

TourMateApp: rotas turísticas urbanas adaptáveis (tema 3)



**Conceção e Análise
de Algoritmos
2ºAno MIEIC**

25 de maio de 2020

Inês Silva, up201806385@fe.up.pt
Mariana Truta, up201806543@fe.up.pt
Rita Peixoto, up201806257@fe.up.pt
Grupo 8, Turma 3

Índice

Introdução	3
Descrição do tema.....	3
Formalização do problema.....	4
Dados de entrada	4
Dados de saída	5
Restrições	5
Função objetivo	6
Perspetiva de solução.....	7
Técnicas de implementação	7
1. Pré-processamento do grafo	7
2. Verificar se é viável a deslocação entre dois pontos	7
3. Encontrar o caminho mais curto entre dois pontos num grafo não dirigido..	7
4. Encontrar o máximo de caminhos possíveis cuja soma de tempo não exceda o especificado pelo utilizador	8
5. Maximizar o número de pontos de interesse visitados	8
Algoritmos a serem considerados.....	9
1. Algoritmo de Pesquisa em Largura	9
2. Algoritmo de Pesquisa em Profundidade.....	10
3. Algoritmo de <i>Dijkstra</i>	10
4. Algoritmo A*	11
Casos de utilização e funcionalidades.....	12
Principais casos de uso implementados.....	13
Estruturas de dados utilizadas	14
Implementação	16
Problema 1	16
Problema 2	17
Problema 3	19
Algoritmos efetivamente implementados.....	20

Algoritmo de Pesquisa em Largura (BFS)	20
Algoritmo de Pesquisa em Profundidade (DFS)	20
Algoritmo de Dijkstra	20
Algoritmo A*	21
Algoritmos de pesquisa bidirecional: A* e Dijkstra	21
Algoritmo de Floyd-Warshall.....	22
Análise temporal empírica dos algoritmos implementados	24
Comparação entre Dijkstra e A*	24
Comparação entre DFS e BFS	24
Comparação entre algoritmos de pesquisa bidirecional: Dijkstra e A*	25
Comparação entre Dijkstra e Floyd-Warshall.....	Erro! Marcador não definido.
Conectividade dos grafos utilizados	26
Notas de implementação.....	27
Conclusão.....	28
Bibliografia	29

Introdução

Descrição do tema

Neste projeto, a intenção é implementar o aplicativo ***TourMateApp***, que permite a construção de itinerários turísticos adaptáveis às preferências e disponibilidade do usuário.

A aplicação mantém uma lista de pontos turísticos de interesses (*POI*) e sugere itinerários que incluem as atrações mais adequadas ao perfil do usuário, num caminho que possa ser realizado no tempo indicado, de uma origem a um destino final, indicado pelo mesmo.

As recomendações maximizam o número de atrações turísticas de acordo com o perfil e as preferências do utilizador, que também pode selecionar circuitos a pé, de carro ou de transporte público.

Para além disso, é necessário ter em atenção se as áreas pelas quais os caminhos passam se encontram, no presente momento, inacessíveis.

Esta aplicação irá utilizar mapas reais extraídos do *OpenStreetMaps* (www.openstreetmap.org) e coordenadas geográficas de alguns pontos de interesse turístico.

Formalização do problema

A rede viária pode ser representada por um **grafo dirigido** em que os **vértices** representam interseções, as **arestas** representam vias e os **pesos** representam distâncias, tempos.

Dados de entrada

G(N, E) - grafo dirigido pesado, representando o mapa da cidade em questão.
Constituído por:

N - Conjunto de vértices que representam os pontos da cidade. Cada ponto é caracterizado por:

id - identificador único do vértice

info - coordenadas do vértice(x, y)

type - tipo de ponto de interesse

duration - duração média do ponto de interesse

adj $\subseteq E$ - conjunto de arestas que partem deste ponto

E - Conjunto de arestas que ligam os vértices entre si, que representam os caminhos entre pontos. Cada aresta é caracterizada por:

id - identificador único da aresta

weight – tempo entre vértices

orig $\in N$ - vértice de origem da aresta

dest $\in N$ - vértice destino da aresta

POIs $\subseteq N$ - conjunto de todos os pontos de interesse

Preferências do utilizador - tempo máximo para realizar o caminho, meio de locomoção selecionado, tipo de pontos turísticos de interesse, ponto inicial do percurso (ponto onde o usuário se encontra, N_i), ponto final do percurso (ponto de destino do usuário, N_f), entre outros.

Dados de saída

WeightTF - peso total de todas as arestas percorridas no percurso (tempo de duração do caminho somado ao tempo de duração dos pontos de interesse efetivamente visitados);

WeightDF - peso total de todas as arestas percorridas no percurso (distância);

P - sequência ordenada dos vértices que representam o melhor caminho entre N_i e N_f , passando pelo maior número de pontos de interesse possíveis de visitar que vão de encontro às preferências selecionadas, sem ordem específica;

VPoi - id dos pontos de interesse efetivamente visitados contidos na sequência ordenada P;

P = {} - não existe caminho possível entre os pontos inseridos pelo utilizador;

VPoi = {} - caso não exista caminho possível entre os pontos inseridos pelo utilizador ou caso o utilizador não tenha tempo de efetivamente visitar algum ponto de interesse ou caso o utilizador não tenha tempo de efetuar o caminho mais rápido entre o seu ponto inicial e final (é retornado o caminho mais rápido que já excede o seu tempo disponível, pelo que não pode visitar nenhum ponto de interesse).

Restrições

- **Para todos os vértices:**

- $type(N[i]) = "information" \vee "hotel" \vee "attraction" \vee "viewpoint" \vee "guest_house" \vee "picnic_site" \vee "artwork" \vee "camp_site" \vee "museum" \vee "*" \vee NULL$.
- $duration \geq 0$, uma vez que se trata de tempo.
- Se $type(N[i]) = NULL$, ou seja, for uma *string* vazia, então *duration* tem que ser obrigatoriamente 0 e vice-versa.

Nota: O facto de o *type* do vértice ser *NULL* (ser uma *string* vazia) significa que é um vértice genérico e, portanto, não representa um ponto de interesse. Da mesma forma, o atributo *duration* é 0 se se tratar de um ponto genérico.

- **Para todas as arestas:**

- $weight(E[i]) \geq 0$;
- $orig \neq dest$, uma vez que uma aresta não pode começar e acabar no mesmo vértice.

- **$WeightTF \geq 0$ e $WeightDF \geq 0$** , uma vez que representam distâncias/tempos;
- **$N_i \in N \wedge N_i = P_0$** , isto é, o ponto inicial tem de ser o primeiro vértice na sequência ordenada de vértices que representa o trajeto ótimo;
- **$N_f \in N \wedge N_f = P_f$** , isto é, o ponto final tem de ser o último vértice na sequência ordenada de vértices que representa o trajeto ótimo;
- O **meio de locomoção** escolhido pelo usuário pode ser “a pé/bicicleta” v “carro” v “transportes públicos”.

Função objetivo

A função objetivo deve retornar um itinerário ótimo que inclua as atrações mais adequadas ao perfil do usuário e que possa ser realizado no tempo indicado, de uma origem a um destino final, indicado pelo utilizador.

A solução ótima maximiza o número de atrações que o usuário consegue visitar dentro das suas restrições de tempo e preferências, encontrando o caminho mais curto entre os pontos turísticos o que consequentemente permite obter o trajeto mais rápido.

Sendo assim, a função objetivo é:

- Maximizar o número de pontos de interesse visitados, $V = \max(\text{size}(vPOI))$;
- $WeightTF \leq \text{tempo disponível do utilizador}$.

Perspetiva de solução

Técnicas de implementação

A resolução deste problema passa por um conjunto de fases sucintamente descritas de seguida.

1. Pré-processamento do grafo

Nesta fase irá ser removida informação desnecessária e redundante.

Tendo em atenção o modo de locomoção do utilizador, serão tomadas diferentes medidas:

- Se o utilizador tiver selecionado transportes públicos, atendendo ao exemplo disponibilizado (figura 1) e sabendo o ID do nó mais próximo das paragens disponíveis, no grafo principal, o atributo *type* deste nó tomará o valor “*pt_stop*”.
- Se o modo de locomoção escolhido pelo utilizador for carro ou a pé, nada será feito nesta fase.

```
METRO:  
(ID nó mais próximo, Nome Paragem, Lista de Linhas)  
(698819576, 'Estádio do Dragão', ['A', 'B', 'E', 'F'])
```

Figura 1 Exemplo da informação relativa a transportes públicos

2. Verificar se é viável a deslocação entre dois pontos

Inicialmente, após serem fornecidos os pontos de partida e de chegada, é necessário verificar se existe pelo menos um caminho possível entre esses pontos, não sendo considerados os pontos de interesse nem o tempo máximo de deslocação.

Nesta fase, não se pretende guardar o caminho encontrado nem o otimizar, mas sim garantir que não estão a ser consideradas zonas inacessíveis e que por isso é possível chegar ao destino. Caso não seja, é necessário informar o utilizador.

3. Encontrar o caminho mais curto entre dois pontos num grafo não dirigido

Na generalidade dos problemas de trajetos, interessa não só encontrar um trajeto que vá de encontro aos requisitos do utilizador, mas também o melhor percurso possível, minimizando a distância percorrida e a duração.

Nesta iteração, ainda não vão ser tidos em conta os pontos de interesse do usuário, focando apenas em otimizar o percurso possível entre dois pontos, isto é, chegar do ponto inicial ao destino dentro dos limites de tempo desejado.

No caso de não haver nenhum caminho possível de se realizar dentro do tempo fornecido, será exposto o melhor caminho conseguinte.

4. Encontrar o máximo de caminhos possíveis cuja soma de tempo não exceda o especificado pelo utilizador

Tendo a certeza da existência de um caminho possível dentro do tempo restringido (ou não, resolvido como esclarecido no ponto anterior), interessa agora encontrar um caminho que maximize o número de pontos de interesse que conferem as preferências do utilizador.

Serão analisados os caminhos possíveis que obedeçam ao modo de locomoção indicado.

5. Maximizar o número de pontos de interesse visitados

Por fim, dentro dos obtidos, é necessário escolher qual o caminho que abrange o maior número de pontos de interesse do usuário.

É importante referir que, ao longo destas fases, é avaliada a conectividade do grafo, uma vez que certas áreas podem encontrar-se inacessíveis.

Algoritmos a serem considerados

A análise da possibilidade de chegar da origem ao destino passa por uma pesquisa no grafo a partir do vértice de origem, a qual pode ser realizada utilizando o algoritmo de pesquisa em profundidade ou de pesquisa em largura.

1. Algoritmo de Pesquisa em Largura

A **pesquisa em largura** (figura 2) é um dos métodos mais simples para a exploração de um grafo e é a base para muitos outros algoritmos, como o de Dijkstra.

Dado um vértice origem, explora-se sistematicamente as arestas do grafo, descobrindo todos os vértices alcançáveis a partir de s (vértices adjacentes), passando posteriormente à exploração do vértice seguinte. Isto é, encontra-se todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$. Para tal, é utilizada uma

```
BFS(G, s):  
1.  for each  $v \in V$  do discovered( $v$ )  $\leftarrow$  false  
2.   $Q \leftarrow \emptyset$   
3.  ENQUEUE( $Q, s$ )  
4.  discovered( $s$ )  $\leftarrow$  true  
5.  while  $Q \neq \emptyset$  do  
6.     $v \leftarrow$  DEQUEUE( $Q$ )  
7.    pre-process( $v$ )  
8.    for each  $w \in \text{Adj}(v)$  do  
9.      if not discovered( $w$ ) then  
10.        ENQUEUE( $Q, w$ )  
11.        discovered( $w$ )  $\leftarrow$  true  
12.    post-process( $v$ )
```

Figura 2 Pseudocódigo do algoritmo de pesquisa em largura

fila em que se processa o vértice da frente da mesma e à qual são adicionados no fim os vértices descobertos através desse processamento.

Este algoritmo tem uma complexidade temporal de $O(|E|+|V|)$, em que $|E|$ é o número de arestas do grafo e $|V|$ o número de vértices. Já em termos de complexidade espacial, este algoritmo tem uma complexidade $O(|V|)$, onde $|V|$ representa o número total de vértices do grafo.

2. Algoritmo de Pesquisa em Profundidade

O algoritmo de pesquisa em profundidade (figura 3) consiste em explorar todas as arestas a partir do último vértice encontrado, sendo implementado de forma recursiva e utilizando o método de *backtracking*. Quando todas as arestas de um vértice forem exploradas, retorna e explora as restantes arestas do vértice que o antecedia. O algoritmo aprofunda a pesquisa até que encontre o vértice pretendido.

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (∀ v ∈ V)

DFS(G):
1. for each v ∈ V
2.   visited(v) ← false
3. for each v ∈ V
4.   if not visited(v)
5.     DFS-VISIT(G, v)

DFS-VISIT(G, v):
1. visited(v) ← true
2. pre-process(v)
3. for each w ∈ Adj(v)
4.   if not visited(w)
5.     DFS-VISIT(G, w)
6. post-process(v)
```

Figura 3 Pseudocódigo do algoritmo de pesquisa em profundidade

Este algoritmo tem uma complexidade temporal de $O(|E|+|V|)$, onde $|E|$ é o número total de arestas do grafo e $|V|$ o número total de vértices do grafo. A sua complexidade espacial é de $O(|V|)$.

3. Algoritmo de Dijkstra

O algoritmo de *Dijkstra* é um algoritmo ganancioso que tem como objetivo calcular o caminho mais curto entre dois vértices de um grafo dirigido pesado, sem arestas de peso negativo.

Este algoritmo (figura 4) é semelhante ao de pesquisa em largura, diferenciando-se no facto de utilizar uma fila de prioridade (alterável) como estrutura de dados auxiliar para guardar a ordem dos próximos vértices a pesquisar, na qual se prioriza os vértices cuja soma do peso das arestas do caminho origina uma distância mínima (em vez de serem processados pela ordem em que foram descobertos, numa fila simples). É um algoritmo ganancioso pois em cada passo procura maximizar o ganho imediato, ou seja, minimizar a distância entre a origem e o destino.

```
DIJKSTRA(G, s): // G=(V,E), s ∈ V
1. for each v ∈ V do
2.   dist(v) ← ∞
3.   path(v) ← nil
4. dist(s) ← 0
5. Q ← ∅ // min-priority queue
6. INSERT(Q, (s, 0)) // inserts s with key 0
7. while Q ≠ ∅ do
8.   v ← EXTRACT-MIN(Q) // greedy
9.   for each w ∈ Adj(v) do
10.    if dist(w) > dist(v) + weight(v,w) then
11.      dist(w) ← dist(v) + weight(v,w)
12.      path(w) ← v
13.    if w ∉ Q then // old dist(w) was ∞
14.      INSERT(Q, (w, dist(w)))
15.    else
16.      DECREASE-KEY(Q, (w, dist(w)))
```

Figura 4 Pseudocódigo do algoritmo de Dijkstra

Em cada vértice processado é guardada a informação relativa ao vértice que o antecede no caminho e, após ser encontrado o vértice final na fila de prioridade,

percorre-se o caminho encontrado no sentido contrário, guardando-o para o retornar, até se regressar ao vértice inicial.

Para analisar a eficiência deste algoritmo, considere-se V o conjunto dos vértices do grafo e E o conjunto das arestas. Sendo que o número de extrações e inserções na fila de prioridades é $|V|$ e cada uma destas operações pode ser realizada em tempo logarítmico no tamanho da fila, que é no máximo $|V|$, o tempo de execução das extrações e inserções na fila de prioridades é de $O(|V| * \log |V|)$.

A operação de reordenação dos vértices na fila de prioridade é feita, no pior caso, uma vez por cada aresta, ou seja, $|E|$ vezes, e pode ser realizada em tempo logarítmico no tamanho da fila, que no máximo é $|V|$, resultando numa complexidade $O(|E| * \log |V|)$.

Assim, pode-se afirmar que a complexidade temporal do **algoritmo de Dijkstra** é $O((|V|+|E|) * \log |V|)$ e a complexidade espacial de $O(|V|)$.

4. Algoritmo A*

O **algoritmo A*** (figura 5) constitui uma otimização do algoritmo de *Dijkstra* que permite um melhoramento (*speedup*) moderado ao utilizar uma função heurística para orientar a busca do vértice de destino, não garantindo, no entanto, que o caminho encontrado seja o ótimo.

O **algoritmo A*** apenas difere no de *Dijkstra* na ordenação dos vértices na fila de prioridade, dando prioridade aos vértices que se encontram mais próximos do destino ao somar à distância mínima conhecida entre o vértice atual e a origem, o valor da estimativa por baixo da distância mínima do próprio vértice ao vértice de chegada.

Pode-se garantir que a solução encontrada será a ótima se, por exemplo, os pesos das arestas forem distâncias em *km* e a estimativa da distância do vértice atual ao destino for calculada usando a distância Euclidiana (em linha reta) entre os vértices.

A complexidade espacial deste algoritmo é $O(|V|)$, sendo $|V|$ o número total de vértices do grafo e a sua complexidade temporal é de $O((|V| + |E|) * \log |V|)$.

```

AStar(G, s, d): // h(v,f) – euclidianDistance

    for each v ∈ V do
        dist(v) ← ∞
        path(v) ← null

    dist(s) ← h(s,d)
    Q ← ∅ //min-priority queue
    INSERT(Q, (s,0)) //inserts s with key 0

    while Q ≠ ∅ do
        v ← EXTRACT-MIN(Q) //greedy

        for each w ∈ Adj(v) do
            f = dist(v) + weight(v,w) – h(v, d) + h(w,d)

            if dist(w) > f then
                dist(w) ← f
                path(w) ← v

                if w ∉ Q then //old sit(w) was ∞
                    INSERT(Q, (w,f))

            else

```

Figura 5 Pseudocódigo do algoritmo de A*

Casos de utilização e funcionalidades

Nesta aplicação, pretende-se utilizar uma interface intuitiva com diversos menus onde o utilizador poderá escolher as opções que mais se adequam aos seus interesses.

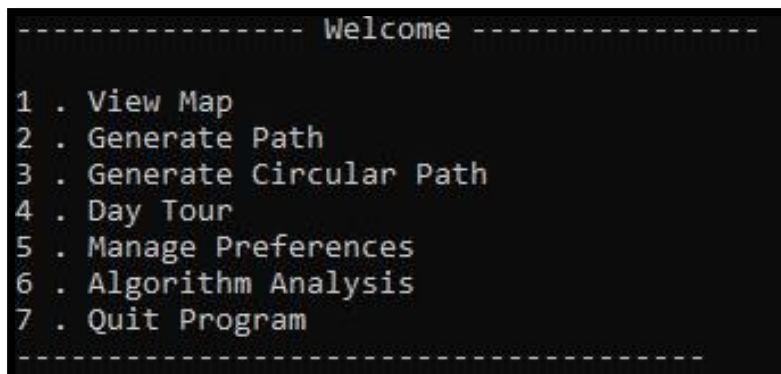
O objetivo é permitir a **visualização de mapas** e a **navegação entre locais**. No primeiro caso, serão pedidas algumas **restrições espaciais** para mostrar o mapa com a informação pretendida. No segundo caso, o utilizador terá de preencher um formulário onde irá especificar as suas **preferências** tais como tempo disponível, meio de transporte a utilizar, pontos de interesse, pontos de partida e de destino.

Na navegação entre locais, após se obter a informação necessária, será indicado se existe ou não um caminho possível entre o ponto inicial e o ponto final e, caso exista, apresenta-se o **melhor caminho**, ou seja, o caminho com o maior número de pontos de interesse que possam ser visitados no tempo fornecido. Realça-se que, se não existir uma possibilidade dentro do tempo máximo escolhido, será apresentada uma mensagem de erro bem como a melhor alternativa possível.

Principais casos de uso implementados

Ao entrar na aplicação, o utilizador depara-se com um menu com as seguintes opções:

- Visualizar o mapa do Porto com alguns pontos reais assinalados;
- Gerar um caminho pela cidade que começa num ponto e termina noutro diferente do inicial, cuja duração está dentro do intervalo a especificar e cujo meio de locomoção é escolhido pelo utilizador;
- Gerar um caminho circular, isto é, um caminho que começa e acaba no mesmo ponto, sendo que aqui o utilizador também especifica o tempo que tem disponível e qual o meio de locomoção (a pé/bicicleta ou carro) a utilizar;
- Ser surpreendido com um roteiro de dia inteiro passando pelos principais pontos emblemáticos da cidade do Porto;
- Gerir as suas preferências: aqui pode adicionar, remover ou visualizar as suas preferências especificadas, isto é, o tipo de pontos de interesse que deseja visitar;
- Analisar os tempos de execução de diferentes algoritmos interessantes para o contexto do problema em diferentes situações.



```
----- Welcome -----  
1 . View Map  
2 . Generate Path  
3 . Generate Circular Path  
4 . Day Tour  
5 . Manage Preferences  
6 . Algorithm Analysis  
7 . Quit Program  
-----
```

Figura 6 Imagem do menu implementado

Estruturas de dados utilizadas

Ao longo do programa são utilizadas as seguintes estruturas de dados:

Stop: esta classe guarda a informação de uma estação de metro, ou seja, o seu id, o seu nome e o número da paragem (definida em *Stop.h*);

Edge: a classe *Edge* que guarda o ponto de início, o ponto de fim e o peso de uma aresta (definida em *Graph.h*);

Vertex: esta classe guarda as coordenadas de um ponto, o seu id, o tipo de ponto de interesse que representa e vários atributos necessários para os algoritmos (definida em *Graph.h*);

Graph: esta classe guarda um vetor de vértices, um mapa de que guarda o id de um vértice e o *index* deste vértice no vetor de vértices, para tornar mais rápido o acesso à informação dos vértices. Guarda também um vetor de estações de metro e um mapa com o id de uma estação de metro e o *index* desta estação (*Stop*) no vetor de estações, um mapa com o id e o nome dos pontos de interesse principais da cidade. Para além disso, tem o máximo e mínimo das coordenadas x e y para poder redimensionar a janela do *GraphViewer* e mais alguns atributos necessários para os algoritmos (definida em *Graph.h*);

As classes *Graph*, *Edge* e *Vertex* constituem uma adaptação do código fornecido nas aulas práticas.

PoiEdge: esta classe guarda arestas e vai ser utilizada na classe seguinte, sendo que guarda o id do vértice de destino, o peso desta aresta e o caminho entre os pontos de interesse que são ligados por esta aresta (definida em *PoiEdge.h*);

PoiVertex: esta classe guarda vértices e vai ser utilizada na classe seguinte, sendo que guarda o seu id e as arestas que começam neste vértice (definida em *PoiVertex.h*);

PoiGraph: esta é uma classe auxiliar mais simples que guarda a informação dos caminhos já calculados entre os pontos já processados. Como o problema se baseia na combinação de diferentes caminhos, acaba-se por repetir muitas vezes o cálculo do mesmo caminho entre dois pontos. Com esta classe garantimos que apenas é necessário calcular cada caminho apenas uma vez, sendo depois possível aceder ao caminho já calculado (definida em *PoiGraph.h*);

GraphViewer: esta classe foi fornecida pelos docentes e é utilizada para visualização dos grafos e dos caminhos obtidos;

Queue: esta estrutura de dados é utilizada para guardar os caminhos ao longo do programa;

MutablePriorityQueue: esta estrutura de dados é utilizada para os algoritmos *Dijkstra* e *A**;

Map: esta estrutura de dados é utilizada em algumas das estruturas de dados anteriormente referidas;

ClientInfo: esta estrutura de dados guarda toda a informação necessária relativa às escolhas e preferências do utilizador (definida em *ClientInfo.h*).

```
class POIGraph {
    vector<POIVertex*> vertexSet;    // vertex set
    unordered_map<int, int> vertexMap; //<id, index>

public:
    POIVertex *findVertex(const int &id) const;
    bool addVertex(const int &id);
    bool addEdge(const int &source, const int &dest, double w, queue<Vertex<coord>*> path);
    double getDist(Graph<coord> &g, const int &source, const int &dest, bool biDir);
    queue<Vertex<coord>*> getPath(Graph<coord> &g, const int &source, const int &dest, bool biDir);
};

class POIVertex {
    int id;
    vector<POIEdge> adj;    // outgoing edges

    void addEdge(const int &dest, double dist, queue<Vertex<coord>*> path) {
        adj.push_back(POIEdge(dest, dist, path));
    }

public:
    POIVertex(int id) {this->id = id;}
    friend class POIGraph;
};

class POIEdge {
    int destID;    // destination vertex
    double dist;    // edge weight
    queue<Vertex<coord>*> path;

public:
    POIEdge(int destID, double dist, queue<Vertex<coord>*> path);
    int getDest();
    double getDist();
    queue<Vertex<coord>*> getPath();
    friend class POIGraph;
    friend class POIVertex;
};
```

Figura 7 Imagem de algumas das estruturas de dados utilizadas

Implementação

O problema proposto foi subdividido em três subproblemas, sempre tentando maximizar o número de pontos de interesse visitados e dentro das restrições/preferências especificadas pelo utilizador:

- Caminho de um ponto inicial a um ponto final, cujo meio de locomoção é a pé/bicicleta ou de carro;
- Caminho de um ponto inicial a um ponto final, utilizando transportes públicos (nesta implementação, de modo a tornar mais simples e menos demoroso todo o processamento de dados, apenas foi considerado o metro e apenas uma linha);
- Caminho circular, ou seja, começando e acabando no mesmo ponto, cujo meio de locomoção ou é a pé/bicicleta ou de carro.

Problema 1

Este primeiro subproblema foi resolvido com os seguintes passos:

1. Verificar se existe caminho entre o ponto inicial e o ponto final

Após esta verificação, caso não exista um caminho, o utilizador é notificado.

2. Verificar que, se existir caminho, o utilizador tem tempo para o percorrer

Se existir um caminho, é calculada a sua duração e caso o utilizador não tenha tempo suficiente disponível, o utilizador é notificado sendo retornado o caminho mais rápido entre estes dois pontos bem como tempo mínimo necessário para percorrer este caminho no meio de locomoção especificado pelo utilizador.

3. Pré-processamento do grafo

Nesta fase, é efetuado um pré-processamento do grafo. Aqui são filtrados os pontos de interesse a serem processados na fase seguinte. São descartados os pontos de interesse que não se consegue atingir a partir do ponto de início, os que não estejam especificados nas preferências do utilizador e os cujo tempo de visita excede o tempo disponível do utilizador.

4. Cálculo de melhor caminho

Por fim, processam-se todos os pontos de interesse não descartados em fases anteriores. Começa-se por verificar se tem tempo de ir da origem ao ponto, depois se

tem tempo de o visitar e por fim se tem tempo de ir do ponto ao destino. Cada vez que uma das condições anteriores não é verificada, o ponto é ignorado. De seguida, guardando o caminho do ponto de início ao ponto que está a ser processado, retira-se dos pontos a serem processados naquela iteração os pontos já visitados no caminho da origem ao ponto e os pontos que não são acessíveis a partir deste ponto e é feito o mesmo processamento com estes pontos já filtrados, tendo como ponto de origem o ponto a ser processado nesta iteração e mantendo-se o destino. Ao fim de cada iteração, verifica-se se foi possível encontrar um caminho e caso tenha sido, compara-se com o anteriormente guardado e se for melhor substitui-o.

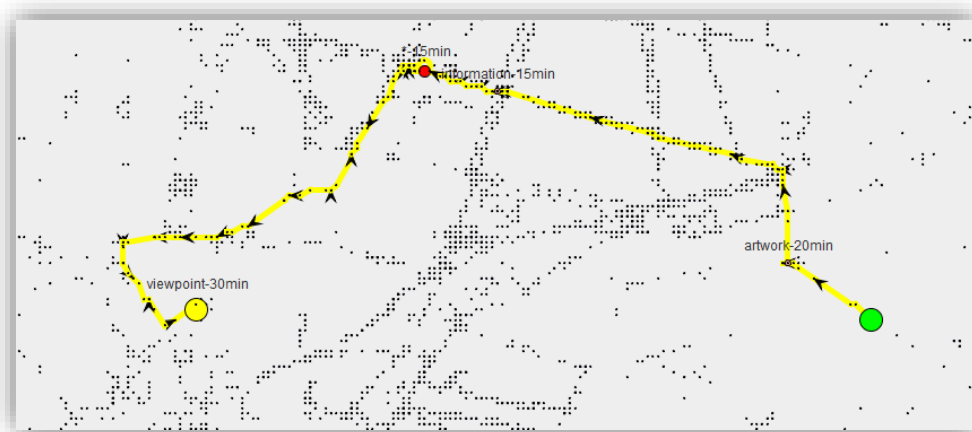


Figura 8 Exemplo de um caminho gerado pela aplicação

Problema 2

Por sua vez, este problema é resolvido com os seguintes passos:

1. Verificar quais as estações de metro mais próximas do ponto de início e do ponto de destino

Neste passo calculam-se as distâncias dos pontos a todas as estações de metro existentes, associando a cada ponto a estação de metro mais perto. Se no fim a estação de metro associada ao ponto de início for igual à estação associada ao ponto de destino, o utilizador é notificado que não compensa utilizar transportes públicos e que deve fazer este caminho a pé, sendo encaminhado ao menu anterior.

2. Verificar se tem tempo de realizar a viagem de metro

É verificado se o tempo da viagem é igual ou superior ao tempo disponível, sendo que, nesse caso, o utilizador é notificado de que não tem tempo para realizar a viagem que pretendia, voltando ao menu anterior.

3. Verificar se tem tempo de ir dos pontos às estações

Após o passo anterior, é verificado se, para além de ter tempo de realizar a viagem de metro, tem também tempo de ir dos pontos de início e destino às respectivas estações de metro. Caso tal não aconteça, é notificado, voltando ao menu anterior.

4. Cálculo do melhor caminho

Nesta fase, é dado um tempo proporcional à distância para o caminho do ponto do início e do ponto de destino às estações associadas. Os pontos de interesse são filtrados de forma semelhante à resolução do problema anterior: são descartados os pontos de interesse que não se consegue atingir a partir do ponto, os que não estejam especificados nas preferências do utilizador e os cujo tempo de visita excede o tempo disponível do utilizador. Após esta filtragem, é realizado o passo 4 da resolução do problema anterior para o caminho do ponto de início à sua estação mais próximas e da estação mais próxima do ponto de destino até ao ponto de destino. Por fim, estes dois caminhos são adicionados ao caminho realizado de metro.

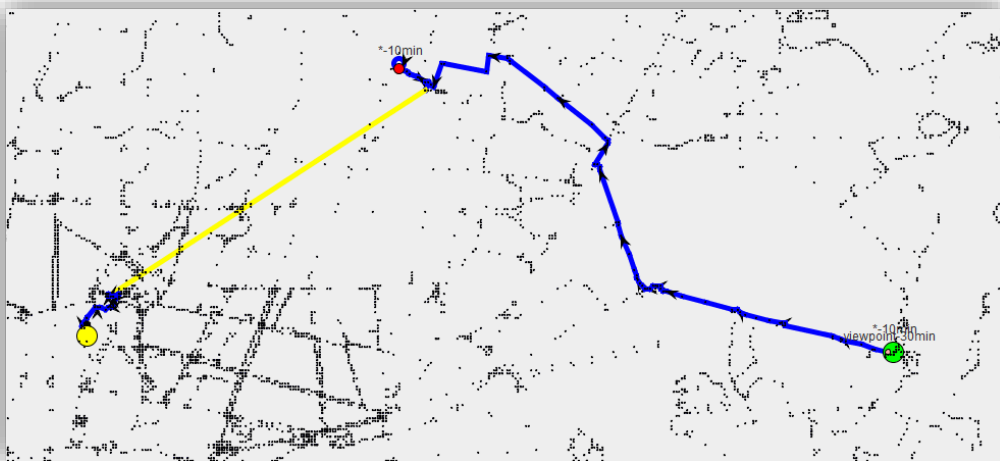


Figura 9 Exemplo de um caminho com metro (linha azul) gerado pela aplicação

Problema 3

As fases de resolução deste problema são:

1. Verificar quais os pontos das preferências do utilizador acessíveis a partir do ponto escolhido no tempo disponível

De forma semelhante aos problemas anteriormente referidos, os pontos de interesse são filtrados, sendo descartados os pontos de interesse que não se consegue atingir a partir do ponto, os que não estejam especificados nas preferências do utilizador e os cujo tempo de visita excede o tempo disponível do utilizador. Caso não haja pelo menos um ponto acessível, o utilizador é notificado de que não consegue visitar nenhum ponto que pretende no tempo que tem disponível.

2. Cálculo do melhor caminho

Todos os pontos não descartados na parte anterior vão ser processados. Começando por verificar se o utilizador tem tempo de ir até ao ponto e visitá-lo, caso contrário, passa-se a processar o ponto seguinte. Se passar esta verificação, são eliminados dos pontos a serem processados nesta iteração todos os que se encontram no caminho do ponto de início ao ponto que está a ser processado nesta iteração. É realizada a mesma filtragem aos pontos de interesse e é realizado algo semelhante ao ponto 4 do problema 1. Se for encontrado um caminho na iteração em que se encontra, é verificado se é melhor que o anteriormente guardado e se for substituí-o. São, por fim, realizadas as restantes iterações até que não haja mais pontos a processar. Se no fim não houver nenhum caminho, o utilizador é notificado de que não há nenhum caminho no tempo que tem disponível para visitar um ponto do seu interesse.

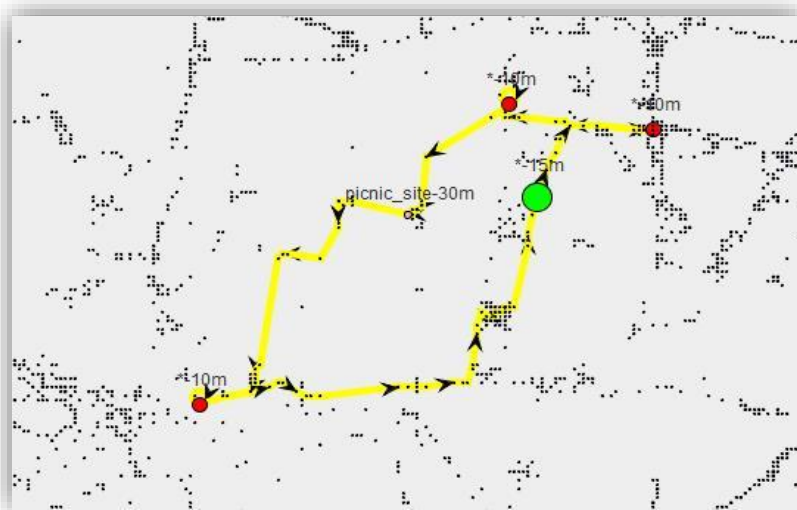


Figura 10 Exemplo de um caminho circular gerado pela aplicação

Algoritmos efetivamente implementados

Para além dos algoritmos referidos na secção **Algoritmos a serem considerados**, foram implementados mais alguns. De seguida, vão ser apresentados todos os algoritmos implementados e os seus casos de utilização, sendo que não vai ser repetido o pseudocódigo e análises temporais e espaciais dos já anteriormente referidos.

Algoritmo de Pesquisa em Largura (BFS)

Foram implementadas três variantes da *BFS* para serem utilizadas ao longo do programa de modo a torná-lo mais eficiente. Um dos casos é a função *bfs()* que verifica se existe um caminho entre um ponto de origem (*source*) e um ponto de destino (*dest*), retornando o primeiro caminho que encontrar. É muitas vezes utilizada para verificar se se deve ou não processar o ponto de interesse seguinte como candidato a fazer parte do caminho final. Existe também a *bfsAll()* que retorna todos os pontos acessíveis a partir de uma origem, o que ajuda na filtragem dos pontos de interesse a processar, uma vez que, se não for possível atingir um dado ponto de interesse a partir de a origem do nosso caminho, então este não deve ser processado, já que não fará parte do caminho final. Por fim, a função *bfsAllPOI()* que, de modo semelhante à função anterior, filtra os pontos interesse. No entanto, esta função faz uma filtragem mais específica, recebendo o tipo dos pontos de interesse nas preferências do utilizador e o tempo que este tem disponível, descarta os pontos que não consegue atingir a partir da origem, os que não tem tempo de visitar (o seu tempo de visita é superior ao tempo disponível do utilizador) e os pontos que não são do tipo de interesse das preferências do utilizador.

Algoritmo de Pesquisa em Profundidade (DFS)

Este algoritmo é utilizado para comparação e análise dos tempos de execução dos diferentes algoritmos em diferentes situações.

Algoritmo de Dijkstra

O algoritmo de Dijkstra é utilizado para comparação e análise dos tempos de execução dos diferentes algoritmos em diferentes situações.

Algoritmo A*

O algoritmo é utilizado para calcular o caminho mais rápido entre dois pontos quando o meio de transporte é carro ou metro, uma vez que se mostrou ser mais eficiente do que *Dijkstra* no contexto do nosso problema. No entanto, é de realçar que, quando os caminhos são muito reduzidos, o *Dijkstra* torna-se mais eficiente que o algoritmo de A*. É também utilizado para comparação e análise dos tempos de execução dos diferentes algoritmos em diferentes situações.

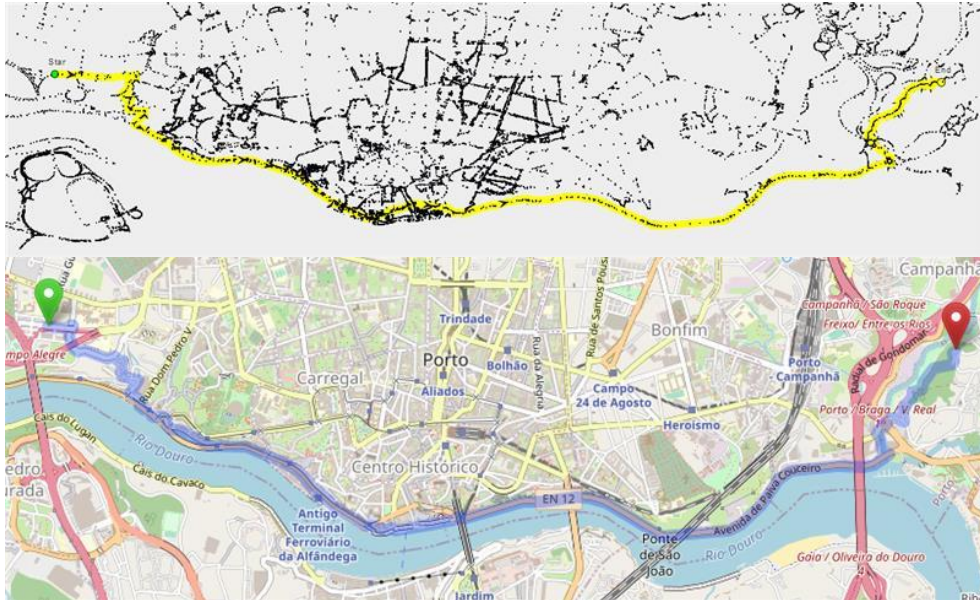


Figura 11 Exemplo de um caminho entre dois pontos gerado por A* em comparação com o gerado pelo OpenStreetMaps (<https://www.openstreetmap.org/>)

Algoritmos de pesquisa bidirecional: A* e Dijkstra

A pesquisa bidirecional é um algoritmo de pesquisa em grafos que calcula o caminho mais curto entre um ponto de origem a um ponto de destino. Este algoritmo utiliza um grafo auxiliar sendo que faz uma *forward search* do ponto de origem ao ponto de destino num grafo e uma *backward search* do ponto de destino ao ponto de origem no outro, sendo que o algoritmo termina quando os dois grafos intersetarem.

```
BIDIRECTIONAL SEARCH
1   $Q_I.Insert(x_I)$  and mark  $x_I$  as visited
2   $Q_G.Insert(x_G)$  and mark  $x_G$  as visited
3  while  $Q_I$  not empty and  $Q_G$  not empty do
4    if  $Q_I$  not empty
5       $x \leftarrow Q_I.GetFirst()$ 
6      if  $x = x_G$  or  $x \in Q_G$ 
7        return SUCCESS
8    forall  $u \in U(x)$ 
9       $x' \leftarrow f(x, u)$ 
10     if  $x'$  not visited
11       Mark  $x'$  as visited
12        $Q_I.Insert(x')$ 
13     else
14       Resolve duplicate  $x'$ 
15  if  $Q_G$  not empty
16     $x' \leftarrow Q_G.GetFirst()$ 
17    if  $x' = x_I$  or  $x' \in Q_I$ 
18      return SUCCESS
19    forall  $u^{-1} \in U^{-1}(x')$ 
20       $x \leftarrow f^{-1}(x', u^{-1})$ 
21      if  $x$  not visited
22        Mark  $x$  as visited
23         $Q_I.Insert(x)$ 
24      else
25        Resolve duplicate  $x$ 
26  return FAILURE
```

Figura 12 Pseudocódigo do algoritmo de pesquisa bidirecional

Ambos os algoritmos são em tudo semelhante ao pseudocódigo apresentado, distinguindo-se pela forma que o cálculo dos pesos das arestas é efetuado, tal como os seus algoritmos de pesquisa unidirecional.

O algoritmo de pesquisa bidirecional de A* é utilizado para encontrar o caminho mais rápido entre dois pontos quando o meio de locomoção é a pé/bicicleta já que, neste caso, o grafo é bidirecional.



Figura 13 Exemplo de um caminho gerado pelo A* bidirecional em comparação com o gerado pelo OpenStreetMaps (<https://www.openstreetmap.org/>)

Ambos os algoritmos de pesquisa bidirecional de A* e Dijkstra são utilizados para comparação e análise dos tempos de execução dos diferentes algoritmos em diferentes situações.

Algoritmo de Floyd-Warshall

Para além dos algoritmos mencionadas na primeira parte do relatório foi também implementado o algoritmo de *Floyd-Warshall*.

O algoritmo de *Floyd-Warshall* também é um algoritmo para encontrar o caminho mais curto entre todos os vértices de um grafo pesado. Funciona tanto para grafos orientados como não orientados, no entanto, não funciona para grafos com ciclos negativos.

Este algoritmo preenche duas matrizes n por n , onde n é o número de vértices do grafo. As linhas são indexadas por i e as colunas por j , em que i e j são os vértices do grafo.

```
for i := 1 to n do
  for j := 1 to n do
    cost[i, j] := c[i, j]; // Let c[u, u] := 0
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        sum = cost[i, k] + cost[k, j];
        if(sum < cost[i, j]) then cost[i, j] := sum;
```

Figura 14 Pseudocódigo do algoritmo de *Floyd-Warshall*

Numa das matrizes, a célula $A[i][j]$ representa a distância mínima do vértice i ao vértice j , sendo 0 se i for igual a j e infinito se não houver caminho entre os dois vértices. Na outra matriz, a célula $B[i][j]$ representa o caminho mais curto do vértice i ao vértice j .

A complexidade temporal deste algoritmo é de $O(|V|^3)$ e a complexidade espacial $O(|V|^2)$, sendo $|V|$ o número total de vértices do grafo.

Ao fim de alguns testes, percebeu-se que algoritmo, apesar de ter um tempo de resposta mais rápido que o *Dijkstra* ($O(1)$ em relação a $O((|V|+|N|)\log(|V|))$), o seu pré-processamento é bem mais pesado, sendo que, no contexto do problema, o grafo é constituído por um número muito elevado de vértices e, portanto, o algoritmo *Dijkstra* tem um menor tempo de execução.

Sendo assim, este algoritmo é apenas utilizado para comparação e análise dos tempos de execução dos diferentes algoritmos em diferentes situações.

Análise temporal empírica dos algoritmos implementados

Comparação entre Floyd-Warshall, Dijkstra e A*

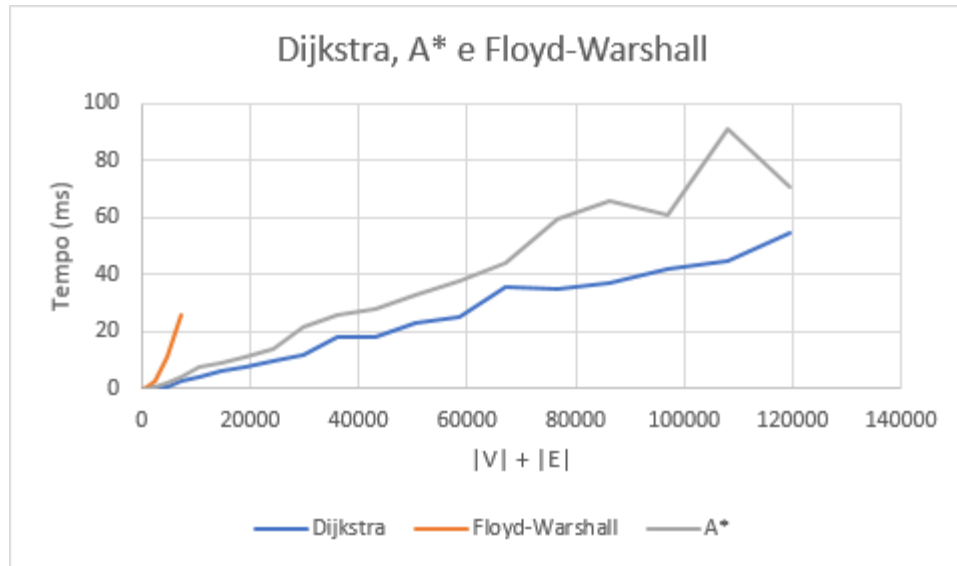


Figura 15 Comparação de *Floyd-Warshall*, *Dijkstra* e *A**

Comparação entre DFS e BFS

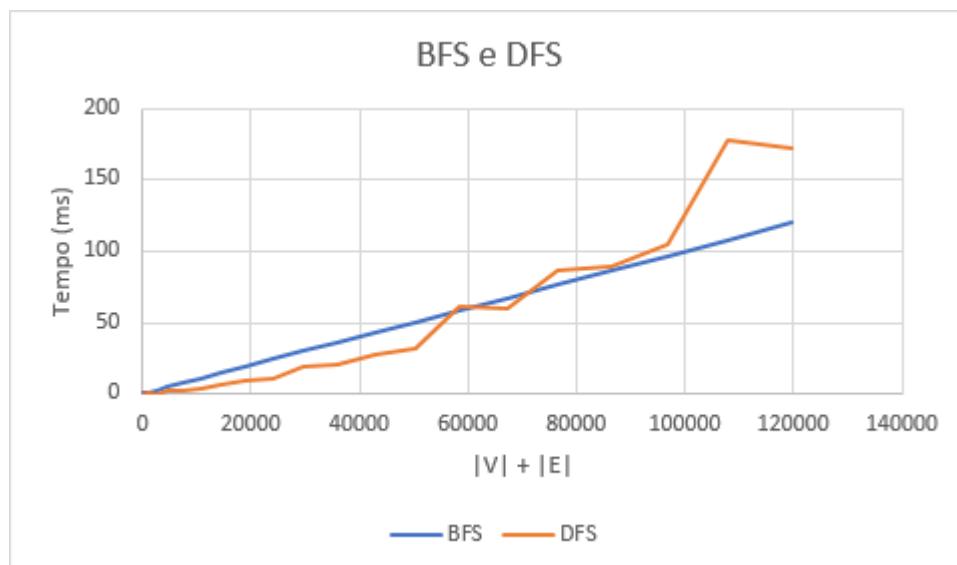


Figura 16 Comparação de *DFS* e *BFS*

Comparação entre algoritmos de pesquisa bidirecional: Dijkstra e A*

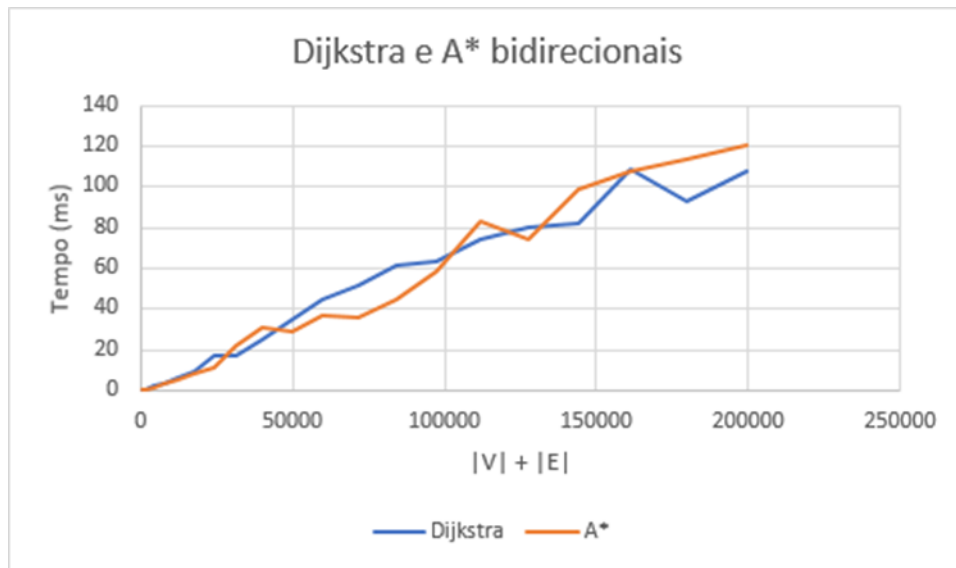


Figura 17 Comparação de algoritmos de pesquisa bidirecional

Conectividade dos grafos utilizados

Um grafo é fortemente conexo quando entre quaisquer dois vértices existe sempre um caminho.

Para testar a conectividade de um grafo não dirigido, deve ser realizado o algoritmo de *BFS* ou *DFS* começando num vértice qualquer e, se a *BFS* ou a *DFS* visitar todos os vértices existentes, então o grafo é fortemente conexo.

A ideia é que, se todos os vértices podem ser atingidos a partir do vértice v e todos os vértices podem atingir v , então o grafo é fortemente conexo.

Utilizando os ficheiros fornecidos de “*porto_full_**” e “*porto_strong_**”, foram realizados testes à conectividade destes grafos e confirmou-se que o grafo nos ficheiros “*porto_strong_**” é um grafo fortemente conexo, ao contrário do grafo nos ficheiros “*porto_full_**”.

```
Running bfs for each node...  
  
Average percentage of nodes reached: 40.2442
```

Figura 18 Resultado dos testes de conectividade com os ficheiros “*porto_full_**”

Notas de implementação

A *BFS* está a ser utilizada para encontrar todos os pontos acessíveis a partir de um ponto de início, no tempo disponível, sendo que coloca os pontos mais próximos primeiro seguindo até aos mais afastados. Por isso, com alguns testes, compreendeu-se que, a partir de certo ponto, nos pontos finais do vetor retornado pela *BFS*, estes pontos já estão bastante longe do ponto inicial e que, muitas vezes, se o tempo disponível fosse relativamente baixo, não compensava estar a processar estes pontos já que não tinha tempo de os atingir.

Para resolver este problema, foi criado um *counterfactor* que desistia de procurar mais caminhos a partir do momento em que um x número de pontos processados não tinha tempo de chegar da origem ao ponto. Como os seguintes pontos estavam ainda mais longe, não fazia sentido explorá-los uma vez que não iriam conseguir alcançar o destino no tempo disponível.

Depois de feitos diversos testes em termos de tempo de execução dos algoritmos criados em resposta ao problema, decidiu-se que o algoritmo devia de “desistir” de encontrar um caminho caso 30% dos pontos a processar falhassem em encontrar um caminho possível no tempo disponível.

Os testes feitos demonstraram que com a rejeição ser feita aos 50% o ganho já era notório em termos de tempo de execução e que 30% mostrou-se ser o ideal, já que com um número inferior poderia se estar a descartar pontos importantes para a resolução do problema.

```
Loading Path...
Execution time processing all points: 17552 milliseconds
found path with 1.4871km and 3 points

Loading Path...
Execution time giving up when 50% of points fail: 9691 milliseconds
found path with 1.4871km and 3 points

Loading Path...
Execution time giving up when 30% of points fail: 7101 milliseconds
found path with 1.4871km and 3 points

Loading Path...
Execution time giving up when 10% of points fail: 3545 milliseconds
found path with 1.68254km and 3 points

Loading Path...
Execution time giving up when 5% of points fail: 1310 milliseconds
found path with 1.0998km and 1 points

Process finished with exit code 0
|
```

Figura 19 Resultado dos testes do *counterFactor*

Conclusão

1ª Parte

Neste relatório, foi discutida uma solução possível para o problema proposto, analisando pormenorizadamente alguns algoritmos, apresentados nas aulas, que possam vir a ser utilizados como uma estratégia para o desenvolvimento do projeto.

Tendo o hábito de realizar relatórios durante ou após a implementação do código, a maior dificuldade encontrada foi conseguir organizar o processo de desenvolvimento sem de facto o implementar.

Toda a pesquisa necessária e a elaboração do relatório foram igualmente divididas pelos três membros do grupo, estando em constante comunicação e discussão. Desta forma, o esforço dedicado por cada elemento é de $\frac{1}{3}$.

2ª Parte

Como era expectável ao fim da primeira entrega do trabalho, a ideia de resolução do problema não estava totalmente correta. Obrigou a muitos testes e implementação diferentes ideias, de modo a escolher a que melhor se adequava ao contexto do problema.

É de salientar que se tornou complicada a realização do trabalho quando o objetivo principal não foi totalmente explicado desde o início o que levou a bastantes mudanças da implementação do mesmo na semana de entrega. Outro fator que dificultou o projeto foi o facto de os mapas corretos só terem sido disponibilizados poucos dias antes da entrega uma vez que sendo bastante mais pesados levou a grandes problemas de execução do programa em tempo útil.

Tal como na entrega anterior, o trabalho foi igualmente dividido pelos três membros do grupo, estando em constante comunicação e discussão ao longo de todo o seu desenvolvimento. O esforço dedicado por cada elemento é, portanto, de $\frac{1}{3}$.

Bibliografia

- Slides das aulas teóricas
- Relatórios fornecidos pelo docente
- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://pt.wikipedia.org/wiki/Busca_em_profundidade
- https://pt.wikipedia.org/wiki/Busca_em_largura
- https://pt.wikipedia.org/wiki/Algoritmo_A*
- <https://www.programiz.com/dsa/floyd-warshall-algorithm>
- <https://www.geeksforgeeks.org/bidirectional-search/>
- <https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/>