

Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algorítmos

**EasyPilot**

Sistema de Navegação

Grupo B, turma 3

Ângelo Miguel Tenreiro Teixeira, [up201606516@fe.up.pt](mailto:up201606516@fe.up.pt)

Henrique Melo Lima, [up201606525@fe.up.pt](mailto:up201606525@fe.up.pt)

Rui Pedro Moutinho Moreira Alves, [up201606746@fe.up.pt](mailto:up201606746@fe.up.pt)

30 de Março, 2018

Table of Contents

[Descrição do Problema 3](#__RefHeading___Toc112_1963847657)

[Iteração 1: Verificação da possibilidade de Navegar entre dois Locais 3](#__RefHeading___Toc117_1963847657)

[Iteração 2: Melhor percurso entre dois Locais, desprezando a existência de POIs no percurso 3](#__RefHeading___Toc120_1963847657)

[Iteração 3: Melhor percurso entre dois Locais, considerando POIs de vários tipos no percurso 4](#__RefHeading___Toc122_1963847657)

[Formalização do Problema 5](#__RefHeading___Toc140_1963847657)

[Dados de Entrada 5](#__RefHeading___Toc142_1963847657)

[Dados de Saída 5](#__RefHeading___Toc144_1963847657)

[Restrições 6](#__RefHeading___Toc146_1963847657)

[Função Objetivo 6](#__RefHeading___Toc148_1963847657)

[Estrutura de Classes do Programa 7](#__RefHeading___Toc181_1963847657)

[Representação de um Grafo 7](#__RefHeading___Toc183_1963847657)

[Algorítmos que operam sobre a estrutura Grafo 7](#__RefHeading___Toc185_1963847657)

[Classes Auxiliares 8](#__RefHeading___Toc187_1963847657)

[Solução Implementada 9](#__RefHeading___Toc196_1963847657)

[Casos de Utilização 10](#__RefHeading___Toc198_1963847657)

[Dificuldades encontradas no desenvolvilmento do Trabalho 11](#__RefHeading___Toc200_1963847657)

[Conclusões 12](#__RefHeading___Toc202_1963847657)

[Bibliografia e outras Fontes de Referência 13](#__RefHeading___Toc204_1963847657)

# Descrição do Problema

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.

Neste trabalho, pretende-se implementar um navegador que identifique o caminho a seguir, numa dada rede, a partir de uma origem até ao destino desejado. O itinerário poderá ser simples, ou ainda incluir vários pontos de interesse (POIs).

Este problema pode ser dividido em três iterações, enumeradas a seguir.

## Iteração 1: Verificação da possibilidade de Navegar entre dois Locais

Nesta primeira iteração o único objetivo é avaliar a possibilidade de, através de um ponto de partida, chegar a um ponto de destino, ou seja, não é importante nesta iteração guardar o caminho percorrido para chegar ao ponto de destino, nem a otimização do mesmo.

Nesta iteração não é também relevante a existência de pontos de interesse no percurso a realizar desde o ponto de partida até ao ponto de destino.

Certas vias podem não poder ser utilizadas no percurso, devido a estarem interrompidas por algum fator externo ou por terem caraterísticas indesejadas para o utilizador (a existência de, por exemplo, portagens).

## Iteração 2: Melhor percurso entre dois Locais, desprezando a existência de POIs no percurso

Num sistema de GPS é importante, para o utilizador, não só encontrar um percurso, mas também encontrar o melhor percurso (seja em termos de minimizar o consumo de combustível do veículo utilizado, minimizar o tempo de viagem, minimizar o custo total da viagem, ou ainda outros critérios. Mais à frente verificar-se-á que todos estes critérios são instâncias do mesmo problema).

Nesta iteração, apesar de não se considararem pontos de interesse no percurso, é importante otimizar o percurso com base na informação das estradas pelo que, ao contrário da iteração anterior, é essencial guardar o caminho percorrido para o mostrar ao utilizador.

## Iteração 3: Melhor percurso entre dois Locais, considerando POIs de vários tipos no percurso

Nesta última iteração o objetivo é não só encontrar o melhor caminho (de acordo com um dado critério) entre um ponto de origem e um ponto de destino, mas também passar por um conjunto de pontos de interesse indicado pelo utilizador.

Nesta Iteração torna-se essencial verificar a conectividade do Grafo, verificando todos os pontos acessíveis através do ponto de origem da viagem (algumas zonas podem-se tornar inacessíveis devido a fatores como, por exemplo, obras, delúvios, …).

É de salientar que, devido às restrições adicionadas pelos vários POIs, que devem obrigatoriamente estar presentes no percurso, torna-se muito mais difícil, em termos computacionais, chegar a uma solução ótima, como será explicado num capítulo posterior.

É também importante notar que a ordem pelos quais os POIs incluidos no percurso são visitados não é relevante para o problema em questão.

# Formalização do Problema

O problema descrito e divido em iterações no capítulo anterior é redutível a uma instância de um problema de grafos, como é demonstrado nos sub-capítulos seguintes.

## Dados de Entrada

* **P** – conjunto de pontos (localidades de um dado mapa). Cada ponto é caraterizado por:
  + Nome – nome da localidade
  + Coordenadas – Coordenadas da localidade no mapa em questão
  + Info – Informação relativa ao tipo de Localidade em questão
* **E** – conjunto das estradas que ligam as localidades
  + Peso – Custo ao percorrer a estrada (seja em termos de distância total, monetários, temporais, quantidade de combustível necessário, …)
* **Pi** ∈ **P** – ponto do mapa em que o utilizador se encontra (ponto inicial do percurso)
* **Pf** ∈ **P**  – ponto que se pretende alcançar (destino)
* **POIs** ⊆ **P**  – conjunto de todos os pontos indicados pelo utilizador que devem estar incluídos no percurso de **Pi** a **Pf**.
* **G(V,E)** – grafo dirigido cíclico pesado, em que os vértices **V** representam os vários pontos do mapa e as arestas **E** representam as estradas (que podem ter apenas um sentido ou ambos) que ligam os vários vértices.

## Dados de Saída

* **C –** conjunto de vértices (ordenado) que representam o melhor caminho entre **Pi** e **Pf**, passando por todos os vértices contidos no conjunto **POIs**, por uma qualquer ordem.
* **W** – peso total de todas as arestas percorridas no caminho (“custo” da viagem)

## Restrições

Os dados acima específicados, quer de entrada, quer de saída, apresentam o seguinte conjunto de restrições subjacentes:

**Restrições nos dados de entrada**

* ∀ **e** ∈ **E**, peso(e) >= 0, visto que o peso representa sempre grandezas positivas (ou nulas no de o peso respresentar, por exemplo, um custo de viagem). A existência de arestas com pesos negativos poderia, visto que o grafo de entrada pode conter cíclos, levar à ocorrência de cíclos com peso negativo, tornando-se o problema de minimização não resolúvel.

**Restrições nos dados de saída**

* **W** >= 0, consequência da restrição de entrada em que ∀ **e** ∈ **E**, peso(e) >= 0.
* **Pi** ∈ **C** ∧ **Pi** = **C**0 , o ponto de partida tem de ser o primeiro vértice no conjunto ordenado de vértice que representa o melhor percurso.
* **Pi** ∈ **C** ∧ **Pi** = **C**f , o ponto de destino tem de ser o último vértice no conjunto ordenado de vértice que representa o melhor percurso.
* ∀ **p** ∈ **POIs**,  **p** ∈ **C**, todos os POIs indicados devem estar contidos no percurso calculado.

## **Função Objetivo**

Como foi já indicado anteriormente, a solução ótima do problema é obtida minimizando o peso total das arestas percorridas para chegar do vértice **Pi** ao vértice **Pf**, passando por todos os vértices **p** ∈ **POIs**. Por este motivo, a solução ótima passa por minimizar a função **h** a seguir descrita:

**h** = ∑ peso(**e**) , **e** ∈ **C**

# Estrutura de Classes do Programa

## Representação de um Grafo

A representação de um grafo foi feita com base na estrutura **Graph** definida em *Graph.h*. Esta estrutura é composta pelo conjunto de vértices (representado por um *std::vector<Node>*, que permite acesso constante pelo número de identificação do vértice, visto que este número de identificação corresponde ao índice do vértice no vector de vértices) que compõe o grafo e possui métodos que permitem a sua manipulação, inserção, remoção, …

Os vértices do grafo são representados pela classe **Node**, que é composto pelo seu número de identificação (elemento que o torna único no grafo), pelo nome do local que representa, pelas suas coordenadas espaciais e pela informação extra subjacente à localidade que representa e um conjunto das arestas que ligam o vértice a outros vértices do grafo (representado por uso *std::unordered\_set<Edge>*, ou seja, uma tabela de dispersão que permite o acesso constante a qualquer aresta com origem no vértice).

As arestas do grafo são representadas pela classe **Edge**, que é composta unicamente pelo seu peso e pelo número de identificação do vértice em que tem destino.

## Algorítmos que operam sobre a estrutura Grafo

Foram desenvolvido um conjunto de classes que representam algorítmo que operam sobre grafos, seguindo um *design* de uma estrutura orientada a classes e objetos.

Desenvolveu-se a classe **GraphSearchAlgorithm**, classe puramente virtual que representa apenas um algorítmo genérico que executa uma pesquisa num grafo.

As classes **BFS** (Breadth-First Search) e **DFS** (Depth-First Search) extendem a classe **GraphSearchAlgorithm**. A classe **BFS** é responsável por realizar uma pesquisa em largura a partir de um vértice inicial, retornando a árvore de expansão em largura desse vértice, utilizando uma *std::queue* para auxiliar na ordem de pesquisa dos vértices. Por sua vez, a classe **DFS** é responsável por realizar uma pesquisa em profundidade a partir de um vértice inicial, retornando a árvore de expansão em profundidade desse vértice. Para aulixiar a ordem de pesquisa dos vértices poderia ser utilizada uma *std::stack*, mas adotou-se alternativamente por implementar uma solução recursiva.

A classe **Dijkstra** é responsável por calcular o caminho ótimo entre dois vértices do grafo, de acordo com a informação dos nós e vértices do grafo. A análise da complexidade temporal e espacial do algorítmo, bem como a apresentação do pseudo-código associado. Cada objeto da classe é composto por uma *std::unordered\_set<DNode>* para armazenar os nós já visitados (de forma a ter acesso em tempo constante aos mesmo) e um *std::set<DNode>* que funciona como uma fila de prioridade e auxilia na ordem de pesquisa dos vértices do grafo, estando estes ordenados no set (Árvore Binária de Pesquisa Vermelha-Preta) por ordem crescente de peso total do percurso atual.

A classe **DNode** utilizada pela classe **Dijkstra** consiste numa classe que extende **Node**, tendo como atributos extra o peso total do melhor caminho utilizado para chegar a esse vértice e o número identificador do vértice de onde provém nesse mesmo caminho.

A classe **A\*** (A-Star) extende a classe **Dijkstra** e é também responsável por calcular o caminho ótimo entre dois vértices do grafo, de acordo com a informação dos nós e vértices do grafo, utilizando uma heurística de decisão a fim de melhorar a ordem de pesquisa dos vértices e minimizar o tempo de execução do algorítmo. O funcionamento mais pormenorizado deste algorítmo será abordado num capítulo posterior. A classe **A\*** é em tudo igual à sua superclasse, à exceção de utilizar objetos da classe **ANode** em vez de objetos da classe **DNode**, que estão ordenado na “fila de prioridade” de forma diferente da superclasse.

A classe **ANode** utilizada pela classe **A\*** consiste numa classe que extende **DNode**, tendo como atributos extra a distância euclidiana ao nó de destino do percurso, atributo utilizado na heurística do algoritmo A\*.

## Classes Auxiliares

Foram também utilizadas algumas classes auxiliares para representar os tipos de exceções lançados pela classe **Grafo** e pelos vários algorítmos que nela operam, como a classe **NodeNotFound** (utilizada para indicar a não existência de um nó) e **InvalidNodeId** (utilizada para indicar que o identificador do nó em questão é inválido no âmbito do grafo). Ambas estas classes extendem a classe **Exception**, classe genérica responsável por representar uma exceção.

# **Solução Implementada**

# Casos de Utilização

# Dificuldades encontradas no desenvolvilmento do Trabalho

# Conclusões

O trabalho foi realizado de forma equalitária por todos os membros do grupo, sendo a maior parte do trabalho desenvolvido em reuniões prensenciais nas quais foram discutidas várias ideias, foram planificadas as várias etapas do trabalho e onde se discutiu eficiência e implementação dos vários algorítmos e heurísticas de optimização. Por este motivo, todos os membros tiveram aproximadamente igual contribuição em todas as etapas do trabalho, não havendo nenhuma distinção evidente entre nenhum dos membros.

# Bibliografia e outras Fontes de Referência

* Apresentações das Aulas Teóricas de Conceção e Análise de Algorítmos 2018, da autoria da Professora Doutora Liliana Ferreira, Professor Doutor João Pascoal Faria e Professor Doutor Rosaldo Rossetti.
* Dijkstra’s Algorithm, [https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm](https://en.wikipedia.org/wiki/Dijkstra's_algorithm)
* A\* Search Algorithm, [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
* Computerphile – Dijkstra’s Algorithm, <https://www.youtube.com/watch?v=GazC3A4OQTE>
* Computerphile – A\* (A Star) Search Algorithm, <https://www.youtube.com/watch?v=ySN5Wnu88nE>
* Travelling Salesman Problem, <https://en.wikipedia.org/wiki/Travelling_salesman_problem>