U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

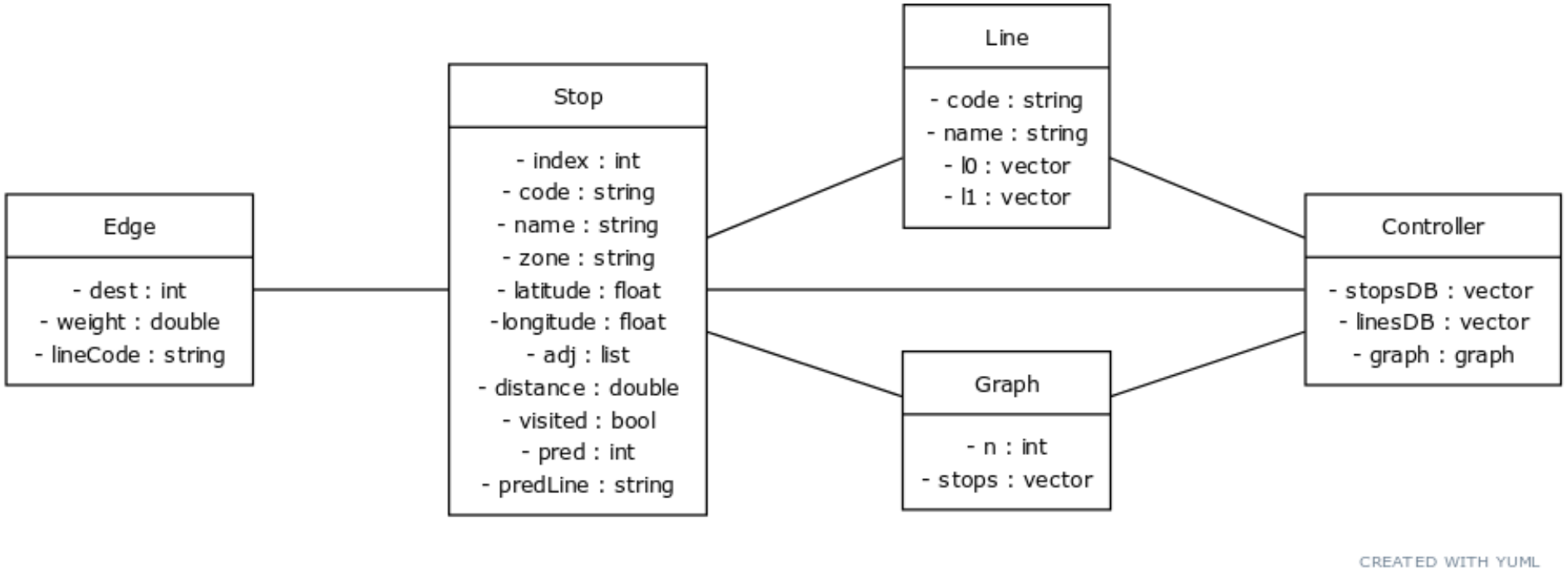
Trabalho prático II

Navegação nos transportes públicos do Porto

Trabalho realizado por:

- Adam Nogueira – 202007519 – up202007519@fe.up.pt
- Ana Sofia Costa – 202007602 – up202007602@fe.up.pt
- Igor Castro – 202000161 – up202000161@fe.up.pt

Diagrama de classes



NOTA : Todos os ficheiros presentes no trabalho estão apenas envolvidos com a classe Controller.

Descrição da leitura do dataset

```
class Controller{
private:
    vector<Stop> stopDB;
    vector<Line> linesDB;
    Graph graph;
    string userName;
    int maxWalingDistance;
    int walkingFactor;
```

```
class Line {
private:
    string code;
    string name;
    vector<int> l0;
    vector<int> l1;
```

Neste projeto a única classe que interage com os ficheiros é o Controller. Através dos seus métodos, esta classe irá criar:

- um vetor de Stops (através da leitura do ficheiro stops.csv);
- um vetor de Lines (através da leitura do ficheiro lines.csv);
- a cada linha será atribuído 2 vetores que representam os seus dois sentidos (através da leitura dos ficheiros de cada linha);
- um grafo com todas as Stops e Edges (representação da ligação que uma linha faz entre apenas duas paragens) da rede de transportes pública (durante todo o processo de leitura de ficheiros).

Além destes atributos, o programa guarda num ficheiro (userData.txt) o username e outras duas variáveis que irão ser introduzidas pelo utilizador, cujos valores condicionam o caminho que o sistema irá retornar.

Exemplo da leitura do ficheiro stops.cvs :

```
void Controller::readStops(){
    graph = Graph( nodes: 2487,  dir: true);
    ifstream stopsFile;
    stopsFile.open( s: "../src/dataset/stops.csv");
    if (stopsFile.fail()) {
        cout << "This file doesn't exist!\n";
    } else {
        string line;
        stopsFile >> line;
        stopsFile.ignore();
        int index=0;
        while (!stopsFile.eof() && stopsFile.peek()!='\n') {
            Stop stop;
            stopsFile >> stop;
            stop.setIndex(index);
            this->stopDB.push_back(stop);
            graph.getStops().push_back(stop);
            index++;
        }
        stopDB.pop_back();
    }
```

Descrição do grafo

Como já foi referido, no programa irá ser criado na classe Controller apenas um objeto da classe Grafo. Após o processo de leitura de todos os ficheiros, o grafo irá conter todas as paragens e consequentemente, as edges que as ligam, estando assim representadas todos os sentidos de cada linha e respetivas paragens da rede de transportes públicos.

Para além disso, depois desta primeira fase estar concluída, serão criados no grafo edges que representarão a ligação de uma paragem a outra a pé. A quantidade de edges adicionadas irá depender do valor da distância máxima que pode ser percorrida numa mudança de linha, que é introduzida pelo utilizador. O peso destas edges será aumentado (para que haja preferência no uso de transportes públicos) e depende da variável que representa o quanto a pessoa gosta de andar (quanto mais a pessoa gosta de andar menor irá ser).

Assim, será apenas neste grafo, que o sistema irá aplicar vários algoritmos para obter todas as informações que irão ser posteriormente disponibilizadas ao utilizador.

```
class Stop {
private:
    int index;
    string code;
    string name;
    string zone;
    float latitude;
    float longitude;
    list<Edge> adj;
    double distance;
    bool visited;   // As t
    int pred;
```

```
class Graph {
    int n;
    vector<Stop> stops;
```

```
class Edge{
private:
    int dest;
    double weight;
    string lineCode;
```

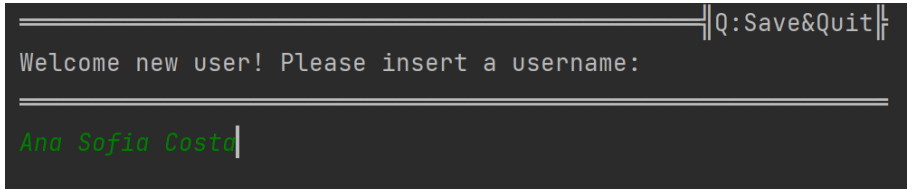
```
void Controller::addEdgesInWalkingDistance() {
    double walkingDistance = (double) maxWalingDistance / 1000;
    for (int i = 0; i < graph.getStops().size()-1; i++) {
        for (int j = i+1; j < graph.getStops().size(); j++) {
            double weight = haversine( lat1: graph.getStops()[i].getLatitude(),
                                       lon1: graph.getStops()[i].getLongitude(),
                                       lat2: graph.getStops()[j].getLatitude(),
                                       lon2: graph.getStops()[j].getLongitude());
            if (!graph.getStops()[i].isInAdj( &: graph.getStops()[j]) && weight <= walkingDistance) {
                graph.addEdge(i, j, weight: weight * (3 + 2*(10-(double) walkingFactor)/10), code: "walking");
            }
            if (!graph.getStops()[j].isInAdj( &: graph.getStops()[i]) && weight <= walkingDistance) {
                graph.addEdge(j, i, weight, code: "walking");
            }
        }
    }
}
```

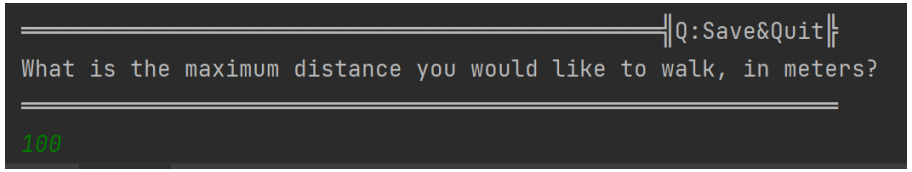
Interface com o utilizador

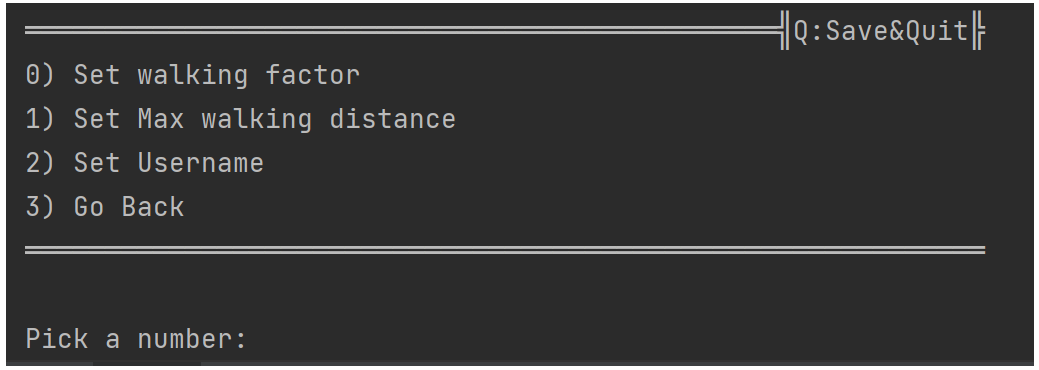
Na primeira vez que o programa será utilizado (ainda não existe o ficheiro userData.txt), irá ser pedido ao utilizador um username, um valor que quantifica o quanto gosta de caminhar e ainda a distância máxima que o percurso devolvido pelo sistema pode incluir, sendo este percorrido a pé. Numa segunda utilização do programa, o ficheiro userData.txt já estará criado sendo que o programa passará logo para o menu principal.

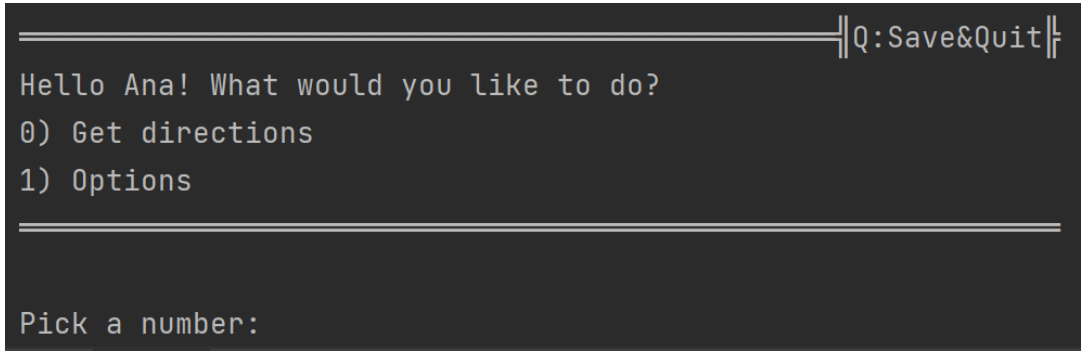
No menu principal, o utilizador depara-se com os dois diferentes campos:

1) Obter direções ;
2) Outras opções – estas irão incluir todas as mudanças das variáveis que se encontram guardadas no ficheiro userData.txt.









Lista de funcionalidades implementadas (1)

- **Origem/Destino:**

Neste projeto, foi dado ao utilizador oportunidade de definir a sua origem e o seu destino de diferentes formas.

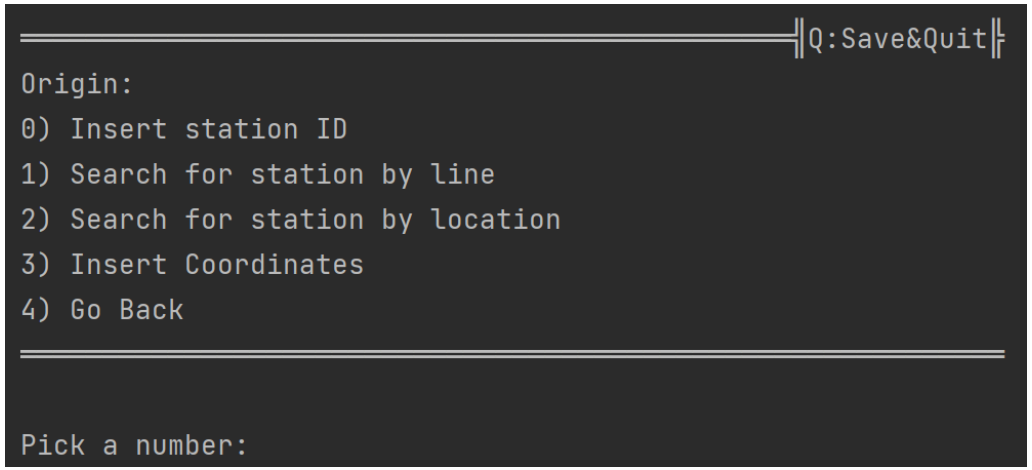
1) Poderá introduzir o código de uma paragem;

2) Poderá selecionar uma paragem de uma determinada linha, onde será disponibilizado a lista de paragens consoante a linha e sentido que escolher;

3) Poderá selecionar uma paragem de uma zona, onde será disponibilizado a lista de paragens consoante a zona que escolher;

4) Poderá introduzir as coordenadas de um local.

Exemplo de inserção da origem :



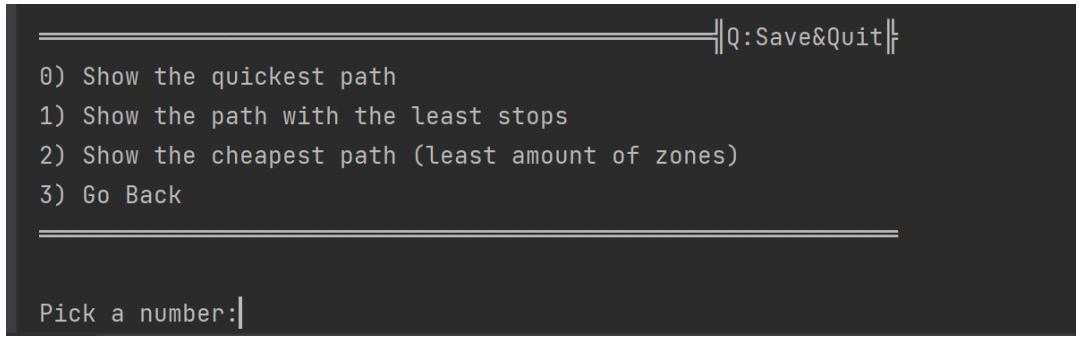
Exemplo de escolha da paragem por linha :

Lista de funcionalidades implementadas (2)

- **Conceito de “melhor” caminho:**

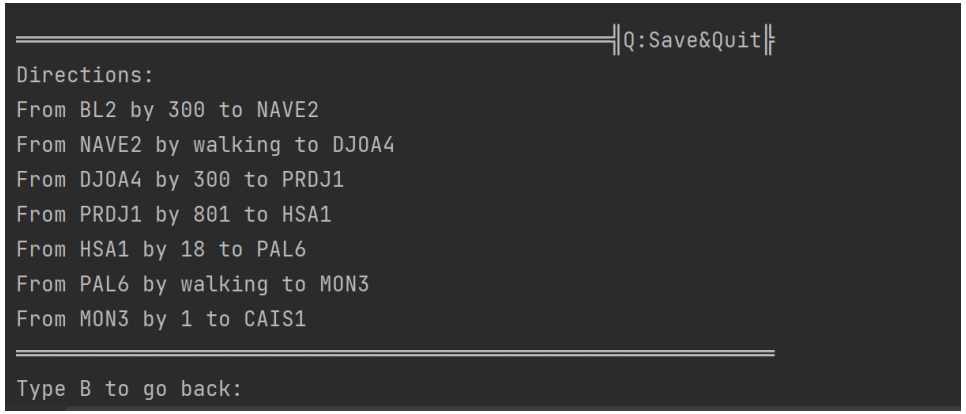
Neste programa, o utilizador vai ter a oportunidade escolher para si o melhor caminho tendo 3 opções :

- Menor distância;
- Menor de paragens;
- Menor número de zonas.



- **Mudança de linha:**

Foi também implementada uma funcionalidade, que permite ao utilizador a introdução da distância máxima que deseja percorrer a pé numa mudança de linha. Logo, para a mesma origem e mesmo destino podem ser retornados diferentes percursos, visto que um percurso onde o valor seja 0 (não haja mudanças de linha) será diferente de um percurso em que a distância seja igual a 50 metros (de acordo com o funcionamento deste programa, irão ser criadas edges entre paragens que distem até 50 metros).



Lista de funcionalidades implementadas (3)

- **Algoritmos desenvolvidos no trabalho :**

Para implementar diferentes funcionalidades, foram desenvolvidos diferentes algoritmos na classe Graph.

I) Algoritmo de dijkstra – Foi criado com duas versões para que o utilizador tivesse acesso tanto ao caminho com menor distância entre a origem e o destino e ao percurso que passaria por menos zonas. Neste algoritmo, foram tidas em conta as distâncias entre paragens através do seu cálculo com as respetivas localizações, que é representado pelo peso de cada edge. Complexidade do algoritmo : O(|E| log |V|);

II) Pesquisa em largura (BFS) – Para obter o caminho que passaria pelo menor número de paragens, foi usado uma bfs. Neste algoritmo, não foram tidas em conta as distâncias entre paragens, sendo apenas importante a contagem de paragens. Complexidade do algoritmo : O(|V| + |E|).

```
public:
    //Constructors
    Graph();
    Graph(int nodes, bool dir = false);

    //Gets
    Stop& getDest(Edge edge);
    Stop& getStop(int index);
    Stop& getStop(string code);
    vector<Stop>& getStops();
    vector<int> getPath(Stop& a, Stop& b);

    //Adds
    void addEdge(int src, int dest, double weight, string code);
    void addStop(Stop& stop);

    //Algorithms
    vector<int> dijkstra_distance (Stop& a, Stop& b);
    vector<int> dijkstra_zones (Stop& a, Stop& b);
    vector<int> bfs(Stop& origin, Stop& dest);

    //Resets
    void resetNodes(int dist);
```

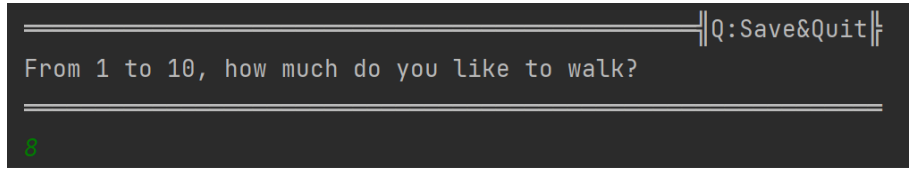
Destaque de funcionalidade

- **Distância máxima que quer andar em mudanças de linha :**

Como já foi referido, as edges criadas para percursos percorridos a pé têm pesos especiais, sendo estes multiplicador por um fator de acordo com a variável que representa o quanto o utilizador gosta de caminhar (quanto mais a pessoa gosta de andar menor irá ser).

Isto implica que para além das mudanças de linha estarem sujeitas a terem uma distância máxima, elas ainda podem variar em outras situações.

Tendo em conta o algoritmo de dijkstra, que tem em conta o peso das edges, a escolha do percurso com menor distância pode variar, visto que o peso das edges de uma pessoa que escolheu 1 é maior relativamente a uma pessoa que escolheu 9, havendo casos em que o valor da distância de uma ligação da rede real fique entre estes dois valores.



Criação das edges com peso especial :

```
graph.addEdge(i, j, weight: weight * (3 + 2*(10-(double) walkingFactor)/10), code: "walking");
```

Excerto de algoritmo dijkstra que mostra como o peso da edge é tido em conta :

```
double tempDist = stops[u].getDistance() + edge.getWeight();
if ((tempDist < getDest( edge: edge).getDistance()) && q.hasKey(edge.getDest())){
    stops[getDest( edge: edge).getIndex()].setDistance(tempDist);
    stops[getDest( edge: edge).getIndex()].setPred(u);
    stops[getDest( edge: edge).getIndex()].setPredLine( line: edge.getLineCode());
    q.decreaseKey( key: getDest( edge: edge).getIndex(),tempDist);
}
```

Principais dificuldades

Neste trabalho, a maior dificuldade foi a organização da lógica do sistema de busca do “melhor” percurso considerando que poderia haver mudanças de linha.

Divisão de tarefas

Adam Nogueira : Interface , Leitura e Escrita em ficheiros, documentação do código e junção das partes;
Ana Sofia Costa : Leitura e Escrita em ficheiros , documentação do código, algoritmos e apresentação;
Igor Castro : Leitura e Escrita em ficheiros, algoritmos e lógica do sistema.