

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

[Introdução e 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)

[Iteração 3.1: Passar por um POI de um tipo especificado 4](#__RefHeading___Toc845_1771646129)

[Iteração 3.2: Passar por todos os POIs por uma qualquer ordem 4](#__RefHeading___Toc847_1771646129)

[Formalização do Problema 6](#__RefHeading___Toc140_1963847657)

[Dados de Entrada 6](#__RefHeading___Toc142_1963847657)

[Dados de Saída 6](#__RefHeading___Toc144_1963847657)

[Restrições 7](#__RefHeading___Toc146_1963847657)

[Restrições nos dados de entrada 7](#__RefHeading___Toc1126_2654923)

[Restrições nos dados de saída 7](#__RefHeading___Toc1128_2654923)

[Função Objetivo 7](#__RefHeading___Toc148_1963847657)

[Estrutura de Classes do Programa 8](#__RefHeading___Toc181_1963847657)

[Representação de um Grafo 8](#__RefHeading___Toc183_1963847657)

[Algorítmos que operam sobre a estrutura Grafo 8](#__RefHeading___Toc185_1963847657)

[User Interface 9](#__RefHeading___Toc1124_2654923)

[Classes Auxiliares 9](#__RefHeading___Toc187_1963847657)

[Solução Implementada 11](#__RefHeading___Toc565_614083082)

[Iteração 1: Verificação da possibilidade de Navegar entre dois Locais 11](#__RefHeading___Toc567_614083082)

[Iteração 2: Melhor percurso entre dois Locais, desprezando a existência de POIs no percurso 13](#__RefHeading___Toc586_330149545)

[Algorítmo Dijkstra 13](#__RefHeading___Toc697_1427114619)

[Algorítmo A\* 15](#__RefHeading___Toc699_1427114619)

[Iteração 3: Melhor percurso entre dois Locais, considerando POIs de vários tipos no percurso 18](#__RefHeading___Toc701_1427114619)

[Iteração 3.1: Passar por um POI de um tipo especificado 18](#__RefHeading___Toc955_1733836673)

[Iteração 3.2: Passar por todos os POIs por uma qualquer ordem 20](#__RefHeading___Toc957_1733836673)

[Dificuldades encontradas no desenvolvilmento do Trabalho 25](#__RefHeading___Toc572_614083082)

[Casos de Utilização 26](#__RefHeading___Toc198_1963847657)

[Conclusões 28](#__RefHeading___Toc202_1963847657)

[Glossário 29](#__RefHeading___Toc703_1427114619)

[Bibliografia e outras Fontes de Referência 30](#__RefHeading___Toc204_1963847657)

# **Introdução e** Descrição do Problema

A navegação GPS é uma tecnologia amplamente utilizada atualmente, equipando cada vez mais veículos e utilizada por 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).

Todos as terminologias sobre a teoria de grafos e sobre algorítmos de *path finding* encontram-se devidamente explicadas no capítulo **Glossário**.

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.

Esta iteração pode-se ainda sub-dividir nas duas seguintes iterações:

### Iteração 3.1: Passar por um POI de um tipo especificado

Nesta sub-iteração o objetivo passa por obter um caminho a começar num ponto inicial e a acabar num ponto de destino, passando por um ponto de interesse de um tipo em específico, de forma a minimizar o custo da viagem (Por exemplo, passar pelo posto de gasolina ou pela farmácia que exige o menor desvio do melhor percurso possível).

O cálculo da solução para este problema pode ser feito utilizando algorítmos utilizados nas iterações prévias, com algumas modificações e detalhes adicionais. Os detalhes relativos à solução para este problema serão abordados num capítulo posterior.

### Iteração 3.2: Passar por todos os POIs por uma qualquer ordem

Nesta sub-iteração o objetivo passa por obter um caminho a começar num ponto inicial e a acabar num ponto de destino, passando por um conjunto de pontos de interesse intermédios por uma qualquer ordem (Por exemplo, passar por um conjunto de monumentos históricos de forma a perder a menor quantidade de tempo em viagem).

O cálculo da solução para este problema não é trivial, como será demonstrado num capítulo posterior, onde serão também abordadas possíveis estratégias para resolver este problema.

É de salientar que o problema de passar por um conjunto de pontos de interesse por uma ordem especificada pelo utilizador se reduz a pequenos sub-problemas de encontrar o melhor caminho entre dois pontos, problema já abordado numa iteração anterior.

Conjugando as soluções de todas as iterações enumeradas previamente é possível responder a qualquer problema típico de *routing*, desde o problema mais simples de encontrar o melhor caminho entre dois pontos específicos até encontrar um caminho que passe por um conjunto de POIs de um tipo específico (ou não) da forma mais eficiente possível.

# 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.
* **Pf** ∈ **C** ∧ **Pf** = **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 vértices e arestas 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 vértices 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 vértices e arestas 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 vértice de destino do percurso, atributo utilizado na heurística do algoritmo A\*.

A classe **DijkstraBiDir** é responsável por calcular o caminho ótimo entre dois vértices, realizando uma expansão a partir do vértice de partida e, simultâneamente, uma expansão inversa a partir do vértice de destino, de forma a minimizar o trabalho total no cálculo do melhor percurso. Pode também ser especificado para a pesquisa um conjunto de pontos de interesse, de forma ao percurso incluir um desses pontos de interesse (aquele que implica um menor custo de viagem, isto é, que implica um menor desvio do percurso ótimo entre o vértice de partida e o vértice de destino).

A classe **TSPNearestNeighbor** é responsável por calcular o caminho entre um vértice de partida e um vértice de destino, passando por todos os pontos de interesse especificados para a pesquisa, utilizando uma heurística de cálculo denominada “vizinho mais próximo” (em inglês, *nearest-neighbor*). Cada objeto da classe é composto por um *std::vector* que especifica a ordem pela qual os POIs têm de ser percorridos (vetor calculado pelo método *calcPath()* da classe) e utiliza, para o cálculo do caminho entre os vários POIs, um objeto da classe **Astar.**

## **Graph Viewer**

Este módulo foi-nos fornecido por parte da equipa docente da cadeira e é responsável por tornar possível a visualização gráfica dos grafos manipulados pelo programa.

## **User Interface**

Apesar de não se tratar de uma classe, os ficheiros do módulo **UI** (UI.cpp e UI.h, onde está declarado o *namespace* **UI**) são responsáveis por gerir toda a interface com o utilizador (tratamento de erros de input, output do resultado dos algorítmos, seleção dos grafos sobre os quais operar, ...), implementando um sistema de menus hierárquico. É também neste módulo que se encontra o *entry-point* da aplicação.

## 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 vértice), **InvalidNodeId** (utilizada para indicar que o identificador do vértice em questão é inválido no âmbito do grafo) e **GraphLoadFailed** (utilizada para indicar a falha ao carregar um grafo para memória do programa). Ambas estas classes extendem a classe **Exception**, classe genérica responsável por representar uma exceção.

A classe **mapParser** é responsável por interpretar um modelo de 3 ficheiros provenientes do *Open Street Maps* e retornar um grafo com base na informação lida nos ficheiros.

# Solução Implementada

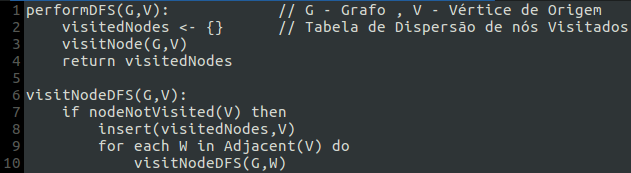
A descrição da solução implementada foi dividida em três partes, cada parte relativa a cada uma dasiterações expostas no capítulo de Descrição do Problema.

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

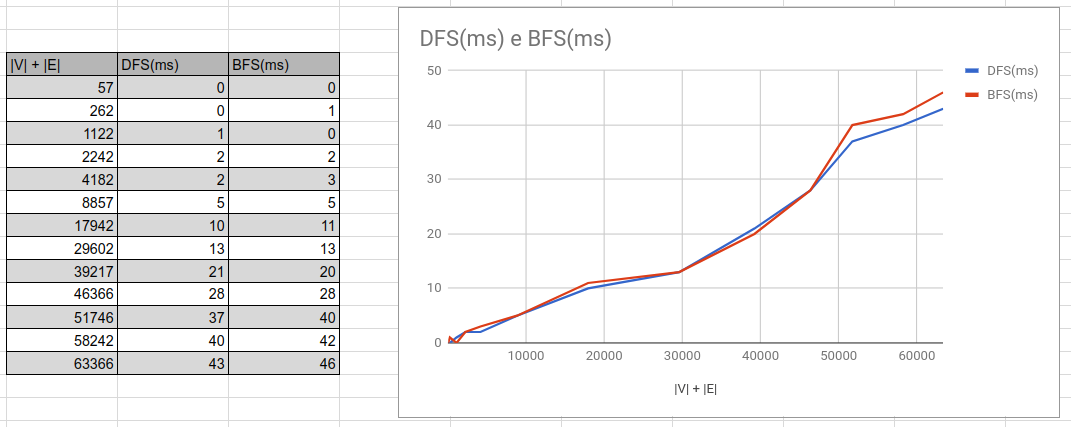
Analisar a possibilidade de chegar de um vértice de origem a um vértice de destino passa por uma simples pesquisa no grafo a partir do vértice de origem.

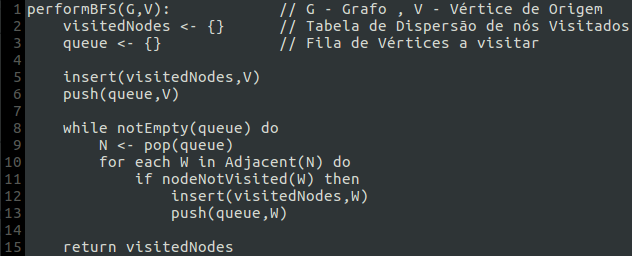
Esta análise pode ser, portanto, efetuada facilmente utilizando o algorítmo de **Pesquisa em Profundidade** ou o algorítmo de **Pesquisa em Largura**, começando no vértice de origem do percuso.

No algorítmo de Pesquisa em Profundidade todas as arestas são exploradas a partir do vértice mais recentemente descoberto. Esta forma de pesquisa apresenta uma estrutura recursiva a sí inerente, sendo que também pode ser implementada com recurso a uma pilha. Optamos, no entanto, de aplicar uma solução recursiva para fazer este tipo de pesquisa, como demonstrado no pseudo-código a seguir apresentado. Os vértices visitados são colocados numa tabela de dispersão após serem descobertos. Se o vértice de destino estiver contido na tabela de dispersão, então é possível navegar do vértice de origem para o vértice de destino.



A **complexidade temporal** deste algorítmo é **O(|V| + |E|)**, ou seja, linear no tamanho total do grafo (em que |V| representa o número de vértices do grafo e |E| o número de arestas). Cada vértice é visitado, no máximo, uma vez e a pesquisa é realizada a partir de cada vértice visitado para todos os seus vértices adjacentes a partir das arestas que os unem. A inserção e remoção na tabela de dispersão é de complexidade constante (O(1)), pelo estas operações não aumentam a complexidade temporal do algorítmo. Quanto ao espaço, por ser um algorítmo recursivo irá ter, no pior caso, |V| entradas na stack de chamada de funções (o caso em que o grafo degenera para uma lista simplesmente ligada). A **complexidade espacial** é, portanto, **O(|V|)**.

No algorítmo de Pesquisa em Largura, ao contrário do algoritmo previamente apresentado, são exploradas todas as arestas a partir do vértice em análise, passando só depois para o vértice seguinte. Esta forma de pesquisa é normalmente implementada com auxílio a uma fila, em que é analisado o vértice na frente da fila e em que todos os seus vértices adjacentes são colocados no fim da fila, e assim sucessivamente, como é evidenciado no pseudo-código a seguir apresentado:



A **complexidade temporal** deste algorítmo é, tal como no algorítmo de Pesquisa em Profundidade, **O(|V| + |E|)**, ou seja, linear no tamanho total do grafo. Cada vértice é, também, visitado no máximo apenas uma vez, sendo os vértices adjacentes ao mesmo visitados também no máximo uma vez através das arestas que os unem. A inserção e remoção na tabela de dispersão e na fila é de complexidade constante (O(1)), pelo estas operações não aumentam a complexidade temporal do algorítmo. Quanto ao espaço, no pior caso a fila que auxlia a implementação do algorítmo terá |V| elementos (o caso em que o vértice a partir do qual é realizada a pesquisa está ligado a todos os outros vértices do grafo). A **complexidade espacial** do algorítmo será, portanto, também **O(|V|)**.

Para corroborar a análise teórica da complexidade destes algorítmos, foram realizados testes experimentais com base na nossa implementação do algorítmo. Os grafos nos quais o algorítmo foram gerados aleatóriamente e são grafos aproximadamente esparsos. Os resultado obtidos podem ser observados no gráfico a seguir representado:

Concluiu-se também experimentalmente que a complexidade temporal de ambos os algorítmos é, portanto, aproximadamente linear. As amostras foram todas geradas aleatoriamente e de acordo com os as mesma condiçõe exteriores (isto é, no mesmo computador, com o número mínimo de processos a correr em “background).

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

O melhor percurso entre dois locais é um problema já muito estudado na teoria de grafos. Para solucionar este problema, recorremos à implementação do algorítmo **Dijkstra** e, posteriormente, à implementação de um “melhoramento” do mesmo, o algorímo **A\***, que tem por base o mesmo funcionamento que o algorítmo Dijkstra, mas que utiliza uma heurística de otimização que será explicada mais à frente.

### Algorítmo Dijkstra

O algorítmo **Dijkstra** original tem por base calcular o melhor caminho entre quaisquer dois vértices do grafo e foi introduzido pela primeira vez em 1956 por **Edsger W. Dijkstra**. No entanto, na nossa implementação, devido à natureza do nosso problema, fixamos o algorítmo a encontrar o melhor caminho apenas desde um vértice de origem até a um vértice de destino. Devido à sua natureza *greedy* e devido ao facto que garante sempre o melhor caminho, o algorítmo torna-se bastante eficiente e fácil de implementar.

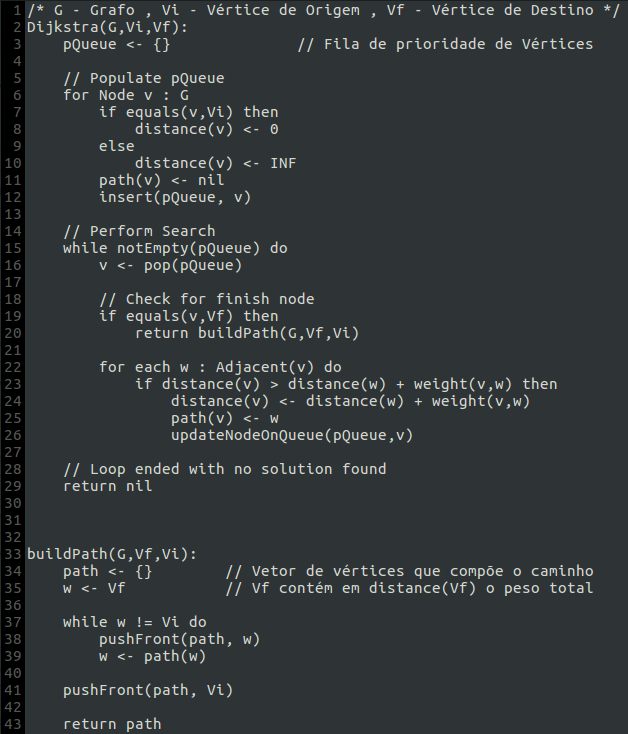
Neste algorítmo, os vértices possuem informação do vértice anterior no melhor caminho até ao próprio vértice, bem como o peso total das arestas do melhor caminho até ao próprio vértice. O algorítmo tem um comportamento semelhante ao de **Pesquisa em Largura** mas, ao invés de utilizar uma *fila* como contentor auxiliar para indicar a ordem dos vértices a pesquisar, utiliza uma ***fila de prioridade***, em que os vértices mais prioritários são aqueles que têm um menor peso total das arestas do melhor caminho (*greedy*).

Após encontrar o vértice de destino no topo da f*ila de prioridade,* o algorítmo está concluido e procede-se a uma reconstrução do caminho, acedendo ao vértice de onde o vértice de destino provém e assim sucessivamente até chegar ao vértice de origem.

É de salientar que estamos a utilizar uma *tabela de dispersão* como estrutura de dados auxiliar para colocar os vértices já analisados após estes saírem da *fila de prioridade,* para posterior reconstrução do caminho (a escolha desta estrutura foi devida à eficiência temporal das operações de inserção e remoção na tabela, O(1)).

O peso das arestas reflete o “custo” de viajar de um vértice até outro através dessa aresta, pelo que, por exemplo, problemas como estradas cortadas ou inacessíveis são abstraídos como eliminando essas arestas do grafo.

O pseudo-código do algorítmo encontra-se a seguir apresentado:



O algorítmo divide-se, portanto, em duas fases cruciais: encontrar o melhor percurso do vértice de origem ao vértice de destino e posterior reconstrução desse caminho através da informação contida nos vértices.

É também de salientar que o custo total do melhor percurso se encontra, no fim do algorítmo, no peso total do vértice de destino.

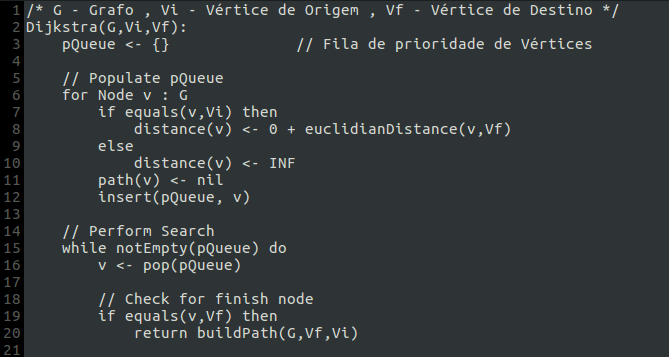
A primeira fase do algorítmo subdivide-se também em duas fases: a primeira sub-fase consiste em preparar os vértices para a execução do algorítmo, sendo esta fase resoluvel em tempo linear relativamente ao número de vértices do grafo, O(|V|). A segunda sub-fase consiste em efetuar a pesquisa propriamente dita. Como já foi mencionado anteriormente, esta fase é em tudo igual a uma Pesquisa em Largura, à exceção de utilizar como estrutura de dados auxiliar uma *fila de prioridade* ao invés de uma *fila.* Ao contrário da *fila*, em que a inserção e remoção de vértices era feita em tempo constante, numa *fila de prioridade* a inserção é feita em tempo logarítmico, O(log n), e a remoção em tempo linear. Por este motivo, a complexidade temporal desta fase do algorítmo é de **O((|V|+|E|)\*log |V|)**, devido às inserções na *fila de prioridade* exigirem re-ordenação dos vértice na estrutura.

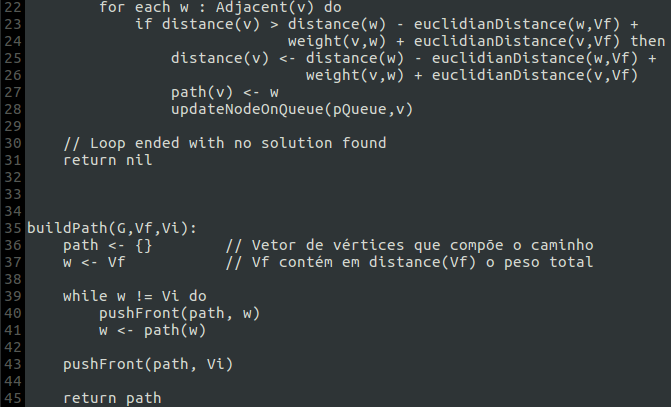
A segunda fase do algorítmo consiste em percorrer todos os vértices que constituem o caminho calculado, pelo que é resoluvel, no pior caso, em tempo linear relativamente ao número de vértices do grafo, **O(|V|).**

Por este motivo, a **complexidade temporal** do algorítmo é **O((|V|+|E|)\*log |V|)**.

### Algorítmo A\*

Após a implementação do algorítmo anteriormente descrito, procedeu-se à implementação de um “melhoramento”, o algorítmo **A\***, introduzido pela primeira vez em 1968 por **Peter Hart**, **Nils Nilsson** e **Bertram Raphael** do Instituto de Investigação de Stanford , que consegue alcançar um melhor desempenho devido a uma heurística de otimização para “guiar” na procura do vértice de destino, avançando no sentido de diminuir também a distância atual ao vértice de destino, como é evidenciado no pseudo-código a seguir apresentado:





O algorítmo é, portanto, idêntico em tudo ao algorítmo **Dijkstra**, à exceção da forma como ordena os vértices na fila de prioridade. Acrescenta ao peso total das arestas do melhor caminho até ao próprio vértice o valor da distância euclidiana do próprio vértice até ao vértice de destino. Desta forma, vértices que estão mais perto do vértice de destino têm prioridade face a vértices que estão mais distantes.

No entanto, o algorítmo **A\*** não garante sempre a solução ótima, visto que a função heurística utilizada é uma **função consistente**. Isto é, este algóritmo asegura a solução ótima se e só se:

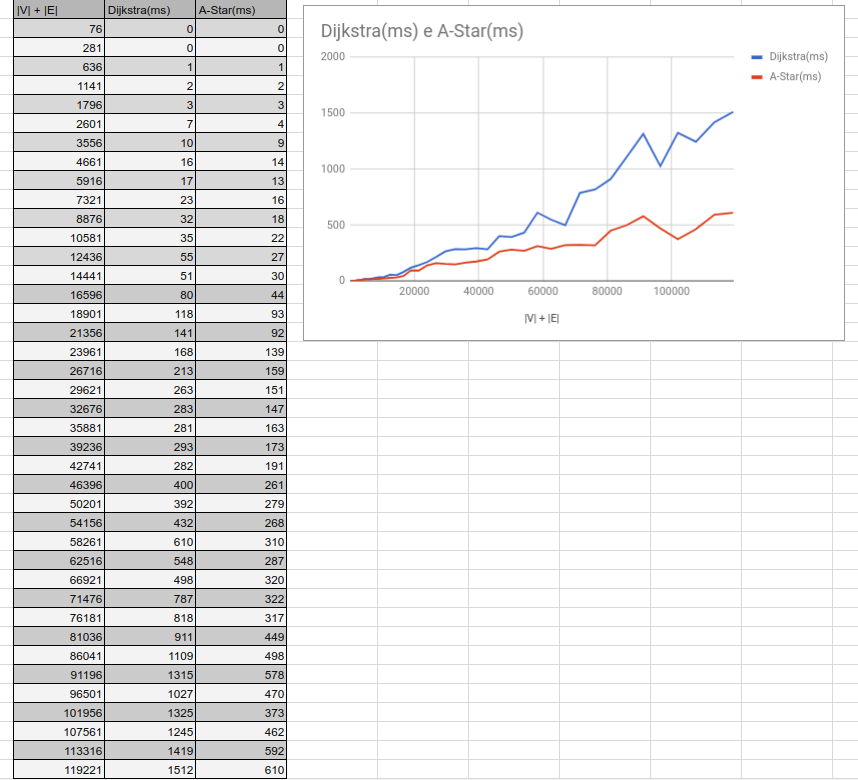
∀ **v** ∈ **G**, ∀ **n** ∈ adjacent(**v**), w(**v,n**) ≥ h(**v,n**)

em que

* **v, n** ∈ **G –** vértice genérico pertencente ao grafo
* **G** – grafo
* w(v1 , v2) - função de peso entre os vértices v1 e v2
* h(v1,v2) – função de heurística entre v1 e v2, neste caso distância euclidiana entre v1 e v2.

Como já foi mencionado previamente, por este algorítmo ser em tudo idêntico ao algorítmo **Dijkstra,** exceto na função cálculo do peso de um vértice, a sua complexidade será igual. Portanto, a **complexidade temporal** é **O((|V|+|E|)\*log |V|)** e a sua **complexidade espacial** é **O(|V|)**.

No entanto, apesar de os algorítmos serem iguais em termos de complexidade, devido à heurística de aproximação do algorítmo **A\***, espera-se que este alcance um melhor desempenho no caso médio do que o algorítmo **Dijkstra**. Para corroborar esta teoria, foram realizados testes de desempenho de ambos os algorítmos em grafos aproximadamente esparsos, analisando o seu tempo de execução:



Após a análise dos resultados obtidos nos quarenta testes, realizados em grafos de tamanhos progressivamente maiores, concluimos que o algorítmo **Dijkstra** obteve um comportamento aproximadamente superior a linear e que o algorítmo **A\*** teve um comportamento aproximadamente linear.

Apesar de o desempenho ter sido melhor do que o que era expectável, o algorítmo **A\*** teve sempre resultados que o algorítmo **Dijkstra,** devido à sua boa heurística de otimização.

Quanto à qualidade dos resultados face ao expectável, é provável que se deva ao facto de os caminhos dos vértices gerados aleatóriamente terem sido relavimente pequenos e devido ao grafo ser pouco denso (|E| [≅](https://pt.wiktionary.org/wiki/≅) |V|).

É de notar também que as flutuações temporais que se verificam perto de |V|+|E| = 100000 e em |V|+|E| se devem provavelmente a caminhos “mais fáceis” (ou seja, cujo vértice de origem se encontra pouco distante do vértice de destino).

É de salientar também, novamente, que o algorítmo **A\*** só apresentou resultados significativamente porque o valor das arestas segue a **função consistente** explicada previamente. Caso contrário, os algorítmos apresentariam resultados muito semelhantes para qualquer valor de |V|+|E|.

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

O problema do deslocamento entre dois pontos passando por um conjunto de pontos de interesse no decorrer do percruso é vulgarmente conhecido como o “Problema do Caixeiro Viajante” (ou, em inglês, como ***T****ravelling* ***S****alesman* ***P****roblem*). Este problema não tem uma resolução trivial e vão neste capítulo ser abordadas várias formas de o abordar.

### Iteração 3.1: Passar por um POI de um tipo especificado

Como já foi explicado num capítulo anterior, esta iteração para por solucionar o problema de encontrar o melhor caminho entre um vértico de partida até a um vértice de destino, passando por um qualquer POI de um tipo especificado, de forma a minimizar o desvio do percurso “principal” (Por exemplo, deslocar-se de um local para outro passando pelo posto de gasolina mais favorável ao percurso de forma a minimizar tempo perdido).

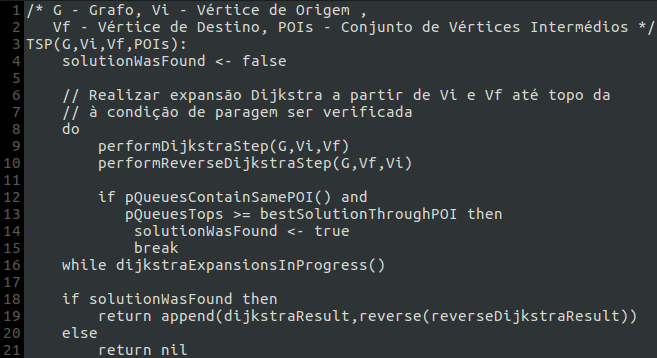
Apesar de ser tratar de um problema com pontos de interesse a visitar no percurso, este problema não se trata de uma instância do problema do caixeiro viajante.

Este problema é facilmente resolvido utilizando o algóritmo **Dijkstra**, utilizado já numa iteração anterior.

A solução passa por fazer uma pesquisa Dijkstra (análoga à pesquisa em largura, mas utilizando o critério *greedy* utilizado pelo algorítmo Dijkstra) a partir do vértice de partida **Vi** e uma outra pesquisa Dijkstra a partir do vértice de destino **Vf** percorrendo, neste caso, as arestas na direção inversa.

A condição de paragem é, resumidamente, quando ambas a pesquisa encontram o mesmo ponto de interesse **P**, sendo que o melhor caminho de **Vi** para **Vf** passando por um dos POIs do tipo específicado se trata do caminho **Vi** → **P** → **Vf**.

O algorítmo mais detalhado com as etapas do seu funcionamento encontra-se especifícado no pseudo-código a seguir apresentado:



É de salientar que a condição de paragem deste algorítmo não é trivial: Para assegurar a solução ótima do algorítmo é necessário que, mesmo após ambas as expansões encontrem um POI em comum, a expansão continue até que o peso total do vértice que se encontra no topo da fila de prioridade de cada uma das expansões seja superior ao peso do caminho **W** de **Vi** até **Vf** passando pelo ponto de interesse em comum **P**.

Isto acontece porque, até ao momento em que o topo da *fila prioridade* de ambas as expansões seja superior ao peso do caminho **W**, há sempre a possibilidade de encontrar um outro ponto de interesse **P2** tal que o caminho **Y** de **Vi** a **Vf** passando pelo ponto de interesse **P2**seja de peso total inferior ao caminho **W**.

Quanto à complexidade do algorítmo em questão, o algorítmo trata-se apenas da dupla utilização do algorítmo **Dijkstra**, acrescido de uma *tabela de dispersão* extra para armazenamento dos pontos de interesse potenciais a serem incluidos no percurso, de tamanho inferior ao número de vértices do grafo (|V|) e de complexidade temporal de inserção, pesquisa e remoção aproximadamente constante.

Por este motivo a **complexidade temporal** do algorítmo é, tal como anteriormente no algorítmo Dijkstra, **O((|V|+|E|)\*log |V|)** e a **complexidade temporal** é **O(|V|)**.

### Iteração 3.2: Passar por todos os POIs por uma qualquer ordem

Este problema trata-se de uma instância direta do problema do caixeiro viajante, com a restrição de haver um vértice de partida e um vértice de destino.

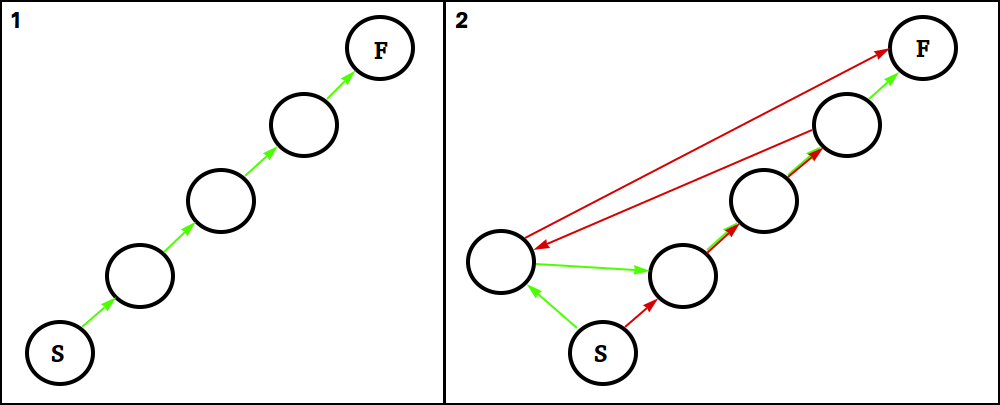
Este problema, no entanto, só tem solução em tempo fatorial: só é possível obter garantidamente a melhor solução com *brute-force*, ou seja, testando exaustivamente todas as possibilidades de caminhos. Por este motivo, a **complexidade temporal** do algorítmo é **O(n!)**, visto que é necessário calcular todos os caminhos possíveis entre todos os vértices e, posteriormente, escolher o melhor caminho de todos os calculados.

Existem, no entanto, diversas heurísticas para encontrar uma solução em tempo aproximadamente linear. No entanto, esta solução trata-se apenas de uma aproximação que depende da qualidade da heurística utilizada.

Uma heurística que resolve este problema em tempo **aproximadamente** linear é a heurística de Vizinho mais Próximo (em inglês *nearest neighbor*), que foi introduzida pela primeira vez por **J. G. Skellam**. Esta abordagem do problema é muito conhecida por ser de fácil implementação e por retornar uma resposta rapidamente. No entanto, a solução é muitas vezes de pouca qualidade face à solução ótima.

O algorítmo associado a esta heurística consiste em escolher sempre o vértice mais próximo ao vértice atual, se for possível completar o caminho a partir desse vértice.

Como mencionado previamente, esta solução pode estar muitas vezes distante da solução ótima. Os diagramas seguintes ilustram o desempenho deste algorítmo face à solução ótima:

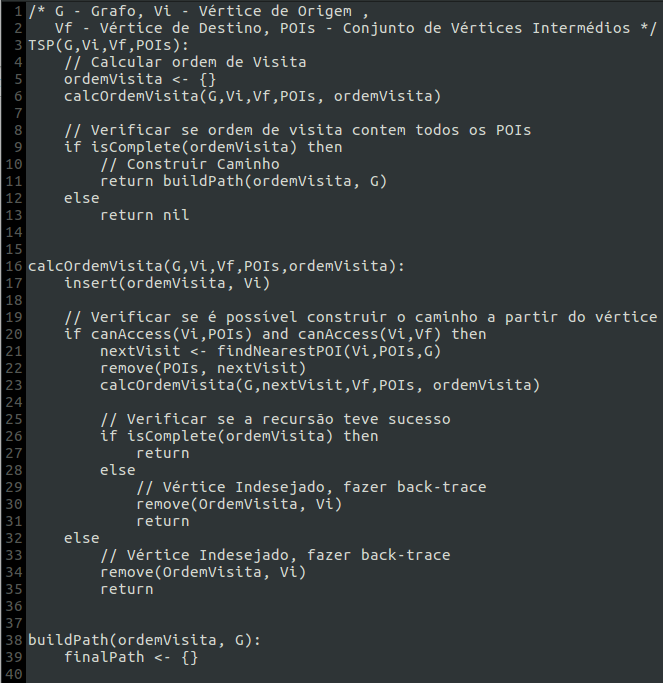


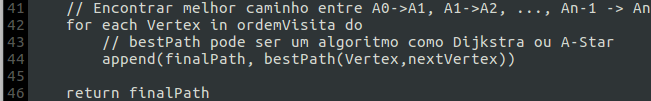
No Diagrama **1**, o algorítmo encontra a solução ótima em tempo linear. No entanto, ao adicionar um novo vértica ao diagrama (Diagrama **2**), o algorítmo não encontra a solução ótima (solução a verde), visto que escolhe sempre como próxima visita o vértice mais próximo (solução a vermelho).

No entanto, num grafo orientado, nem sempre este algorítmo leva a uma solução possível, visto que o vértice mais próximo pode não conseguir alcançar os outros vértices de interesse ou o vértice de destino (verificação feita obtendo a árvore de pesquisa .

Para contornar este problema, o algorítmo implementado é recursivo, diminuindo progressivamente a dimensão do problema, recorrendo a estratégias de *back tracking* quando o vértice atual não consegue alcançar os outros vértices.

O algorítmo mais detalhado com todas as etapas do seu funcionamento encontra-se especifícado no pseudo-código a seguir apresentado:





O algorítmo é composto, como é evidenciado no pseudo-código, por duas fases principais.

Na primeira fase é calculada, se possível, a ordem dos pontos de interesse a visitar, utilizando a heurística de “vizinho mais próximo”. Para o cálculo desta ordem de visita é utilizado um algorítmo **recursivo** que recorre a mecanismos de ***back-tracking*** para comunicar eventuais impossibilidades de percurso.

Na segunda fase é calculado o melhor percurso entre os vértices da estrutura *ordemVisita*, pela ordem em que eles se encontram nessa estrutura (foi utilizado na nossa implementação um vetor), utilizando um algorítmo de melhor caminho entre dois pontos, comoo algorítmo **Dijkstra** ou **A-Star** (foi utilizado na nossa implementação o algorítmo A-Star, face à sua melhor performance no caso médio relativamente ao algoritmo Dijkstra). Isto é, seja a estrutura *ordemVisita* composta pelo pontos **Vi**, **P1**, **P2**, …, **Pn**, **Vf**, o caminho calculado neste percurso é o caminho entre **Vf** e **P1**, seguido do caminho entre **P1** e **P2**, e assim sucessivamente até ao caminho entre **Pn** e **Vf** (em que o custo total do caminho é igual a soma do custo de todos os sub-caminhos).

Este algorítmo é, pela sua natureza, facilmente resolvido utilizando uma abordagem recursiva, pois o problema de partir do vértice **Vi**, passando pelos vértices **P1** e **P2** com destino ao vértice **Vf**, após a escolha como próximo vértice a visitar de, por exemplo, **P1**, é reduzido ao problema de partir do vértice **P1** passando pelo vértice **P2** com destino ao vértice **Vf**, e assim sucessivamente (recorrendo, como já foi mencionado previamente, a mecanismos de *back-tracking* sempre que se encontra um vértice que não consegue aceder a todos os outros pontos de interesse ou ao vértice de destino).

Quanto à complexidade espacial, este algorítmo utiliza como estrutura de dados auxiliares um vetor da ordem dos pontos interesse a visitar (de tamanho sempre inferior ou igual ao número de vértices do grafo, |V|) e alcança um nível de recursão na *pilha* de chamadas de função de tamanho igual ao tamanho do vetor de pontos de interesse. Por este motivo, a **complexidade temporal** deste algorítmo é linear relativamente ao número de vértices do grafo, **O(|V|)**.

Quanto à complexidade temporal do algorítmo, esta análise pode-se dividir em duas fases.

Na primeira fase do algorítmo, pode ser necessário fazer uma pesquisa em profunidade em todos os vértices do grafo (no caso em que todos os vértices são pontos de interesse). Por este motivo, a pesquisa pode ser realizada, no pior caso, |V| vezes. Visto que a complexidade temporal do algorítmo de pesquisa em profundidade é de O(|V|+|E|), a complexidade temporal desta fase do algorítmo é, portanto, no pior caso, de **O((|V|+|E|)\*|V|)**.

Quanto à segunda fase do algorítmo, no pior caso o vetor *ordemVisita* contém todos os vértices do grafo, sendo portanto necessário utilizar |V| - 1 vezes o algorítmo A-Star, cuja complexidade temporal é de O((|V|+|E|)\*log |V|) . Por este motivo, no pior caso, a complexidade temporal desta segunda fase é de **O((|V|+|E|)\*log |V|\*|V|)**. Como este termo da complexidade temporal da segunda fase do algorítmo é dominante, em termos de notação *Big-O,* relativamente ao termo da primeira fase, a **complexidade temporal** deste algorítmo é, no pior caso, de **O((|V|+|E|)\*log |V|\*|V|).**

É de notar que num grafo denso, em que todas os vértices estão conectado a todos os outros vértices por arestas bi-direcionais, a obtenção da ordem de pesquisa (primeira fase do algorítmo) é resolúvel em tempo linear no tamanho do grafo, O(|V|+|E|), visto que não é necessária a verificação da possibilidade de alcançar um outro vértice a partir do vértice atual (não sendo realizada qualquer pesquisa em profundidade). Por este motivo, **num contexto real** (por exemplo, num mapa de estradas), **este problema tem frequentemente complexidade linear** recorrendo a esta heurística.

Quanto à **qualidade** deste algorítmo, a solução obtida apresenta um peso aproximadamente 25% superior à solução ótima (Nilsson, Christian – p. 1), sendo um valor bastante bom face à simplicidade de cálculo comparativamente a uma solução exaustiva (*brute force*).

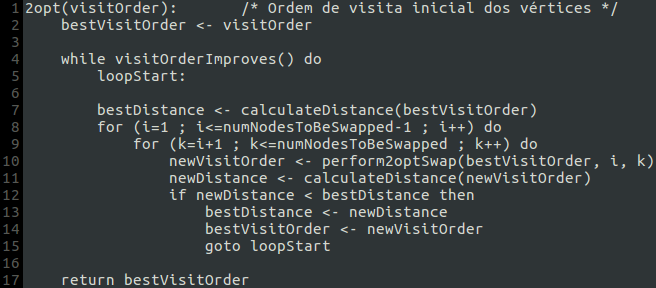
No entanto, a solução obtida com a heurística previamente explicada pode ser melhorada. Em 1958, **Croes** apresentou o algorítmo **2-Opt** a fim de resolver o problema do caixeiro viajante iterativamente.

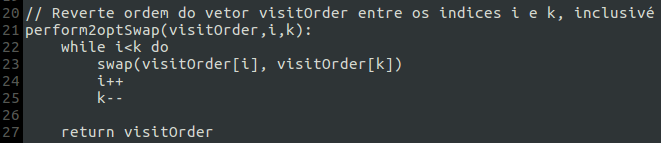
Este algorítmo, ao contrário de todos os algorítmos discutidos previamente, parte de uma solução inicial (seja esta solução aleatória ou, idealmente, calculada com um algorítmo leve e rápido como a heurística de vizinho mais próximo) e, iterativamente, realiza modificações sobre essa solução de forma a tentar obter uma solução melhor.

Este algorítmo é conhecido por ser de fácil implementação, rápido e por realizar uma otimização relativamente boa com pouco trabalho computacional.

A ideia principal do algorítmo é de eliminar rotas no percurso que se cruzem entre si e reordena-las de forma a que não o façam. No entanto, podem ser realizadas mais permutações na tentativa de encontrar uma melhor solução.

O pseudo-código com o funcionamento mais detalhado deste algorítmo encontra-se a seguir:





O algorítmo consiste em, portanto, fazer diversas iterações, nas quais testa diversas permutações diferentes, comparando a qualidade da ordem de visita após as permutações face à qualidade da ordem de visita original.

Ao encontrar uma ordem de visita melhor, a ordem de visita antiga é abandonada e a nova é adotada, passando-se à iteração seguinte e assim sucessivamente.

É de salientar que, apesar de não estar assim especificado no pseudo-código apresentado anteriormente, pode ser especificado um número de iterações específico para o cálculo do algorítmo.

Quando à complexidade espacial do algorítmo, visto que não usa quaisquer estruturas de dados para quaisquer cálculos, a sua **complexidade espacial** é constante, **O(1)**.

Quanto à complexidade temporal do algorítmo, sendo **T** o número de iterações a realizar e **N** o número de vértices que se encontram no vetor *visitOrder* (isto é, o número de pontos de interesse), visto que os dois ciclos internos do algorítmo percorrem N vértices cada um e são corridos tantas vezes quanto o número de iterações T, a **complexidade temporal** do algóritmo é de **O(T\*N2)**.

# Dificuldades encontradas no desenvolvilmento do Trabalho

No decorrer do trabalho encontramo-nos com algumas dificuldades, que conseguiram ser ultrapassados por esforço em equipa por parte de todos os membros.

A principal dificuldade foi o limite de tempo para a realização do trabalho. A quantidade de tempo que nos foi proporcionada para a realização do trabalho foi, na nossa opinião, demasiado reduzida.

Outra dificuldade foi o desfasamento do período de realização do trabalho face ao conteúdo das aulas, visto que os temas de trabalho saíram antes de ser introduzida a teoria dos grafos e que o prazo de entrega foi marcado para pouco depois da segunda aula prática sobre grafos, o que agravou um pouco a nossa dificuldade de gerir o tempo para realizar o trabalho. Este desfasamento obrigou também que fizessemos um estudo prévio de alguns tópicos antes de estes serem lecionados (o que não é necessariamente um problema).

Surgiram também algumas dificuldades na implementação de alguns dos algorítmos, dificuldades essas que foram ultrapassadas com o estudo cuidado dos algorítmos por parte de todos os membros do grupos e com reuniões presenciais para resolução dos problemas em equipa.

# Casos de Utilização

A aplicação implementada utiliza uma interface muito simples para interagir com o utilizador, utilizando um sistema de menus. Permite ter em memória um mapa (sob a forma de um grafo), carregado para memória do programa pelo utilizador, no qual podem ser realizadas diversas operações à escolha do utilizador.

As funcionalidades específicas do programa encontram-se a seguir especificados, com uma estrutura semelhante à estrutura de menus do programa, a partir do menu principal:

* **Carregar um mapa** (um grafo) para a memória do programa, para ser posteriormente utilizado pelo programa.
* **Operar sobre o mapa** que está atualmente na memória do programa.
  + **Visualização do Grafo**, que pode ser feita textualmente (na linha de comandos) ou graficamente (através do *GraphViewer*).
  + **Acessibilidade de um Ponto**, que, recorrendo aos algorítmos de Pesquisa em Profundidade e de Pesquisa em Largura, calcula o número de pontos acessíveis a partir do ponto inicial especificado pelo utilizador, mostrando o tempo de execução, em milissegundos, de ambos os algorítmos. O utilizador pode também escolher visualizar todos os pontos acessíveis.
  + **Caminho ótimo entre dois Pontos**, utilizando o algorítmo Dijkstra, o algorítmo A-Star, ou ambos os algorítmos, mostrando comparativamente os tempos de execução de ambos os algorítmos.
  + **Caminho ótimo entre dois Pontos passando por um Ponto de Interesse**, em que o utilizador especifica um conjunto de pontos de interesse, um ponto de partida e um ponto de destino. De seguida, o programa calcula, recorrendo a um algorítmo de Dijksta Bi-direcional, o melhor percurso entre o ponto de partida e o ponto de destino, passando pelo ponto de interesse que implica o menor desvio relativamente ao melhor percurso entre esses pontos (problema de deslocar entre dois pontos passando, por exemplo, por um posto de gasolina para reabastecimento).
  + **Caminho entre dois Pontos passando por um conjunto de Pontos de Interesse**, em que o utilizador especifica um conjunto de pontos de interesse, um ponto de partida e um ponto de destino. De seguida, o programa calcula um caminho com início no ponto de partida, passando por todos os pontos de interesse especificado e com termino no ponto de destino. Este calculo pode ser feito de múltiplas formas:
    - Utilizando uma heurística de vizinho mais próximo (em inglês, *nearest neighbor*), em que a ordem de visita dos pontos de interesse é a ordem de menor distância euclidiana entre os pontos.
    - ...

# 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.

# Glossário

(1) **Grafo Esparso** – Grafo com baixa densidade de arestas face à quantidade de vértices que tem, isto é, |E| [≅](https://pt.wiktionary.org/wiki/≅) |V|.

(2) **Grafo Denso** – Grafo com elevada densidade de arestas face à quantidade de vértices que tem, isto é, |E| [≅](https://pt.wiktionary.org/wiki/≅) |V|2 (a maior parte dos vértice encontra-se ligado a quase todos os restantes vértices do grafo.

(3) **Grafo Dirigido** – Grafo no qual as arestas têm uma direção definina, isto é:

∃ **v, w** ∈ **G** : **v** ∈ adjacent(**w**) ∧ w ∉ adjacent(**v**)

(4) **Grafo Cíclico** – Grafo em que contêm pelo menos um ciclo, isto é, em que é possível navegar de um vértice para ele próprio, passando por pelo menos um outro vértice, ou seja:

∃ **v** ∈ **G** : **v** ∈ DFS(**v**) , em que DFS representa a àrvore de pesquisa em profundidade resultante de fazer uma pesquisa em profundidade a partir de **v**.

(5) **Grafo Pesado** – Grafo no qual os vértice têm um valor numérico associado, que representa habitualmente o “custo” de navegar entre os vértices que a aresta conecta.

(6) **Função Heurística** – Função que ignora/manipula a informação de forma a tomar escolhas no sentido de alcançar uma solução mais rápida e fácil, isto é, uma função que tenta otimizar (em termos temporais) o processo de decisão de um algorítmo.

(7) **Função Consistente** – No contexto de algorítmos de *path-finding*, um **função heurística** diz-se **consistente** se a sua estimativa num vértice for sempre menor ou igual do que essa mesma estimativa em qualquer vértice vizinho acrescida do valor da aresta que conecta esses mesmos dois vértices e se o valor da função heurística no vértice de destino for nulo, isto é:

h(**v**) [≤](https://pt.wikipedia.org/wiki/Desigualdade) w(**v**,**n**) + h(**n**) ∧ h(**V**f) = 0

em que

* **v, n**, **V**f∈ **G**
* **n** ∈ adjacent(**v**)
* **V**f é o vértice de destino do algorítno
* **w**(v,n)∈ edge(**v**) é o valor da aresta que conecta **v** a **n**.
* **h**(v) representa representa a função heurística consistente em questão.

# 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>
* Consistent Heuristics, <https://en.wikipedia.org/wiki/Consistent_heuristic>
* Besan A. AlSalibi, Marzieh Babaeian Jelodar and Ibrahim Venkat - A Comparative Study between the Nearest Neighbor and Genetic Algorithms: A revisit to the Traveling Salesman Problem, <https://pdfs.semanticscholar.org/c45c/04e923ca478e49499ac02d8b0066b3c206e8.pdf>
* Christian Nilsson - Heuristics for the Traveling Salesman Problem, <https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf>
* Amanur Rahman Saiyed - The Traveling Salesman problem, <http://cs.indstate.edu/~zeeshan/aman.pdf>