



**MIEIC**  
**Conceção e Análise de Algoritmos**

# **Easy Pilot**

## **Sistema de Navegação**

***Turma 2MIEIC01***  
Bernardo Belchior-up201405381  
Maria João Mira Paulo-up201403820  
Pedro Costa-up201403291

# 1. Índice

1.	ÍNDICE.....	2
1.	INTRODUÇÃO .....	3
2.	METODOLOGIA.....	4
3.	DESCRIÇÃO DO PROBLEMA .....	5
3.1	Introdução de dados.....	5
3.2	Input .....	6
3.3	Output.....	6
3.4	Objetivo .....	6
4.	FORMALIZAÇÃO DO PROBLEMA.....	7
4.1	Input .....	7
4.2	Output.....	7
4.3	Objetivo .....	7
5.	SOLUÇÃO .....	8
5.1	Determinação do menor caminho entre dois vértices de um grafo. ....	8
5.1.1	Explicação do código implementado.....	8
5.2	Pesquisa em “Strings”.....	10
6.	ESQUEMA DO PROGRAMA.....	12
7.	DIAGRAMA DE CLASSES .....	14
8.	CASOS DE UTILIZAÇÃO .....	15
9.	PRINCIPAIS DIFICULDADES .....	16
10.	CONCLUSÃO .....	17

# 1. Introdução

A navegação GPS é uma tecnologia amplamente utilizada atualmente, equipando cada vez mais veículos, e disponível em diferentes *apps* para dispositivos móveis, como smartphones, tablets, e mesmo relógios de pulso. As funcionalidades básicas de um navegador geralmente incluem a detecção da posição atual, a partir da qual se escolhe um destino, para o qual se calcula um caminho. O navegador enumera a sequência de ações e detalha o itinerário a seguir, muitas vezes com recurso ao processamento de voz.

No âmbito da disciplina de Conceção e Análise de Algoritmos do curso Mestrado Integrado em Engenharia Informática e de Computação foi-nos proposto a realização de um Sistema de Navegação. O nosso programa permite assim que o utilizador insira os nomes das ruas relativas ao ponto de partida e aos pontos de destino, sendo calculado o melhor caminho entre estes pontos. A aplicação fornece assim todas as ruas por onde o utilizador deverá seguir. Além disso o utilizador deve inserir o nome do concelho a que pertence a rua onde quer chegar, sendo avisado se essa rua não pertencer ao concelho dado. Quando o utilizador insere a rua de partida e rua de destino, o programa pode retornar, caso encontre, a rua com o mesmo nome exato, caso contrário retorna a rua com o nome mais idêntico.

Neste relatório iremos aprofundar o problema dado, explicando os métodos usados para a resolução do mesmo.

## 2. Metodologia

O primeiro procedimento foi a interpretação dos três ficheiros de texto obtidos através do programa OSM2TXT Parser e do OpenStreetMaps. Toda a informação dos ficheiros de texto foi guardada em estruturas de dados adequadas para mais tarde ser interpretadas e armazenada em Grafos.

O segundo procedimento foi a realização de um algoritmo que determine o caminho mais curto entre dois pontos.

O terceiro foi a implementação de uma API de visualização de Grafos para mais fácil e intuitiva interpretação.

O último passo foi a implementação de um algoritmo que compare strings, verificando o quão parecidas estas são.

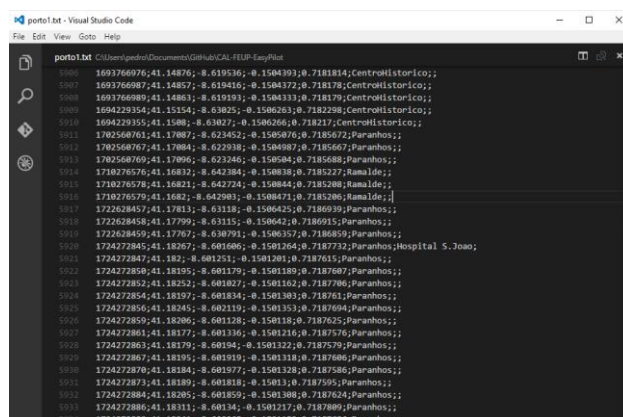
# 3. Descrição do Problema

De uma forma geral, pretendemos que a nossa aplicação, recebendo a rua de Origem, de Destino e pontos de passagem, consiga devolver ao utilizador o itinerário mais curto.

## 3.1 Introdução de dados

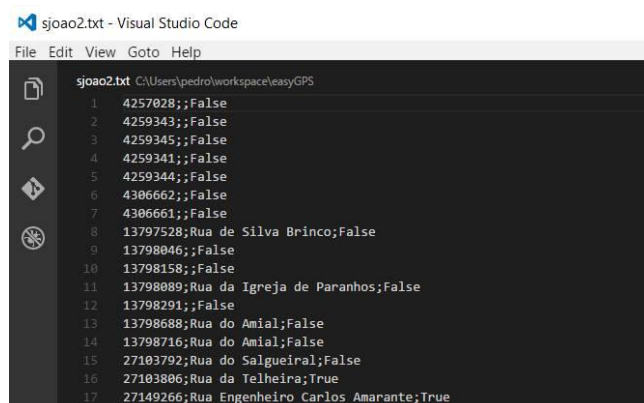
Para a implementação do Sistema de Navegação recorreremos à utilização de mapas reais extraídos do OpenStreetMaps ([www.openstreetmap.org](http://www.openstreetmap.org)) , os quais foram depois convertidos em três ficheiros de texto através do programa OSM2TXT Parser fornecido na página da disciplina. No entanto, com o objetivo de adicionar pontos de interesse aos vértices, alteramos o código do OSM2TXT Parser de forma a imprimir mais um parâmetro no primeiro ficheiro de texto.

O primeiro ficheiro de texto fornece todos os pontos do mapa: o seu *id* e respetiva latitude, longitude, latitude projetada em x, longitude projetada em y, concelho a que o ponto pertence e o nome do ponto de interesse, caso exista.



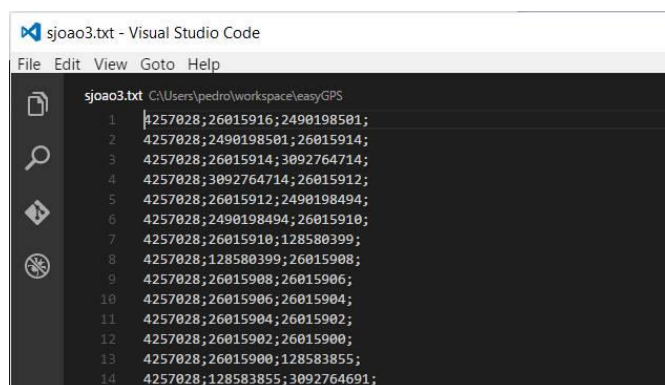
```
porto1.txt
1693766976;41.14876;-8.619536;-0.1504303;0.7181814;CentrolHistorico;;
1693766987;41.14857;-8.619416;-0.1504372;0.718179;CentrolHistorico;;
1693766989;41.14863;-8.619393;-0.1504333;0.718179;CentrolHistorico;;
1694229354;41.15154;-8.63825;-0.1506263;0.7182258;CentrolHistorico;;
1694229355;41.1508;-8.63827;-0.1506266;0.718217;CentrolHistorico;;
1702560761;41.17887;-8.623452;-0.1505876;0.7185672;Paranhos;;
1702560767;41.17884;-8.622938;-0.1504887;0.7185647;Paranhos;;
1702560769;41.17869;-8.623246;-0.150594;0.7185688;Paranhos;;
1710276576;41.16832;-8.642384;-0.150838;0.7185227;Ramalde;;
1710276578;41.16821;-8.642724;-0.150844;0.7185208;Ramalde;;
1710276579;41.1682;-8.642983;-0.1508471;0.7185206;Ramalde;;
1722628457;41.17833;-8.63118;-0.1506425;0.7186939;Paranhos;;
1722628458;41.17799;-8.63115;-0.150642;0.7186915;Paranhos;;
1722628459;41.17767;-8.638791;-0.1506357;0.7186859;Paranhos;;
1724272845;41.18267;-8.601686;-0.1501264;0.7187732;Paranhos;Hospital S. Joao;
1724272847;41.182;-8.601251;-0.1501281;0.7187615;Paranhos;;
1724272850;41.18195;-8.601179;-0.1501189;0.7187687;Paranhos;;
1724272852;41.18252;-8.601827;-0.1501162;0.7187786;Paranhos;;
1724272854;41.18197;-8.601834;-0.1501383;0.718761;Paranhos;;
1724272856;41.18245;-8.602119;-0.1501353;0.7187694;Paranhos;;
1724272859;41.18286;-8.601126;-0.1501189;0.7187625;Paranhos;;
1724272861;41.18177;-8.601136;-0.1501216;0.7187576;Paranhos;;
1724272863;41.18179;-8.60194;-0.1501322;0.7187579;Paranhos;;
1724272867;41.18195;-8.601919;-0.1501118;0.7187886;Paranhos;;
1724272870;41.18184;-8.601977;-0.1501328;0.7187586;Paranhos;;
1724272873;41.18189;-8.601818;-0.150113;0.7187595;Paranhos;;
1724272884;41.18205;-8.601859;-0.1501388;0.7187624;Paranhos;;
1724272886;41.18111;-8.60134;-0.1501217;0.7187889;Paranhos;;
1724272889;41.18231;-8.600905;-0.1501152;0.7187688;Paranhos;;
```

O segundo ficheiro de texto fornece o *id* de cada rua, o nome da rua respetiva, assim como um booleano que especifica se a rua é ou não de dois sentidos.



```
sjoao2.txt
1 4257028;;False
2 4259343;;False
3 4259345;;False
4 4259341;;False
5 4259344;;False
6 4306662;;False
7 4306661;;False
8 13797528;Rua de Silva Brinco;False
9 13798046;;False
10 13798158;;False
11 13798889;Rua da Igreja de Paranhos;False
12 13798291;;False
13 13798688;Rua do Amial;False
14 13798716;Rua do Amial;False
15 27103792;Rua do Salgueiral;False
16 27103806;Rua da Telheira;True
17 27149266;Rua Engenheiro Carlos Amarante;True
```

O terceiro ficheiro de texto fornece o *id* de uma rua e os *ids* de dois pontos consecutivos desta rua.



```
sjoao3.txt - Visual Studio Code
File Edit View Goto Help

sjoao3.txt C:\Users\pedro\workspace\easyGPS
1 4257028;26015916;2490198501;
2 4257028;2490198501;26015914;
3 4257028;26015914;3092764714;
4 4257028;3092764714;26015912;
5 4257028;26015912;2490198494;
6 4257028;2490198494;26015910;
7 4257028;26015910;128580399;
8 4257028;128580399;26015908;
9 4257028;26015908;26015906;
10 4257028;26015906;26015904;
11 4257028;26015904;26015902;
12 4257028;26015902;26015900;
13 4257028;26015900;128583855;
14 4257028;128583855;3092764691;
```

### 3.2 Input

Construção de um grafo,  $G = (V, E)$ , de Pontos e Ruas no qual:

- G - Grafo - representa um mapa de uma determinada zona;
- V – Vértices – representam todas os pontos do mapa, com uma certa latitude e longitude;
- E – Arestas – representam todas as ruas, contendo informação como o nome e características desta;
- Vértice de início de viagem;
- Vértices de pontos a percorrer, caso existam;
- Vértice de destino de viagem.

O utilizador pode, ao escolher o seu trajeto optar por adicionar pontos de interesse ao invés do nome da rua, como por exemplo bomba de gasolina, Faculdade de Engenharia ou o Hospital de São João.

### 3.3 Output

O percurso por onde o utilizador deve passar por forma a otimizar o seu trajeto, a distância entre cada ponto e o valor final da distância da viagem.

### 3.4 Objetivo

Facilitar os utilizadores na escolha do melhor trajeto.

## 4. Formalização do Problema

Formalizamos agora o problema, de acordo com aquela que achamos ser a melhor forma para resolver aquilo a que nos propusemos.

### 4.1 *Input*

$G < V, E >$ ,

$V$ : Pontos das ruas.

$E$ : *ligações entre pontos* (distância entre estes);

$P_0$ : *ponto inicial*

$P_i, i=1\dots n$

$P_f$ : *ponto final*

### 4.2 *Output*

$Caminho = \{ V_i \}, i = 1 \dots n$

$Valor$

### 4.3 *Objetivo*

$Min(valor)$ :

$$valor = f(x) = \sum_{i=1}^n (E_{ij})$$

$ij \in \underline{Caminho}$

## 5. Solução

### 5.1 Determinação do menor caminho entre dois vértices de um grafo.

A melhor solução encontrada para o problema é o algoritmo de *Dijkstra*, capaz de retornar o menor caminho entre dois vértices de um grafo.

Este algoritmo é um algoritmo ganancioso porque toma sempre as decisões que parecem ótimas no momento, determinando assim o conjunto de melhores caminhos intermediários. O valor de cada aresta está associado à distância das ruas ou à distância entre dois pontos de uma rua, calculada através da latitude e da longitude. Este algoritmo não garante a exatidão dos resultados caso o peso das arestas possa ser negativo, mas, dado que, neste caso, a distância entre dois pontos da rua nunca pode ser negativa, este algoritmo aplica-se bem ao nosso problema.

A complexidade temporal do algoritmo de Dijkstra para grafos conectados possui um tempo computacional de  $O([n^{\circ} \text{ arestas}] \log (n^{\circ} \text{ vértices}))$ .

#### 5.1.1 Explicação do código implementado

A nossa solução baseia-se no algoritmo de *Dijkstra* (função *computePaths*) para calcular as distâncias do ponto inicial a todos os vértices seguido de uma função, *getShortestPath*, que retorna os vértices a percorrer.

A função ***computePaths*** segue os seguintes pontos:

1. Através da função ***resetPathfinding***, coloca, em todos os vértices, o atributo *previous* (\*Vertex) a nulo e coloca no atributo *minDistance* de todos os vértices o valor de infinito (dado que esta atribuição não é possível, o algoritmo coloca *minDistance* com o valor de DBL\_MAX);
2. Coloca no atributo *minDistance* do vértice Inicial (*source*) o valor de zero;
3. Coloca numa fila de prioridade o vértice inicial.
4. Entra num ciclo *while* que termina quando a fila de prioridade está vazia.
5. Guarda numa variável (*beingProcessed*) o primeiro vértice da fila e retira-o da mesma.
6. Entra num ciclo que processa todos os vértices adjacentes do vértice *beingProcessed*, colocando na fila de prioridade aqueles em que a soma da distancia ao ponto a ser processado e o ponto adjacente é menor que a *minDistance* desse ponto.
7. Assim que percorre todos os pontos adjacente do *beingProcessed*, retoma o processo em 4 com o próximo vértice da fila de prioridade.

```
void Graph::resetPathfinding() {  
    for(unsigned int i = 0; i < vertexSet.size(); i++) {  
        vertexSet[i].minDistance = DBL_MAX;  
        vertexSet[i].previous = NULL;  
    }  
}
```



```

void Graph::computePaths(Vertex* source) {
    resetPathfinding();

    source->minDistance = 0;

    priority_queue<Vertex*> toBeProcessed=priority_queue<Vertex*>();
    toBeProcessed.push(source);

    while(!toBeProcessed.empty()) {
        Vertex* beingProcessed = toBeProcessed.top();
        toBeProcessed.pop();

        for(unsigned int i=0; i<beingProcessed->adj.size(); i++) {
            Vertex* dest = beingProcessed->adj[i]->destination;
            double distanceToDest=beingProcessed->adj[i]
                ->distance + beingProcessed->minDistance;

            if(distanceToDest < dest->minDistance) {
                dest->minDistance = distanceToDest;
                dest->previous = beingProcessed;

                toBeProcessed.push(dest);
            }
        }
    }
}

```

A função **getShortestPath** começa no vértice final e vai adicionando sucessivamente o vértice anterior ao início da lista que contém o caminho percorrido até chegar ao vértice inicial.

De modo a tornar o código mais eficaz, é guardado o último vértice a partir do qual foram calculados os caminhos. Isto permite que dois caminhos que tenham o mesmo início não necessitem de executar a função **computePaths** duas vezes, diminuindo o tempo de computação em metade.

```

list<Vertex*> Graph::getShortestPath(Vertex* source, Vertex* goal) {
    if(lastComputedPath == NULL || source != lastComputedPath)
        computePaths(source);

    list<Vertex*> path = list<Vertex*>();
    Vertex* v = goal;

    if(goal->getDistance() == DBL_MAX)
        return path;

    while(v->previous != NULL) {
        path.push_front(v->previous);
        v = v->previous;
    }

    return path;
}

```

## 5.2 Pesquisa em “Strings”

### 5.2.1 Explicação do código implementado

```
void EasyPilot::computePrefix(const string &pattern, int prefix[]) {
    unsigned int length = pattern.length();
    int k = -1;

    prefix[0] = -1;

    for (unsigned int i = 1; i < length; i++) {
        while (k > -1 && tolower(pattern[k+1]) != tolower(pattern[i])) {
            k = prefix[k];
        }

        if (tolower(pattern[k+1]) == tolower(pattern[i]))
            k++;

        prefix[i] = k;
    }
}
```

A função **computePrefix** é utilizada no algoritmo de Knuth-Morris-Pratt de modo a aumentar a eficiência da pesquisa, quando comparada com o método de força bruta. Esta computação permite que o algoritmo avance mais do que um caráter por iteração, podendo diminuir o tempo de execução.

```
int EasyPilot::exactMatch(string text, string pattern) {
    int num = 0;
    int prefix[pattern.length()];

    computePrefix(pattern, prefix);

    int q = -1;
    for (unsigned int i = 0; i < text.length(); i++) {
        while (q > -1 && tolower(pattern[q+1]) != tolower(text[i]))
            q = prefix[q];

        if (tolower(pattern[q+1]) == tolower(text[i]))
            q++;

        if (q == pattern.length() - 1) {
            num++;
            q = prefix[q];
        }
    }

    return num;
}
```

Após analisar o prefixo, a função **exactMatch** é utilizada para verificar se a *string pattern* está contida na variável *text*. O programa percorre *text* e compara com *pattern* para decidir se a segunda efetivamente pertence à primeira.

```

unsigned int EasyPilot::editDistance(const string &pattern, const string &text) {
    unsigned int distance[text.length()+1];
    unsigned int oldDistance, newDistance;

    for (unsigned int i = 0; i <= text.length(); i++)
        distance[i] = i;

    for (unsigned int i = 1; i <= pattern.length(); i++) {
        oldDistance = distance[0];
        distance[0] = i;

        for (unsigned int j = 1; j <= text.length(); j++) {
            //Check is not case sensitive
            if (tolower(pattern[i-1]) == tolower(text[j-1]))
                newDistance = oldDistance;
            else {
                newDistance = min(oldDistance, min(distance[j], distance[j-1]));
                newDistance++;
            }

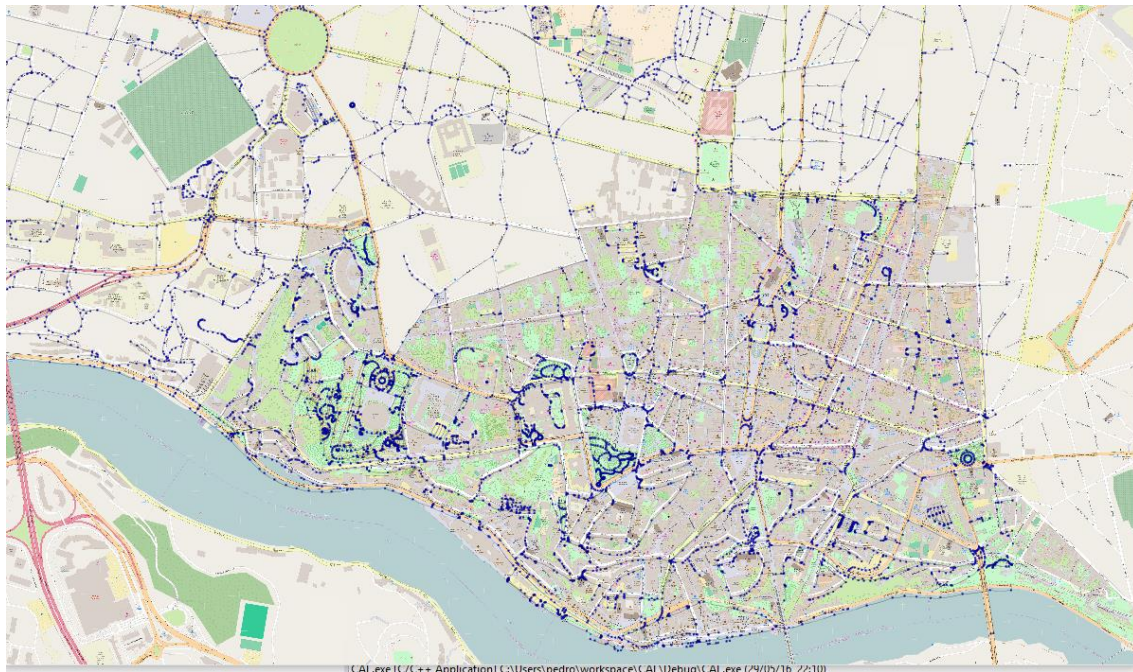
            oldDistance = distance[j];
            distance[j] = newDistance;
        }

        return distance[text.length()];
    }
}

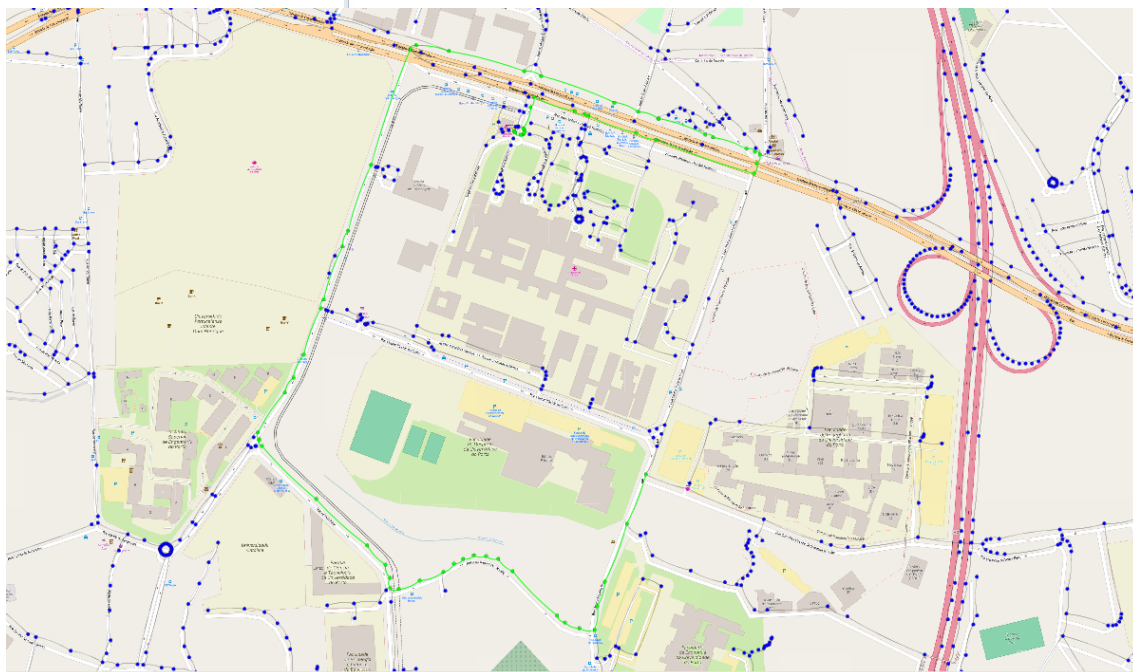
```

A função **editDistance** é utilizada com o intuito de decidir a distância de edição entre duas *strings*. Esta é calculada a partir das operações necessárias para transformar *pattern* em *text*. As operações primitivas são substituição, inserção ou eliminação de um caracter e têm um custo de 1. A distância de edição é o menor número de operações necessário para que *pattern* se transforme em *text*.

## 6. Esquema do Programa



```
CAL.exe [C++ Application] C:\Users\pedro\workspace\CAL\Debug\CAL.exe (29/05/16, 22:10)
Reading files...
Finished reading files ...
Populating graph viewer...
Adding nodes to graph viewer...
Finished adding nodes to graph viewer...
Adding edges to graph viewer...
Finished adding edges to graph viewer...
Finished populating graph viewer...
Please introduce the starting point:
```



```
CAL.exe [C++ Application] C:\Users\pedro\workspace\CAL\Debug\CAL.exe (29/05/16, 22:05)
Finished populating graph viewer...
Please introduce the starting point: Hospital S.Joao
Road name: Hospital S.Joao      Edit distance: 1
What suburb does the road belong to?
Paranhos
Please introduce next point: FEUP
Found an exact match.
What suburb does the road belong to?
Paranhos

Distance between Hospital S.Joao and FEUP is 2.66082 kilometers.
Do you want to add another place? (Y/N) N

Total path distance is 2.66082 kilometers.
```

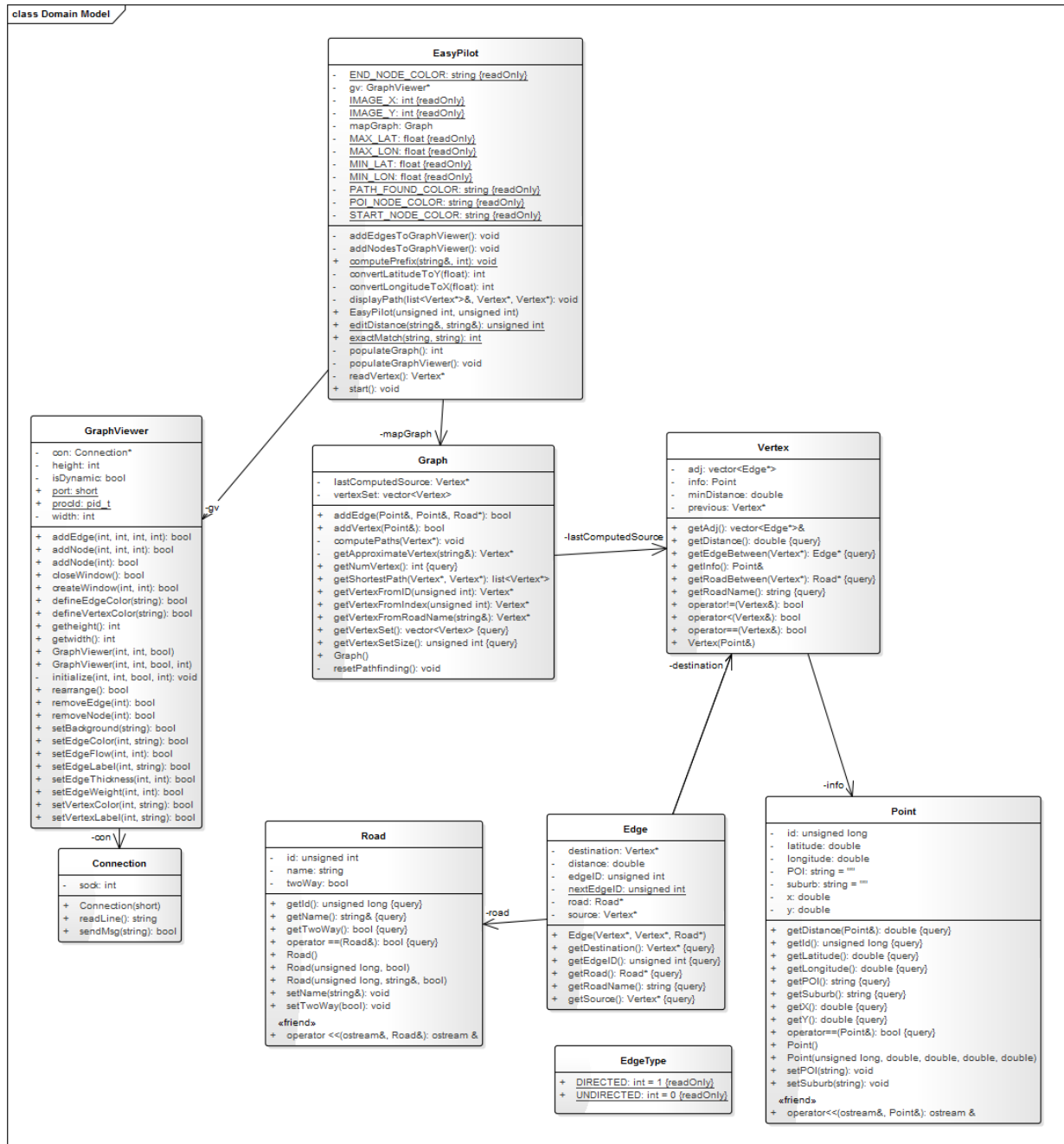
Ao iniciar o programa, começamos por introduzir todos os dados nos grafos, respetivos Vértices, pintados a azul (pontos do mapa) e Arestas (ruas). Além disso são carregados pontos de interesse, representados com a cor laranja e com a legenda respetiva.

Depois de carregados todos os dados, é a altura do utilizador escolher o Ponto de Partida e sucessivamente os Pontos por onde quer passar. Conforme o utilizador vai escolhendo os pontos por onde pretende passar, o menor caminho entre esses pontos vai sendo desenhado, através da representação gráfica (o ponto inicial encontra-se a amarelo e o Ponto de destino encontra-se a cor-de-rosa). Pela consola, o utilizador pode ter acesso à distância entre cada caminho escolhido e a distância final.

Sempre que é pedido ao utilizador o nome de uma rua, de partida, ponto de passagem ou destino, é pedido a seguir nome do concelho a que pertence essa rua. Caso a rua encontrada não pertença ao concelho especificado, o utilizador terá a oportunidade de verificar se a rua que inseriu é, efetivamente, a desejada. Caso não seja, será possível escolher outra.

Sempre que o utilizador insere o nome de uma rua, o programa pode encontrar uma rua com igual nome (pesquisa exata) ou, caso contrário, tenta encontrar a rua com o nome mais parecido ao dado (pesquisa aproximada), assim, se o utilizador se enganar e escrever “Hopital S.Joao” ao invés de “Hospital de S.João” o programa assume que a rua que o utilizador pretendia escrever era de facto “Hospital de S.João”.

# 7. Diagrama de Classes



## 8. Casos de Utilização

- Leitura e interpretação de dados de ficheiros relativos a um mapa;
- Escolha do melhor percurso em termos de distância percorrida;
- Visualização através do GraphViewer de toda o mapa;
- Visualização do melhor percurso através do GraphViewer.

## 9. Principais Dificuldades

Ao longo da realização do trabalho encontramos algumas dificuldades. Uma delas foi a análise dos ficheiros de texto fornecidos pelo OSM2TXT Parser. No entanto, aquela que se tornou mais difícil de contornar foi a implementação do Algoritmo de *Dijkstra*, de pesquisa em Grafos.

Outros desafios incluíram a apresentação dos pontos na posição certa, através do GraphViewer e a correta representação das arestas.



## 10. Conclusão

A realização deste trabalho permitiu nos obter uma melhor compreensão da matéria em questão, particularmente do modo de funcionamento de algoritmos de pesquisa em grafos, nomeadamente o Algoritmo de *Dijkstra*.

Estando a nossa faculdade localizada no Porto, optamos por seleccionar esta área como mapa do nosso projeto.

Todos os membros contribuíram empenhadamente de igual forma, sendo que trabalharam, maioritariamente e em conjunto, na biblioteca ou em salas de estudo.