

# DA PROJETO 1

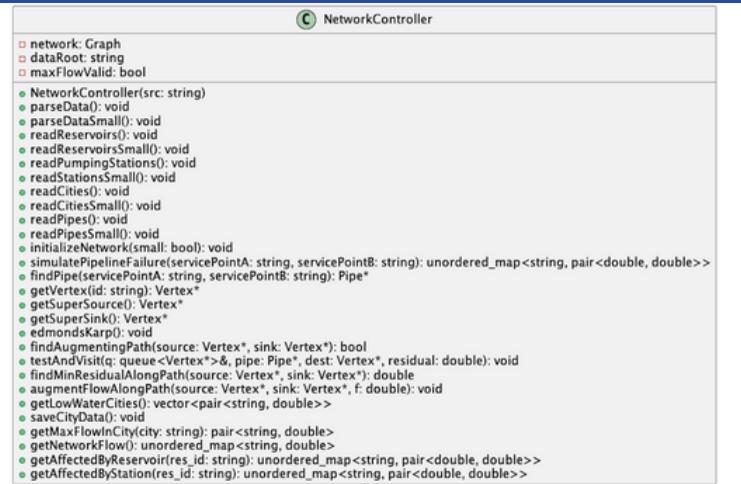
UMA FERRAMENTA DE ANÁLISE PARA GESTÃO DE  
ABASTECIMENTO DE ÁGUA

- Rodrigo Ferreira Alves - up202207478  
- Gonçalo de Abreu Matias - up202108703  
- David Tavares Simões - up202210329

Class: 2LEIC06  
Group: G06\_6

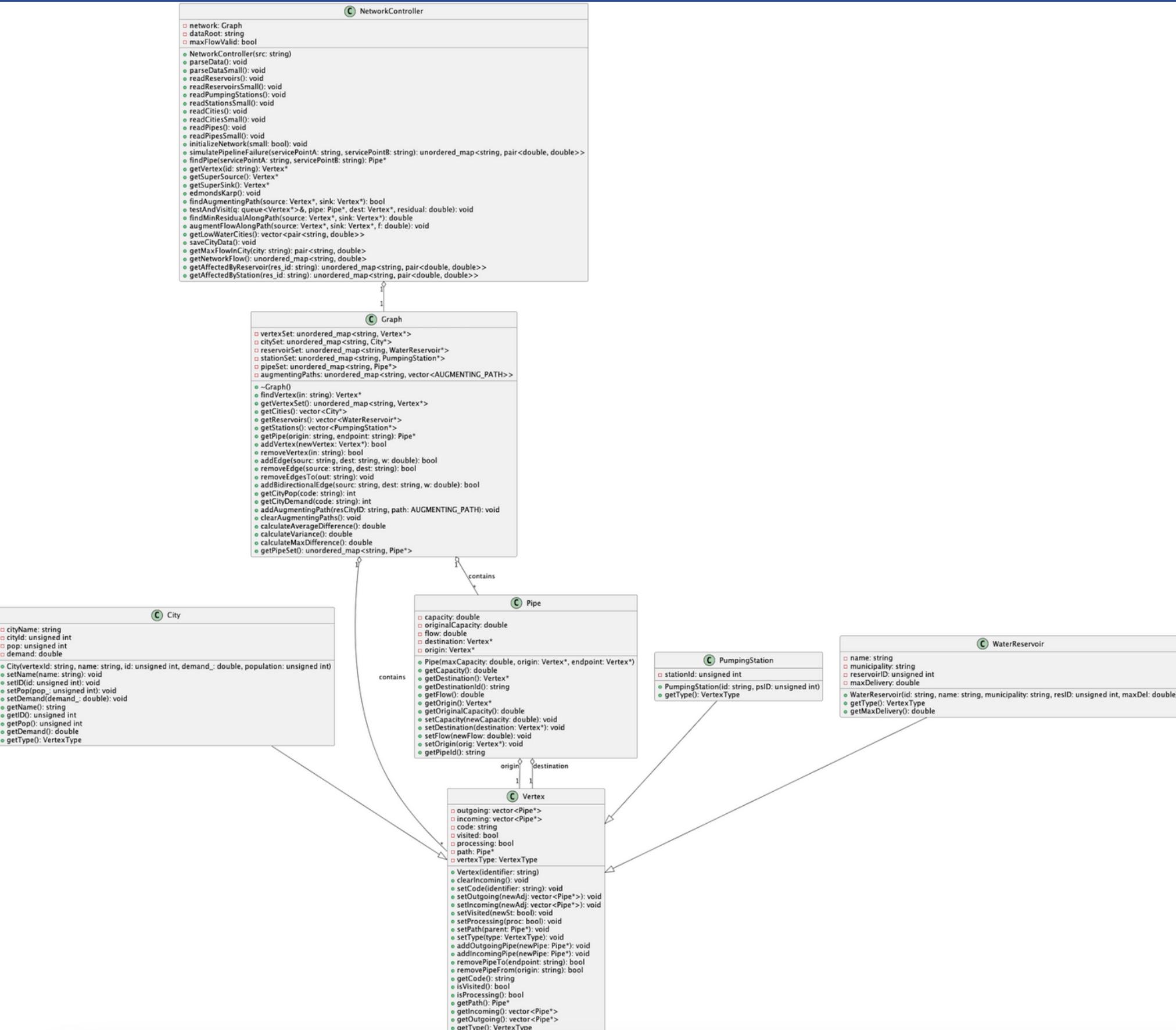
# DIAGRAMA DE CLASSE

**Classe NetworkController ->** Orquestra a interação entre as várias classes e executa algoritmos de análise de rede, como o cálculo do fluxo máximo, simulações de falhas e otimizações de distribuição de água.



**Classe Menu ->** Interface dos menus responsável pela interação com o usuário, apresentando opções e recolhendo comandos para a execução das funcionalidades do sistema.

**Classe Graph ->** Representa a estrutura de grafos onde todos os nós e arestas são gerenciados, permitindo a realização de operações como busca de caminhos, cálculo de fluxo máximo e outras análises de rede.



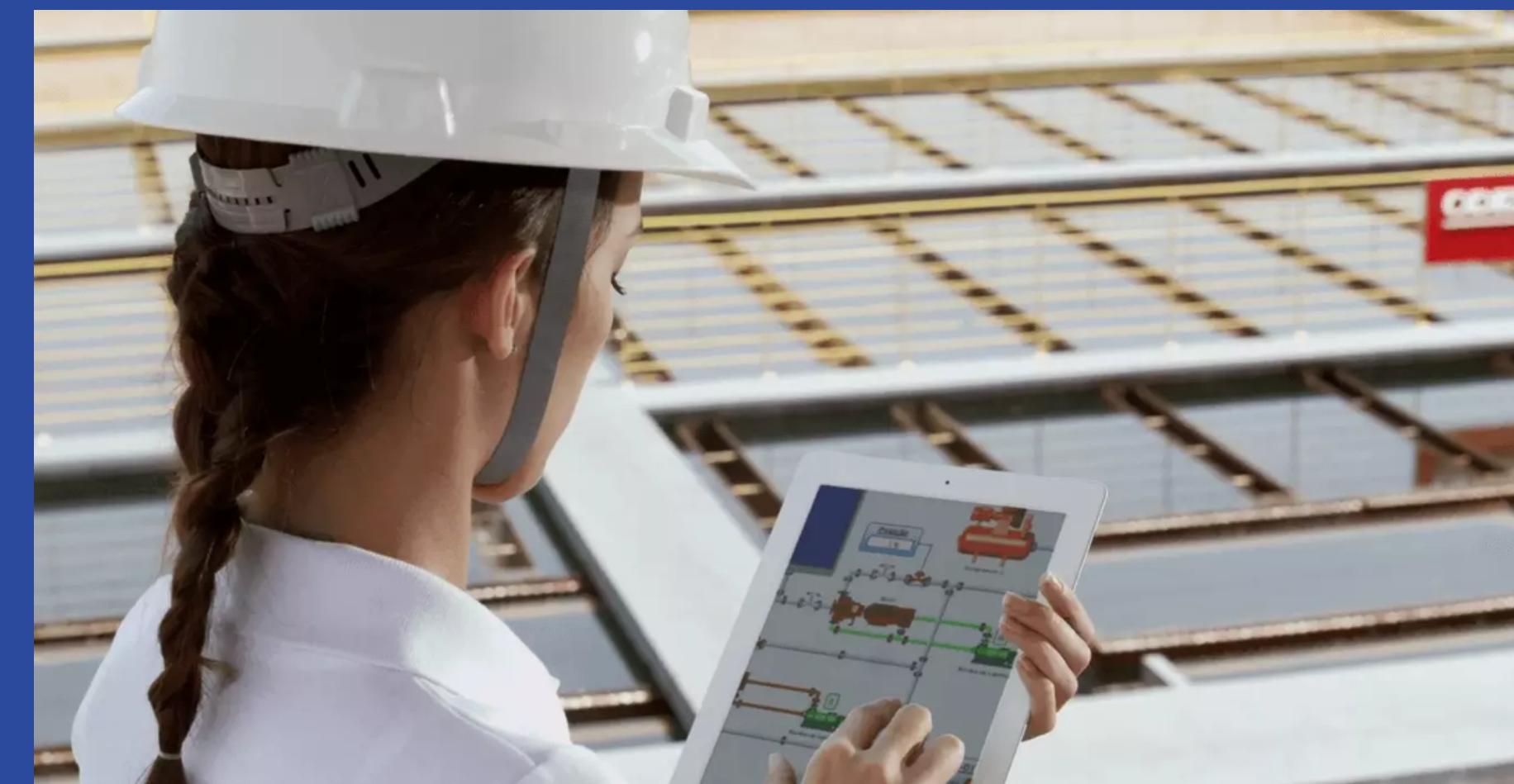
**Classe City ->** Armazena e gerencia dados referentes aos municípios, incluindo nome, identificador, população e procura de água.

**Classe WaterReservoir ->** Mantém as informações dos reservatórios de água, como nome, localização, capacidade máxima de entrega, entre outras características relevantes.

**Classe PumpingStation ->** Controla as propriedades das estações de bombeamento, fundamentais para o direcionamento do fluxo de água na rede.

**Classe Classe Pipe ->** Encapsula as informações dos canos ou tubulações que conectam diferentes elementos da rede, regulando a capacidade e o fluxo atual.

# LEITURA DO DATASET



A leitura e análise dos ficheiros .csv são realizadas nas classes “NetworkController.cpp” e “Graph.cpp”.

Nestas classes, foram implementadas funções específicas para processar os dados do sistema de gestão de recursos hídricos, tais como `readCities()`, `readReservoirs()`, `readPumpingStations()` e `readPipes()`, que analisam os ficheiros "Cities.csv", "Reservoirs.csv", "Stations.csv" e "Pipes.csv", respetivamente.

As funções armazenam a informação recolhida em estruturas específicas dentro do Graph e do NetworkController:

- `readCities()` armazena os dados dos municípios, incluindo procura e população, na estrutura de vértices representando as cidades. Também adiciona esses municípios como nós no Graph.
- `readReservoirs()` gerencia os dados dos reservatórios de água e adiciona-os como nós relevantes no Graph.
- `readPumpingStations()` guarda as informações das estações de bombeamento na sua respetiva estrutura e insere-as no Graph.
- `readPipes()` analisa as conexões de tubulações e canos, adicionando-as como arestas no Graph para representar as ligações entre os elementos da rede de abastecimento de água.

Estas funções são essenciais para a inicialização do sistema, sendo chamadas no ponto de entrada do programa ("main.cpp") para garantir que todos os dados sejam carregados e estruturados corretamente no Graph e no NetworkController antes de iniciar a análise e gerenciamento do fornecimento de água.

Portanto, as classes “NetworkController.cpp” e “Graph.cpp” são fundamentais no processo de carregamento e estruturação dos dados, fazendo com que o sistema de gerenciamento de recursos hídricos opere de forma eficaz desde o momento em que o programa é executado.

# GRAPH

No nosso projeto, optámos por desenvolver um grafo personalizado para visualizar e gerenciar a rede de distribuição de água. Este grafo é composto por diversas classes fundamentais:

- **Vertex**: Representa os elementos principais da rede, tais como cidades, reservatórios e estações de bombeamento.
- **City**: Subclasse de Vértice que especifica as localidades dentro do nosso sistema.
- **WaterReservoir**: Subclasse de Vértice que denota os pontos de armazenamento da nossa rede.
- **PumpingStation**: Uma subclasse de Vértice designada para as instalações responsáveis pelo bombeamento de água.
- **Pipe**: Representa as conexões físicas entre os vértices, representando as tubulações através das quais a água flui. Cada pipe tem uma capacidade máxima de fluxo, determinando a quantidade máxima de água que pode ser transportada.
- **Graph**: Esta é a estrutura central que mantém todos os componentes interligados. O graph é responsável por facilitar operações essenciais, como cálculos de fluxo máximo e análises de eficiência do sistema. Ele fornece a infraestrutura necessária para compreender e otimizar a distribuição de água em nossa rede.

```
class Vertex {
protected:
    std::vector<Pipe*> outgoing; // Outgoing pipes
    std::vector<Pipe*> incoming; // Incoming pipes
    std::string code; // Identifier of the vertex
public:
    Vertex(const std::string& identifier);
    void addOutgoingPipe(Pipe* newPipe);
    void addIncomingPipe(Pipe* newPipe);
    bool removePipeTo(const std::string& endpoint);
    virtual VertexType getType() const;
    // ... other members and methods ...
};
```

# ESTRUTURA

```
class Graph {
protected:
    std::unordered_map<std::string, Vertex*> vertexSet; // Set of vertices
    std::unordered_map<std::string, Pipe*> pipeSet; // Set of pipes

public:
    ~Graph();
    Vertex *findVertex(const std::string& in) const;
    bool addVertex(Vertex* newVertex);
    bool removeVertex(const std::string &in);
    bool addEdge(const std::string &sourc, const std::string &dest, double w);
    // ... other members and methods ...
};
```

```
class Pipe{
private:
    double capacity; // Maximum capacity of the pipe
    double flow; // Current flow through the pipe
    Vertex* destination; // Destination vertex of the pipe
    Vertex* origin; // Origin vertex of the pipe
public:
    Pipe(double maxCapacity, Vertex* origin, Vertex* endpoint);

    double getCapacity() const;
    Vertex* getDestination() const;
    const std::string& getDestinationId() const;
    double getFlow() const;
    Vertex* getOrigin();
    void setCapacity(double newCapacity);
    void setDestination(Vertex* destination);
    void setFlow(double newFlow);
    void setOrigin(Vertex* orig);
    std::string getPipeId() const;
};
```

# FUNCIONALIDADES IMPLEMENTADAS

**void initializeNetwork(bool small)**

*Inicia a rede com dados reduzidos ou completos, executa o algoritmo Edmonds-Karp para calcular o fluxo máximo, e guarda os dados das cidades.*

*Complexidade:  $O(V + E)$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas.*

**void parseData()**

*Processa dados detalhados de cidades, reservatórios, estações de bombeamento e tubagens a partir de ficheiros CSV para construir o grafo da rede.*

*Complexidade:  $O(V+E)$ , onde  $V$  é o número de vértices e  $E$  o número de arestas.*

**void readCities()**

*Extrai e adiciona informações das cidades de um ficheiro CSV ao grafo representando-as como vértices.*

*Complexidade:  $O(N)$ , sendo  $N$  o número de cidades.*

**void readPumpingStations()**

*Lê informações das estações de bombeamento de um ficheiro CSV e adiciona-as ao grafo como vértices.*

*Complexidade:  $O(M)$ , onde  $M$  é o número de estações de bombeamento.*

# CONTINUAÇÃO...

*Carrega os dados dos reservatórios de um ficheiro CSV e insere-os no grafo como vértices.*

**void readReservoirs()**

*Complexidade:  $O(R)$ , sendo  $R$  o número de reservatórios.*

*Importa as tubagens do ficheiro CSV e adiciona-as ao grafo, estabelecendo as conexões entre pontos de serviço.*

**void readPipes()**

*Complexidade:  $O(P)$ , onde  $P$  é o número de tubagens.*

**void balanceNetwork()**

*Equilibra a rede ajustando as capacidades das tubagens com base no fluxo e capacidades existentes, buscando otimização do fluxo geral.*

*Complexidade:  $O(V^*E^2)$ , assumindo ajuste de capacidades e repetição do Edmonds-Karp para avaliar o impacto.*

```
std::unordered_map<std::string, std::pair<double, double>>
NetworkController::simulatePipelineFailure(const
std::string& servicePointA, const std::string& servicePointB)
```

*Simula uma falha de tubagem entre dois pontos de serviço e analisa o impacto no fluxo de água na rede, comparando o fluxo antes e após a falha.*

*Complexidade:  $O(F + E)$ , onde  $F$  é o número de operações para encontrar a tubagem e  $E$  o número de operações do algoritmo Edmonds-Karp.*

```
std::unordered_map<std::string, std::pair<double, double>>
NetworkController::getAffectedByStation(const std::string
&res_id)
```

*Identifica as cidades afetadas pela desativação temporária de uma estação de bombeamento e o impacto no fluxo de água para essas cidades, comparando o fluxo antes e após a desativação.*

*Complexidade:  $O(F + E)$ , onde  $F$  é o número de operações para ajustar a estação e  $E$  o número de operações do algoritmo Edmonds-Karp.*

# CONTINUAÇÃO...

**Vertex\* getVertex(const std::string& id);**

*Fornece um vértice do gráfico pelo seu identificador.*

*complexity O(1)*

**void edmondsKarp()**

*Implementa o algoritmo Edmonds-Karp para encontrar o fluxo máximo na rede.*

*Complexidade:  $O(V * E^2)$ , onde V é o número de vértices e E é o número de arestas no gráfico.*

**bool findAugmentingPath(Vertex\* source,  
Vertex\* sink);**

*Encontra um augmenting path desde a source até ao sink usando um BFS.*

*Complexidade:  $O(E)$ , onde E é o número de arestas no graph, já que o BFS percorre cada aresta no máximo uma vez.*

**void testAndVisit(std::queue<Vertex\*>&, Pipe\* pipe,  
Vertex\* dest, double residual);**

*Testa e visita vértices ao longo de um caminho para determinar o fluxo residual.*

*Complexidade:  $O(P)$ , onde P é o comprimento do path.*

# CONTINUAÇÃO...

*Encontra o resíduo mínimo ao longo de um caminho entre uma source e uma sink.*

```
double findMinResidualAlongPath(Vertex  
*source, Vertex *sink);
```

*Complexidade:  $O(P)$ , onde  $P$  é o comprimento do path.*

*Aumenta o fluxo ao longo do augmenting path encontrado.*

```
void augmentFlowAlongPath(Vertex* source,  
Vertex* sink, double f);
```

*Complexidade:  $O(P)$ , onde  $P$  é o número de arestas do augmenting path*

*Procura pelas cidades com baixos níveis de água.*

```
std::vector<std::pair<std::string, double>>  
getLowWaterCities();
```

*Complexidade:  $O(C)$ , onde  $C$  é o número de cidades conectadas ao super sink*

*Salva os dados da cidade nos arquivos correspondentes.*

```
void saveCityData();
```

*Complexidade:  $O(C)$ , onde  $C$  é o número de cidades.*

*Calcula o fluxo máximo numa cidade.*

```
std::pair<std::string, double>  
getMaxFlowInCity(const std::string& city);
```

*Complexidade:  $O(V^*E^2)$  já que o algoritmo edmondsKarp() será corrido*

# CONTINUAÇÃO...

```
std::unordered_map<std::string, double>
    getNetworkFlow();
```

*Recupera o fluxo da rede.*

*Complexidade: O(C), onde C é o número de cidades.*

```
std::unordered_map<std::string, std::pair<double,
    double>> getAffectedByReservoir( const
        std::string& res_id);
```

*Recupera vértices afetados por um dado reservatório.*

*Complexidade: O(V\*E^2), devido à necessidade de reexecutar o algoritmo de Edmonds-Karp após modificar a rede.*

```
Vertex* getSuperSource();
```

*Cria uma super source que conecta todas as sources a isso com capacidade INF*

*Complexidade: O(R), onde R é o número de reservatórios, devido à criação de arestas da super source até cada reservatório.*

```
Vertex* getSuperSink();
```

*Cria uma super sink que conecta todas as sinks a ele com capacidade de borda INF*

*Complexidade: O(C), onde C é o número de cidades, devido à criação de arestas de cada cidade até o super sink.*

# BALANCEAMENTO E OTIMIZAÇÃO DA NETWORK (T2.3)

```
void NetworkController::addAugmentingPathToGraph(Graph *graph, std::vector<Pipe *> path) {
    Vertex* superSink = path.front()->getDestination();
    graph->addVertex(new Vertex(superSink->getCode()));

    for(Pipe* pipe: path){
        // Add vertex endpoint to partialGraph
        if(pipe->getOrigin()->getType() == CityVertex){
            City* curr_vertex = dynamic_cast<City*>(pipe->getOrigin());
            City* vertex = new City(curr_vertex);
            graph->addVertex(vertex);
        }
        else{
            Vertex* vertex = new Vertex(pipe->getOrigin()->getCode());
            graph->addVertex(vertex);
        }

        // Add connection between endpoints
        graph->addEdge(pipe->getOrigin()->getCode(), pipe->getDestinationId(), pipe->getCapacity());
    }
}
```

```
std::unordered_map<std::string, std::pair<double, double>> NetworkController::removeReservoirPartial(const std::string &res_id) {
    Graph partialGraph;
    std::unordered_set<std::string> citiesAffected;

    // CREATING PARTIALGRAPH

    std::vector<AUGMENTING_PATH> augPaths = this->network.getAugmentingPaths()[res_id];

    // Add direct vertices of pair RESERVOIR-CITY
    for(AUGMENTING_PATH path: augPaths){
        citiesAffected.insert(path.front()->getOrigin()->getCode());
        this->addAugmentingPathToGraph(&partialGraph, path);
    }

    // Add reservoirs that serve the same city as res_id
    for(std::pair<std::string, std::vector<AUGMENTING_PATH>> path: this->network.getAugmentingPaths()){
        // Path from reservoir x
        augPaths = path.second;
        for(AUGMENTING_PATH path: augPaths){
            std::string cityInAugmentingPath = path.front()->getOrigin()->getCode();
            if(citiesAffected.find(cityInAugmentingPath) != citiesAffected.end()){
                // City in current augmenting path is also affected by reservoir res_id removal
                this->addAugmentingPathToGraph(&partialGraph, path);
            }
        }
        // Prevent reservoir from being used
        for(Pipe* out: partialGraph.findVertex("SuperSource")->getOutgoing()){
            if(out->getDestination()->getCode() == res_id) out->setCapacity(0);
        }
    }

    std::cout << partialGraph.getVertexSet().size() << " - " << this->network.getVertexSet().size() << std::endl;

    // For each pumping station i should reduce its capacity by the flow that is not present in the partial graph
    // -----
    std::unordered_map<std::string, double> flowBeforeRemoval = this->getNetworkFlow();
    Graph backupOriginal = this->network;

    this->network = partialGraph;
    this->edmondsKarp();

    std::unordered_map<std::string, double> flowAfterRemoval = this->getCitiesFlow();

    std::unordered_map<std::string, std::pair<double, double>> result;

    for(std::pair<std::string, double> afterPair: flowAfterRemoval){
        // If it has its demand met
        City* cityVertex = dynamic_cast<City*>(this->network.findVertex(afterPair.first));
        if(cityVertex->getDemand() <= afterPair.second) continue;

        // If the flow didn't change i.g. not directly dependent -> skip
        if(afterPair.second == flowBeforeRemoval[afterPair.first]) continue;

        // It does not have its demand met and the flow changed!!!
        result[afterPair.first] = {flowBeforeRemoval[afterPair.first], flowAfterRemoval[afterPair.first]};
    }

    this->maxFlowValid = false;
    this->network = backupOriginal;
    return result;
}
```

# INTERFACE COM O UTILIZADOR

A interface com o utilizador é na sua maioria feita através de menus, tendo sido implementados 6 menus diferentes de modo a que a experiência do utilizador seja simples e direta.

Menu Principal

```
Choose a dataset:  
1. Madeira (small)  
2. Continental Portugal (big)  
Enter your choice (1 or 2):
```

Menu das Estatísticas:

```
==== Water Supply Management Analysis System Menu ====  
1. Calculate Maximum Flow of the network  
2. Calculate Maximum Flow to a City  
3. List Cities with Water Deficits  
4. Analyze Impact of Reservoir Removal  
5. Analyze Impact of Pumping Station Removal  
6. Simulate Pipeline Failure  
7. Exit  
Select an option:
```

# INTERFACE COM O UTILIZADOR

Consoante a escolha, temos pedidos de input diferentes:

```
== Water Supply Management Analysis System Menu ==
1. Calculate Maximum Flow of the network
2. Calculate Maximum Flow to a City
3. List Cities with Water Deficits
4. Analyze Impact of Reservoir Removal
5. Analyze Impact of Pumping Station Removal
6. Simulate Pipeline Failure
7. Exit
Select an option: 4
```

```
Select an option: 4
Enter the reservoir code to evaluate its impact: R_1
City          Old Flow      New Flow
C_20           50            50
C_17          5650          4650
Press Enter to continue...
```

```
== Water Supply Management Analysis System Menu ==
1. Calculate Maximum Flow of the network
2. Calculate Maximum Flow to a City
3. List Cities with Water Deficits
4. Analyze Impact of Reservoir Removal
5. Analyze Impact of Pumping Station Removal
6. Simulate Pipeline Failure
7. Exit
Select an option: 3
```

Select an option: 3	C_6 - Castelo Branco ● Demand: 230 ● Actual Flow: 229 ● Deficit: 0	C_13 - Lagos ● Demand: 158 ● Actual Flow: 121 ● Deficit: 36
Code, Demand, Actual Flow, Deficit (m³)	C_20 - Viana do Castelo ● Demand: 168 ● Actual Flow: 50 ● Deficit: 117	C_10 - Évora ● Demand: 313 ● Actual Flow: 166 ● Deficit: 146
C_20 - Viana do Castelo	C_22 - Viseu ● Demand: 397 ● Actual Flow: 277 ● Deficit: 119	C_5 - Bragança ● Demand: 152 ● Actual Flow: 107 ● Deficit: 44
● Demand: 168	C_17 - Porto ● Demand: 6324 ● Actual Flow: 5650 ● Deficit: 674	C_7 - Coimbra ● Demand: 896 ● Actual Flow: 672 ● Deficit: 224
● Actual Flow: 50	C_8 - Covilhã ● Demand: 122 ● Actual Flow: 11 ● Deficit: 110	● Deficit: 0
● Deficit: 117		

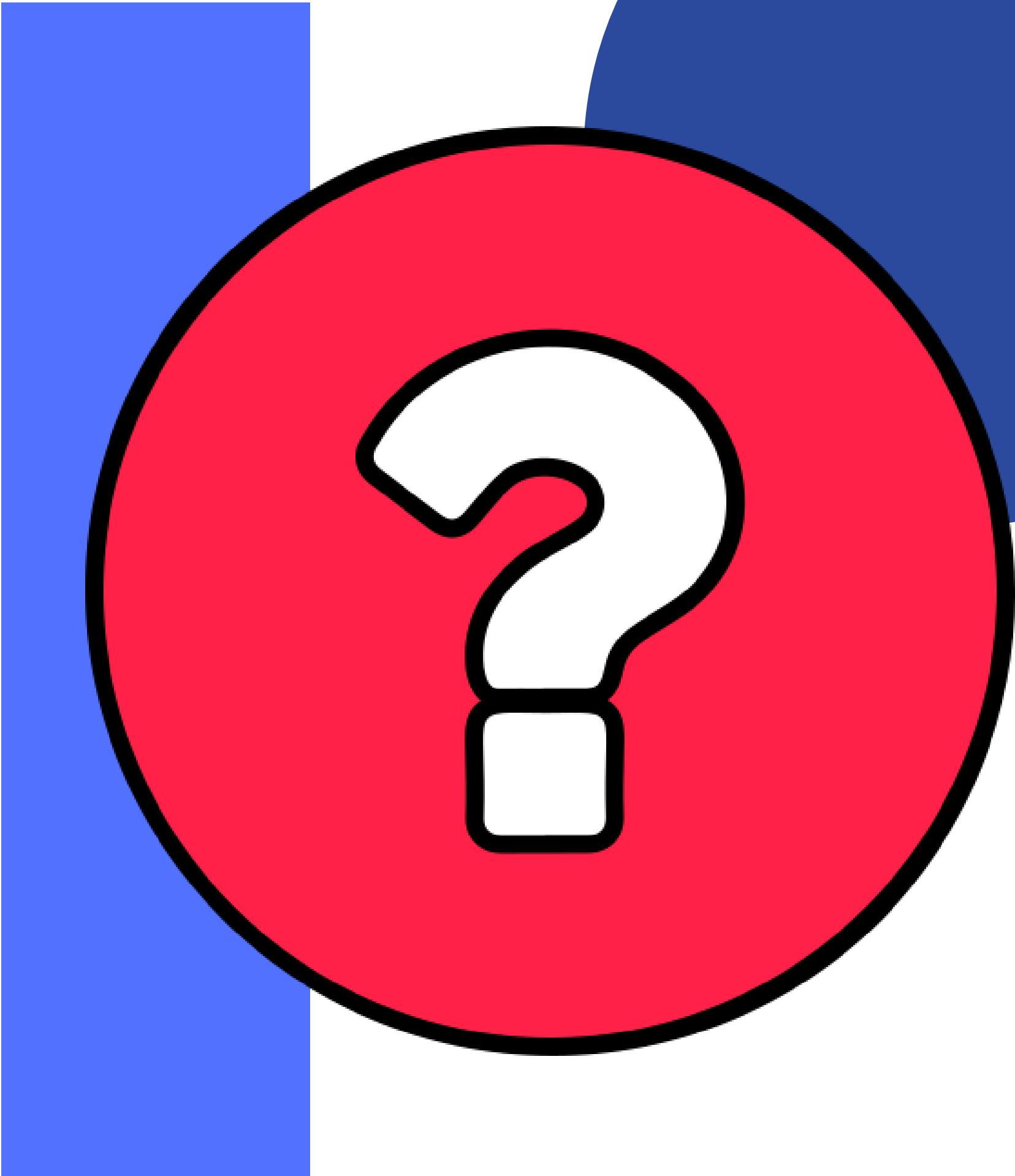
# DESTAQUE DE FUNCIONALIDADES

- Desenvolvimento de um Menu organizado e de fácil compreensão pelo usuário;
- Criação de algoritmos eficientes que fornecem ao usuário informação desejada sobre a rede;
- Estruturação organizada e detalhada do programa;



# DIFICULDADES ENCONTRADAS

- No decorrer do projeto, cada integrante do grupo desempenhou um papel ativo e significativo, contribuindo de maneira efetiva e igual para a sua conclusão.
- A tentativa de implementação de um algoritmo heurístico para a redução da variância na distribuição de água provou ser uma tarefa complexa e intrincada. O desafio residia em encontrar um equilíbrio ótimo que minimizasse as diferenças sem afetar a eficiência geral do sistema.
- O desenvolvimento do conceito de 'Partial Edmonds' revelou-se um desafio significativo, particularmente na construção de um grafo parcial que capturasse todas as vertentes relacionadas à remoção de um reservatório.



# FIM

