



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	Explicação do código implementado.....	8
6.	ESQUEMA DO PROGRAMA.....	11
7.	DIAGRAMA DE CLASSES	12
8.	CASOS DE UTILIZAÇÃO	13
9.	PRINCIPAIS DIFICULDADES	14
10.	CONCLUSÃO	15

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 todos as ruas por onde o utilizador deverá seguir.

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 e último passo foi a implementação de uma API de visualização de Grafos para mais fácil e intuitiva interpretação.

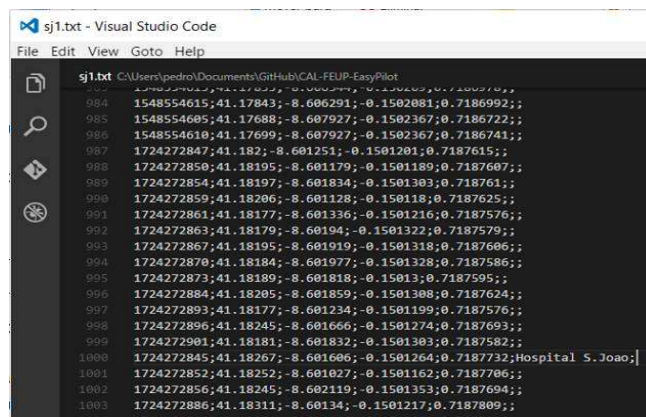
3. Descrição do Problema

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

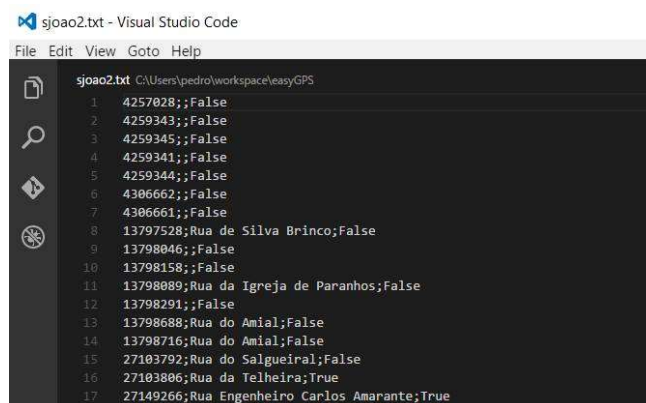
Para a implementação do Sistema de Navegação recorreremos à utilização de mapas reais extraídos do OpenStreetMaps (www.openstreetmap.org), os quais foram depois convertidos em três ficheiros de texto através do programa OSM2TXT Parser fornecido na página da disciplina. No entanto, com o objetivo de adicionar 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 e o nome do ponto de interesse, caso exista.



```
sj1.txt  C:\Users\pedro\Documents\GitHub\CAL-FEUP-EasyPilot
File Edit View Goto Help
984 1548554615;41.17843;-8.606291;-0.1502081;0.7186992;;
985 1548554605;41.17688;-8.607927;-0.1502367;0.7186722;;
986 1548554610;41.17699;-8.607927;-0.1502367;0.7186741;;
987 1724272847;41.182;-8.601251;-0.1501201;0.7187615;;
988 1724272850;41.18195;-8.601179;-0.1501189;0.7187607;;
989 1724272854;41.18197;-8.601834;-0.1501303;0.718761;;
990 1724272859;41.18206;-8.601128;-0.150118;0.7187625;;
991 1724272861;41.18177;-8.601336;-0.1501216;0.7187576;;
992 1724272863;41.18179;-8.60194;-0.1501322;0.7187579;;
993 1724272867;41.18195;-8.601919;-0.1501318;0.7187606;;
994 1724272870;41.18184;-8.601977;-0.1501328;0.7187586;;
995 1724272873;41.18189;-8.601818;-0.15013;0.7187595;;
996 1724272884;41.18205;-8.601859;-0.1501308;0.7187624;;
997 1724272893;41.18177;-8.601234;-0.1501199;0.7187576;;
998 1724272896;41.18245;-8.601666;-0.1501274;0.7187693;;
999 1724272901;41.18181;-8.601832;-0.1501303;0.7187582;;
1000 1724272845;41.18267;-8.601606;-0.1501264;0.7187732;Hospital S.Joao;
1001 1724272852;41.18252;-8.601027;-0.1501162;0.7187706;;
1002 1724272856;41.18245;-8.602119;-0.1501353;0.7187694;;
1003 1724272886;41.18311;-8.60134;-0.1501217;0.7187809;;
1004 1724272889;41.18241;-8.600955;-0.1501152;0.7187686;;
```

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  C:\Users\pedro\workspace\easyGPS
File Edit View Goto Help
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 13798089;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.

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

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 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;

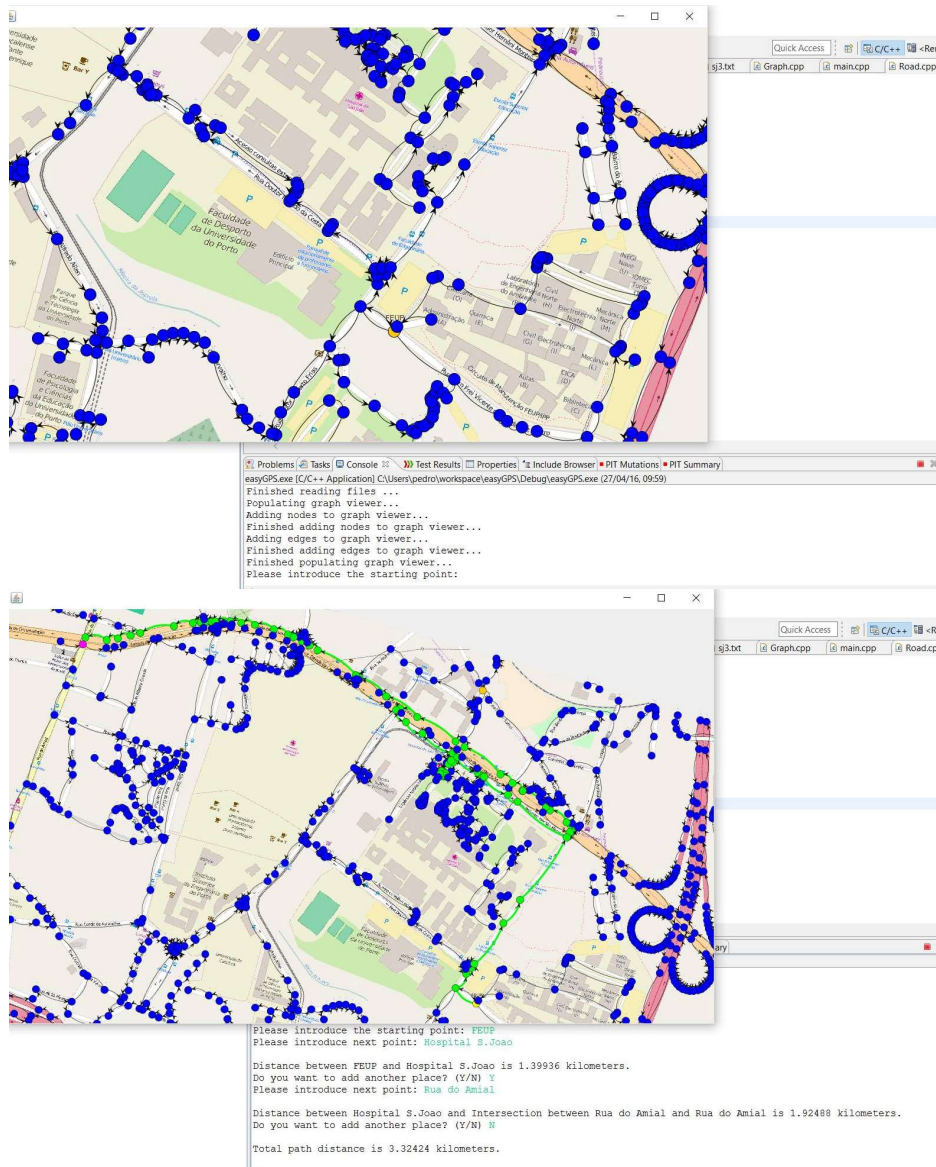
    if(goal->getDistance() == DBL_MAX)
        return path;

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

    return path;
}

```

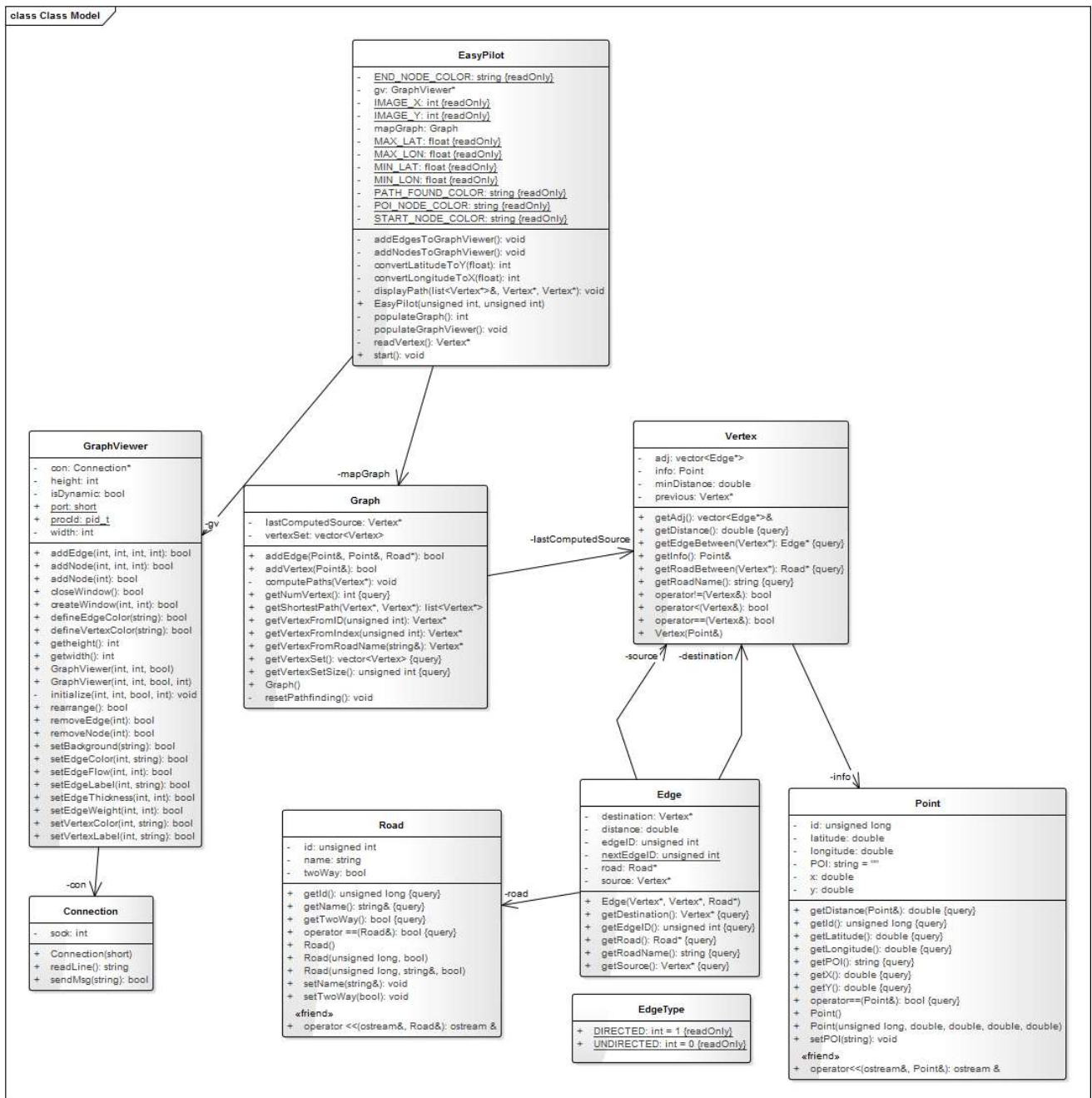
6. Esquema do Programa



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.

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 na zona de Paranhos, 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.