

**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 deteção 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. 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.

1. 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.

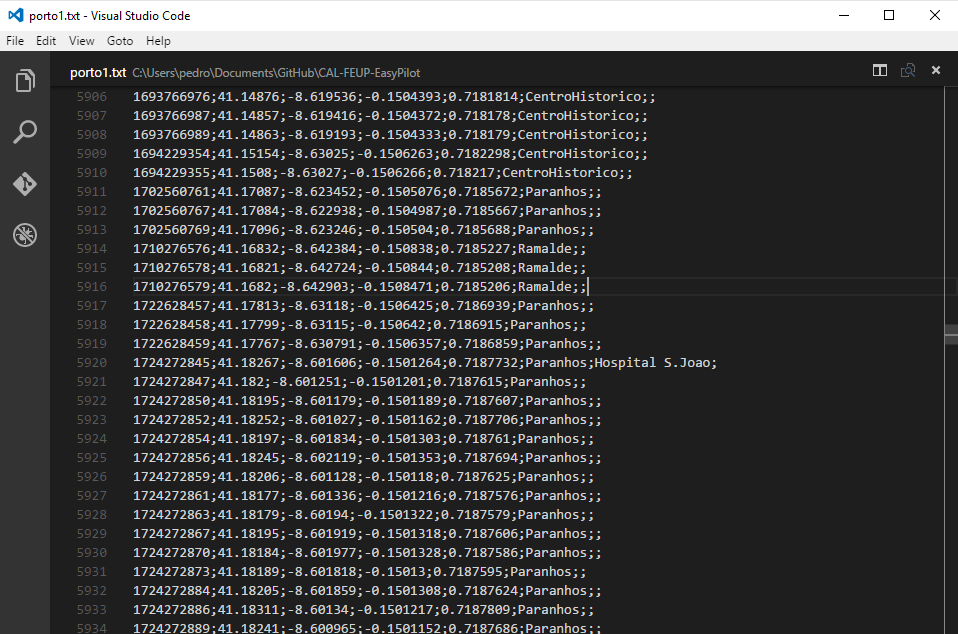
1. Descrição do Problema

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.

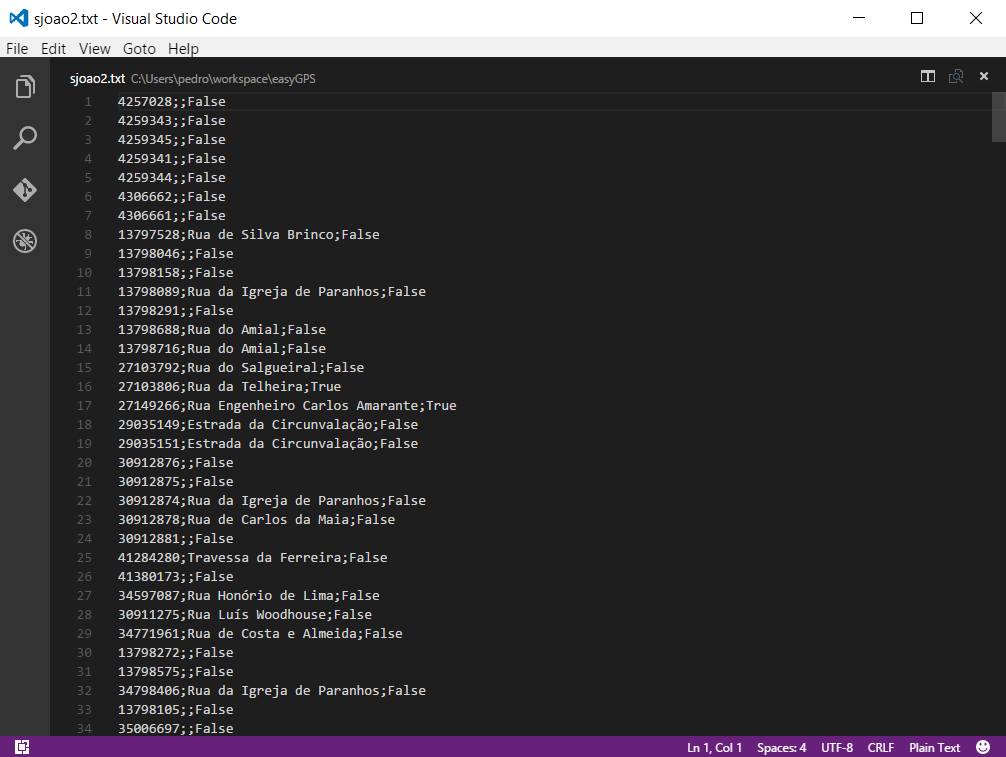
* 1. Introdução de dados

Para a implementação do Sistema de Navegação recorremos 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.

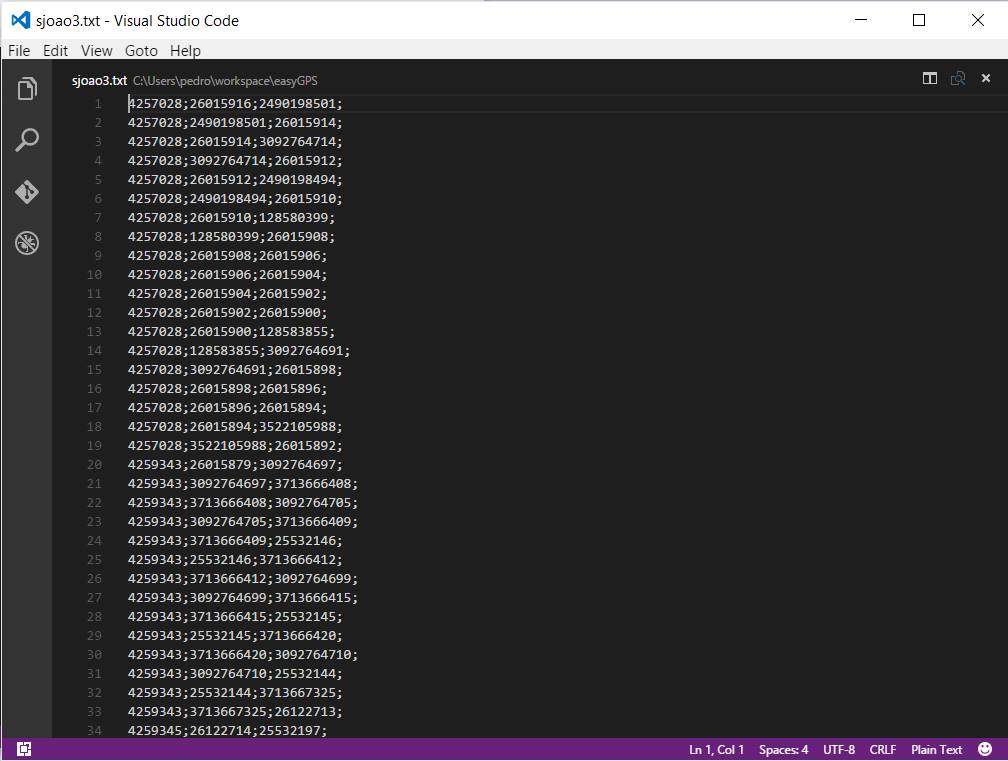
O primeiro ficheiro de texto fornece todos os pontos do mapa: o seu *id* e respetiva latitude, longitude, latitude projetada em x, longitude projetada em y, concelho a que o ponto pertence e o nome do ponto de interesse, caso exista.



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.



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



* 1. 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.

* 1. *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.

* 1. Objetivo

Facilitar os utilizadores na escolha do melhor trajeto.

1. Formalização do Problema

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

* 1. *Input*

𝐺 < 𝑉, 𝐸 >,

𝑉: Pontos das ruas.

𝐸: 𝑙𝑖𝑔𝑎çõ𝑒𝑠 𝑒𝑛𝑡𝑟𝑒 𝑝𝑜𝑛𝑡𝑜𝑠 (distância entre estes);

P0: 𝑝𝑜𝑛𝑡𝑜 𝑖𝑛𝑖𝑐𝑖𝑎𝑙

Pi , I=1...n

Pf : 𝑝𝑜𝑛𝑡𝑜 𝑓𝑖𝑛𝑎l

* 1. *Output*

𝐶𝑎𝑚𝑖𝑛h𝑜 = { 𝑉i }, 𝐼 = 1 … 𝑛

𝑉𝑎𝑙𝑜r

* 1. Objetivo

𝑀𝑖𝑛 (𝑣𝑎𝑙𝑜𝑟):

𝑣𝑎𝑙𝑜𝑟 =

𝑖𝑗 ∈ 𝐶𝑎𝑚𝑖𝑛h𝑜

1. Solução
   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 óptimas 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([nº arestas]log (nº vértices)).

* + 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.





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.

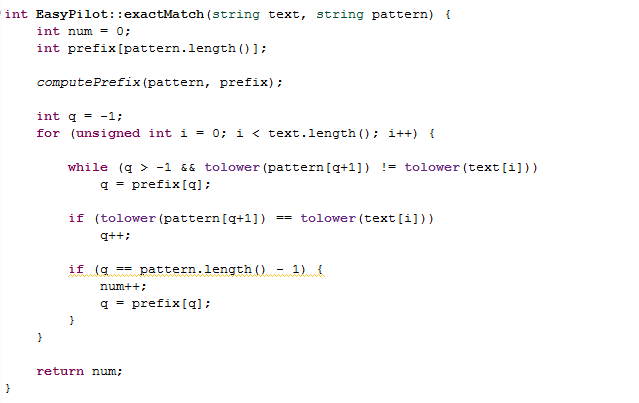
De modo a tornar o código mais eficaz, é guardado o último vértice a partir do qual foram calculados os caminhos. Isto permite que dois caminhos que tenham o mesmo início não necessitem de executar a função ***computePaths*** duas vezes, diminuindo o tempo de computação em metade.



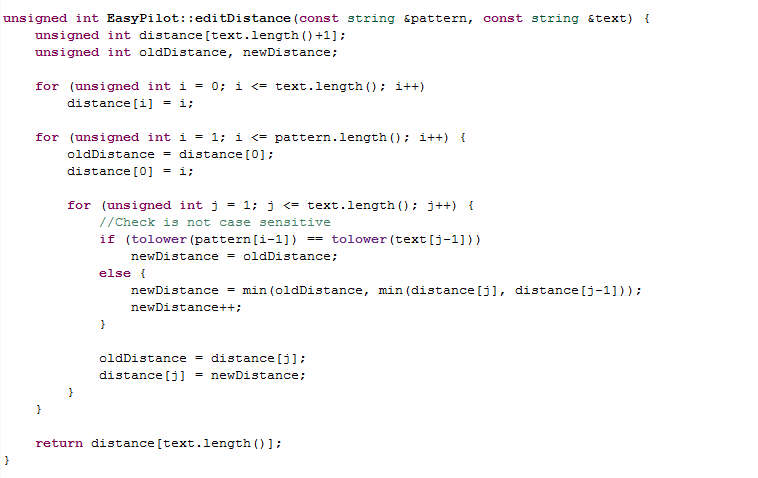
* 1. Pesquisa em “Strings”
     1. Explicação do código implementado



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.

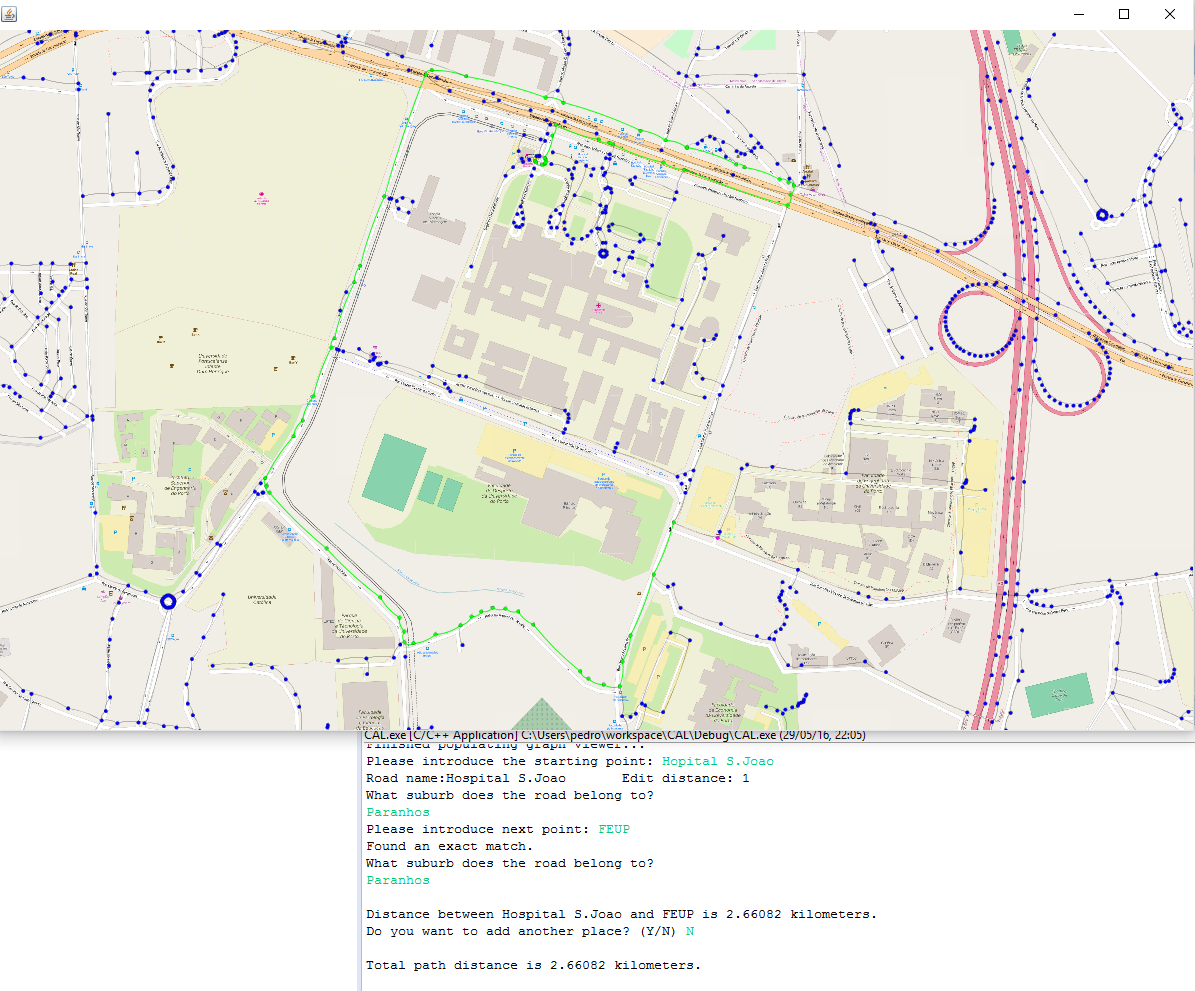
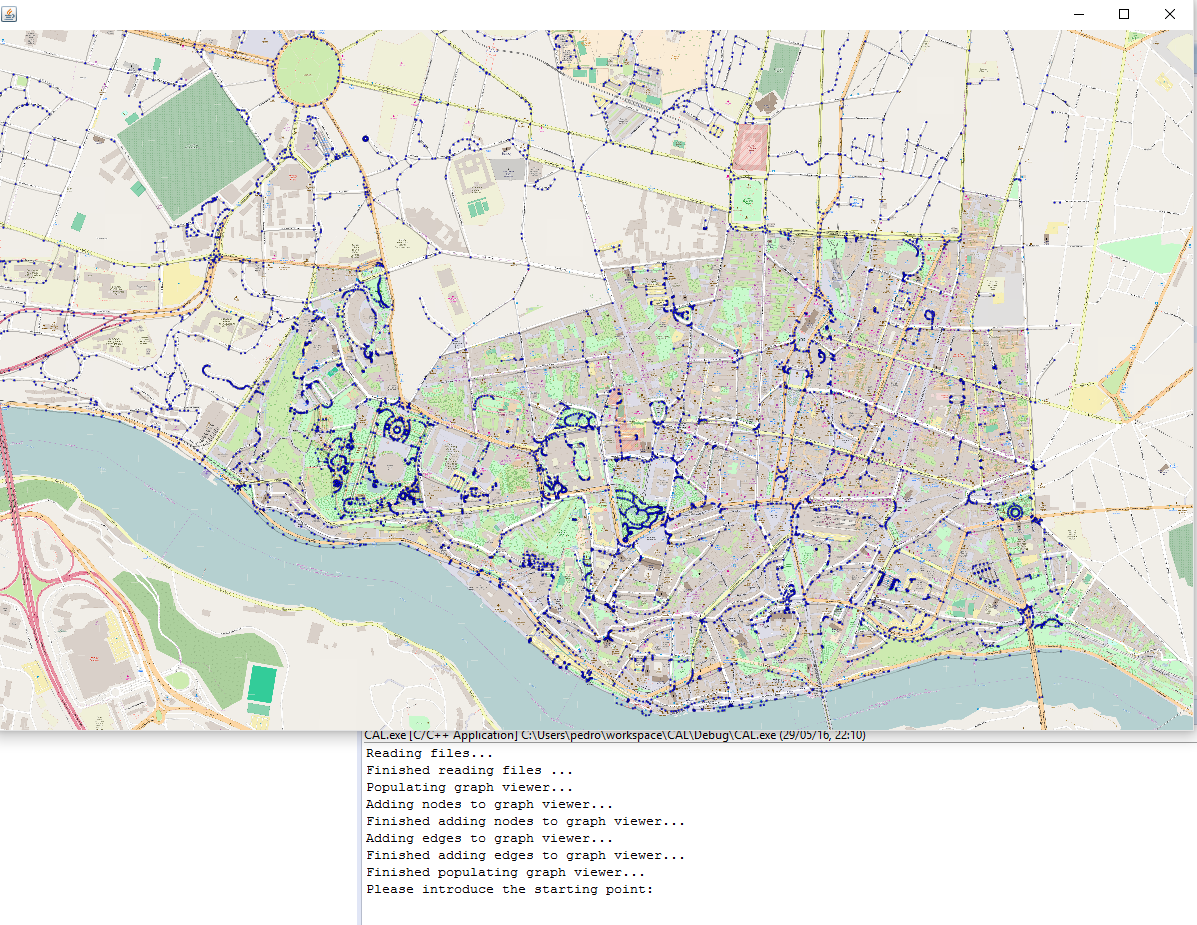


Após analisar o prefixo, a função **exactMatch** é utilizada para verificar se a *string pattern* está contida na variável *text*. O programa percorre *text* e compara com *pattern* para decidir se a segunda efetivamente pertence à primeira.



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*.

1. Esquema do Programa



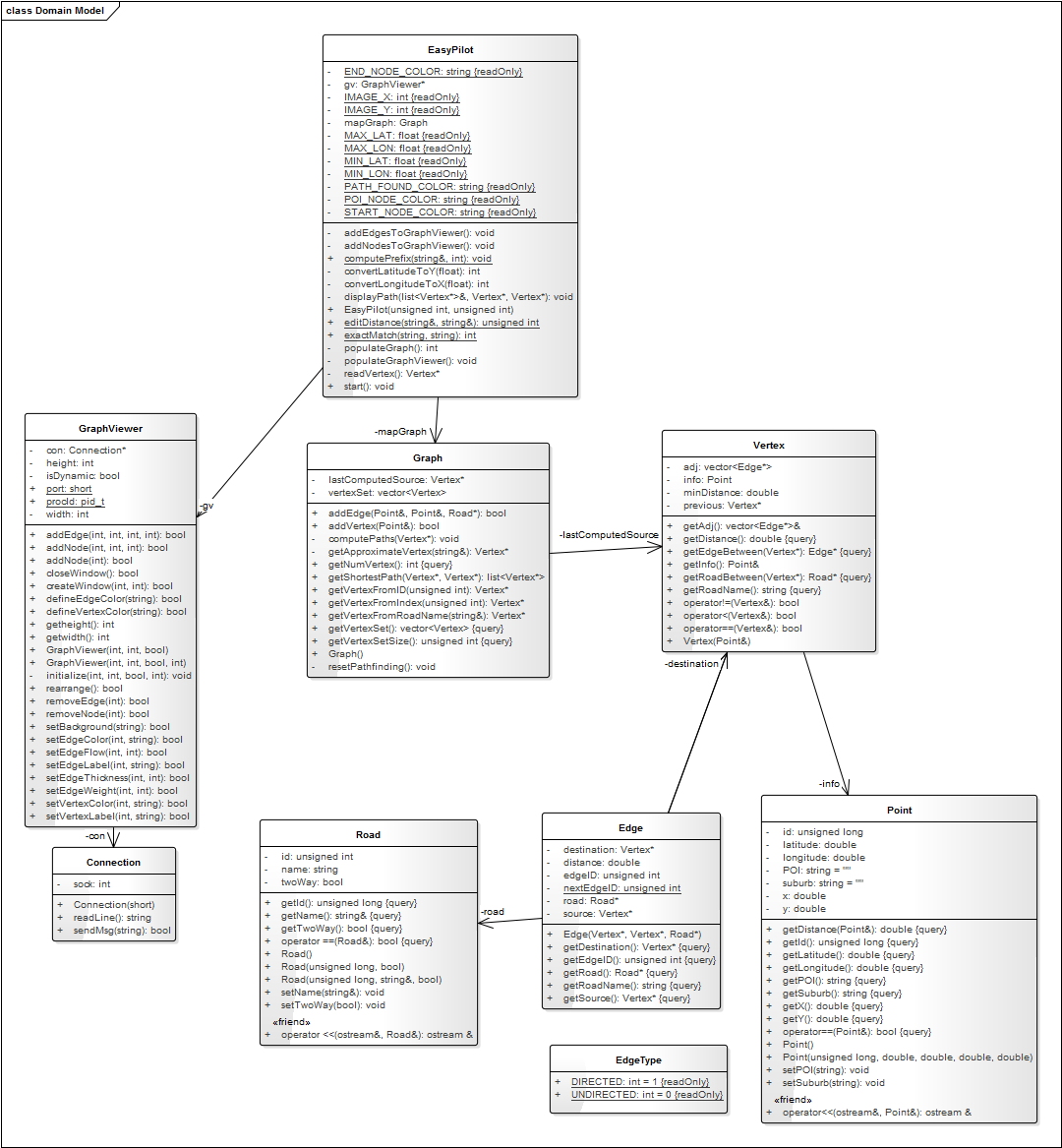
Ao iniciar o programa, começamos por introduzir todos os dados nos grafos, respetivos Vértices, pintados a azul (pontos do mapa) e Arestas (ruas). Além disso são carregados pontos de interesse, representados com a cor laranja e com a legenda respetiva.

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.

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”.

1. Diagrama de Classes



1. 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.

1. 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 D*ijkstra*, de pesquisa em Grafos.

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

1. 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 selecionar 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.