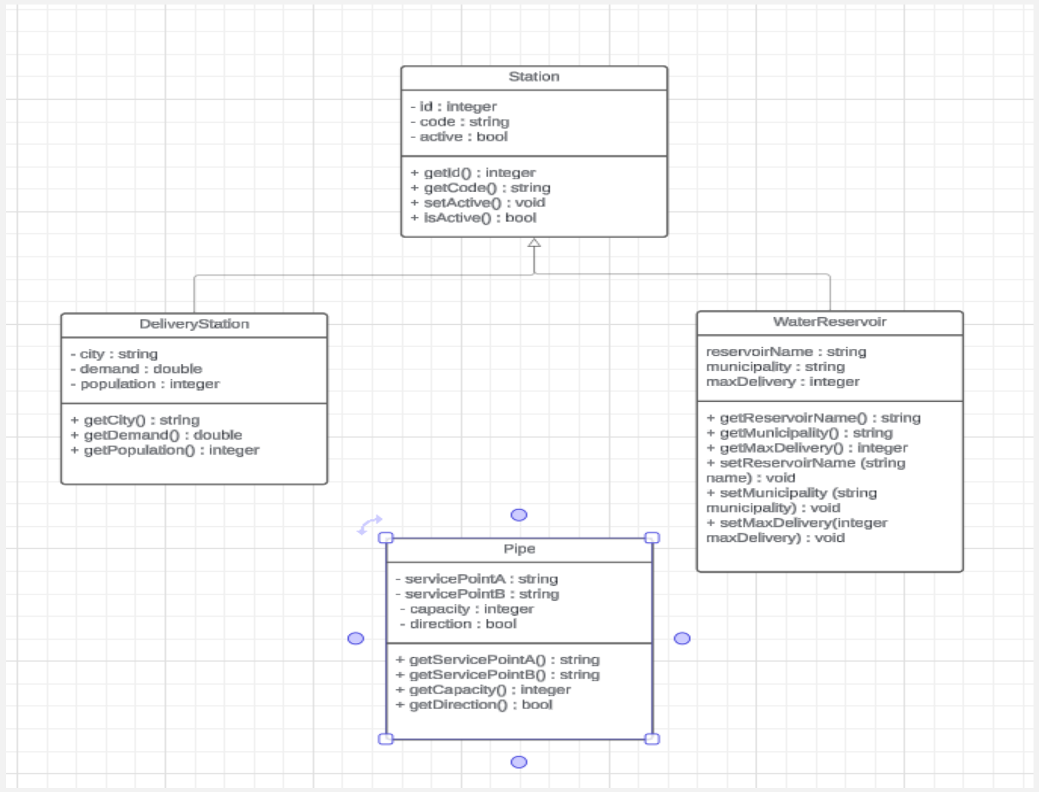
# ANALYSIS TOOL FOR WATER SUPPLY MANAGEMENT

Carlos Filipe Oliveira Sanches Pinto up202107694

David dos Santos Ferreira up202006302

João Maria Correia Rebelo up202107209

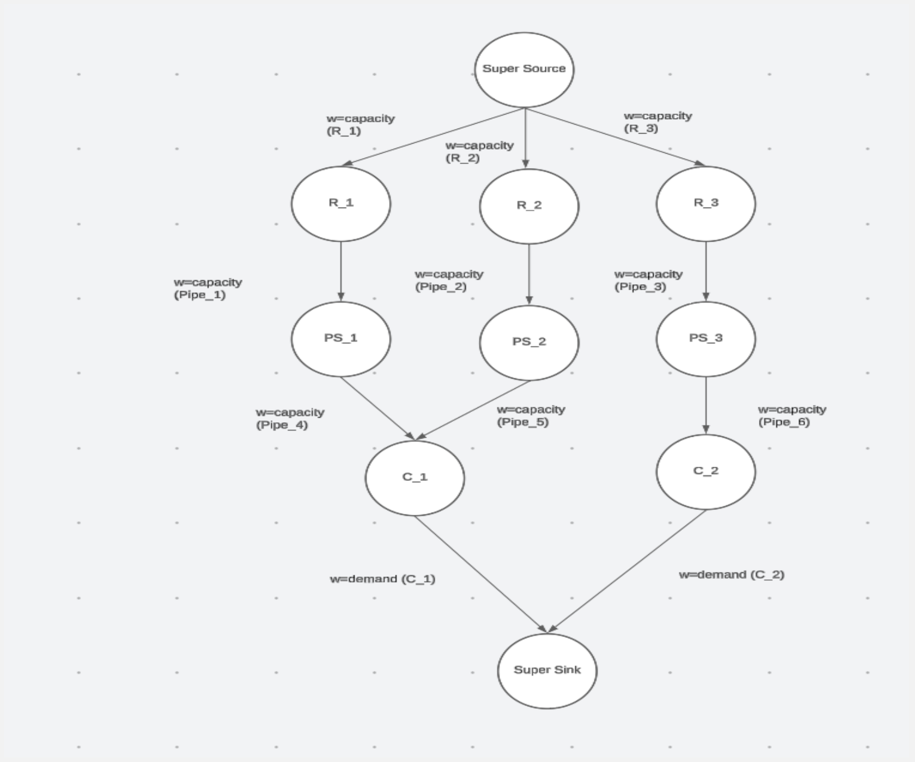
# CLASS DIAGRAM



## READING THE GIVEN DATASET

- The program consists of a set of functions implemented in C++ whose purpose is to read and analyse data from various CSV files in order to create specific objects and include them in a graph.
- The method called readAndParsePS() is responsible for extracting information from the Stations.csv file in order to create objects of the Station type. Similarly, the readAndParseDS() method is responsible for processing the data in the Cities.csv file to generate objects of the DeliveryStation type. Likewise, the readAndParseWR() method performs the same operation, but for objects of the WaterReservoir type, while the readAndParsePipes() method establishes the connections between the stations in the graph based on the information contained in the Pipes.csv file.
- Finally, the addSuperSourceAndSink() method is responsible for adding super source and super destination vertices to the graph, connecting them to the relevant vertices. In addition, auxiliary methods are provided for retrieving station and reservoir objects from different maps, using their codes, IDs or names as a reference.

# GRAPH


Super Source
w=capacity (R_1)
w=capacity (R_2)
w=capacity (R_3)
R_1
R_2
R_3
w=capacity (Pipe_1)
w=capacity (Pipe_2)
w=capacity (Pipe_3)
PS_1
PS_2
PS_3
w=capacity (Pipe_4)
w=capacity (Pipe_5)
w=capacity (Pipe_6)
C_1
C_2
w=demand (C_1)
w=demand (C_2)
Super Sink

## MAXIMUM AMOUNT OF WATER THAT CAN REACH EACH OR A SPECIFIC CITY

- Solving the problem of determining the maximum amount of water that can reach each city, or a specific city, involves introducing a "super source" vertex and a "super sink" vertex into the graph. The super sink is connected to all the cities via directed edges, where the weight of each edge corresponds to the demand of the corresponding city. On the other hand, the super source is connected to all the water reservoirs via directed edges, and the weight of each edge is determined by the reservoir's capacity.

- The Edmonds-Karp algorithm is then applied to find the maximum flow in the graph. Subsequently, a function called getFlowCity is used to return the maximum flow that passes through the city in question. This procedure is carried out by iterating over all the incoming edges of that city and calculating the sum of the flows. This approach makes it possible to accurately determine the maximum amount of water that can reach each city, taking into account reservoir capacities and city demands.

## MINIMIZE THE DIFFERENCES OF FLOW TO CAPACITY ON EACH PIPE ACROSS THE ENTIRE NETWORK

- The proposed function addresses the challenge of minimising disparities between flow and capacity in a pipeline network. Initially, it calculates key metrics to assess the current distribution, including total differences and squared differences between capacity and flow. These metrics inform the next steps, in which pipes with a flow rate lower than capacity are identified and categorised based on the disparity. The function then redistributes the flow, prioritising pipes with significant differences above the average, while taking the variance into account for precise adjustments. By seeking a more uniform distribution of flow, the approach increases the efficiency and resilience of the network in supplying water to the destination stations, effectively optimising the use of pipe capacity.

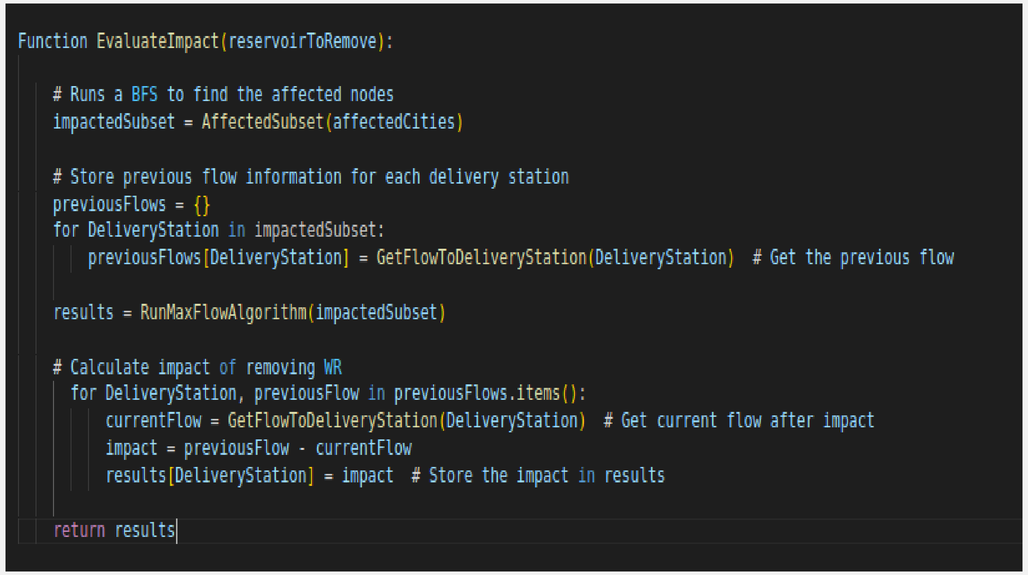
- This method stands out for its systematic approach to identifying and rectifying flow imbalances in the network. By using metrics such as mean and variance to understand the current distribution of flow and strategically adjusting the flow in pipes with significant disparities, the solution ensures a more equitable distribution of flow. Ultimately, this approach promotes the efficient use of resources, leading to a more robust and effective water distribution network.

## RELIABILITY AND SENSITIVITY TO FAILURES: RESERVOIR FAILURE

- The methodology adopted to determine the cities affected by the deactivation of a water reservoir follows a systematic procedure. Firstly, we remove the vertex corresponding to the reservoir from the graph by setting the weight of the edge connecting the main source (supersource) to the deactivated reservoir to zero. We then used the Edmonds-Karp algorithm to calculate the maximum flow in each city in the system. The discrepancy between the results obtained before and after the elimination of the vertex provides significant insights into the impact of the reservoir's deactivation on adjacent cities. This rigorous, iterative process allows for an accurate assessment of the implications of the reservoir's unavailability on the water supply of the surrounding areas.(The user has the possibility to remove 1 or more water reservoirs).

## RESERVOIR FAILURE : THEORETICAL EFFICIENT APPROACH

- Our proposal of an algorithm that would solve this efficiency issue, begins by conducting a breadth-first search (BFS) calling the findAffectedSubset function to identify the subset of the network impacted by the removal of a reservoir. This BFS starts from the affected cities and explores neighboring nodes, effectively determining the reach of the impact. Finally, the Max-Flow algorithm is applied to this reduced subset, and we check the impact it has on the delivery stations of this affected subset. This prevents inefficiency by running the max-flow algorithm on a smaller subset of the original graph.



```
Function EvaluateImpact(reservoirToRemove):

    # Runs a BFS to find the affected nodes
    impactedSubset = AffectedSubset(affectedCities)

    # Store previous flow information for each delivery station
    previousFlows = {}
    for DeliveryStation in impactedSubset:
        previousFlows[DeliveryStation] = GetFlowToDeliveryStation(DeliveryStation)  # Get the previous flow

    results = RunMaxFlowAlgorithm(impactedSubset)

    # Calculate impact of removing WR
      for DeliveryStation, previousFlow in previousFlows.items():
          currentFlow = GetFlowToDeliveryStation(DeliveryStation)  # Get current flow after impact
          impact = previousFlow - currentFlow
          results[DeliveryStation] = impact  # Store the impact in results

    return results
```

## RELIABILITY AND SENSITIVITY TO FAILURES: PUMPING STATIONS FAILURE

- The algorithm we have developed, examines the impact of temporarily removing pumping stations from a network by setting their incoming edges' capacity to zero. It identifies the affected cities and their water supply deficits, displaying relevant information. It aims to determine which pumping stations can be removed without causing deficits. By simulating the removal and recalculating flow using the edmonds-karp algorithm we check the impact of disabling a pumping station. Finally, it reports the number of stations that can be safely removed without water deficits.

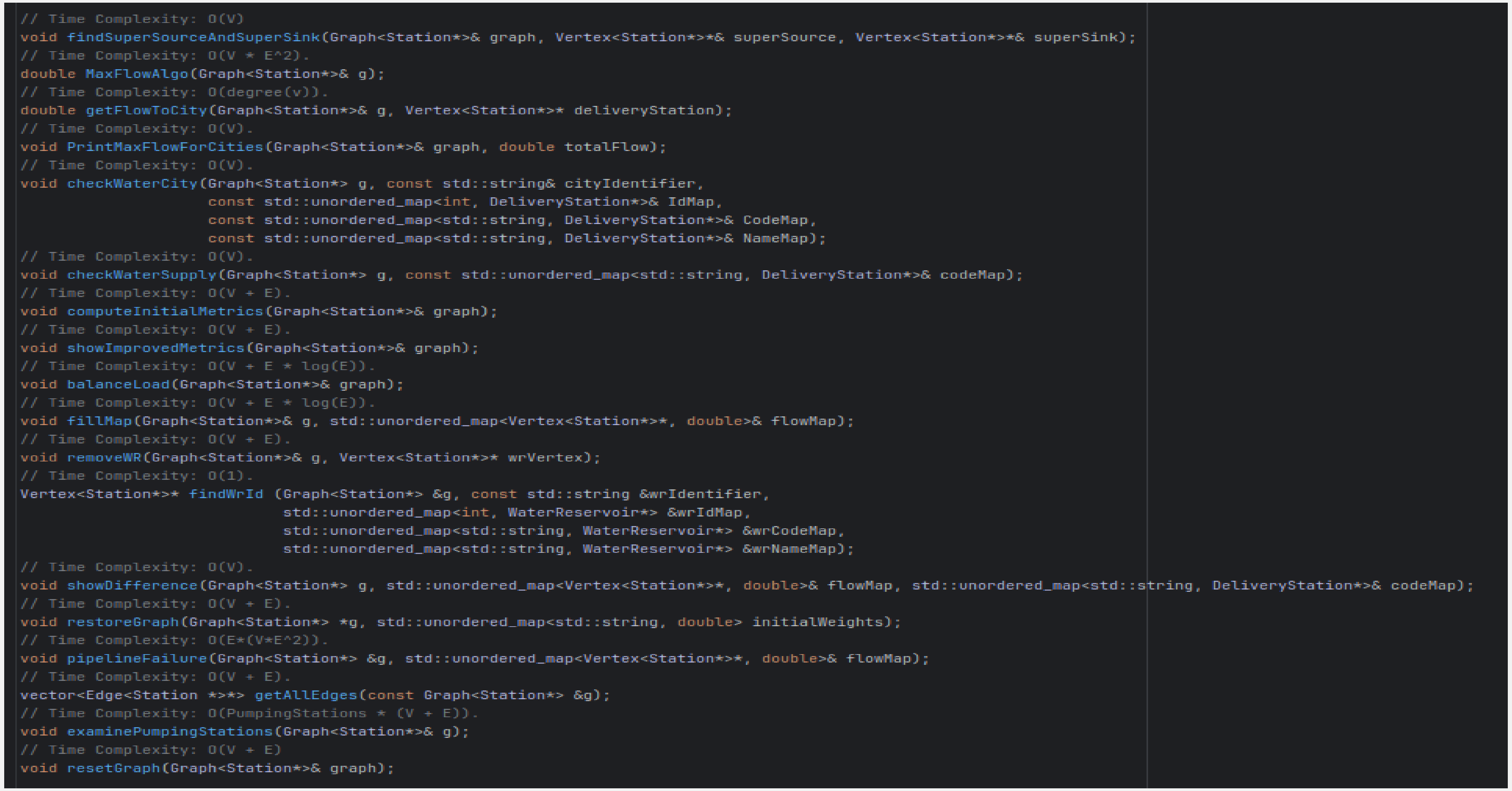
## RELIABILITY AND SENSITIVITY TO FAILURES: PIPELINE FAILURE

- The pipelineFailure function is designed to solve the problem of identifying the cities affected when a pipe in the water distribution system is out of service. The algorithm traverses all the edges of the graph representing the distribution network, temporarily deactivating each pipe individually (by setting its capacity to 0) and recalculating the maximum flow using the Edmonds-Karp algorithm.
- For each city connected to the network, it is checked whether the demand for water can be met after the pipe has been deactivated. If the demand cannot be fully met, the city is considered affected and the relevant information is displayed, including the water deficit. At the end of the process, the function restores the original graph configurations.
- This systematic method provides a comprehensive assessment of the implications of a pipe failure in the water distribution network, aiding decision-making to mitigate the negative impacts on the cities involved.

## USER INTERFACE

- The application's main menu plays the fundamental role of providing the user with a range of options for analysing and operating the water distribution network. After the initial loading process, where the data from the CSV files is read and interpreted to create specific objects and incorporate information into the graph, the main menu is presented. This menu offers the user a set of numbered options, each corresponding to different functionalities.
- Within the main menu, the user has the possibility of choosing between different functionalities, such as viewing basic service metrics, checking the network's reliability and sensitivity to faults, or closing the session. Each option takes the user to a corresponding sub-menu, where additional options are available for interaction. For example, by selecting the "Basic Service Metrics" option, the user is directed to a sub-menu where they can choose from a number of options, including calculating the maximum amount of water that can be delivered to each city, checking that the network configuration matches water needs, or balancing the load on the network to optimise flow. Each sub-menu provides a series of alternatives specific to the functionality selected, giving the user an interactive experience aimed at the efficient operation and management of the water distribution network.

# TIME COMPLEXITY



```
// Time Complexity: O(V)
void findSuperSourceAndSuperSink(Graph<Station*>& graph, Vertex<Station*>*& superSource, Vertex<Station*>*& superSink);
// Time Complexity: O(V * E^2).
double MaxFlowAlgo(Graph<Station*>& g);
// Time Complexity: O(degree(v)).
double getFlowToCity(Graph<Station*>& g, Vertex<Station*>* deliveryStation);
// Time Complexity: O(V).
void PrintMaxFlowForCities(Graph<Station*>& graph, double totalFlow);
// Time Complexity: O(V).
void checkWaterCity(Graph<Station*> g, const std::string& cityIdentifier,
                    const std::unordered_map<int, DeliveryStation*>& IdMap,
                    const std::unordered_map<std::string, DeliveryStation*>& CodeMap,
                    const std::unordered_map<std::string, DeliveryStation*>& NameMap);
// Time Complexity: O(V).
void checkWaterSupply(Graph<Station*> g, const std::unordered_map<std::string, DeliveryStation*>& codeMap);
// Time Complexity: O(V + E).
void computeInitialMetrics(Graph<Station*>& graph);
// Time Complexity: O(V + E).
void showImprovedMetrics(Graph<Station*>& graph);
// Time Complexity: O(V + E * log(E)).
void balanceLoad(Graph<Station*>& graph);
// Time Complexity: O(V + E * log(E)).
void fillMap(Graph<Station*>& g, std::unordered_map<Vertex<Station*>*, double>& flowMap);
// Time Complexity: O(V + E).
void removeWR(Graph<Station*>& g, Vertex<Station*>* wrVertex);
// Time Complexity: O(1).
Vertex<Station*>* findWrId (Graph<Station*> &g, const std::string &wrIdentifier,
                            std::unordered_map<int, WaterReservoir*> &wrIdMap,
                            std::unordered_map<std::string, WaterReservoir*> &wrCodeMap,
                            std::unordered_map<std::string, WaterReservoir*> &wrNameMap);
// Time Complexity: O(V).
void showDifference(Graph<Station*> g, std::unordered_map<Vertex<Station*>*, double>& flowMap, std::unordered_map<std::string, DeliveryStation*>& codeMap);
// Time Complexity: O(V + E).
void restoreGraph(Graph<Station*> *g, std::unordered_map<std::string, double> initialWeights);
// Time Complexity: O(E*(V*E^2)).
void pipelineFailure(Graph<Station*> &g, std::unordered_map<Vertex<Station*>*, double>& flowMap);
// Time Complexity: O(V + E).
vector<Edge<Station *>*> getAllEdges(const Graph<Station*> &g);
// Time Complexity: O(PumpingStations * (V + E)).
void examinePumpingStations(Graph<Station*>& g);
// Time Complexity: O(V + E)
void resetGraph(Graph<Station*>& graph);
```

## FUNCTIONALITY TO HIGHLIGHT

- Our system for evaluating network resiliency features a unique capability allowing users to remove consecutive water reservoirs to simulate cascading failures. Users can reset the graph after each removal, facilitating iterative analysis of the network's response to multiple failures. This feature offers a streamlined approach for exploring various failure scenarios and understanding the impact on water supply to affected cities. There is also the possibility of just removing one reservoir then resetting the graph and running a removal again to access the impact of removing a single water reservoir.

## MAIN DIFICULTES

- Our group faced significant difficulties when selecting the most optimal solution to challenge 4.1.3, which consisted of developing a simple algorithm or heuristic to balance the load across the network, minimising the flow differences in relation to the capacity in each pipe as much as possible. Initially, we considered approaching the problem by calculating the average flow on the adjacent edges of each vertex and redistributing this average between them.
- However, when we implemented this approach, we noticed that the variance of the results increased even more, indicating a less uniform distribution of the flow. Faced with this challenge, we opted to explore other strategies, taking into account the complexity of the network and the need to minimise flow disparities in an efficient and robust way.