

# ANALYSIS TOOL FOR WATER SUPPLY MANAGEMENT

*DA Projeto 1 - 2023/2024 – Turma 4*

Elementos do Grupo:

1. Ângelo Oliveira(202207798)
2. Bruno Fortes(202209730)
3. José Costa(202207871)

# Objetivo

O objetivo deste trabalho é o desenvolvimento de diversos algoritmos para análise e gerência do abastecimento de água em Portugal

# Classes

**Graph:** Responsável por definir a estrutura do grafo usada como base para a elaboração do projeto.

**Algorithm:** Este módulo contém os algoritmos utilizados para realizar operações específicas no grafo, como cálculo de rotas mais curtas, determinação de fluxos máximos. Ele fornece a lógica necessária para a aplicação dos algoritmos no contexto do problema.

**DeliverySites:** Objeto usado no grafo para representar e fornecer funcionalidades relacionadas aos locais de escoamento.



# Classes

**LoadingFunctions:** Ponto de partida da nossa aplicação, trata do processamento dos dados fornecidos nos ficheiros .csv lendo-os e organizando-os nas estruturas por nós escolhidas

**Logic:** Esta envolvida com a manipulação e gerenciamento de uma rede de distribuição de água, implementando funcionalidades como a identificação de fontes e destinos, criação de super-nós para cálculos de fluxo máximo e recuperação de informações sobre os tubos da rede.

**UI:** Interface servil do utilizador para a navegação do programa.





# O Grafo

Vertex é parte de uma estrutura de dados que representa um Delivery Site, onde cada vértice contém informações sobre estações de bombeamento, as suas arestas adjacentes e seu estado durante a execução de diversos algoritmos.

```
1  template <class T>
2  class Vertex {
3  public:
4      Vertex(T in);
5      bool operator<(Vertex<T> & vertex) const; // required by MutablePriorityQueue
6
7      T getInfo() const;
8      std::vector<Edge<T>*> getAdj() const;
9      bool isVisited() const;
10     bool isProcessing() const;
11     unsigned int getIndegree() const;
12     int getIncomingFlow() const;
13     int getOutgoingFlow() const;
14     double getDist() const;
15     Edge<T> *getPath() const;
16     std::vector<Edge<T>*> getIncoming() const;
17
18     void setInfo(T info);
19     void setVisited(bool visited);
20     void setProcessing(bool processing);
21     void setIndegree(unsigned int indegree);
22     void setDist(double dist);
23     void setIncomingFlow(int flow);
24     void setPath(Edge<T> *path);
25     Edge<T> *addEdge(Vertex<T> *dest, double w);
26     bool removeEdge(T in);
27     void removeOutgoingEdges();
28
29     double calculateIncomingFlow() const;
30     double calculateOutgoingFlow() const;
31     bool noOutgoingFlow() const;
32     bool noIncomingFlow() const;
33
34     friend class MutablePriorityQueue<Vertex>;
35 protected:
36     T info; // info node
37     std::vector<Edge<T>*> adj; // outgoing edges
38
39     // auxiliary fields
40     bool visited = false; // used by DFS, BFS, Prim ...
41     bool processing = false; // used by isDAG (in addition to the visited attribute)
42     unsigned int indegree; // used by topsort
43     double dist = 0;
44     Edge<T> *path = nullptr;
45     int incomingFlow = 0;
46
47     std::vector<Edge<T>*> incoming; // incoming edges
48
49     int queueIndex = 0; // required by MutablePriorityQueue and UFDS
50
51     void deleteEdge(Edge<T> *edge);
```

```
1  template <class T>
2  class Edge {
3  public:
4      Edge(Vertex<T> *orig, Vertex<T> *dest, double w);
5
6      Vertex<T> *getDest() const;
7      double getWeight() const;
8      bool isSelected() const;
9      Vertex<T> *getOrig() const;
10     Edge<T> *getReverse() const;
11     double getFlow() const;
12
13     void setSelected(bool selected);
14     void setReverse(Edge<T> *reverse);
15     void setFlow(double flow);
16 protected:
17     Vertex<T> *dest; // destination vertex
18     double weight; // edge weight, can also be used for capacity
19
20     // auxiliary fields
21     bool selected = false;
22
23     // used for bidirectional edges
24     Vertex<T> *orig;
25     Edge<T> *reverse = nullptr;
26
27     double flow; // for flow-related problems
```

Edge é parte de uma estrutura de dados que representa uma aresta, neste caso um pipe, onde cada vértice contém informações sobre as cidades ou pumping stations.

# O Grafo

Essa classe encapsula as operações e algoritmos comuns em grafos, fornecendo métodos para manipular vértices e arestas, realizar buscas e processamento de grafos, calcular métricas e realizar análises específicas.

```
1  template <class T>
2  class Graph {
3  public:
4      ~Graph();
5      /*
6       * Auxiliary function to find a vertex with a given the content.
7       */
8      Vertex<T> *findVertex(const T &in) const;
9      /*
10     * Adds a vertex with a given content or info (in) to a graph (this).
11     * Returns true if successful, and false if a vertex with that content already exists.
12     */
13     bool addVertex(const T &in);
14     bool removeVertex(const T &in);
15
16     /*
17     * Adds an edge to a graph (this), given the contents of the source and
18     * destination vertices and the edge weight (w).
19     * Returns true if successful, and false if the source or destination vertex does not exist.
20     */
21     bool addEdge(const T &source, const T &dest, double w);
22     bool removeEdge(const T &source, const T &dest);
23     bool addBidirectionalEdge(const T &source, const T &dest, double w);
24
25     int getNumVertex() const;
26     std::vector<Vertex<T>> *getVertexSet() const;
27
28     std::vector<T> dfs() const;
29     std::vector<T> dfs(const T &source) const;
30     const std::vector<std::vector<Edge<T>>*> allPaths(const T &source, const T &target) const;
31     void allPathsAux(Vertex<T> *current, Vertex<T> *target, std::vector<Edge<T>>*> &currentPath,
32                     std::vector<std::vector<Edge<T>>*> &allPaths) const;
33     void dfsVisit(Vertex<T> *v, std::vector<T> &res) const;
34     std::vector<T> bfs(const T &source) const;
35
36     bool isDAG() const;
37     bool dfsIsDAG(Vertex<T> *v) const;
38     std::vector<T> topsort() const;
39
40     int calculateFlowAcrossEdges() const;
41     bool checkEdgesFlow() const;
42     Metrics calculateMetrics() const;
43     std::vector<Edge<T>*> getEdges() const;
44
45     void printMetrics(Metrics metrics) const;
46 protected:
47     std::vector<Edge<T>> *edgeSet;
48     std::vector<Vertex<T>> *vertexSet; // vertex set
49
50     double **distMatrix = nullptr; // dist matrix for Floyd-Warshall
51     int **pathMatrix = nullptr; // path matrix for Floyd-Warshall
52
53     /*
54     * Finds the index of the vertex with a given content.
55     */
56     int findVertexIdx(const T &in) const;
```

# Descrição da leitura do dataset a partir dos ficheiros dados

A leitura e processamento dos datasets são cruciais para o funcionamento do sistema de gerência de abastecimento de água. Esta operação é realizada por meio de várias classes e métodos específicos, garantindo que os dados sejam carregados de forma correta e eficientemente no sistema.

Função responsável por carregar os dados das cidades a partir de um arquivo CSV e criar objetos DeliverySite para representar essas cidades.

```
void LoadCities() {
    std::string path = "SmallDataSet";
    std::ifstream file("SmallDataSet/Cities_Madeira.csv");
    if (!file.is_open()) {
        std::cerr << "Failed to open the CSV file." << std::endl;
    }
    std::string line;
    getline(file, line);
    while (getline(file, line)) {
        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
        std::istringstream lineStream(line);
        std::vector<std::string> tokens;
        std::string token;
        while (getline(lineStream, token, ',')) {
            tokens.push_back(token);
        }
        std::string name = tokens[0];
        std::string municipality;
        std::string code = tokens[2];
        int id = stoi(tokens[1]);
        int maxDelivery = 0;
        int demand = std::stoi(tokens[3]);
        int population = 0;
        if (path == "SmallDataSet") {
            NormaliseString(tokens[4], tokens[5]);
            population = std::stoi(tokens[4]);
        } else {
            Remove_terminations(tokens[4]);
            population = std::stoi(tokens[4]);
        }
        DeliverySite deliverySite(name, municipality, code, id, maxDelivery, demand, population, CITY);
        nodesToAdd.insert(deliverySite);
    }
    file.close();
}
```

```
void LoadWaterReservoirs() {
    std::ifstream file("SmallDataSet/Reservoirs_Madeira.csv");
    if (!file.is_open()) {
        std::cerr << "Failed to open the CSV file." << std::endl;
    }
    std::string line;
    getline(file, line);
    while (getline(file, line)) {
        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
        std::istringstream lineStream(line);
        std::vector<std::string> tokens;
        std::string token;
        while (getline(lineStream, token, ',')) {
            tokens.push_back(token);
        }
        Remove_terminations(tokens[4]);
        std::string name = tokens[0];
        std::string municipality = tokens[1];
        std::string code = tokens[3];
        int id = stoi(tokens[2]);
        int maxDelivery = stoi(tokens[4]);
        int demand = 0;
        int population = 0;
        //mandatory
        DeliverySite deliverySite(name, municipality, code, id, maxDelivery, demand, population, WATER_RESERVOIR);
        nodesToAdd.insert(deliverySite);
    }
    file.close();
}
```

Função responsável por carregar os dados dos canos a partir de um arquivo CSV e criar objetos PumpingStations para representar os pipes.

# Descrição da leitura do dataset a partir dos ficheiros dados

Esta função é responsável por carregar os dados dos reservatórios de água a partir de um arquivo CSV e criar objetos DeliverySite para representar esses reservatórios.

```
void LoadPipes() {
    std::ifstream file("SmallDataSet/Pipes_Madeira.csv");
    if (!file.is_open()) {
        std::cerr << "Failed to open the CSV file." << std::endl;
    }
    std::string line;
    getline(file, line);
    while (getline(file, line)) {
        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
        std::istringstream lineStream(line);
        std::vector<std::string> tokens;
        std::string token;
        while (getline(lineStream, token, ',')) {
            tokens.push_back(token);
        }
        std::string servicePointA = tokens[0];
        std::string servicePointB = tokens[1];
        int capacity = stoi(tokens[2]);
        Remove_terminations(tokens[3]);
        bool direction = stoi(tokens[3]);
        PumpingStations pumpingStation(servicePointA, servicePointB, capacity, direction);
        edges.push_back(pumpingStation);
    }
    file.close();
}
```

```
void LoadFireStations()
{
    std::ifstream file("SmallDataSet/Stations_Madeira.csv");
    if (!file.is_open()) {
        std::cerr << "Failed to open the CSV file." << std::endl;
    }
    std::string line;
    getline(file, line);
    while (getline(file, line)) {
        std::istringstream lineStream(line);
        std::vector<std::string> tokens;
        std::string token;
        while (getline(lineStream, token, ',')) {
            if (!token.empty())
                tokens.push_back(token);
        }
        if (!tokens.empty()) {
            std::string name;
            std::string municipality;
            Remove_terminations(tokens[1]);
            std::string code = tokens[1];
            int id = stoi(tokens[0]);
            int maxDelivery = 0;
            int demand = 0;
            int population = 0;
            //mandatory
            DeliverySite deliverySite(name, municipality, code, id, maxDelivery, demand, population, FIRE_STATION);
            nodesToAdd.insert(deliverySite);
        }
    }
    file.close();
    //std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

Função responsável por carregar os dados das estações de bombeamento a partir de um arquivo CSV e criar objetos DeliverySite para representar essas estações.



# Edmonds Karp

O algoritmo de Edmonds-Karp é uma implementação para encontrar o fluxo máximo num grafo. Ele usa uma BFS para encontrar caminhos na rede, onde o fluxo pode ser aumentado. Ao encontrar um caminho ela percorre-o, aumentando o fluxo à medida que passa. O processo continua até que não seja mais possível encontrar mais caminhos de onde o fluxo pode ser aumentado. O fluxo máximo é então determinado somando-se todos os fluxos aumentados ao longo dos diversos caminhos encontrados. Este algoritmo garante que o fluxo máximo seja encontrado em redes de fluxo cujas capacidades definidas estão definidas nas suas arestas.

```
1 ouble edmondsKarp(Graph<DeliverySite> *g, const DeliverySite& source, const DeliverySite& target, const DeliverySite& removed) {
2      double maxFlow = 0;
3      // Find source and target vertices in the graph
4      Vertex<DeliverySite>* s = g->findVertex(source);
5      Vertex<DeliverySite>* t = g->findVertex(target);
6      Vertex<DeliverySite>* remove = g->findVertex(removed);
7      // Validate source and target vertices
8      if (s == nullptr || t == nullptr || s == t)
9          throw std::logic_error("Invalid source and/or target vertex");
10     // Initialize flow on all edges to 0
11     for (auto v : g->getVertexSet()) {
12         for (auto e : v->getAdj()) {
13             e->setFlow(0);
14             e->setSelected(false);
15         }
16     }
17
18     // While there is an augmenting path, augment the flow along the path
19     while(findAugmentingPath(g, s, t, remove) ) {
20         double f = findMinResidualAlongPath(s, t);
21         maxFlow += f;
22         augmentFlowAlongPath(s, t, f);
23     }
24     return maxFlow;
25 }
26
```

# Heurística

A heurística tem como objetivo rebalancear e otimizar o fluxo ao longo da rede de modo a que ele seja mais uniforme, buscando minimizar a atributos como a média e variância que servem de auxílio ao algoritmo, conservando sempre o fluxo final de cada ponto do grafo. Para tal a heurística começa ordenando os canos consoante o seu fluxo, procurando outro trajeto "mais vazio" a partir do ponto inicial até ao destino do cano principal. De seguida procede a redistribuir a bombear fluxo por esse caminho escolhido, repetindo o processo enquanto a média ou variância ao longo do grafo melhorarem.

```
1 Metrics heuristic(GraphDeliverySite* g){
2     std::vector<EdgeDeliverySite*> edges;
3
4     edges = g->getEdges(); //O(V+E)
5
6     Metrics finalMetrics = g->calculateMetrics();
7     Metrics initialMetrics = finalMetrics;
8
9     g->printMetrics(initialMetrics);
10    initialMetrics = {DBL_MAX, DBL_MAX, DBL_MAX, DBL_MAX};
11    while(finalMetrics.variance < initialMetrics.variance || finalMetrics.avg < initialMetrics.avg){
12
13        std::sort(edges.begin(), edges.end(), [](EdgeDeliverySite* a, EdgeDeliverySite* b) {
14
15            if(a->getWeight() - a->getFlow() == b->getWeight() - b->getFlow()){
16                return a->getWeight() > b->getWeight();
17            }
18
19            return a->getWeight() - a->getFlow() < b->getWeight() - b->getFlow();
20        }); //O(E log E)
21
22        //O(E)
23        for(EdgeDeliverySite* e : edges){
24            std::vector<EdgeDeliverySite*> path;
25            std::vector<std::vector<EdgeDeliverySite*>> allPaths;
26
27            //O(V+E)
28            allPaths = g->allPaths(e->getOrig()->getInfo(), e->getDest()->getInfo());
29
30            double maxDiff = -1;
31            if(allPaths.empty())
32                continue;
33
34            for(std::vector<EdgeDeliverySite*> tempPath : allPaths){
35                double minFlow = minLeftOverCap(tempPath);
36                if (minFlow > maxDiff) {
37                    maxDiff = minFlow;
38                    path = tempPath;
39                }
40            }
41
42            double waterToPump = maxDiff;
43            if(e->getFlow() - waterToPump < 0)
44                waterToPump = e->getFlow();
45
46            e->setFlow(e->getFlow() - waterToPump);
47
48            pumpWater(path, waterToPump);
49        }
50
51        initialMetrics = finalMetrics;
52        finalMetrics = g->calculateMetrics();
53
54    }
55
56    finalMetrics = g->calculateMetrics();
57    g->printMetrics(finalMetrics);
58
59    for(auto e : g->getEdges()){
60        if(e->getFlow() > e->getWeight()){
61            print("SOBRECARREGA", false);
62        }
63        if(e->getFlow() < 0)
64            print("DESCEU O CANO", false);
65    }
66    return finalMetrics;
67 }
68
69 }
```

# Resiliência

A resiliência implementa algoritmos que permitem retirar quer uma cidade , uma fire station ou um pipe para testar se a rede é resistente ou sensível a esta alterações. No caso específico de tirar uma cidade é possível ao utilizador escolher se quer recalcular todo o algoritmo de max flow da rede ou optar por um algoritmo alternativo que não recalcula toda rede , mas tenta redistribuir alguma da água perdida nas cidades , por água que está disponível em outros reservatórios, para as cidades que foram afetadas consigam tentar combater a diferença significativa que originou a retirada do reservatório

```
1 double redistributeWaterWithoutMaxFlow2(Graph<DeliverySite>*g, std::vector<std::vector<Edge<DeliverySite>>>& paths){
2     if(paths.empty()) return 0;
3
4     for(auto v : g->getVertexSet()){
5         v->setInNeed(0);
6         v->setAlreadyHas(0);
7         v->setVisited(false);
8         for(Edge<DeliverySite>* edge: v->getAdj()){
9             edge->setSelected(false);
10            edge->setNeeds(0);
11        }
12    }
13
14    std::vector<Vertex<DeliverySite>> cities;
15    for(auto path : paths){
16        if(path.size() < 2) continue;
17        Edge<DeliverySite>* edge = path[path.size() - 1];
18        if(!edge->isSelected()){
19            Vertex<DeliverySite>* cityFound = edge->getDest();
20            cityFound->setInNeed(cityFound->getInNeed() + edge->getFlow());
21            edge->setNeeds(edge->getFlow());
22            cities.push_back(edge->getDest());
23        }
24    }
25
26    for(std::vector<Edge<DeliverySite>>* path : paths){
27        double minFlow = DBL_MAX;
28        for(auto & i : path) minFlow = std::min(minFlow,i->getFlow());
29
30        for(auto & i : path) i->setFlow(i->getFlow() - minFlow);
31    }
32
33    for(Vertex<DeliverySite>* v: g->getVertexSet()){
34        for(Edge<DeliverySite>* e : v->getAdj()){
35            e->setSelected(false);
36        }
37    }
38
39    for(std::vector<Edge<DeliverySite>>* path : paths){
40        if(path.size() < 2) continue;
41        Edge<DeliverySite>* edge = path[path.size() - 1];
42        if(!edge->isSelected()){
43            Vertex<DeliverySite>* city = edge->getDest();
44            city->setInNeed(city->getInNeed() - edge->getFlow());
45            edge->setNeeds(edge->getNeeds() - edge->getFlow());
46            edge->setSelected(true);
47        }
48    }
49
50    Vertex<DeliverySite>* water_reservoir = paths[0][0]->getOrig();
51
52    for(Vertex<DeliverySite>* city : cities){
53        while(true){
54            Vertex<DeliverySite>* augment = findAugPath(g,city,water_reservoir);
55            if(augment == nullptr){
56                break;
57            }
58            double flow = minResidualAugPath(g,augment,city);
59
60            if(flow > city->getInNeed()){
61                flow = city->getInNeed();
62            }
63            augmentFlowPath(augment,city,flow);
64            city->setInNeed(city->getInNeed() - flow);
65            city->setAlreadyHas(city->getAlreadyHas() + flow);
66        }
67    }
68
69    double flow = 0;
70    for(Vertex<DeliverySite>* v : g->getVertexSet()){
71        if(v->getInfo().getNodeId() == CITY){
72            for(auto e : v->getIncoming()){
73                flow += e->getFlow();
74            }
75        }
76    }
77
78    return flow;
79 }
80
81 }
```

# Ui

## Menu Interativo:

O sistema conta com um menu interativo implementado no arquivo UI.cpp. Este menu oferece uma interface de usuário amigável e fácil de usar, que contém a implementação das funções necessárias para interagir com o usuário e conduzir as operações principais do sistema de análise de gerenciamento de abastecimento de água.

```
Press A to start the program:a

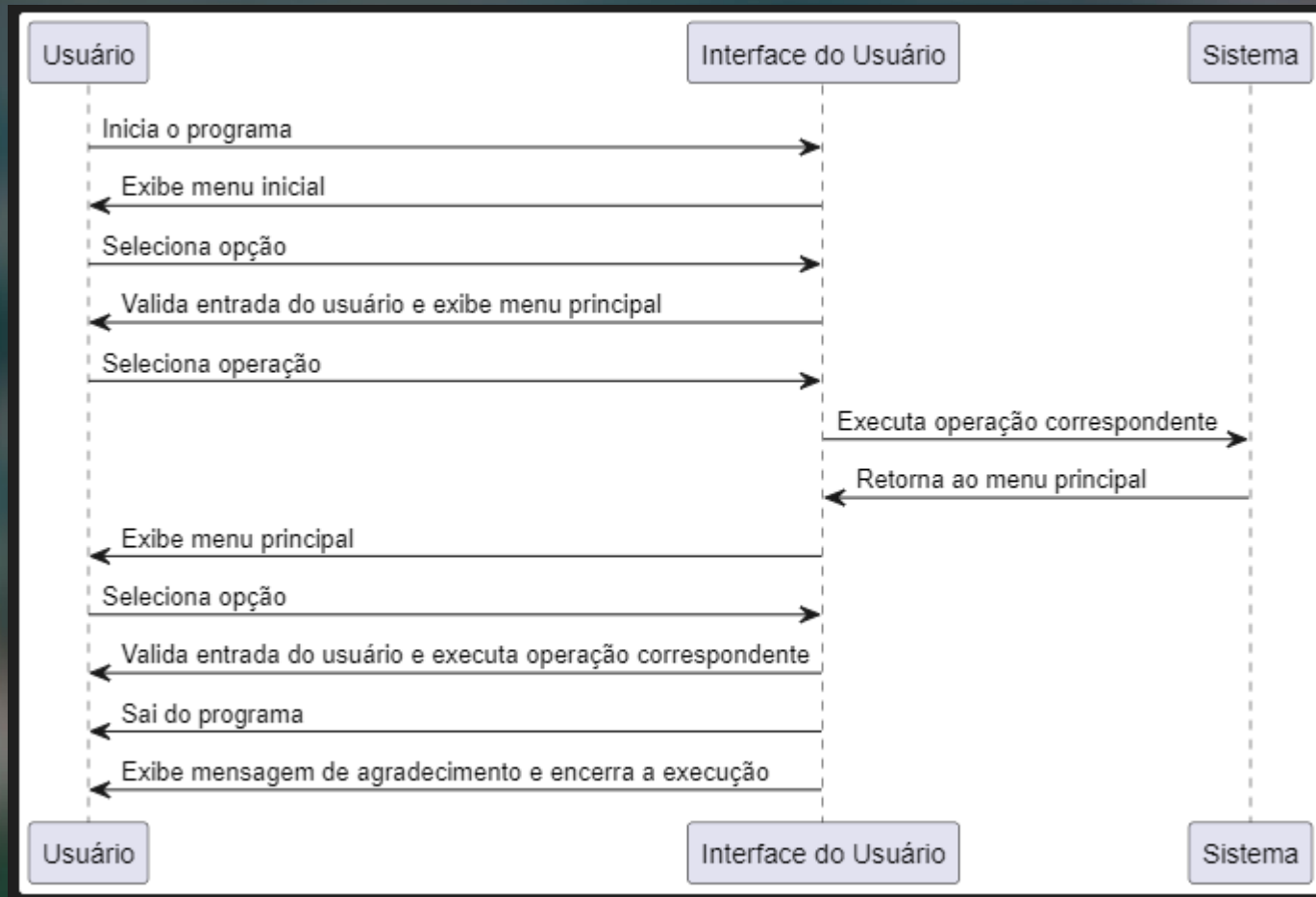
#####
@ @ @ @@@@@ @@@@@ @@@@@ @@@@@ @@@@@ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
@ @ @ @@@@@ @ @@@@@ @@@@@ @@@@@ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
  @ @ @ @ @ @ @@@@@ @ @ @ @ @ @ @ @ @ @ @ @ @ @
#####

Welcome to the Analysis Tool for Water Supply Management, what would you like to do?
A. Proceed to the application
B. Close the application
Insert the letter:a

What would you like to know?
A. Run Max Flow algorithm
B. Check if every city meets it's water demand
C. Check heuristic stats of the max flow
D. Evaluate network's resiliency
E. Exit the program
Insert your choice:|
```



# Ui



# Test Cases

## Max Flow:

```
The max flow for the entire network is: 24163
```

```
Insert the code of the city:
```

```
C_1
```

```
The max flow for the city C_1 is: 64
```

## Water Demand

```
The following cities don't receive enough water :
```

City Name	City Code	Demand	Flow	Defecit
Lagos	C_13	158	123	35
Évora	C_10	313	220	93
Viana do Castelo	C_20	168	100	68
Vila real	C_21	161	135	26
Viseu	C_22	397	330	67
Bragança	C_5	152	125	27
Porto	C_17	6324	5650	674
Covilhã	C_8	122	100	22
Beja	C_3	160	110	50

# Test Cases

## Heuristica:

```
-----  
Average: 862.534  
Variance: 3.51703e+06  
Maximum Difference: 14000  
-----
```

```
Average: 673.447  
Variance: 2.83952e+06  
Maximum Difference: 14000  
-----
```

## Resiliência

The max flow of the network removing PS\_4 is: 23955

City Name	City Code	Required Units	New Flow	Old Flow
Braga	C_4	208	1000	1208

# Destaque de Funcionalidades

Os Destaques são:

**O algoritmo de Edmonds-Karp:** Após intensa discussão e várias iterações de testes, finalmente conseguimos implementar o algoritmo de Edmonds-Karp com sucesso, que no início apresentava inúmeros problemas.

**A Heurística:** O nosso objetivo era encontrar uma abordagem inteligente e eficiente para balancear o fluxo e melhorar o status geral da rede. Após uma série de iterações e ajustes, finalmente conseguimos desenvolver uma heurística que oferece resultados significativamente melhores e contribui para uma gestão mais eficaz dos recursos de água.

**A Resiliência da Rede:** Este foi um dos desafios mais complexos que enfrentamos. Passamos dias a trabalhar na avaliação da resiliência da rede, buscando identificar e mitigar possíveis pontos de falha. Após muita pesquisa e testes, conseguimos desenvolver uma estratégia robusta para lidar com falhas em diferentes componentes da rede, garantindo assim uma operação mais confiável e resiliente no fornecimento de água.



# Dificuldades do Trabalho e Participação

Dificuldades do Trabalho e Participação As principais dificuldades deste trabalho foram:

**O algoritmo de Edmonds karp:** houve muita discussão sobre o resultado do maxflow, o que levou a uma intensa caça ao erro no algoritmo.

**A Heurística:** foi um dos aspetos mais complicados do projeto, não na teorização em si do algoritmo mas na implementação do mesmo que foi extremamente complicada.

**A Resiliência da rede:** foi certamente a parte mais complicada do trabalho uma vez que demoramos vários dias a pensar numa implementação eficiente e correta, que depois na prática foi mais complicado implementar do que aquilo que esperávamos.

Up202207798 – Ângelo Oliveira – 33,3%

Up202209730 – Bruno Fortes – 33,3%

Up202207871 – José Costa – 33,3%

# Considerações finais

Este projeto foi uma oportunidade valiosa para aplicar os algoritmos que aprendemos em sala de aula na resolução de problemas do mundo real. Foi uma prova concreta de que esses algoritmos não são apenas teoria acadêmica, mas ferramentas poderosas que têm aplicações práticas em diversos contextos. Pudemos também perceber como os algoritmos de grafos, cálculo de rotas mais curtas e determinação de fluxos máximos podem ser utilizados para resolver problemas reais. Essa experiência preparou-nos para enfrentar desafios semelhantes no futuro, seja na academia, em ambientes profissionais ou até mesmo em projetos pessoais. Além disso, essa vivência mostrou-nos a importância de compreender não apenas a teoria por trás dos algoritmos, mas também sua aplicação prática. Isso permite-nos não apenas resolver problemas de forma eficiente, mas também nos dá uma base sólida para propor soluções inovadoras e eficazes em diversos cenários.

**FIM**