# AN ANALYSIS TOOL FOR WATER SUPPLY MANAGEMENT

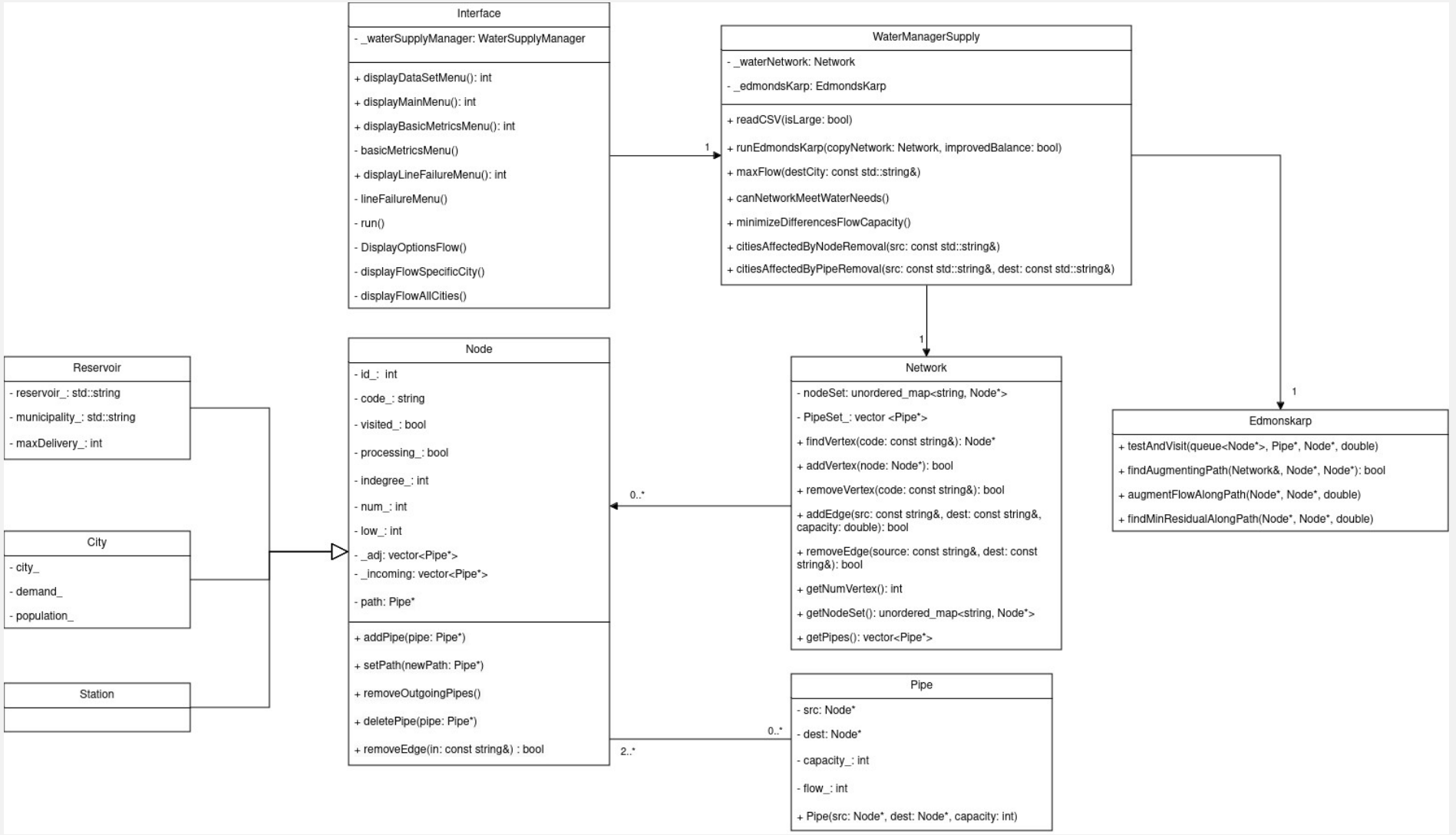
Done By:

Carlos Oliveira

Yves Bonneau

Sara Azevedo

# CLASS DIAGRAM



# DATASET

```
void WaterSupplyManager::readCSV(bool isLarge) {
    std::string line;
    std::ifstream citiesFile;
    std::ifstream stationsFile;
    std::ifstream pipesFile;
    std::ifstream reservoirsFile;

    if (isLarge) {
        citiesFile.open( s: "../dataset/Project1LargeDataSet/Cities.csv");
        stationsFile.open( s: "../dataset/Project1LargeDataSet/Stations.csv");
        pipesFile.open( s: "../dataset/Project1LargeDataSet/Pipes.csv");
        reservoirsFile.open( s: "../dataset/Project1LargeDataSet/Reservoir.csv");
    }
    else {
        citiesFile.open( s: "../dataset/Project1DataSetSmall/Cities_Madeira.csv");
        stationsFile.open( s: "../dataset/Project1DataSetSmall/Stations_Madeira.csv");
        pipesFile.open( s: "../dataset/Project1DataSetSmall/Pipes_Madeira.csv");
        reservoirsFile.open( s: "../dataset/Project1DataSetSmall/Reservoirs_Madeira.csv");
    }

    getline( &: citiesFile,  &: line);
    getline( &: stationsFile,  &: line);
    getline( &: pipesFile,  &: line);
    getline( &: reservoirsFile,  &: line);
```

```
    // Lines with empty fiels at the end should be getting automatically ignored
    while (getline( &: citiesFile,  &: line)) {
        std::stringstream lineStream( s: line);
        string id, demand, population, code, cityName;
        if (std::getline( &: lineStream,  &: cityName,  dlm: ',')
            && getline( &: lineStream,  &: id,  dlm: ',')
            && getline( &: lineStream,  &: code,  dlm: ',')
            && getline( &: lineStream,  &: demand,  dlm: ',')
            && getline( &: lineStream,  &: population)) {
            if (population.back() == '\r') population.pop_back(); // Remove last character (which is a '\r' character)
            if (!isLarge) {
                population.erase( first: std::remove( first: population.begin(),  last: population.end(),  value: '\"'),  last: population.end());  // Remove double quo
                population.erase( first: std::remove( first: population.begin(),  last: population.end(),  value: ','),  last: population.end());
            }
            auto *city = new City( city: cityName,  id: stoi( str: id), code,  demand: stoi( str: demand),  population: stoi( str: population));
            _waterNetwork.addVertex( node: city);
        }
    }

    while (getline( &: stationsFile,  &: line)) {
        // Skip empty lines (like last line in Stations_Madeira.csv)
        if (std::all_of( first: line.begin(),  last: line.end(),  pred: [](const char c) -> bool { return c == ','; })) {
            continue;
        }

        std::stringstream lineStream( s: line);
        string id, code;
        if (std::getline( &: lineStream,  &: id,  dlm: ',') && getline( &: lineStream,  &: code,  dlm: ',')) {
            if (code.back() == '\r') code.pop_back(); // Remove last character (which is a '\r' character)
```

# DATASET

```
while (getline( &: stationsFile, &: line)) {
    // Skip empty lines (like last line in Stations_Madeira.csv)
    if (std::all_of( first: line.begin(), last: line.end(), pred: [](const char c) -> bool { return c == ','; })) {
        continue;
    }

    std::stringstream lineStream( s: line);
    string id, code;
    if (std::getline( &: lineStream, &: id, dlm: ',') && getline( &: lineStream, &: code, dlm: ',')) {
        if (code.back() == '\r') code.pop_back(); // Remove last character (which is a '\r' character)
        auto station = new Station( id: stoi( str: id), code);
        _waterNetwork.addVertex( node: station);
    }
}

while (getline( &: reservoirsFile, &: line)) {
    std::stringstream lineStream( s: line);
    string name, municipality, id, code, maxDelivery;
    if (std::getline( &: lineStream, &: name, dlm: ',')
        && getline( &: lineStream, &: municipality, dlm: ',')
        && getline( &: lineStream, &: id, dlm: ',')
        && getline( &: lineStream, &: code, dlm: ',')
        && getline( &: lineStream, &: maxDelivery, dlm: ',')) {
        if (maxDelivery.back() == '\r') maxDelivery.pop_back(); // Remove last character (which is a '\r' character)
        auto *reservoir = new Reservoir( reservoir: name, municipality, id: stoi( str: id), code, maxDelivery: stoi( str: maxDelivery));
        _waterNetwork.addVertex( node: reservoir);
    }
}
```

```
    while (getline( &: pipesFile, &: line)) {
        std::stringstream lineStream( s: line);
        string direction, src, dest, capacity;
        if (getline( &: lineStream, &: src, dlm: ',')
            && getline( &: lineStream, &: dest, dlm: ',')
            && getline( &: lineStream, &: capacity, dlm: ',')
            && getline( &: lineStream, &: direction, dlm: ',')) {
            //if last character is a '\r' character, remove it
            if (direction.back() == '\r') direction.pop_back();

            auto source : Node * = _waterNetwork.findVertex( code: src);
            auto destination : Node * = _waterNetwork.findVertex( code: dest);

            if (source == nullptr || destination == nullptr) {
                cout << "Error: Source or destination vertex does not exist" << endl;
            }
            _waterNetwork.addEdge(src, dest, capacity: stoi( str: capacity));
            if (stoi( str: direction) == 0) {
                _waterNetwork.addEdge( src: dest, dest: src, capacity: stoi( str: capacity));
            }
        }
    }
}
```

# GRAPHS USED FOR THE DATASET

The graph used is very similar to the provided in the lectures, with some modifications: an unordered map mapping the structure code to a Node pointer was used instead of a vector. We also used inheritance for the vertexes (City, Reservoir, Pumping Station all inherit Node).

```
class Node {
    friend class Pipe;

protected:
    int id_;
    string code_;
    bool visited_{false};
    bool processing_{false};
    int indegree_{0};
    int num_{0};
    int low_{0};
    vector<Pipe *> _adj;
    vector<Pipe *> _incoming;
    Pipe *path{nullptr};

public:
    Node(int id, string cod);
    virtual ~Node();
    // Getters
    int getID() const;
    string getCode() const;
    bool isVisited() const;
    bool isProcessing() const;
    int getIndegree() const;
    int getNum() const;
    int getLow() const;
    [[nodiscard]] const vector<Pipe *> &getAdj() const;
    [[nodiscard]] const vector<Pipe *> &getIncoming() const;
    [[nodiscard]] Pipe *getPath() const;
```

```
    //setters
    void setID(int newID);
    void setCod(string newCod);
    void setVisited(bool visited);
    void setIsProcessing(bool processing);
    void setIndegree(int newIndegree);
    void setNum(int num);
    void setLow(int low);
    void setAdj(const vector<Pipe *> &adj);
    bool operator==(const Node &rhs) const;
    bool operator!=(const Node &rhs) const;
    void addPipe(Pipe *pipe);
    void setPath(Pipe *newPath);
    void removeOutgoingPipes();
    void deleteEdge(Pipe * pipe);
    bool removeEdge(const string &in);
;
```

# GRAPHS USED FOR THE DATASET

```cpp
class City final : public Node {
public:
    City(const string &city, const int id, const string &code, const int demand,
         const int population): Node(id, cod: code),
        city_(city),
        demand_(demand),
        population_(population) {
    }
    string getCity() const { return city_; }
    int getDemand() const { return demand_; }
    int getPopulation() const { return population_; }
    void setCity(const string &newCity) { city_ = newCity; }
    void setDemand(const int newDemand) { demand_ = newDemand; }
    void setPopulation(const int newPopulation) { population_ = newPopulation; }

private:
    string city_;
    int demand_;
    int population_;
};
```

City.h

```cpp
class Station final : public Node {
public:
    //Constructor
    Station(int id, string code) : Node(id, cod: code){
    }

};
```

Station.h

```cpp
class Reservoir final : public Node {
public:
    Reservoir(const string &reservoir, const string &municipality, const int id, const string &code,
              const int maxDelivery) : Node(id, cod: code),
                                       reservoir_(reservoir),
                                       municipality_(municipality),
                                       maxDelivery_(maxDelivery) {
    }
    string getReservoir() const { return reservoir_; }
    string getMunicipality() const { return municipality_; }
    int getMaxDelivery() const { return maxDelivery_; }
    void setReservoir(const string &newReservoir) { reservoir_ = newReservoir; }
    void setMunicipality(const string &newMunicipality) { municipality_ = newMunicipality; }
    void setMaxDelivery(const int newMaxDelivery) { maxDelivery_ = newMaxDelivery; }

private:
    string reservoir_;
    string municipality_;
    int maxDelivery_;
};
```

Reservoir.h

# FUNCTIONALITIES AND ASSOCIATED ALGORITHM'S

The Edmonds Karp algorithm was the main algorithm used in the project. In the first exercise, we used the algorithm provided in the lessons to calculate the maxflow for each city.

```
class Edmonskarp {
public:
    static void testAndVisit(queue<Node *> &q, Pipe *pipe, Node *dest,
                             double residual);
    static bool findAugmentingPath(Network &network, Node *s, Node *t,
                                   bool improvedBalance);
    static double findMinResidualAlongPath(Node *s, Node *t);
    static void augmentFlowAlongPath(Node *s, Node *t, double f);
    static void edmondsKarp(Network &network, const std::string &sourceCode,
                            const std::string &targetCode, bool improvedBalance) ;

    };
```

```
void WaterSupplyManager::maxFlow(const std::string &destCity) {
    Network copyNetwork = _waterNetwork;
    runEdmondsKarp(copyNetwork, improvedBalance: false);

    // Print the flow for a specific city or for all of them, depending on the choice of the user
    if (!destCity.empty()) {
        City *cityNode = nullptr;
        for (auto [fst : const string , snd : Node * ] : copyNetwork.getNodeSet()) {
            if (auto city = dynamic_cast<City *>(snd)) {
                if (city->getCity() == destCity) {
                    cityNode = dynamic_cast<City *>(snd);
                    break;
                }
            }
        }
        if (cityNode) {
            int flow = 0;
            for (const auto pipe : Pipe *const  : cityNode->getIncoming()) {
                flow += pipe->getFlow();
            }
            cout << "City: " << cityNode->getCity() << " | Code: " << cityNode->getCode() << " | Flow: " << flow << endl;
        } else {
            cout << "City not found. " << destCity << endl;
        }
    } else {
        for (auto [fst : const string , snd : Node * ]: copyNetwork.getNodeSet()) {
            Node *node = snd;
            if (const auto city = dynamic_cast<City *>(node)) {
                int flow = 0;
```

# FUNCTIONALITIES AND ASSOCIATED ALGORITHM'S

For the 2.2 the approach was similar.

```
void WaterSupplyManager::canNetworkMeetWaterNeeds() {
    //Using the same approach as before
    Network copyNetwork = _waterNetwork;
    runEdmondsKarp(copyNetwork, improvedBalance: false);

    // Check if each city's demand is met
    for (auto &[_ : const string , node : Node * ] : _waterNetwork.getNodeSet()) {
        if (City *city = dynamic_cast<City *>(node)) {
            int demand = city->getDemand();
            int actualFlow = 0;
            for (const auto pipe : Pipe *const : city->getIncoming()) {
                actualFlow += pipe->getFlow();
            }
            int deficit = demand - actualFlow;
            if (deficit > 0) {
                std::cout << city->getCode() << "-" << city->getCity() << std::endl;
                std::cout << "Deficit: " << deficit << std::endl;
                std::cout << std::endl;
            }
        }
    }
}
```

In 2.3, we implemented a calculateMetrics function to discovery the average, variance and max difference but we were unable to find a correct solution

```
    void calculateMetrics(Network &network) {
        std::vector<Pipe *> pipes = network.getPipes();
        int n = pipes.size();
        double sum = 0.0;
        double sumSq = 0.0;
        maxDifference = -std::numeric_limits<double>::max();

        for (Pipe *pipe: pipes) {
            double difference = pipe->getCapacity() - pipe->getFlow()
            sum += difference;
            sumSq += difference * difference;
            maxDifference = std::max(maxDifference, difference);
        }

        average = sum / n;
        variance = (sumSq / n) - (average * average);
    }
};
```

# FUNCTIONALITIES AND ASSOCIATED ALGORITHM'S

For exercise 3 we had difficulties implementing a solution where running Edmonds Karp on the entire network was not necessary. We have although found a possible solution, which would involve calculating the strongly connected components of the network, finding out if the structure to be removed is part of a SCC and if so, to isolate and run Edmonds Karp on this smaller network. We would then compute t[illegible]hich are affected.

```
void WaterSupplyManager::citiesAffectedByNodeRemoval(const std::string &src) {
    runEdmondsKarp( copyNetwork: _waterNetwork, improvedBalance: false);

    unordered_map<string, int> citiesFlow;
    //Store a copy of the flow for each city
    for (auto &[_ : const string , node : Node * ] : _waterNetwork.getNodeSet()) {
        if (auto city = dynamic_cast<City *>(node)) {
            int flow = 0;
            for (const auto pipe : Pipe *const : city->getIncoming()) {
                flow += pipe->getFlow();
            }
            citiesFlow[city->getCode()] = flow;
        }
    }

    //Removed the reservoir from the network (make a copy before so you cann add it back later)
    _waterNetwork.removeVertex( code: src);
    runEdmondsKarp( copyNetwork: _waterNetwork, improvedBalance: false);

    //Check the flow for each city and compare it with the previous flow (if it is smaller, print the city)
    bool flag = false;
    for (auto &[_ : const string , node : Node * ] : _waterNetwork.getNodeSet()) {
        if (auto city = dynamic_cast<City *>(node)) {
            int flow = 0;
            for (const auto pipe : Pipe *const : city->getIncoming()) {
                flow += pipe->getFlow();
            }
            if (flow < citiesFlow[city->getCode()]) {
                cout << "City: " << city->getCity() << " | Code: " << city->getCode() << " | Old Flow: " <<
```

# USER INTERFACE

```cpp
void Interface::run() {
    system("clear");
    bool running = true;
    while (running) {
        switch (displayDataSetMenu()) {
            case 1:
                _waterSupplyManager.readCSV( isLarge: false);
                running = false;
                break;
            case 2:
                _waterSupplyManager.readCSV( isLarge: true);
                running = false;
                break;
            default:
                std::cout << "Please, insert a valid number" << std::endl;
                std::cout << "Press enter to continue...";
                std::cin.ignore();
                std::cin.get();
                system("clear");
        }
    }

    running = true;

    while (running) {
        switch (displayMainMenu()) {
            case 0:
                std::cout << "Goodbye!" << std::endl;
                std::cout << "Press Enter to exit.";
```

```cpp
    while (running) {
        switch (displayMainMenu()) {
            case 0:
                std::cout << "Goodbye!" << std::endl;
                std::cout << "Press Enter to exit.";
                std::cin.ignore();
                std::cin.get();
                system("clear");
                running = false;
                break;
            case 1:
                basicMetricsMenu();
                break;
            case 2:
                lineFailureMenu();
                break;
            default:
                std::cout << "Please, insert a valid number" << std::endl;
                std::cout << "Press enter to continue...";
                std::cin.ignore();
                std::cin.get();
                system("clear");
        }
    }
}
```

# USER INTERFACE

```
Please, select the dataset you want to use:
1. Small Dataset
2. Large Dataset
```

Menu to select the database

```
========== Main Menu ==========
=======================================================
 1. Basic Service Metrics
 2. Reliability and Sensitivity to Line Failures

 Enter your choice (1-2)
 Press '0' to exit.

=======================================================
```

Main Menu

```
========== Basic Service Metrics ==========
=============================================================

 1. Maximum amount of water that can reach each or a specific city.
 2. Can an existing network configuration meet the water needs of its customer?
 3. Show that you can minimize the differences of flow to capacity on each pipe across the entire network.

 Enter your choice (1-3)
 Press '0' to return

=============================================================
```

Basic Service Metrics

```
========= Reliability and Sensitivity to Line Failures =========
================================================================

 1. Which cities are affected if one water reservoir is out of service
 2. Which cities are affected if one pumping station is out of service
 3. Which cities are affected if on pipe is out of service

 Enter your choice (1-3)
 Press '0' to exit.

================================================================
```

Reliablility and Sensitivity to Line Failures

# MAIN DIFFICULTIES

During the project implementation, we faced significant challenges in comprehending and executing exercise 2.3. The exercise required us to balance the load across the network in a way that minimizes the flow to capacity differences on all pipes throughout the entire network. Moreover, we encountered difficulty in exercise 3, as we struggled to find an alternative method that could produce the same outcome without having to rerun the algorithm.

# TEST CASES AND EXAMPLES

This are the test cases for the exercises 2.1 and 2.2 with the small dataset, respectively :

C_1-Porto Moniz 18

C_2-São Vicente 34

C_3-Santana 46

C_4-Machico 137

C_5-Santa Cruz 295

C_6-Funchal 664

C_7-Câmara de Lobos 225

C_8-Ribeira Brava 89

C_9-Ponta do Sol 59

C_10-Calheta 76

```
The calculation is in process...
City: Ribeira Brava | Code: C_8 | Flow: 89
City: Ponta do Sol | Code: C_9 | Flow: 59
City: Câmara de Lobos | Code: C_7 | Flow: 225
City: Funchal | Code: C_6 | Flow: 664
City: Machico | Code: C_4 | Flow: 137
City: Calheta | Code: C_10 | Flow: 76
City: Santa Cruz | Code: C_5 | Flow: 295
City: Santana | Code: C_3 | Flow: 46
City: Porto Moniz | Code: C_1 | Flow: 18
City: São Vicente | Code: C_2 | Flow: 34
```

C_6-Funchal

- Demand:  740
- Actual Flow: 664
- Deficit: 76

```
C_6-Funchal
Deficit: 76
```