After a quick calculation from the system, the user can check the cities affected by the new change.

5. CONCLUSIONS

5. CONCLUSIONS - HIGHLIGHTS

- A basic yet useful setting we would like to point out, is the fact that our inputs are stricted, that is, if the user tries to input something that is not asked, the system won't take it.
- Another feature to mention is the saving of results from the general max flow into a .csv file. This is done automatically after the user selects the option for "General Max Flow" on the second menu.

5. CONCLUSIONS - DIFFICULTIES

- The main difficulty we faced while developing the system was in functionality 2.3, more specifically, in the creation of a balancing algorithm.

PARTICIPATION

- We divided the tasks for the three of us and everyone completed their goals and when necessary, we helped each other.