



# An Analysis Tool for Water Supply Management

up20206409 - Filipe Cardoso

up201506074 - Miguel Charchalis

up201900839 - Artur Santos



# Highlighted Functionalities

## 2.3 Algorithm:

The algorithm developed in 2.3 is the one we are the most proud of, since we struggled a lot to accomplish it. Therefore, we developed a “guided local search” like approach using a neighboring function where we found the edges which had the highest discrepancy in regard to their neighbors. So, by diminishing the capacity to be equal to the second highest capacity neighbour we could yield very good results. By going through the neighbourhood we were able to identify some problematic vertexes. Although, since it was like a “guided local search” algorithm some neighbours could lead to worse results, so we had to save the best yielded result and on which iteration it was found in order to always get the best result out of our algorithm.



# Main Difficulties and Contributions:

## Main Difficulties:

One of the main difficulties faced was during the problem 2.3 where we were not sure how we could keep the problem simple and achieve great results. Since the problem had no direct relation with what we have seen in the classes we felt like we were left on a cliffhanger. Still, after some brainstorming we were able to achieve great results.

## Contributions:

- Filipe Cardoso: 50% (2.3, 3.1, 3.2, 3.3)
- Miguel Charchalis: 40% (1.1, 1.2, 2.1, 2.2)
- Artur Santos: 15% (1.3)



# Description of the User Interface (UI)

The user interface is very simple:

- From points 1-5 you are supposed to select which exercise to execute
  - (1) - 2.1 AND 2.2
  - (2) - 2.3
  - (3) - 3.1
  - (4) - 3.2
  - (5) - 3.3
- Then you are asked to select which dataset
  - “big” for Portugal’s Flow Network Dataset
  - “small “ for Madeira’s Flow Network Dataset




# Description of reading the dataset

The dataset is read using the function `populate_graph(Graph<T> &g, string dataset)`. It parses the information of each dataset file to a vector using the function `parseCSV(const &string filename)`.

Then the objects City, Station, Reservoir (inherit from Vertex) and Edge (pipes) are created based on the dataset and inserted into the graph.

At the end of the `populate_graph` function there is a loop whose purpose is to set all edges' flow to zero.




# Description of the graph used to represent the dataset

The code structure we used for the graph was heavily influenced by the code structure we used in the practical exercises.

We created the classes City, Station and Reservoir that inherit from the class Vertex, since they are all nodes but each have different kinds of variables. And the class Edge represents the pipes.

```
class City: public Vertex<string> {
private:
    string city;
    int id;
    string code;
    float demand;
    int population;
public:
    City(string city, int id, string code, float demand, int population)
        : Vertex(code), city(city), id(id), code(code), demand(demand)
    {
        this->setDemand(demand);
    }
}
```



# Description of the graph used to represent the dataset

```
class Station: public Vertex<string> {
private:
    int id;
    string code;
public:
    Station(int id, string code)
        : Vertex(code), id(id), code(code) {}
};

class Reservoir: public Vertex<string> {
private:
    string reservoir;
    string city;
    int id;
    string code;
    int maxDelivery;
public:
    Reservoir(string reservoir, string city, int id, string code, int maxDelivery)
        : Vertex(code), reservoir(reservoir), city(city), id(id), code(code), maxDelivery(maxDelivery){}
};
```



# Description of implemented functionalities and associated algorithms

## Edmonds Karp:

We use the `Graph::bfs(source, map discoveryMap)` to find an augmenting path.

DiscoveryMap is a map that stores the destination nodes of the path and the corresponding incoming edge. So the path can be traced back by getting the node of origin of the incoming edge of the sink node, and then get the incoming edge of that node, etc. The process is repeated until the source node is found on the map.

After finding an augmenting path, the max flow is calculated and attributed to the edges of the augmenting path.

Once there are no more augmenting paths (there is no `discoveryMap['source']`), the cycle breaks and the algorithm is finished.





# Description of implemented functionalities and associated algorithms

T2\_1:

The function T2\_1 basically performs the edmonds karp algorithm using the master-source node that is connected to every reservoir node with edges with “unlimited” capacity, and using the master-sink node that is connected to every city node, again, with edges with “unlimited” capacity.

list\_affected\_cities:

At the end of the T2\_1 function, the list\_affected\_cities function is run. This function sums all the flow from the incoming edges of a city and checks if its value meets the cities' demands.



# Description of implemented functionalities and associated algorithms

T2\_3:

This algorithm uses a heuristic-based approach to optimize flow in a graph. It identifies critical edges based on differences between capacity and flow, and adjusts the capacities of these edges to improve flow balance.

Complexity:  $O(V^2 * E * n)$ , where  $V$  is the number of vertices,  $E$  is the number of edges, and  $n$  is the number of iterations required until all critical edges are identified and optimizations applied. This is because the function performs Edmonds-Karp and computes metrics in each iteration, as well as various operations on graphs and vectors.



# Description of implemented functionalities and associated algorithms

T3\_1:

The function **T3\_1** aims to simulate the removal of a water reservoir from the network represented by the graph 'g'. It first checks if the **waterReservoir** parameter is provided.

If not, it returns false. Then, it attempts to remove the specified reservoir from the graph. If successful, it sets up the graph for analysis using the Edmonds-Karp algorithm, simulates the network's flow after the reservoir removal, prints the resulting graph, and lists affected cities.

Finally, it returns true to indicate successful execution. If the reservoir is not found, it returns false.

Time Complexity :  $O(V + E)$



# Description of implemented functionalities and associated algorithms

T3\_2:

The function T3\_2 simulates the removal of a pumping station from the water supply network represented by the graph 'g'. It initializes the graph, runs the Edmonds-Karp algorithm to simulate the network's flow after the pumping stations removal, and lists the affected cities.

It compares the deficits before and after the removal, identifying any worsening conditions due to the removal. If a worsening deficit is found, it prints the city name along with the new deficit.

Finally, it returns a boolean indicating whether any worsening deficits were found.

Time Complexity :  $O(V + E)$



# Description of implemented functionalities and associated algorithms

T3\_3:

The function T3\_3 simulates the failure of each pipeline in the water supply network represented by the graph 'g'. For each pipeline, it temporarily reduces its capacity to zero to simulate a rupture or failure.

Then, it runs the Edmonds-Karp algorithm to determine how this failure affects the network's ability to meet the water demand of cities. It identifies cities that are affected by each pipeline failure and calculates the deficit in water supply for each affected city.

Finally, it prints the results sorted by both pipeline and city, displaying the pipelines affecting each city and the corresponding deficits, as well as the cities affected by each pipeline.

Time Complexity:  $O(V^3 * E^2)$