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

objetivo deste trabalho é para o desenvolvimento de um algoritmo para análise e gerenciamento de 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

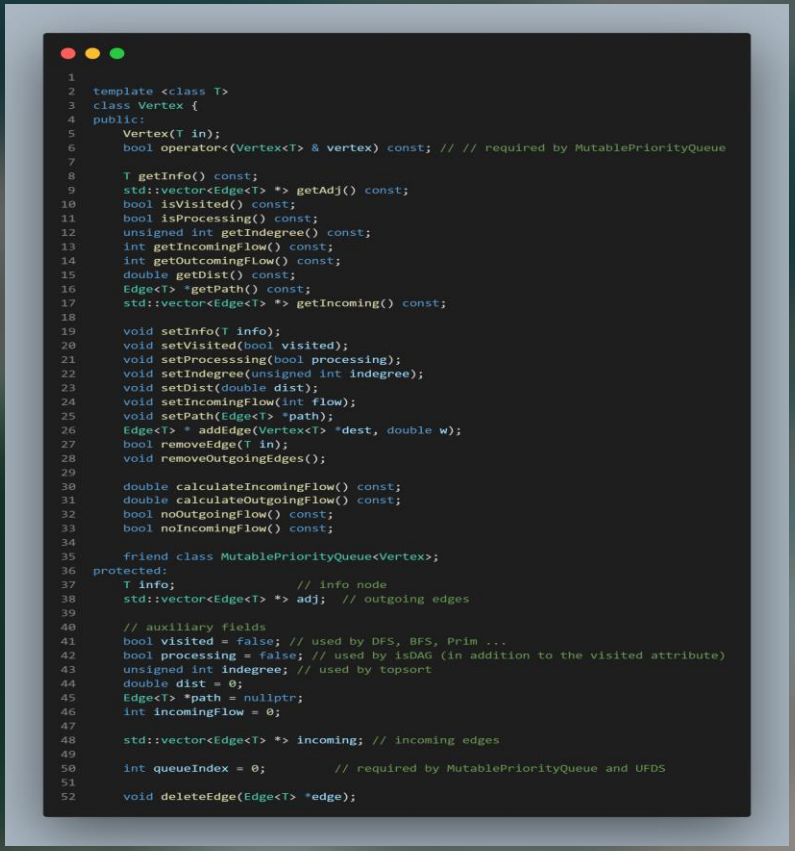
LoadingFunctions:Ponto de partida da nossa aplicação, trata do processamento dos dados fornecidos nos ficheiros .csv lendo-os e organizando-os nas estruturas por nós escolhidas

Logic: Esta envolvida com a manipulação e gerenciamento de uma rede de distribuição de agua, 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.

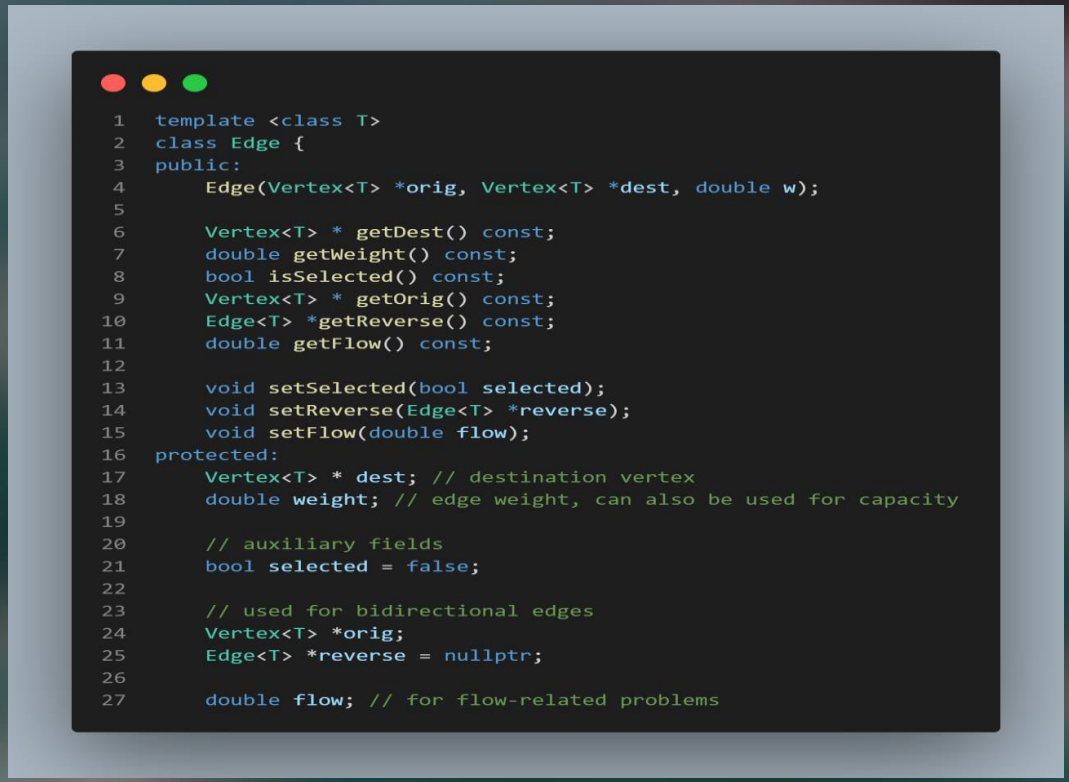
UI: Interface servil do utilizador para a navegação do programa.

O Grafo

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



```
template <class T>
class Vertex {
public:
    Vertex(T in);
    bool operator<(Vertex<T> & vertex) const; // // required by MutablePriorityQueue

    T getInfo() const;
    std::vector<Edge<T> *> getAdj() const;
    bool isVisited() const;
    bool isProcessing() const;
    unsigned int getIndegree() const;
    int getIncomingFlow() const;
    int getOutcomingFLow() const;
    double getDist() const;
    Edge<T> *getPath() const;
    std::vector<Edge<T> *> getIncoming() const;

    void setInfo(T info);
    void setVisited(bool visited);
    void setProcesssing(bool processing);
    void setIndegree(unsigned int indegree);
    void setDist(double dist);
    void setIncomingFlow(int flow);
    void setPath(Edge<T> *path);
    Edge<T> * addEdge(Vertex<T> *dest, double w);
    bool removeEdge(T in);
    void removeOutgoingEdges();

    double calculateIncomingFlow() const;
    double calculateOutgoingFlow() const;
    bool noOutgoingFlow() const;
    bool noIncomingFlow() const;

    friend class MutablePriorityQueue<Vertex>;
protected:
    T info;                // info node
    std::vector<Edge<T> *> adj;  // outgoing edges

    // auxiliary fields
    bool visited = false; // used by DFS, BFS, Prim ...
    bool processing = false; // used by isDAG (in addition to the visited attribute)
    unsigned int indegree; // used by topsort
    double dist = 0;
    Edge<T> *path = nullptr;
    int incomingFlow = 0;

    std::vector<Edge<T> *> incoming; // incoming edges

    int queueIndex = 0;         // required by MutablePriorityQueue and UFDS

    void deleteEdge(Edge<T> *edge);
```



```
template <class T>
class Edge {
public:
    Edge(Vertex<T> *orig, Vertex<T> *dest, double w);

    Vertex<T> * getDest() const;
    double getWeight() const;
    bool isSelected() const;
    Vertex<T> * getOrig() const;
    Edge<T> *getReverse() const;
    double getFlow() const;

    void setSelected(bool selected);
    void setReverse(Edge<T> *reverse);
    void setFlow(double flow);
protected:
    Vertex<T> * dest; // destination vertex
    double weight; // edge weight, can also be used for capacity

    // auxiliary fields
    bool selected = false;

    // used for bidirectional edges
    Vertex<T> *orig;
    Edge<T> *reverse = nullptr;

    double flow; // for flow-related problems
```

Edge é parte de uma estrutura de dados que representa um grafo, onde cada vértice contém informações sobre as cidades.

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.

```cpp
template <class T>
class Graph {
public:
    ~Graph();
    /*
    * Auxiliary function to find a vertex with a given the content.
    */
    Vertex<T> *findVertex(const T &in) const;
    /*
     *  Adds a vertex with a given content or info (in) to a graph (this).
     *  Returns true if successful, and false if a vertex with that content already exists.
     */
    bool addVertex(const T &in);
    bool removeVertex(const T &in);

    /*
     * Adds an edge to a graph (this), given the contents of the source and
     * destination vertices and the edge weight (w).
     * Returns true if successful, and false if the source or destination vertex does not exist.
     */
    bool addEdge(const T &sourc, const T &dest, double w);
    bool removeEdge(const T &source, const T &dest);
    bool addBidirectionalEdge(const T &sourc, const T &dest, double w);

    int getNumVertex() const;
    std::vector<Vertex<T> *> getVertexSet() const;

    std:: vector<T> dfs() const;
    std:: vector<T> dfs(const T & source) const;
    const std::vector<std::vector<Edge<T> *>> allPaths(const T & source , const T & target) const;
    void allPathsAux(Vertex<T> *current, Vertex<T> *target, std::vector<Edge<T>*> &currentPath,
                     std::vector<std::vector<Edge<T>*>> &allPaths) const;
    void dfsVisit(Vertex<T> *v,  std::vector<T> & res) const;
    std::vector<T> bfs(const T & source) const;

    bool isDAG() const;
    bool dfsIsDAG(Vertex<T> *v) const;
    std::vector<T> topsort() const;

    int calculateFlowAcrossEdges() const;
    bool checkEdgesFlow() const;
    Metrics calculateMetrics() const;
    std::vector<Edge<T>*> getEdges() const;


    void printMetrics(Metrics metrics) const;
protected:
    std::vector<Edge<T> *> edgeSet;
    std::vector<Vertex<T> *> vertexSet;    // vertex set

    double ** distMatrix = nullptr;   // dist matrix for Floyd-Warshall
    int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall

    /*
     * Finds the index of the vertex with a given content.
     */
    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 gerenciamento 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.

funcao 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());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();
}
```

Funcao 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.

```cpp
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));
}
```

```cpp
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();
}
```

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 em uma rede de fluxo. Ele usa a (BFS) para encontrar caminhos de aumento na rede, onde o fluxo pode ser aumentado. Ele encontra um caminho de aumento usando e aumenta o fluxo ao longo desse caminho. O processo continua até que não seja mais possível encontrar caminhos de aumento na rede. O fluxo máximo é então determinado somando-se todos os fluxos aumentados ao longo dos caminhos de aumento encontrados. Este algoritmo garante que o fluxo máximo seja encontrado em redes de fluxo com capacidades definidas em 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
```

Heuristica

A heurística é projetada para redistribuir a água de forma eficiente em uma rede de entrega, considerando tanto a capacidade máxima de entrega em cada local de entrega quanto a demanda de água em cada cidade. Ela realiza o cálculo das métricas do grafo, que são utilizadas para acompanhar o progresso da redistribuição de água. Essas métricas incluem valores como a média e a variância do fluxo de água, entre outros, e são atualizadas ao longo do processo de redistribuição. O objetivo é otimizar a distribuição de água de modo a atender às demandas das cidades e respeitar as capacidades máximas de entrega em cada local, buscando minimizar a variância e outras métricas relevantes do fluxo.

```
Metrics heuristic(Graph<DeliverySite>*g){
    std::vector<Edge<DeliverySite>*> edges;

    edges = g->getEdges(); //O(V+E)

    Metrics finalMetrics = g->calculateMetrics();
    Metrics initialMetrics = finalMetrics;

    g->printMetrics(initialMetrics);
    initialMetrics = {DBL_MAX , DBL_MAX , DBL_MAX , DBL_MAX};
    while(finalMetrics.variance < initialMetrics.variance || finalMetrics.avg < initialMetrics.avg){

        std::sort(edges.begin(), edges.end(), [](Edge<DeliverySite>* a, Edge<DeliverySite>* b) {

            if(a->getWeight() - a->getFlow() == b->getWeight() - b->getFlow()){
                return a->getWeight() > b->getWeight();
            }

            return a->getWeight() - a->getFlow() < b->getWeight() - b->getFlow();
        });//O(E log E)

        //O(E)
        for(Edge<DeliverySite>* e : edges){
            std::vector<Edge<DeliverySite>*> path;
            std::vector<std::vector<Edge<DeliverySite>*>> allPaths;

            //O(V+E)
            allPaths = g->allPaths(e->getOrig()->getInfo() , e->getDest()->getInfo());

            double maxDiff = -1;
            if(allPaths.empty())
                continue;

            for(std::vector<Edge<DeliverySite>*> tempPath : allPaths){
                double minFlow = minLeftOverCap(tempPath);
                if (minFlow > maxDiff) {
                    maxDiff = minFlow;
                    path = tempPath;
                }
            }

            double waterToPump = maxDiff;
            if(e->getFlow() - waterToPump < 0)
                waterToPump = e->getFlow();

            e->setFlow(e->getFlow() - waterToPump);

            pumpWater(path , waterToPump);
        }

        initialMetrics = finalMetrics;

        finalMetrics = g->calculateMetrics();

    }

    finalMetrics = g->calculateMetrics();

    g->printMetrics(finalMetrics);

    for(auto e : g->getEdges()){
        if(e->getFlow() > e->getWeight()){
            print("SOBRECARGAAAA" , false);
        }
        if(e->getFlow() < 0)
            print("DESCEU O CANOOOOO" , false);
    }
    return  finalMetrics;
}

```

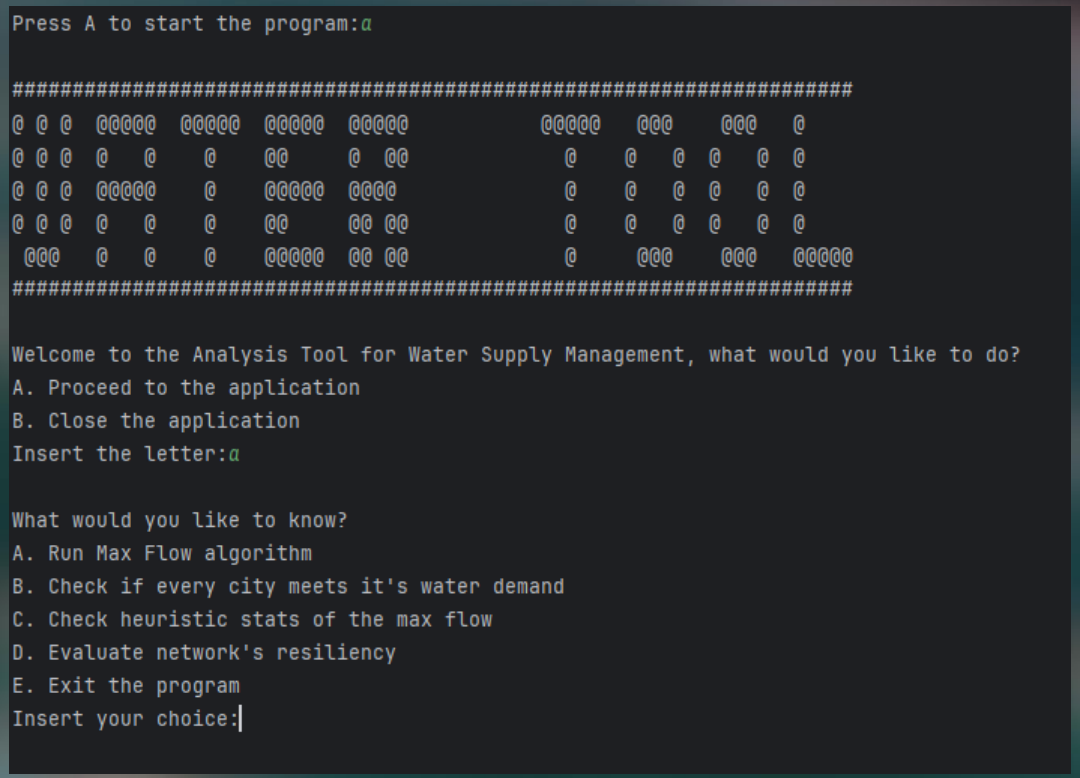
Resiliência

implementa a redistribuição de água em uma rede de entrega sem usar o algoritmo de fluxo máximo. Ela define as necessidades e quantidades de água já distribuídas para cada vértice na rede, Identifica as cidades e atualiza as suas necessidades de água com base nos caminhos fornecidos. Redução do fluxo ao longo dos caminhos fornecidos, ajustando as quantidades de água distribuídas. Determina caminhos de aumento de capacidade e ajusta os fluxos de água ao longo desses caminhos para satisfazer as necessidades das cidades.

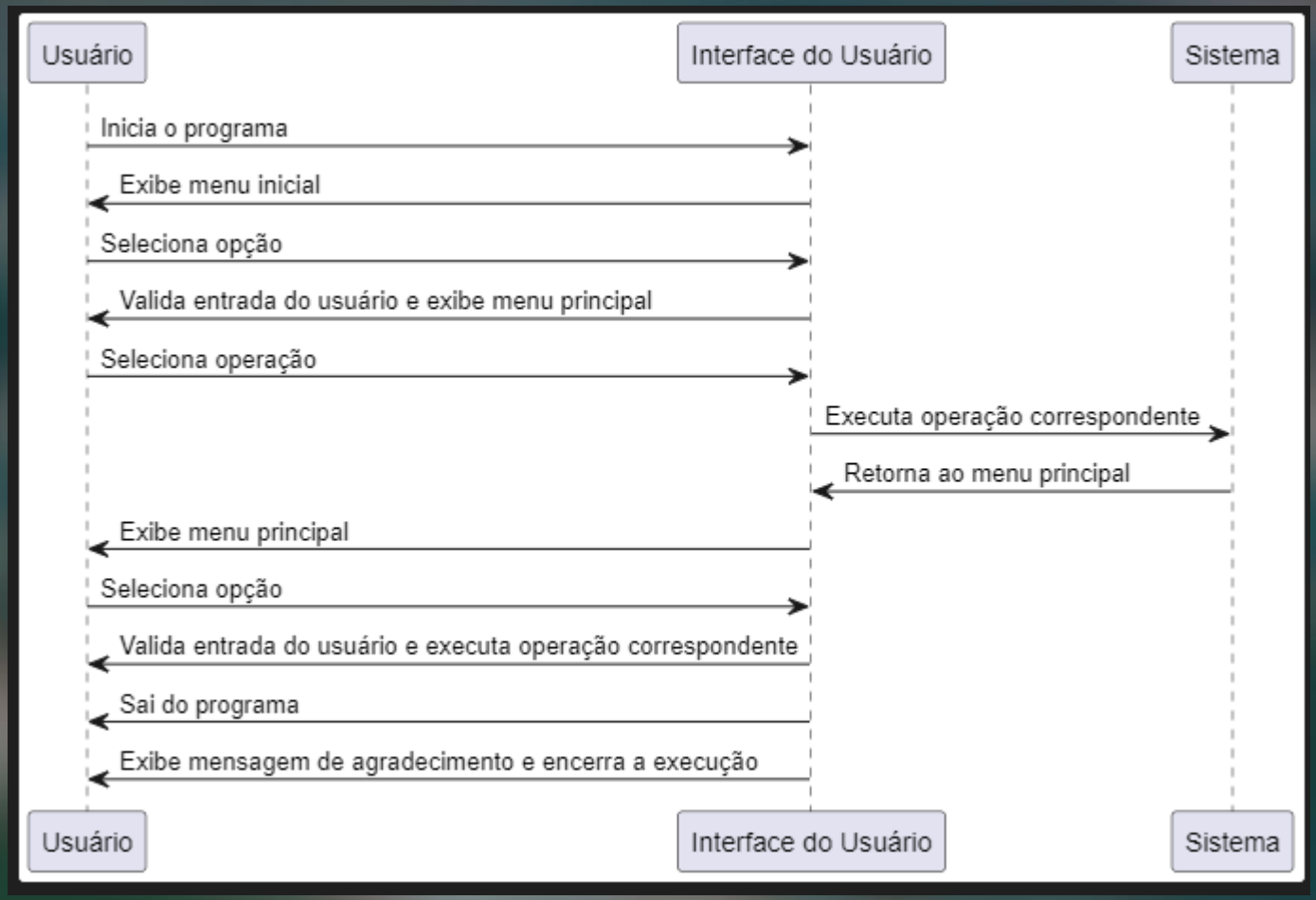
```
double redistributeWaterWithoutMaxFlow2(Graph<DeliverySite>*g, std::vector<std::vector<Edge<DeliverySite>*>>& paths){
    if(paths.empty()) return 0;

    for(auto v : g->getVertexSet()){
        v->setInNeed(0);
        v->setAlreadyHas(0);
        v->setVisited(false);
        for(Edge<DeliverySite>* edge: v->getAdj()){
            edge->setSelected(false);
            edge->setNeeds(0);
        }
    }

    std::vector<Vertex<DeliverySite>*> cities;
    for(auto path : paths){
        if(path.size() < 2) continue;
        Edge<DeliverySite>* edge = path[path.size() - 1];
        if(!edge->isSelected()){
            Vertex<DeliverySite>* cityFound = edge->getDest();
            cityFound->setInNeed(cityFound->getInNeed() + edge->getFlow());
            edge->setNeeds(edge->getFlow());
            cities.push_back(edge->getDest());
        }
    }

    for(std::vector<Edge<DeliverySite>*> path : paths){
        double minFlow = DBL_MAX;
        for(auto & i : path) minFlow = std::min(minFlow,i->getFlow());

        for(auto & i : path) i->setFlow(i->getFlow() - minFlow);
    }

    for(Vertex<DeliverySite>* v: g->getVertexSet()){
        for(Edge<DeliverySite>* e : v->getAdj()){
            e->setSelected(false);
        }
    }

    for(std::vector<Edge<DeliverySite>*> path : paths){
        if(path.size() < 2) continue;
        Edge<DeliverySite>* edge = path[path.size() - 1];
        if(!edge->isSelected()){
            Vertex<DeliverySite>* city = edge->getDest();
            city->setInNeed(city->getInNeed() - edge->getFlow());
            edge->setNeeds(edge->getNeeds() - edge->getFlow());
            edge->setSelected(true);
        }
    }

    Vertex<DeliverySite>* water_reservoir = paths[0][0]->getOrig();

    for(Vertex<DeliverySite>* city : cities){
        while(true){
            Vertex<DeliverySite>* augment = findAugPath(g,city,water_reservoir);
            if(augment == nullptr){
                break;
            }
            double flow = minResidualAugPath(g,augment,city);

            if(flow > city->getInNeed()){
                flow = city->getInNeed();
            }
            augmentFlowPath(augment,city,flow);
            city->setInNeed(city->getInNeed() - flow);
            city->setAlreadyHas(city->getAlreadyHas() + flow);

        }
    }

    double flow = 0;
    for(Vertex<DeliverySite>* v : g->getVertexSet()){
        if(v->getInfo().getNodeType() == CITY ){
            for(auto e : v->getIncoming()){
                flow += e->getFlow();
            }
        }
    }

    return flow;
}

```

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.



Ui


Usuário
Interface do Usuário
Sistema
Inicia o programa
Exibe menu inicial
Seleciona opção
Valida entrada do usuário e exibe menu principal
Seleciona operação
Executa operação correspondente
Retorna ao menu principal
Exibe menu principal
Seleciona opção
Valida entrada do usuário e executa operação correspondente
Sai do programa
Exibe mensagem de agradecimento e encerra a execução
Usuário
Interface do Usuário
Sistema

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
Porto               C_17                6324                5650                674
├évora              C_10                313                 220                 93
Covilh├ú            C_8                 122                 100                 22
Viseu               C_22                397                 330                 67
Lagos               C_13                158                 123                 35
Vila real           C_21                161                 135                 26
Bragan├ºa           C_5                 152                 125                 27
Viana do Castelo    C_20                168                 100                 68
Beja                C_3                 160                 110                 50
```

Test Cases

Heuristica:

```
←[0;32m------------------←[0m
Average: 862.438
Variance: 3.51817e+06
Maximum Difference: 14000
←[0;32m------------------←[0m←[0;32m------------------←[0m
Average: 677.101
Variance: 2.88951e+06
Maximum Difference: 14000
←[0;32m------------------←[0m
```

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 foram:

O algoritmo de Edmonds-Karp: Após intensa discussão e várias iterações de testes, finalmente conseguimos implementar o algoritmo de Edmonds-Karp com sucesso.

A Heurística: Nosso objetivo era encontrar uma abordagem mais 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 trabalhando na avaliação da resiliência da rede, buscando identificar e mitigar possíveis pontos de falha. Após muita pesquisa e experimentação, conseguimos desenvolver estratégias robustas 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

As dificuldades deste trabalho foram:

O algoritmo de Edmonds karp: pois houve muita discussao sobre o resultado do maxflow, e com isso tentou-se chegar a moda do resultado.

A Heuristica: pois queriamos econtrar uma melhor forma de balancear e melhorar os status.

A Resiliência da rede: houve uma grande dificuldade em determiner sem recalcular o maxflow.

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 tambem 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 nos preparou para enfrentar desafios semelhantes no futuro, seja na academia, em ambientes profissionais ou até mesmo em projetos pessoais. Além disso, essa vivência nos mostrou a importância de compreender não apenas a teoria por trás dos algoritmos, mas também sua aplicação prática. Isso nos permite não apenas resolver problemas de forma eficiente, mas também nos dá uma base sólida para propor soluções inovadoras e eficazes em diversos cenários.

FIM