U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

EasyPilot

Sistema de Navegação

Grupo B, turma 3

Ângelo Miguel Tenreiro Teixeira, up201606516@fe.up.pt
Henrique Melo Lima, up201606525@fe.up.pt
Rui Pedro Moutinho Moreira Alves, up201606746@fe.up.pt

30 de Março, 2018

Índice

Introdução e Descrição do Problema

A navegação GPS é uma tecnologia amplamente utilizada atualmente, equipando cada vez mais veículos e dispositivos e utilizada por diferentes *apps* móveis, em *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 é calculado 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 algoritmos de *path finding* encontram-se devidamente explicadas no capítulo **Glossário**.

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

Iteração 0: Pré-Processamento do Grafo

De forma a simplifcar os algorítmos que operam no Grafo, nesta iteração procurou-se reduzir o número de arestas e vértices, a fim de evitar processamento desnecessários em vértices consecutivos ligados por uma só aresta.
Uma descrição mais detalhada deste pré-processamento encontra-se num capítulo posterior.

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 ser válidas para pertenceram a um percurso, devido a estarem interrompidas por algum fator externo ou por terem caraterísticas indesejadas para o utilizador (a existência de, por exemplo, portagens), o que significa uma pequena alteração nos dados de entrada, sendo que os algoritmos utilizados para a resolução dos problemas são os mesmos. Tais dados de entrada não estão considerados nos exemplos apresentados na nossa aplicação.

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 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 considerarem 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 ou mais pontos de um conjunto de pontos de interesse indicado pelo utilizador.

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 subdividir 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 algoritmos 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 o resolver.

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.

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, dilúvios, …).

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 dividido em iterações no capítulo anterior é redutível a uma instância de um problema de grafos, como é demonstrado nos subcapítulos seguintes.

Dados de Entrada

- **P** – conjunto de pontos (localidades de um dado mapa). Cada ponto é caraterizado por:
  - Nome – nome identificativo do ponto (ex: localidade, edifício, etc)
  - Coordenadas – Coordenadas da localidade no mapa 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** $\in$ **P** – ponto do mapa em que o utilizador se encontra (ponto inicial do percurso)
- **Pf** $\in$ **P** – ponto que se pretende alcançar (destino)
- **POIs** $\subseteq$ **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 ($\in$ **P**) 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 um ou todos os vértices (conforme indicado pelo utilizador) contidos no conjunto de **POIs**, por uma qualquer ordem.
- **W** – peso total de todas as arestas percorridas no caminho (“custo” da viagem)

Restrições

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

Restrições nos dados de entrada

- $\forall$ **e** $\in$ **E**, peso(e) $\geq$ 0, visto que o peso representa sempre grandezas positivas (ou nulas no de o peso representar, por exemplo, um custo de viagem). A existência de arestas com pesos negativos poderia, visto que o grafo de entrada pode conter ciclos, levar à ocorrência de ciclos com peso negativo, tornando-se o problema de minimização não resolúvel.

Restrições nos dados de saída

- **W** $\geq$ 0, consequência da restrição de entrada em que $\forall$ **e** $\in$ **E**, peso(e) >= 0.
- **Pi** $\in$ **C** $\wedge$ **Pi** = $\mathbf{C}_0$, o ponto de partida tem de ser o primeiro vértice no conjunto ordenado de vértice que representa o melhor percurso.
- **Pf** $\in$ **C** $\wedge$ **Pf** = $\mathbf{C}_f$, o ponto de destino tem de ser o último vértice no conjunto ordenado de vértice que representa o melhor percurso.
- $\forall$ **p** $\in$ **POIs**, **p** $\in$ **C**, todos os POIs indicados devem estar contidos no percurso calculado (exceto na iteração **3.1**, como será especificado posteriormente)

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** $\in$ **POIs**. Por este motivo, a solução ótima passa por minimizar a função **h** a seguir descrita:

$$\mathbf{h} = \sum \text{peso}(\mathbf{e}) \text{ , } \mathbf{e} \in \mathbf{C}$$

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** definida em *Node.h*, 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 um conjunto das arestas que ligam o vértice a outros vértices do grafo (representado por um *std::vector<Edge>*).

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

Algoritmos que operam sobre a estrutura Grafo

Foi desenvolvido um conjunto de classes que representam algoritmos 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 algoritmo 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 auxiliar 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 algoritmo, bem como a apresentação do pseudocódigo 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 algoritmo. O funcionamento mais pormenorizado deste algoritmo 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 **Dijkstra** a partir do vértice de partida e, simultaneamente, uma expansão **Dijkstra** 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 **AStarBiDir** é responsável por calcular o caminho ótimo entre dois vértices, realizando uma expansão **A*** a partir do vértice de partida e, simultaneamente, uma expansão **A*** 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 Unidade Curricular 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 algoritmos, 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 algoritmos 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). Todas estas classes estendem 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 das iterações expostas no capítulo de Descrição do Problema.

Iteração 0: Pré-Processamento do Grafo

Como já foi explicado anteriormente, foi idealizado um pré-processamento da estrutura Grafo a fim de reduzir o número de vértices e arestas, a fim de diminuir o tempo de execução dos algorítmos que nesta estrutura operam.

Inicialmente, procedeu-se a eliminar todas as arestas intermédias de um caminho de vértices consecutivos que não continham quaisquer ramificações para outros caminhos. Isto é, num grafo dirigido constituido pelos vértices **A**, **B**, e **C**, em que **A**→**B** e **B**→**C**, o as duas arestas intermédias seriam eliminadas, resultando apenas numa aresta única ligando o vértice **A** ao vértice **C**.

Verificamos, contudo, que este procedimento levava a uma redução drástica (cerca de 50%) do número de arestas e que tornava muitos vértices inacessíveis (isto é, isolados), perdendo-se possíveis caminhos com início ou com destino a estes vértices isolados (isto é, deixaria de ser possível especificar um caminho com início ou destino em **B**).

Seguidamente, considerou-se remover apenas as ligações intermédias cujo ângulo face à direção inicial não excedesse os 45º (para garantir que apenas ligações dentro de uma mesma rua eram eliminadas). Todavia, este algorítmo eliminava um número reduzido de edges face ao tamanho do grafo (cerca de 1%), mantendo-se também o problema dos caminhos com início ou destino nos vértices eliminados.

Por este motivo, concluimos que este pré-processamento do Grafo não nos era vantajoso, pelo que decidimos não o fazer.

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 algoritmo de **Pesquisa em Profundidade** ou o algoritmo de **Pesquisa em Largura**, começando no vértice de origem do percurso.

No algoritmo 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 si 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 pseudocódigo 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.

```
performDFS(G,V):            // G - Grafo , V - Vértice de Origem
    visitedNodes <- {}      // Tabela de Dispersão de nós Visitados
    visitNode(G,V)
    return visitedNodes

visitNodeDFS(G,V):
    if nodeNotVisited(V) then
        insert(visitedNodes,V)
        for each W in Adjacent(V) do
            visitNodeDFS(G,W)
```

A **complexidade temporal** deste algoritmo 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 algoritmo. Quanto ao espaço, por ser um algoritmo 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|)**.

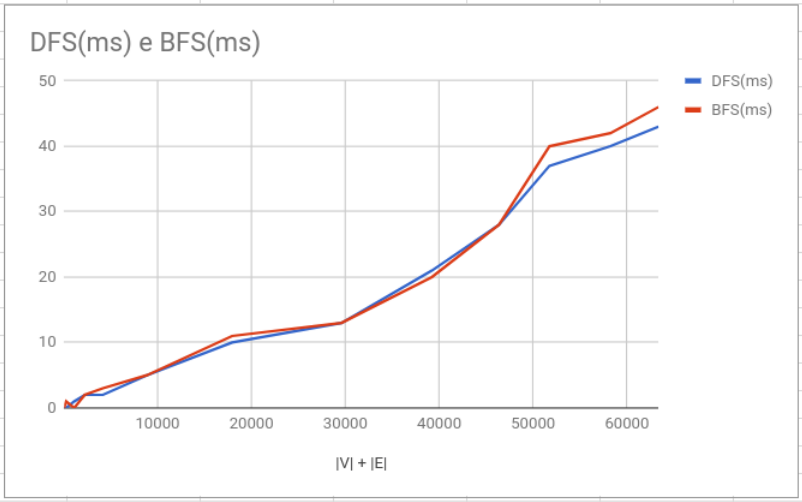
No algoritmo de Pesquisa em Largura, ao contrário do algoritmo previamente apresentado, são exploradas todas as arestas a partir do vértice em análise, passando só depois para o vértice seguinte. Esta forma de pesquisa é normalmente implementada com auxílio a uma fila, em que é analisado o vértice na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

```
performBFS(G,V):            // G - Grafo , V - Vértice de Origem
    visitedNodes <- {}      // Tabela de Dispersão de nós Visitados
    queue <- {}             // Fila de Vértices a visitar

    insert(visitedNodes,V)
    push(queue,V)

    while notEmpty(queue) do
        N <- pop(queue)
        for each W in Adjacent(N) do
            if nodeNotVisited(W) then
                insert(visitedNodes,W)
                push(queue,W)

    return visitedNodes
```

A **complexidade temporal** deste algoritmo é, tal como no algoritmo 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 algoritmo. Quanto ao espaço, no pior caso a fila que auxilia a implementação do algoritmo 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 algoritmo será, portanto, também **O(|V|)**.

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

|V| + |E|	DFS(ms)	BFS(ms)
57	0	0
262	0	1
1122	1	0
2242	2	2
4182	2	3
8857	5	5
17942	10	11
29602	13	13
39217	21	20
46366	28	28
51746	37	40
58242	40	42
63366	43	46



Concluiu-se também experimentalmente que a complexidade temporal de ambos os algoritmos é, portanto, aproximadamente linear. As amostras foram todas geradas aleatoriamente e de acordo com os as mesma condições 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 algoritmo **Dijkstra** e, posteriormente, à implementação de um "melhoramento" do mesmo, o algorímo **A***, que tem por base o mesmo funcionamento que o algoritmo Dijkstra, mas que utiliza uma heurística de otimização que será explicada mais à frente.

A fim de reduzir o tempo de execução dos algorítmos apresentados neste capítulo, é realizada uma pesquisa em profundidade antes da execução dos mesmos, a fim de evitar processamento desnecessário quando não existe nenhum caminho possível entre os vértices em questão.

Algoritmo Dijkstra

O algoritmo **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 algoritmo 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 algoritmo torna-se bastante eficiente e fácil de implementar.

Neste algoritmo, 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 algoritmo 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 algoritmo está concluído 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 pseudocódigo do algoritmo encontra-se a seguir apresentado:

```
/* G - Grafo , Vi - Vértice de Origem , Vf - Vértice de Destino */
Dijkstra(G,Vi,Vf):
    pQueue <- {}              // Fila de prioridade de Vértices

    // Populate pQueue
    for Node v : G
        if equals(v,Vi) then
            distance(v) <- 0
        else
            distance(v) <- INF
        path(v) <- nil
        insert(pQueue, v)

    // Perform Search
    while notEmpty(pQueue) do
        v <- pop(pQueue)

        // Check for finish node
        if equals(v,Vf) then
            return buildPath(G,Vf,Vi)

        for each w : Adjacent(v) do
            if distance(v) > distance(w) + weight(v,w) then
                distance(v) <- distance(w) + weight(v,w)
                path(v) <- w
                updateNodeOnQueue(pQueue,v)

    // Loop ended with no solution found
    return nil



buildPath(G,Vf,Vi):
    path <- {}        // Vetor de vértices que compõe o caminho
    w <- Vf           // Vf contém em distance(Vf) o peso total

    while w != Vi do
        pushFront(path, w)
        w <- path(w)

    pushFront(path, Vi)

    return path
```

O algoritmo 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 algoritmo, no peso total do vértice de destino.

A primeira fase do algoritmo subdivide-se também em duas fases: a primeira sub-fase consiste em preparar os vértices para a execução do algoritmo, sendo esta fase resolúvel 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 algoritmo 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 algoritmo consiste em percorrer todos os vértices que constituem o caminho calculado, pelo que é resolúvel, no pior caso, em tempo linear relativamente ao número de vértices do grafo, **O(|V|).**

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

Algoritmo A*

Após a implementação do algoritmo anteriormente descrito, procedeu-se à implementação de um "melhoramento", o algoritmo **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 pseudocódigo a seguir apresentado:

```
1  /* G - Grafo , Vi - Vértice de Origem , Vf - Vértice de Destino */
2  Dijkstra(G,Vi,Vf):
3      pQueue <- {}              // Fila de prioridade de Vértices
4
5      // Populate pQueue
6      for Node v : G
7          if equals(v,Vi) then
8              distance(v) <- 0 + euclidianDistance(v,Vf)
9          else
10             distance(v) <- INF
11         path(v) <- nil
12         insert(pQueue, v)
13
14     // Perform Search
15     while notEmpty(pQueue) do
16         v <- pop(pQueue)
17
18         // Check for finish node
19         if equals(v,Vf) then
20             return buildPath(G,Vf,Vi)
21
```

```
22          for each w : Adjacent(v) do
23              if distance(v) > distance(w) - euclidianDistance(w,Vf) +
24                               weight(v,w) + euclidianDistance(v,Vf) then
25                  distance(v) <- distance(w) - euclidianDistance(w,Vf) +
26                                 weight(v,w) + euclidianDistance(v,Vf)
27                  path(v) <- w
28                  updateNodeOnQueue(pQueue,v)
29
30      // Loop ended with no solution found
31      return nil
32
33
34
35  buildPath(G,Vf,Vi):
36      path <- {}          // Vetor de vértices que compõe o caminho
37      w <- Vf             // Vf contém em distance(Vf) o peso total
38
39      while w != Vi do
40          pushFront(path, w)
41          w <- path(w)
42
43      pushFront(path, Vi)
44
45      return path
```

O algoritmo é, portanto, idêntico em tudo ao algoritmo **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 algoritmo **A*** não garante sempre a solução ótima, visto que a função heurística utilizada é uma **função consistente**. Isto é, este algoritmo assegura a solução ótima se e só se:

$$\forall\ \mathbf{v} \in \mathbf{G},\ \forall\ \mathbf{n} \in \text{adjacent}(\mathbf{v}),\ w(\mathbf{v,n}) \geq h(\mathbf{v,n})$$

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 algoritmo ser em tudo idêntico ao algoritmo **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 algoritmos serem iguais em termos de complexidade, devido à heurística de aproximação do algoritmo **A***, espera-se que este alcance um melhor desempenho no caso médio do que o algoritmo **Dijkstra**. Para corroborar esta teoria, foram realizados testes de desempenho de ambos os algoritmos em grafos aproximadamente esparsos, analisando o seu tempo de execução:

|V| + |E|	Dijkstra(ms)	A-Star(ms)
76	0	0
281	0	0
636	1	1
1141	2	2
1796	3	3
2601	7	4
3556	10	9
4661	16	14
5916	17	13
7321	23	16
8876	32	18
10581	35	22
12436	55	27
14441	51	30
16596	80	44
18901	118	93
21356	141	92
23961	168	139
26716	213	159
29621	263	151
32676	283	147
35881	281	163
39236	293	173
42741	282	191
46396	400	261
50201	392	279
54156	432	268
58261	610	310
62516	548	287
66921	498	320
71476	787	322
76181	818	317
81036	911	449
86041	1109	498
91196	1315	578
96501	1027	470
101956	1325	373
107561	1245	462
113316	1419	592
119221	1512	610

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

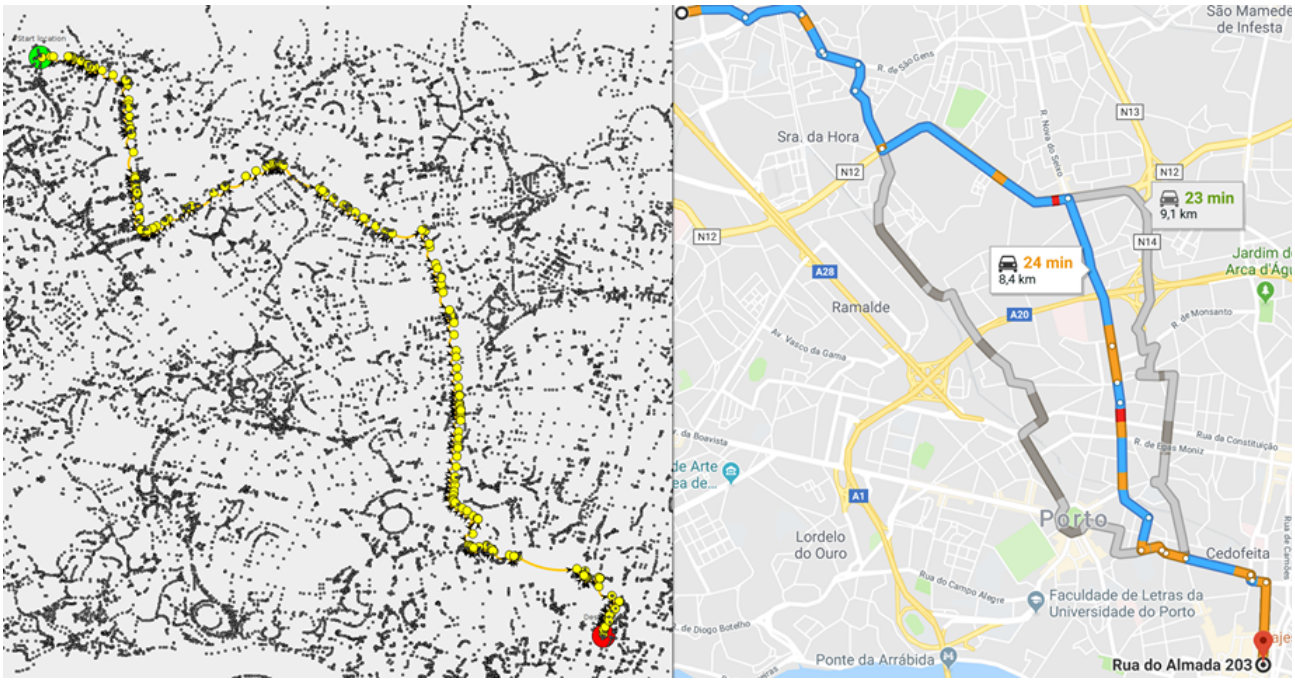
Apesar de o desempenho ter sido melhor do que o que era expectável, o algoritmo **A*** teve sempre resultados que o algoritmo **Dijkstra,** devido à sua boa heurística de otimização.

Quanto à qualidade dos resultados face ao expectável, é provável que se deva ao facto de os caminhos dos vértices gerados aleatoriamente terem sido relativamente pequenos e devido ao grafo ser pouco denso (|E| ≅ |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).

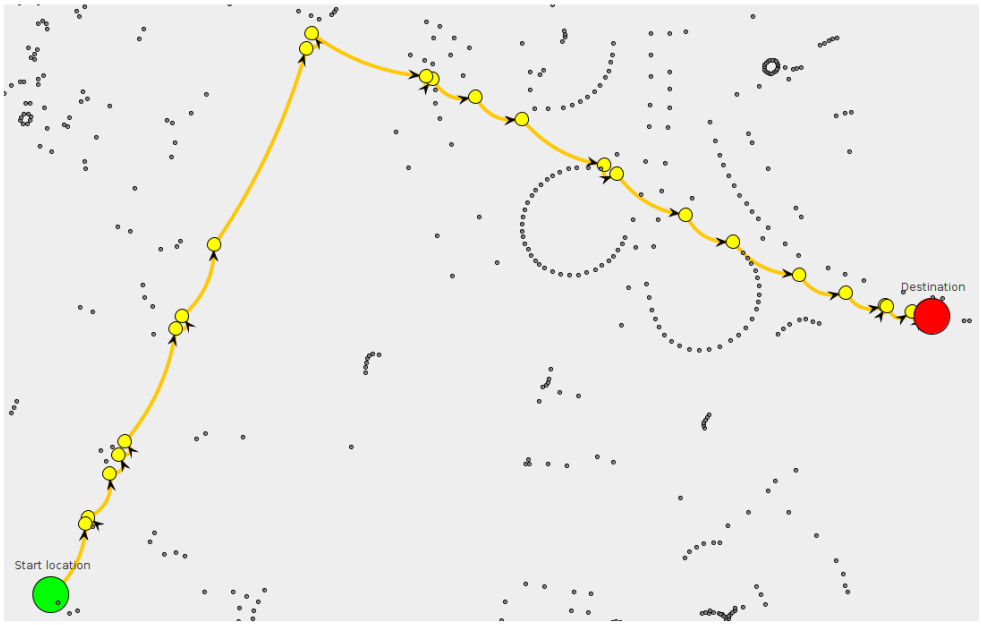
É de salientar também, novamente, que o algoritmo **A*** só apresentou resultados significativamente melhores porque o valor das arestas segue a **função consistente** explicada previamente. Caso contrário, os algoritmo apresentariam resultados muito semelhantes para qualquer valor de |V|+|E|.

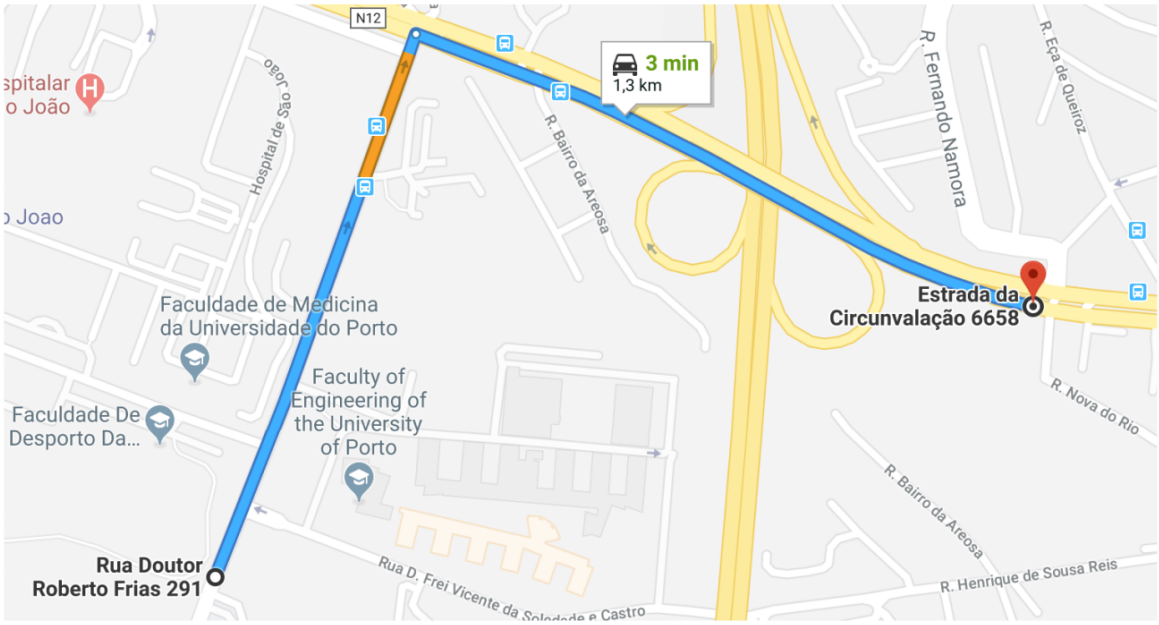
Ao correr tanto o algoritmo **Dijkstra** como o **A*** num grafo que contém, aproximadamente, 41 000 vértices (em que |V| + |E| ≅ 90 000), que representa todo o mapa da região do centro do **Porto**, parte de **Vila Nova de Gaia** e parte de **Matosinhos** (extraído do ***Open Street Maps***), obtemos resultados muito idênticos aos resultados do ***Google Maps*** para o mesmo percurso. Por exemplo, numa viagem da Avenida dos aliados até à Rua da Lagoa (na Senhora da Hora, Matosinhos), foi obtido um percurso de 8,9 Km, apenas 300m mais longo do que o caminho de 8,6 Km do *Google Maps* (devido, possivelmente, a diferenças dos mapas do *Open Street Maps* e do *Google Maps*), como se pode ver na imagem seguinte:



Também se realizaram testes em grafos mais pequenos, como no grafo que representa o mapa de toda a zona da **Asprela**, que contém grande parte do polo Universitário da Asprela, Hospital de São João e parte da Areosa, grafo esse com aproximadamente 2 700 vértices (em que |V| + |E| ≅ 5500). Por exemplo, numa viagem do cruzamento da rua Doutor Roberto Frias com a rua Dr. Júlio de

Matos, ao lado da FEUP, até à Liga Portuguesa contra o Cancro, perto da Areosa, foi obtido um caminho exatamente igual ao caminho sugerido pelo *Google Maps*, caminho esse com pouco menos de 1,3 Km, como se pode ver na imagem seguinte:





A semelhança entre as soluções da nossa plataforma e do *Google Maps* verificou-se também em muitos outros casos de teste, quer em grafos de pequenas e dimensões, quer em grafos densos ou esparsos, que em grafos fortemente conexos ou pouco conexos.

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 percurso 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 vértice 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 algoritmo **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 algoritmo 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 especificado se trata do caminho **Vi → P → Vf**.

O algoritmo mais detalhado com as etapas do seu funcionamento encontra-se especificado no pseudocódigo a seguir apresentado:

```
1  /* G - Grafo, Vi - Vértice de Origem ,
2     Vf - Vértice de Destino, POIs - Conjunto de Vértices Intermédios */
3  TSP(G,Vi,Vf,POIs):
4      solutionWasFound <- false
5
6      // Realizar expansão Dijkstra a partir de Vi e Vf até topo da
7      // à condição de paragem ser verificada
8      do
9          performDijkstraStep(G,Vi,Vf)
10         performReverseDijkstraStep(G,Vf,Vi)
11
```

```
12          if pQueuesContainSamePOI() and
13             pQueuesTops >= bestSolutionThroughPOI then
14              solutionWasFound <- true
15              break
16      while dijkstraExpansionsInProgress()
17
18      if solutionWasFound then
19          return append(dijkstraResult,reverse(reverseDijkstraResult))
20      else
21          return nil
```

É de salientar que a condição de paragem deste algoritmo não é trivial: Para assegurar a solução ótima do algoritmo 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 $\mathbf{P_2}$ tal que o caminho **Y** de **Vi** a **Vf** passando pelo ponto de interesse $\mathbf{P_2}$ seja de peso total inferior ao caminho **W**.

Para tornar a execução do algoritmo mais rápida e eficiente, recorremos a *multi-threading*, utilizando um *core* do processador para cada uma das duas expansões que, devido ao facto de se poderem realizar de forma assíncrona, poupando bastante tempo de processamento e alcançando melhores resultados.

Quanto à complexidade do algoritmo em questão, o algoritmo trata-se apenas da dupla utilização do algoritmo **Dijkstra**, acrescido de uma *tabela de dispersão* extra para armazenamento dos pontos de interesse potenciais a serem incluídos 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 algoritmo é, tal como anteriormente no algoritmo Dijkstra, **O((|V|+|E|)*log |V|)** e a **complexidade temporal** é **O(|V|)**.

Foram realizados testes em grafos provenientes de mapas do *Open Street Maps*, a fim de comparar a qualidade da nossa solução à solução do *Google Maps*. Um deles foi o grafo que representa o mapa de toda a zona da **Asprela**, que contém grande parte do polo Universitário da Asprela, Hospital de São João e parte da Areosa, grafo esse com aproximadamente 2 700 vértices (em que |V| + |E| ≅ 5500). Por exemplo, numa viagem do cruzamento da rua Doutor Roberto Frias com a rua Dr. Júlio de Matos, ao lado da FEUP, até à Liga Portuguesa contra o Cancro, perto da Areosa, **passando pelo posto de gasolina que exige um menor desvio do caminho ótimo**, foi obtido um caminho exatamente igual ao caminho sugerido pelo *Google Maps*, passando pelo posto de gasolina da *Cepsa* que se situa em frente ao Hospital de São João (que se trata do posto de gasolina

evidentemente mais próximo relativamente a qualquer outro posto na área da Asprela e Areosa), caminho esse com pouco cerca de 2,8 Km de comprimento, como se pode observar na imagem seguinte:

É de salientar que o desvio associado a visitar este posto de gasolina é devido às estradas em frente ao Hospital de São João serem vias de um único sentido, assim como grande parte das estradas na zona da Asprela. Por este motivo, a maior parte das arestas associadas ao grafo que representa esta zona são unidirecionais.

Para obter um melhor desempenho temporal, procedeu-se de seguida à implementação do algoritmo **A***, também utilizando uma expansão bi-direcional.

Este algoritmo é em tudo semelhante ao algorítmo Dijkstra com expansão bi-direcional, à exceção que utiliza uma heurística de distância euclidiana ao vértice de destino, conforme já foi explicado numa iteração anterior.

Por este motivo, a **complexidade temporal** e a **complexidade** espacial deste algorítmo são iguais à do algorítmo Dijkstra com expansão bi-direcional.

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.

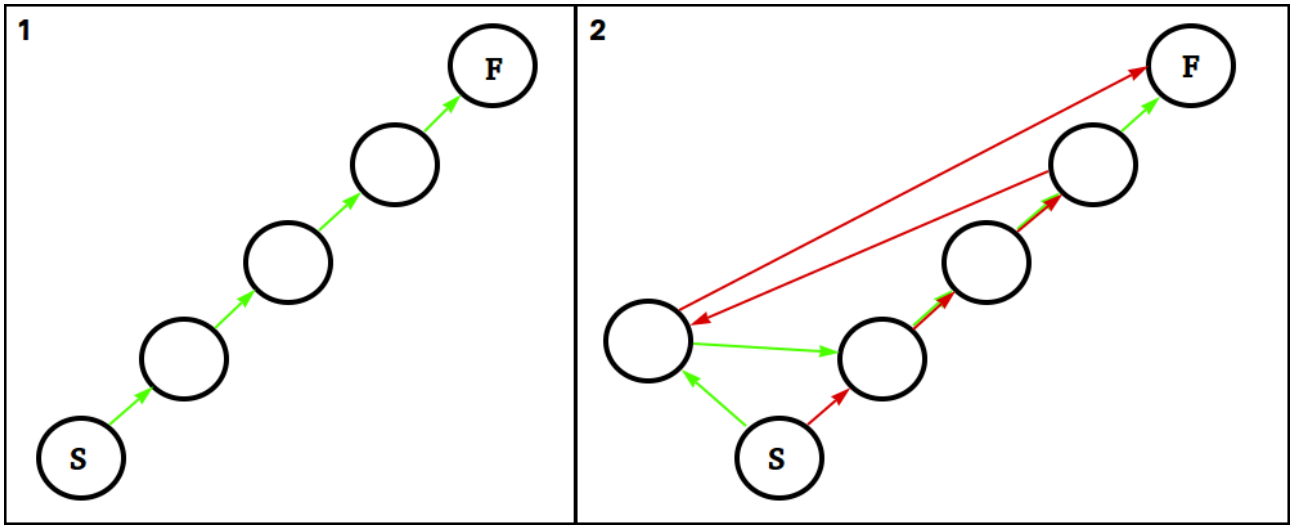
Este problema, no entanto, só tem solução em tempo fatorial: só é possível obter garantidamente a melhor solução com *brute-force*, ou seja, testando exaustivamente todas as possibilidades de caminhos. Por este motivo, a **complexidade temporal** do algoritmo é **O(n!)**, visto que é necessário calcular 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 algoritmo 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 um distante da solução ótima. Os diagramas seguintes ilustram o desempenho deste algoritmo face à solução ótima:



No Diagrama **1**, o algoritmo encontra a solução ótima em tempo linear. No entanto, ao adicionar um novo vértice ao diagrama (Diagrama **2**), o algoritmo 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 algoritmo 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 algoritmo 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 algoritmo mais detalhado com todas as etapas do seu funcionamento encontra-se especificado no pseudocódigo a seguir apresentado:

```
/* G - Grafo, Vi - Vértice de Origem ,
   Vf - Vértice de Destino, POIs - Conjunto de Vértices Intermédios */
TSP(G,Vi,Vf,POIs):
    // Calcular ordem de Visita
    ordemVisita <- {}
    calcOrdemVisita(G,Vi,Vf,POIs, ordemVisita)

    // Verificar se ordem de visita contem todos os POIs
    if isComplete(ordemVisita) then
        // Construir Caminho
        return buildPath(ordemVisita, G)
    else
        return nil


calcOrdemVisita(G,Vi,Vf,POIs,ordemVisita):
    insert(ordemVisita, Vi)

    // Verificar se é possível construir o caminho a partir do vértice
    if canAccess(Vi,POIs) and canAccess(Vi,Vf) then
        nextVisit <- findNearestPOI(Vi,POIs,G)
        remove(POIs, nextVisit)
        calcOrdemVisita(G,nextVisit,Vf,POIs, ordemVisita)

        // Verificar se a recursão teve sucesso
        if isComplete(ordemVisita) then
            return
        else
            // Vértice Indesejado, fazer back-trace
            remove(OrdemVisita, Vi)
            return
    else
        // Vértice Indesejado, fazer back-trace
        remove(OrdemVisita, Vi)
        return


buildPath(ordemVisita, G):
    finalPath <- {}

```

```
41      // Encontrar melhor caminho entre A0->A1, A1->A2, ..., An-1 -> An
42      for each Vertex in ordemVisita do
43          // bestPath pode ser um algoritmo como Dijkstra ou A-Star
44          append(finalPath, bestPath(Vertex,nextVertex))
45
46      return finalPath
```

O algoritmo é composto, como é evidenciado no pseudocódigo, 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 algoritmo **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 algoritmo de melhor caminho entre dois pontos, como o algoritmo **Dijkstra** ou **A-Star** (foi utilizado na nossa implementação o algoritmo 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 subcaminhos).

Este algoritmo 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 algoritmo 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 algoritmo é linear relativamente ao número de vértices do grafo, **O(|V|)**.

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

Na primeira fase do algoritmo, pode ser necessário fazer uma pesquisa em profundidade 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 algoritmo de pesquisa em profundidade é de O(|V|+|E|), a complexidade temporal desta fase do algoritmo é, portanto, no pior caso, de **O((|V|+|E|)*|V|)**.

Quanto à segunda fase do algoritmo, no pior caso o vetor *ordemVisita* contém todos os vértices do grafo, sendo portanto necessário utilizar |V| - 1 vezes o algoritmo 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 algoritmo é dominante, em termos de notação *Big-O*, relativamente ao termo da primeira fase, a **complexidade temporal** deste algoritmo 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 bidirecionais, a obtenção da ordem de pesquisa (primeira fase do algoritmo) 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 algoritmo, 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 algoritmo **2-Opt** a fim de resolver o problema do caixeiro viajante iterativamente.

Este algoritmo, ao contrário de todos discutidos previamente, parte de uma solução inicial (seja esta solução aleatória ou, idealmente, calculada com um algoritmo 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 algoritmo é conhecido por ser de fácil implementação, rápido e por realizar uma otimização relativamente boa com pouco trabalho computacional.

A ideia principal do algoritmo 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 pseudocódigo com o funcionamento mais detalhado encontra-se a seguir:

```
2opt(visitOrder):       /* Ordem de visita inicial dos vértices */
    bestVisitOrder <- visitOrder

    while visitOrderImproves() do
        loopStart:

        bestDistance <- calculateDistance(bestVisitOrder)
        for (i=1 ; i<=numNodesToBeSwapped-1 ; i++) do
            for (k=i+1 ; k<=numNodesToBeSwapped ; k++) do
                newVisitOrder <- perform2optSwap(bestVisitOrder, i, k)
                newDistance <- calculateDistance(newVisitOrder)
                if newDistance < bestDistance then
                    bestDistance <- newDistance
                    bestVisitOrder <- newVisitOrder
                    goto loopStart

    return bestVisitOrder
```

```
20 // Reverte ordem do vetor visitOrder entre os indices i e k, inclusivé
21 perform2optSwap(visitOrder,i,k):
22     while i<k do
23         swap(visitOrder[i], visitOrder[k])
24         i++
25         k--
26 
27     return visitOrder
```

O algoritmo 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 algoritmo.

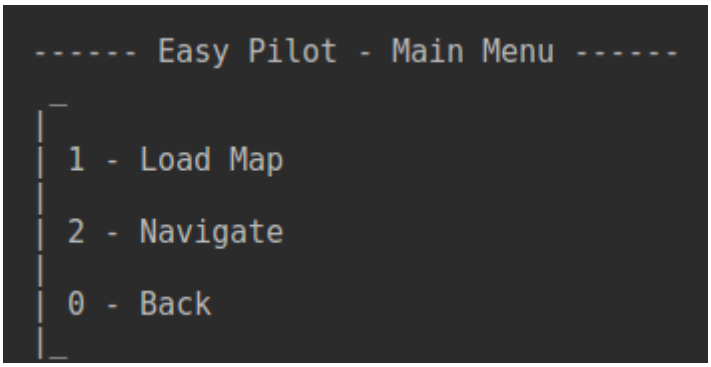
Quanto à complexidade espacial do algoritmo, visto que não usa quaisquer estruturas de dados para quaisquer cálculos, a sua **complexidade espacial** é constante, **O(1)**.

Quanto à complexidade temporal do algoritmo, 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 algoritmo percorrem N vértices cada um e são corridos tantas vezes quanto o número de iterações T, a **complexidade temporal** do algoritmo é de **$O(T*N^2)$**.

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:

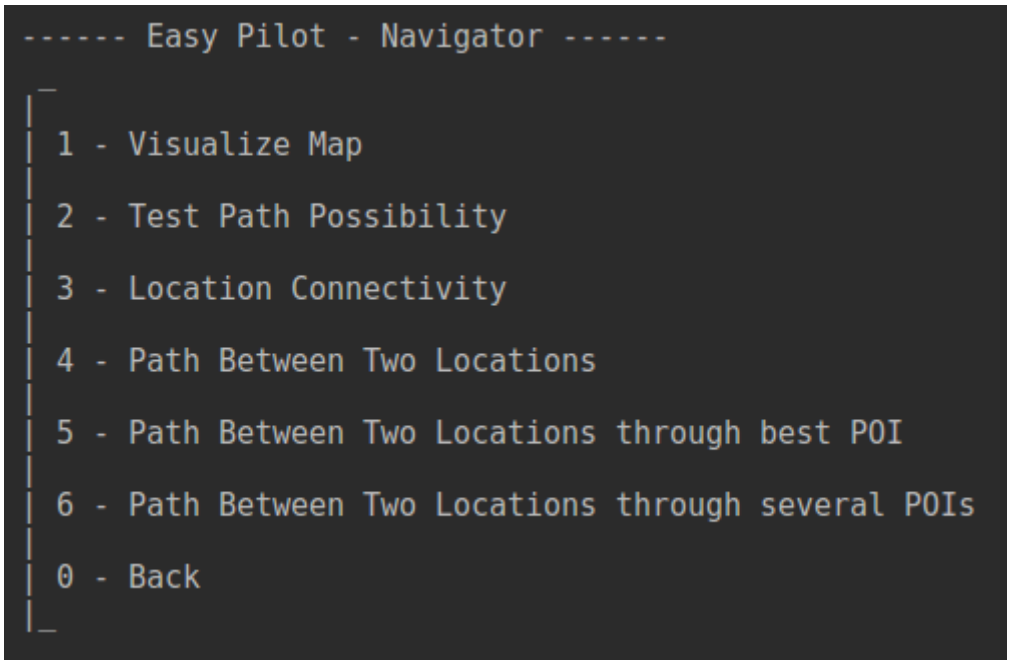


Menu Principal

- **Carregar um mapa** (um grafo) para a memória do programa, para ser posteriormente utilizado pelo programa.

Menu de Carregamento de Mapas

- **Operar sobre o mapa** que está atualmente na memória do programa, isto é, realizar algorítmos sobre o grafo.



Menu de Navegação no Mapa

  - **Visualização do Grafo**, que pode ser feita textualmente (na linha de comandos) ou graficamente (através do *GraphViewer*).
  - **Possibilidade de realizar um Caminho**, que, recorrendo ao algoritmo de Pesquisa em Profundidade, verifica se é possível a deslocação entre os pontos de partida e de destino introduzidos pelo utilizador.
  - **Conectividade de um Ponto**, que, recorrendo aos algoritmos 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 algoritmos. O utilizador pode também escolher visualizar todos os pontos acessíveis.
  - **Caminho ótimo entre dois Pontos**, que pode ser realizado de 4 formas diferentes:
    - Utilizando o algoritmo Dijkstra
    - Utilizando o algoritmo Dijsktra Bi-Direcional, isto é, realizando expansão a partir do vértice de origem e do vértice de destino simultaneamente.
    - Utilizando o algoritmo A*

        - Utilizando o algoritmo A* Bi-Direcional, isto é, realizando expansão a partir do vértice de origem e do vértice de destino simultaneamente.
        - Utilizando ambos os algoritmos Dijkstra e A*, comparando o tempo de execução de ambos.
        - Utilizando ambos os algoritmos A* e A* com expansão Bi-Direcional, comparando o tempo de execução de ambos.
    - **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 algoritmo 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 término no ponto de destino. Este cálculo 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.
        - Utilizando o algoritmo iterativo 2-opt que, iterativamente, melhora a solução obtida pela heurística de vizinho mais próximo, realizando permutações entre a ordem pontos de interesse a visitar. O número de iterações pode ser indicado pelo utilizador e é apresentado o melhoramento.

Todos os caminhos calculados pela aplicação podem ser observados graficamente, recorrendo ao *Graph Viewer* disponibilizado pela equipa docente da Unidade Curricular .

Foi realizada uma manipulação no *display* do grafo, redimensionando a janela conforme o necessário de forma a mostrar só as regiões pertinentes aos caminho e, também, colorindo de forma diferente os vértices e as arestas pertencentes aos caminhos, de forma a tornar o percurso mais visível.

Dificuldades encontradas no desenvolvimento 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 algoritmos, dificuldades essas que foram ultrapassadas com o estudo cuidado dos algoritmos por parte de todos os membros do grupos e com reuniões presenciais para resolução dos problemas em equipa.

Conclusões

Quanto aos objetivos do trabalho, todos estes foram cumpridos:

- Foram implementados todos os **algoritmos abordados nas aulas teóricas**, desde os mais simples como **Pesquisa em Largura** e **Pesquisa em Profundidade** para testar conectividade entre vértices do grafo, até a alguns mais complexos como o **Dijkstra** e o **A*** para o cálculo do caminho ótimo entre dois vértices do grafo, **Dijkstra Bi-Direcional** para incluir o melhor ponto de interesse de um conjunto de pontos de interesse no caminho ótimo entre dois vértices do grafo e **Nearest Neighbor** para calcular um percurso entre dois vértices passando por um conjunto de pontos de interesse por uma qualquer ordem (problema do caixeiro viajante).

- Foram também implementados algoritmos **não abordados no âmbito da Unidade Curricular**, como o algoritmo iterativo 2-opt, que realiza um melhoramento iterativo sobre uma solução já existente para o problema do Caixeiro Viajante.

- Foram também utilizadas **estratégias de conceção de algoritmos** abordadas nas aulas teóricas, como a recursividade, memorização (na Pesquisa em Profundidade, por exemplo) e *back-tracking* (no Nearest Neighbor, por exemplo).

- Foi abordado o conceito de **programação paralela,** tendo sido utilizado *multi-threading* para a implementação do algoritmo de **Dijkstra Bi-Direcional**.

- Foram realizadas **análises de complexidade** temporal e espacial de todos os algoritmos implementados e foi apresentado o **pseudo-código** de todos os algoritmos, com base nas apresentações das aulas teóricas da disciplina.

- Foram elaborados **Gráficos de desempenho** para comparação da eficiência dos algoritmos e para **análise experimental** da complexidade temporal dos algoritmos, que correspondeu à análise teórica. Estes testes levaram também a outras conclusões. Concluímos, por exemplo, que o algoritmo **A*** consegue, num grafo que representa um contexto real (conexo), em média, tempo de execução cerca de 3 vezes menores do que o algoritmo **Dijkstra**.

- Foi implementada **Visualização Gráfica** dos Grafos e dos resultados obtidos obtidos pelos algoritmo, utilizando o ***GraphViewer*** fornecido pela equipa docente da Unidade Curricular.

- Foram realizadas **comparações a outras plataformas**, comparando os resultados obtidos pela nossa aplicação aos resultados do ***Google Maps***. Os resultados obtidos foram sempre muito semelhantes, sendo grande parte das vezes exatamente iguais, quer em caminhos de pequenas dimensões, quer em caminhos de dimensões superiores.

- Foi também implementada uma **Interface de Utilizador** bastante intuitiva e fácil de utilizar, que permite facilmente recorrer a todas as funcionalidades implementadas, utilizando um sistema de menus.
- Foi utilizado um **Design Orientado a Classes e Objetos**, sendo que foram implementadas classes para a realização de todos os algoritmo, recorrendo a hierarquias de classes (por exemplo, as classes DFS – *Depth First Search* e BFS – *Breadth First Search* estendem a superclasse GraphSearchAlgorith).
- O código foi divido em vários **Módulos**, divididos por responsabilidades independentes, pelo que o código está organizado de forma bastante modular.

O trabalho foi realizado de forma equalitária por todos os membros do grupo, sendo a maior parte do trabalho desenvolvido em reuniões presenciais nas quais foram discutidas várias ideias, foram planificadas as várias etapas do trabalho e onde se discutiu eficiência e implementação dos vários algoritmos e heurísticas de otimização. 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.

Apesar de diversos percalços no desenvolvimento do projeto, o grupo conseguiu enfrentar todos os desafios com muitas horas de trabalho.

Glossário

(1) **Grafo Esparso** – Grafo com baixa densidade de arestas face à quantidade de vértices que tem, isto é, $|E| \cong |V|$.

(2) **Grafo Denso** – Grafo com elevada densidade de arestas face à quantidade de vértices que tem, isto é, $|E| \cong |V|^2$ (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 definida, isto é:

$\exists\ \mathbf{v}, \mathbf{w} \in \mathbf{G}\ :\ \mathbf{v} \in \text{adjacent}(\mathbf{w}) \wedge w \notin \text{adjacent}(\mathbf{v})$

(4) **Ciclo** – Caminho que tem início e acaba no mesmo vértice.

(5) **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:

$\exists\ \mathbf{v} \in \mathbf{G}\ :\ \mathbf{v} \in \text{DFS}(\mathbf{v})$, em que DFS representa a árvore de pesquisa em profundidade resultante de fazer uma pesquisa em profundidade a partir de **v**.

(6) **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.

(7) **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 algoritmo.

(8) **Função Consistente** – No contexto de algoritmos de *path-finding*, um **função heurística** diz-se **consistente** se a sua estimativa num vértice for <u>sempre</u> 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:

$$h(\mathbf{v}) \leq w(\mathbf{v},\mathbf{n}) + h(\mathbf{n}) \quad \wedge \quad h(\mathbf{V}f) = 0$$

em que

- $\mathbf{v}, \mathbf{n}, \mathbf{V}f \in \mathbf{G}$
- $\mathbf{n} \in \text{adjacent}(\mathbf{v})$
- $\mathbf{V}f$ é o vértice de destino do algoritmo
- $\mathbf{w}(v,n) \in \text{edge}(\mathbf{v})$ é o valor da aresta que conecta **v** a **n**.
- $\mathbf{h}(v)$ representa representa a função heurística consistente em questão.

(9) **Grafo Conexo** – Um grafo é conexo se todos os pares de vértices estão ligados por um caminho.

(10) **Grafo Completo –** Um grafo é completo se qualquer vértice estiver conectado por uma aresta a todos os vértices do grafo.

Bibliografia e outras Fontes de Referência

- Apresentações das Aulas Teóricas de Conceção e Análise de Algoritmo 2018, da autoria da Professora Doutora Liliana Ferreira, Professor Doutor João Pascoal Faria e Professor Doutor Rosaldo Rossetti.
- Thomas H. Cormen... [et al.]; Introduction to algorithms. ISBN: 978-0-262-53305-8
- Dijkstra's Algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- 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
- Google Maps, https://www.google.com/maps/
- Open Street Maps, https://www.openstreetmap.org/
- 2-Opt, https://en.wikipedia.org/wiki/2-opt
- Heuristics for the Traveling Salesman Problem, https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf