U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC 2020/2021

Concepção e Análise de Algoritmos

À procura de estacionamento

Abordagem formal e implementação

Turma 3, Grupo 1
Adelaide Santos up201907487@fe.up.pt
Bruno Mendes up201906166@fe.up.pt
Rita Mendes up201907877@fe.up.pt

24 de maio de 2021

Conteúdo

1. Introdução

Na procura de aliviar o tráfego citadino causado, em certa parte, pela procura incessante de estacionamento, pretende-se com este projeto implementar um sistema que construa um trajeto eficiente - com base em critérios ajustáveis - com passagem em parques de estacionamento, cujo preço é variável de parque para parque, perto de uma ou mais etapas do percurso do condutorou ou com base em, por exemplo, serviços de *drive through*, em que não seja preciso realizar a procura de estacionamento, apenas a passagem pelo ponto desejado.

A abstração deste problema no contexto de um grafo pesado dirigido resulta numa variante do *problema do caixeiro-viajante*, com uma nuance significativa: não há que passar em todas as etapas do percurso, mas em parques de estacionamento perto destas.

No contexto deste relatório, começa-se por explanar formalmente o problema em mãos, tendo em conta os diferentes critérios de que o condutor dispõe na escolha do seu percurso. Estando definido o objetivo, são propostas abordagens de resolução do problema, que se refletem em diferentes algoritmos, cuja complexidade será analisada, a fim de selecionar o mais adequado em cada situação.

Por fim, e tendo já implementado o programa, apresentam-se considerações relativas às escolhas dos algoritmos e respetiva performance.

2. Descrição do problema

2.1 O *problema do caixeiro-viajante*

No *problema do caixeiro-viajante*, um indivíduo visita um conjunto prescrito de cidades, voltando a casa, tal que a distância total percorrida seja a menor possível.

Este problema pode ser reduzido a um problema de grafos, em que cada ponto de estrada registado representa um nó, e respetivos caminhos perfazem arestas, com peso igual à distância a percorrer ou, noutra variante, ao tempo estimado na travessia entre os dois pontos.

Quando o conjunto de cidades a percorrer é reduzido - e admitindo que se sabe a distância correta entre estas -, uma simples análise combinatória é suficiente para obter a solução ótima. No entanto, se o conjunto é extenso e não se sabe *a priori* a distância em estrada entre cada cidade, como é o caso do problema em mãos, a complexidade aumenta exponencialmente.

Na verdade, o *problema do caixeiro-viajante* é considerado *NP-hard*: não existe um algoritmo conhecido capaz de resolver o problema em tempo polinomial. Consequentemente, qualquer solução num contexto real (em que há um número elevado de paragens) será uma aproximação, que se espera razoável, se a estratégia tomada for adequada.

2.2 O *problema do condutor citadino*

O *problema do condutor citadino*, batizado à falta de um nome mais expressivo recomendado pela bibliografia, é uma generalização do *problema do caixeiro-viajante*. Neste caso, não é necessário regressar ao ponto de partida após chegar ao destino e, mais significativamente, o utilizador substitui (salvo indicação do contrário) cada ponto do percurso por um parque de estacionamento próximo, onde estacionará o seu carro, fazendo o percurso até ao ponto exato a pé.

Esta diferença exige um pré-processamento na zona dos vértices em que se pretende parar, a fim de determinar o parque de estacionamento a recomendar ao condutor, que representará um acréscimo na complexidade do problema geral do caixeiro-viajante.

A procura far-se-á de modo a minimizar uma das seguintes variáveis:

- Distância percorrida a pé desde o parque até ao vértice
- Preço a pagar pelo estacionamento
- Distância percorrida de carro até ao parque de estacionamento

3. Formalização do problema

Antes de analisar os algoritmos candidatos para a resolução do problema, identificam-se, neste capítulo, os dados que o programa irá tratar - com as devidas propriedades e restrições -, o formato dos dados de saída e a função-objetivo.

3.1 Dados de entrada

$G_i(V_i, E_i)$ - Grafo dirigido pesado que representa o mapa

V_i - Conjunto de vértices que representam locais no mapa, que contêm:

- Y - Coordenadas euclidianas do ponto no mapa
- $Adj \subseteq E_i$ - Arestas adjacentes

E_i - Conjunto de arestas que representam caminhos entre locais, que contêm:

- W - Distância entre os vértices de partida e destino
- $Dest \in V_i$ - Vértice para o qual a aresta aponta

$P \subseteq V_i$ - Conjunto de parques de estacionamento, que contêm:

- C - Capacidade total
- N_{Vagas} - Número de lugares vagos
- P_{Base} - Preço base do estacionamento
- Q_{Disp} - Coeficiente a multiplicar pela percentagem de lugares ocupados para calcular o preço dinâmico

$S \in V_i$ - Ponto de partida do utilizador

$D \in V_i$ - Ponto de destino do utilizador

$I \subseteq V_i$ - Conjunto de pontos por onde o utilizador deseja passar antes do destino, que contêm:

- Q - Booleano se deve ser procurado um parque próximo

O - Parâmetro a otimizar: distância percorrida até ao parque de estacionamento, preço a pagar pelo estacionamento, distância a percorrer a pé até ao ponto de destino

Z - Limite de distância de um parque de estacionamento ao ponto onde há que realizar a tarefa

3.2 Dados de saída

$G_f(V_f, E_f)$ - Grafo dirigido pesado que representa a trajetória percorrida

V_f - Conjunto de vértices que representam os pontos onde parar, contendo:

- Y - Coordenadas euclidianas do ponto no mapa
- $C \subseteq E_f$ - Caminho mais curto (conjunto de arestas) até ao próximo ponto onde parar
- F - Distância a percorrer a pé até ao ponto onde há que realizar a tarefa (nula se não se estaciona neste ponto)

E_f - Conjunto ordenado de arestas do grafo

- W - Distância entre o vértice de partida e destino
- $Dest \in V_f$ - Vértice para o qual a aresta aponta

L_{carro} - Distância total percorrida de carro

L_{pe} - Distância total percorrida a pé

L_{preco} - Preço total gasto em estacionamento

3.3 Restrições

3.3.1 Sobre os dados de entrada

- $\forall x \in I \cup \{S, D\}$, x faz parte do mesmo componente fortemente conexo do grafo, garantindo que é possível transitar entre todos os pontos relevantes do percurso - partida, pontos intermédios e destino
- $\forall x \in E_i, x.W > 0$, ou seja, a distância entre vértices é maior ou igual a zero, dado que representa uma unidade de distância

- $\forall x \in P, x.C > 0$, ou seja, a capacidade total de cada parque de estacionamento é maior que zero
- $\forall x \in P, 0 \leq x.N_{Vagas} \leq x.C$, ou seja, o número de vagas num parque de estacionamento está entre zero e o valor da sua capacidade
- $\forall x \in P, x.P_{Base} \geq 0$, ou seja, o preço base do estacionamento é maior ou igual a zero
- $\forall x \in P, x.Q_{Disp} \geq 0$, ou seja, o coeficiente para calcular o preço dinâmico é maior ou igual a zero
- $Z > 0$, ou seja, o limite de distância entre um ponto de paragem e o respetivo parque é positivo (de preferência, um valor não muito elevado, para não aumentar significativamente o tempo de processamento)

3.3.2 Sobre os dados de saída

- $\forall x \in V_f, x \in V_i$, ou seja, todos os pontos onde parar pertencem ao conjunto de vértices inicial
- $\forall x \in V_f, x.F \geq 0$, ou seja, a distância a percorrer a pé até ao ponto onde há que realizar a tarefa é maior ou igual a zero
- $\forall x \in V_f, len(x.C) > 0$, ou seja, o próximo ponto não é o próprio vértice
- $\forall x \in E_f, x.W > 0$, pela mesma razão da anterior
- E_f forma um caminho contínuo, isto é, todo o nó é adjacente ao seguinte, se este existir
- $\forall x \in E_f, x \in E_i$, ou seja, cada aresta do grafo de saída pertence ao conjunto inicial de arestas
- $L_{carro} > 0$, isto é, a distância total percorrida é não nula
- $L_{pe} \geq 0$, ou seja, a distância total a pé é nula (caso não se tenha pretendido estacionar em nenhum ponto) ou positiva
- $L_{preco} \geq 0$, ou seja, ou preço dispendido em estacionamento é nulo (caso todos os parques sejam grátis, ou não se tenha estacionado) ou positivo
- $V_f[0] = S$, ou seja, o primeiro vértice do conjunto de pontos de paragem retornados é o ponto de partida
- $V_f[len(V_f) - 1] = D$, ou seja, o último vértice do conjunto de pontos de paragem retornados é o ponto de chegada

3.4 Função-objetivo

É importante não tomar cegamente que há que minimizar uma e uma só das variáveis referidas na descrição do problema. Se, por exemplo, o utilizador pretende minimizar o preço a pagar, estará disposto a abrir mão de alguma proximidade, mas não demasiado, ao ponto de ter de estacionar, por exemplo, numa localidade distante. A conjugação destes critérios implica a definição de uma função flexível a minimizar:

- $f := k * L_{carro} + j * L_{pe} + l * L_{preco}$, em que os coeficientes $0 <= k <= 1$, $0 <= j <= 1$ e $0 <= l <= 1$, com $k + j + l = 1$, representam os pesos a dar a cada critério

A somar à necessidade de minimização desta função, vem o requisito de um tempo de processamento aceitável - idealmente, na ordem dos segundos -, ou seja, terá de existir um balanço entre os dois fatores.

4. Abordagem ao problema

4.1 Fases de resolução do problema

1. **Análise da conectividade do grafo** - como explanado nas restrições dos dados de entrada, é importante garantir que é possível transitar entre todos os pontos de paragem relevantes.

2. **Procura de parques de estacionamento** - junto de cada ponto de paragem em que há que estacionar, enumeram-se os parques de estacionamento das redondezas (com limite de distância Z) e efetua-se a escolha do melhor parque, destes.

3. **Cálculo dos caminhos a pé** - depois de escolher o parque onde se estaciona junto ao ponto onde se realiza a tarefa, é necessário calcular o caminho mais curto entre estes, que o utilizador irá percorrer, duas vezes, a pé - do parque para o ponto da tarefa, e deste para o parque, para continuar a travessia de carro.

4. **Cálculo dos caminhos entre pontos de paragem** - o cálculo dos caminhos mais curto entre os pontos de paragem, a partida e o destino permitirá converter o problema numa instância do problema do caixeiro-viajante.

5. **Cálculo da travessia de carro** - por fim, encontra-se o caminho mais curto começando em S, passando pelos pontos em I (ou pelos respetivos parques de estacionamento próximos) e terminando em D.

4.2 Algoritmos candidatos

4.2.1 Análise da conectividade do grafo

Com vista a garantir que é possível uma travessia passando por todos os pontos de interesse - partida, pontos intermédios e destino -, procede-se a uma pesquisa, em cada ponto deste conjunto, por todos os outros. Caso se verifique que um dos pontos de paragem é inalcançável - por exemplo, por motivos de obras na via ou áreas rurais sem acesso em estrada -, o problema não tem solução.

> *Componente fortemente conexo* - zona de um grafo em que há um caminho possível entre todos os pares de vértices.

Se todo o grafo forma um componente fortemente conexo, verifica-se imediatamente que a travessia é possível - por exemplo, com o *algoritmo de Kosaraju* -; caso contrário, é

imperativo que exista pelo menos um caminho entre todos os pares de pontos de paragens - pesquisa em profundidade ou largura.

Algoritmo de Kosaraju

O *algoritmo de Kosaraju* visa avaliar se o grafo é fortemente conexo, através do seguinte processo: [1]

1. Inicializar todos os vértices como vértices não visitados.

2. Partindo de um vértice aleatório do grafo, realizar uma *pesquisa em profundidade* - ver secção seguinte - no grafo.

3. Se não se tiver visitado todos os vértices do grafo, concluir que o grafo não é fortemente conexo; senão, continuar com o cálculo o grafo transposto, trocando o sentido de todas as arestas, e marcar todos os vértices do grafo transporto como não visitados.

4. Realizar uma *pesquisa em profundidade* no grafo transposto, começando pelo mesmo vértice escolhido no ponto 1, e avaliar se esta pesquisa chega a todos os vértices do grafo: se for o caso, o grafo é fortemente conexo.

Este algoritmo tem a complexidade temporal da *pesquisa em profundidade*, que é $O(V + E)$, se o grafo for representado por uma matriz de adjacências.

Se, depois deste processo, se verificar que o grafo não é fortemente conexo, é necessário verificar se o grafo é conexo para os pontos de interesse - por exemplo, com pesquisa em largura ou profundidade entre todos os pares de pontos de interesse.

4.2.2 Seleção de parques de estacionamento

Procura de parques de estacionamento

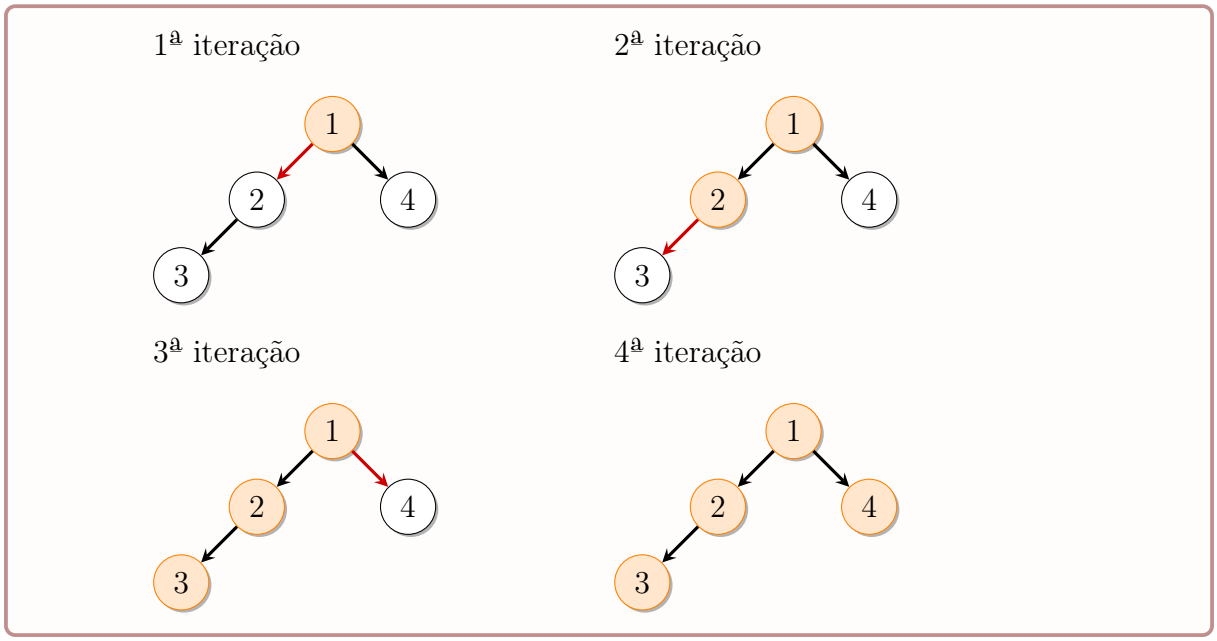
Nesta fase, para enumerar os parques de estacionamento nas redondezas de cada ponto, recorre-se a uma pesquisa em profundidade ou em largura. A pesquisa deve parar quando não há nenhum vértice a processar com distância ao ponto da tarefa inferior a Z.

Na ***pesquisa em profundidade***, visita-se cada ramo do grafo até ao seu fim antes de passar para o seguinte. Sendo necessário visitar cada vértice e os seus vizinhos, a complexidade temporal será na ordem de $O(V + E)$.

Algoritmo 1: Pesquisa em profundidade

```
Q ← ∅;
Function DFS (G)
    foreach v ∈ V do
        visited(v) ← false;
    end foreach
    foreach v ∈ V do
        if not visited(v) then
            DFS_VISIT(G, v);
        end if
    end foreach
Function DFS_ VISIT(G, v)
    visited(v) ← true;
    foreach w ∈ Adj(v) do
        if not visited(w) then
            DFS_VISIT(G, v);
            ENQUEUE(Q, w);
        end if
    end foreach
```

No exemplo abaixo, considere-se que se sai do ponto marcado pelo nó 1, procurando em profundidade um caminho válido para o nó 4. Os nós visitados até ao momento, em cada iteração, são marcados a laranja, e a aresta para o próximo nó a visitar a vermelho.

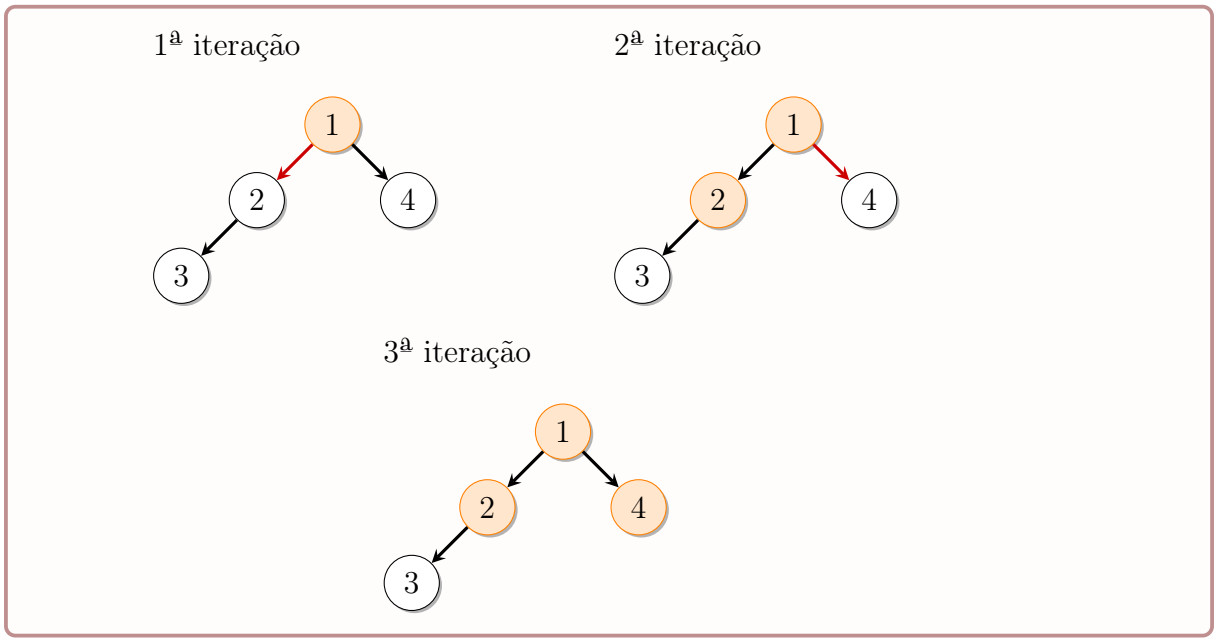


Na ***pesquisa em largura***, visitam-se todos os vértices de um nível da árvore antes de passar para o nível seguinte. Tal como na pesquisa em profundidade, a complexidade temporal será $O(V + E)$.

Algoritmo 2: Pesquisa em largura

```
G(V, E);
F ← ∅;
foreach v ∈ V do
    discovered(v) ← false;
end foreach
Q ← ∅;
ENQUEUE(Q, S);
discovered(S) ← true;
while Q ≠ ∅ do
    v ← DEQUEUE(Q);
    foreach w ∈ Adj(v) do
        if not discovered(w) then
            discovered(w) ← true;
            F ← F ∪ {w};
            ENQUEUE(Q, w);
        end if
    end foreach
end while
```

Considere-se novamente o grafo usado como exemplo para a pesquisa em profundidade, desta vez percorrido com uma pesquisa em largura.



Neste grafo, a pesquisa em largura revelou-se vantajosa, mas a escolha entre a pesquisa em profundidade e a pesquisa em largura dependerá da natureza do grafo do problema: se o grafo é largo (ou seja, cada vértice tem um número elevado de arestas adjacentes), como no caso de um mapa real, é possível que a pesquisa em largura resulte em menos iterações; por outro lado, é bastante menos eficiente em termos espaciais (por ter de guardar todos os vértices a processar numa fila).

Escolha do melhor parque

A escolha do melhor parque, a substituir pelo ponto da tarefa na travessia final de carro (a calcular numa fase posterior), dependerá da função de prioridade usada, que atribua pesos adequados aos diferentes critérios de escolha (de acordo com a opção do utilizador):

$$g := k * (P_{Base} + ((C - V)/C) * Q_{Disp}) + j * Dist_{ponto-tarefa} + l * Dist_{destino}$$

Note-se que a distância ao destino será sempre uma heurísitica para ter a menor distância de carro, dado que, nesta fase, ainda não se calculou a travessia de carro. Para corrigir esta debilidade, seria necessário deslocar este passo para o fim das tarefas, obrigando depois a um recálculo dos caminhos, que seria penoso.

Sendo N o número de parques encontrados até um dado momento, a estrutura de dados a utilizar para manter informação sobre os parques poderá ser:

- Fila de prioridade (*min heap*)
  - Complexidade temporal
    - Inserções: $O(N * log(N))$
    - Extração do menor elemento: $O(1)$
  - Complexidade espacial
    - $O(N)$

- Lista ligada
  - Complexidade temporal
    - Inserções: $O(N * 1)$
    - Extração do menor elemento: $O(N)$
  - Complexidade espacial
    - $O(N)$

Dado que o número de parques de estacionamento pode ser muito elevado, poderá recorrer-se a uma lista ligada ou estrutura semelhante, guardando-se o mínimo atual em cada inserção, de que vem uma complexidade temporal $O(N)$.

4.2.3 Cálculo dos caminhos a pé

O caminho mais curto (a pé) dos parques até ao ponto onde há que realizar a tarefa pode ser dado por um dos tradicionais algoritmos de caminho mais curto.

Algoritmo de Dijkstra

No *algoritmo de Dijkstra*, calculam-se os caminhos mais curtos de um vértice para todos os outros. A cada iteração, enfileiram-se os vértices adjacentes ao vértice de momento processado para futuro processamento, atualizando-se os caminhos até estes caso seja encontrado um caminho mais curto.

O vértice a processar a cada passo é aquele para o qual houve aresta com menor peso. Ainda que gananciosa, esta estratégia resulta numa solução ótima, dado que o grafo em que abstratizamos o mapa não contém arestas de peso negativo (ver restrições).

Algoritmo 3: Algoritmo de Dijkstra

```
foreach v ∈ V do
    dist(v) ← ∞;
    path(v) ← nil;
end foreach
Q ← ∅;
INSERT(Q, (s, 0));
while Q ≠ ∅ do
    v ← EXTRACT_MIN(Q);
    foreach w ∈ Adj(v) do
        if dist(w) > dist(v) + weight(v, w) then
            dist(w) ← dist(v) + weight(v, w);
            path(w) ← v;
            if w ∉ Q then
                INSERT(Q, (w, dist(w)));
            end if
            else
                DECREASE_KEY(Q, (w, dist(w)));
            end if
        end if
    end foreach
end while
```

A complexidade temporal depende da implementação da fila de prioridade: [2]

- *Array* - prioridade de cada vértice numerado de 1 a $|V|$ nos slots 0 a $|V-1|$
  - Inserção ($|V|$ vezes): $O(1)$
  - Redução de prioridade ($|E|$ vezes): $O(1)$
  - Extração do mínimo ($|V|$ vezes): $O(V)$

  Tempo total: $O(V^2)$

- *Min heap binária*
  - Inserção ($|V|$ vezes): $O(logV)$
  - Redução de prioridade ($|E|$ vezes): $O(logV)$
  - Extração do mínimo ($|V|$ vezes): $O(logV)$

  Tempo total: $O((V+E)logV)$

- *Heap de Fibonacci* - tempo amortizado
  - Inserção ($|V|$ vezes): $O(logV)$
  - Redução de prioridade ($|E|$ vezes): $O(1)$
  - Extração do mínimo ($|V|$ vezes): $O(logV)$

  Tempo total: $O(E + VlogV)$

Algoritmo de A*

O *algoritmo de A**, construído a partir do algortimo de Dijkstra por Hart et al., para aquando do primeiro caminho encontrado até ao destino, evitando percorrer todo o grafo, com elevado custo computacional.

Para melhorar a qualidade da solução (que nunca será ótima), a busca do algoritmo é «informada» [3]: a prioridade dos vértices a processar depende também da sua distância ao destino, facilitando uma aproximação ao destino.

A heurística a utilizar poderá ser a distância euclidiana ao destino ou a diferença das somas das coordenadas cartesianas do ponto e do destino, mais precisa em cidades com estradas em grelha, como Manhattan, onde se reduz a liberdade de movimento de 8 para 4 direções [4].

Algoritmo 4: Algoritmo de A*

```
foreach v ∈ V do
    dist(v) ← ∞;
    path(v) ← nil;
end foreach
Q ← ∅;
INSERT(Q, (s, 0));
while Q ≠ ∅ do
    v ← EXTRACT_MIN(Q);
    if v = D then
        break;
    end if
    foreach w ∈ Adj(v) do
        if dist(w) > dist(v) + weight(v, w) then
            dist(w) ← dist(v) + weight(v, w);
            path(w) ← v;
            if w ∉ Q then
                INSERT(Q, (w, dist(w) + heuristic(w)));
            end if
            else
                DECREASE_KEY(Q, (w, dist(w) + heuristic(w)));
            end if
        end if
    end foreach
end while
```

A complexidade temporal do *algoritmo de A** depende da heurística utilizada, mas, no pior caso, o número de nós expandido será exponencial ao tamanho do caminho mais curto [5]. No caso médio, espera-se que seja consideravelmente mais rápido do que o algoritmo de Dijkstra.

Como forma de melhorar o *algoritmo de A** (ou de *Dijkstra*, pelas suas semelhanças), existem variantes em que a pesquisa é bidirecional, parando quando um dos lados para [6].

4.2.4 Cálculo dos caminhos entre pontos de paragem

Nesta fase do problema, pretende-se calcular a menor distância de cada ponto de paragem aos outros pontos de paragem, através do cálculo do caminho mais curto entre cada ponto de paragem aos demais existentes. Serão, portanto, apresentados nesta secção algoritmos passíveis de resolver este problema.

Algoritmo de Dijkstra

Já apresentado anteriormente, o *algoritmo de Dijkstra* pode também ser utilizado para resolver o problema agora em análise, realizando $|C| + 2$ iterações deste mesmo algoritmo, sendo $|C|$ o número de pontos de paragem mais o ponto de partida, S e o ponto de destino, D. Portanto, já que o *algoritmo de Dijkstra* calcula o caminho mais curto de um vértice a todos os outros vértices do grafo, nesta situação, este seria realizado N vezes, em todos vértices selecionados.

Assumindo que se utiliza uma fila de prioridade implementada através de uma *min heap binária*, a complexidade temporal desta algoritmo será $O((V + E)log(V))$, pelo que, ao ser realizado N vezes, esta passará a ser $O((|C| + 2)(V + E)log(V))$.

Algoritmo de Floyd-Warshall

No *algoritmo de Floyd-Warshall*, o grafo é representado, preferencialmente, usando uma matriz de adjacências, que contém o peso das arestas entre cada par de vértices.

Primeiramente, a matriz de adjacências é inicializada com as distâncias entre vértices diretamente ligados. Depois, atualiza-se o menor caminho encontrado de um vértice i para um vértice j, passando por um ou mais vértices intermédios k, caso daqui resulte melhoria na distância total percorrida.

Algoritmo 5: Algoritmo de Floyd-Warshall

```
n ← number of vertices in graph;
distance ← n * n array of minimum distances initialized to ∞;
foreach v ∈ V do
    distance[v][v] ← 0;
end foreach
foreach e(u, v) ∈ E_i do
    distance[u][v] ← weight(u, v);
end foreach
for k ← 1, n do
    for i ← 1, n do
        for j ← 1, n do
            if distance[i][j] > distance[i][k] + distance[k][j] then
                distance[i][j] ← distance[i][k] + dist[k][j];
            end if
        end for
    end for
end for
```

Como é facilmente dedutível, a complexidade temporal deste algoritmo é $O(V^3)$.

Dijkstra* vs *Floyd-Warshall

Apresentados estes dois algoritmos, e a sua respetiva complexidade temporal anteriormente, pode-se concluir que o *algoritmo de Dijkstra* apresenta, em geral, uma complexidade temporal melhor que a do *algoritmo de Floyd-Warshall*.

Considerando que a variável N, anteriormente apresentada como o número de iterações do *algoritmo de Dijkstra*, tem o mesmo valor que V, o número de iterações do *algoritmo de Dijkstra* será igual ao número de vértices do grafo, resultando numa complexidade temporal de $O(V * (V + E) * log(V))$. Considere-se, agora, que o número de arestas do grafo é aproximadamente igual ao número de vértices ao quadrado, ou seja, $E \approx V^2$. Substituindo E por V^2, as complexidades ficam:

Dijkstra: $O(V * (V + V^2) * log(V)) \iff (V^3 * log(V))$

Floyd-Warshall: $O(V^3)$

Analisando as novas complexidades, verifica-se que no caso de se estar perante um grafo denso, em que o *algoritmo de Dijkstra* seja iterado V vezes, fazendo exatamente o mesmo que o *algoritmo de Floyd-Warshall*, ou seja, calcular o caminho mais curto entre todos os pares de vértices, o *algoritmo de Floyd-Warshall* é mais eficiente que o *algoritmo de Dijkstra*.

No entanto, e analisando o contexto real do problema em mãos, que consiste em calcular a menor distância de todos os pontos de paragem uns em relação aos outros, incluindo o ponto de paragem e de destino, é pouco provável, se não quase irrealista, que o contudor pretenda que todos - ou aproximadamente todos - os vértices do grafo sejam locais de paragem.

4.2.5 Cálculo da travessia de carro

Nesta fase, o problema está reduzido ao *problema do caixeiro-viajante*, sem necessidade de voltar ao ponto inicial: há que encontrar um caminho hamoltiano - caminho em que cada vértice do grafo é visitado exatamente uma vez [7] - entre os pontos de paragem, com a restrição de que há que começar em S e acabar em D, e o objetivo de reduzir a distância total percorrida.

Algoritmos inexatos

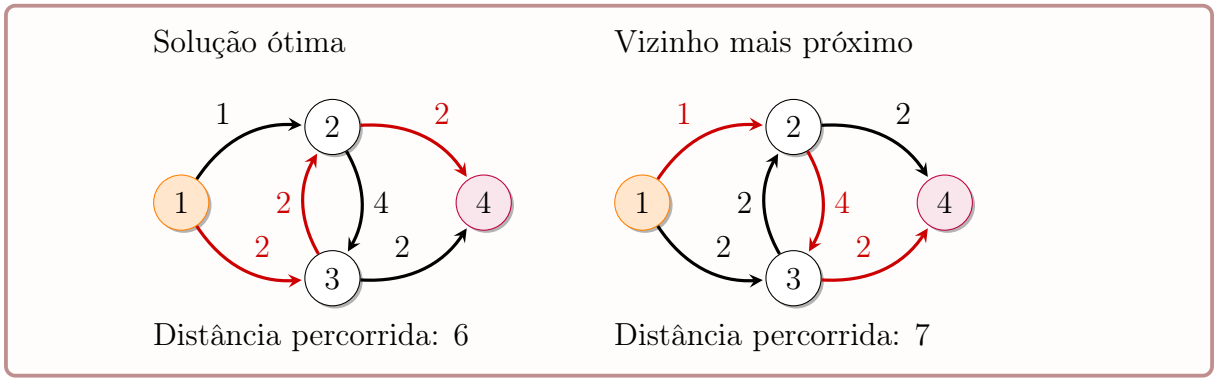
Algoritmo do vizinho mais próximo

No algoritmo do vizinho mais próximo, uma solução é aproximada a partir da heurística:

Do vértice V_1, transita-se para o vértice V_2, tal que a aresta $E(V_1, V_2)$ é a aresta adjacente a $V1$ com menor peso, e $V2$ ainda não foi visitado.

Seguindo repetidamente este critério (com a natural restrição de o vértice D de destino não ser incluído até que o número de vértices visitado seja o total menos um), obtém-se uma solução razoável e muito rápida: a complexidade temporal é o número de cidades a visitar, $O(C)$.

No exemplo abaixo, compara-se a solução ótima (à esquerda) com a solução aproximada por esta metodologia (à direita), começando a travessia no vértice 1 (a laranja) e com destino no vértice 4 (a violeta). As arestas selecionadas são assinaladas a vermelho.



Algoritmo de Christofides

O *algoritmo de Christofides* tem as condições necessárias:

Seja $G = (V, w)$ um grafo completo com o conjunto de vértices V e w a função que tribui um peso (valor real não negativo) a cada aresta de G. De acordo com a desigualdade triangular, para três vértices u, v e x quaisquer, é válido dizer que $w(uv) + w(vx) \geq w(ux)$, ou seja, o caminho direto de u para x é sempre mais curto do que qualquer caminho entre aqueles que inclua um vértice intermédio.

Seguindo os passos seguintes, obtém-se uma aproximação no máximo 50% mais longa do que a solução ótima. [8]

1. Encontrar a árvore de expansão mínima T para G.
2. Calcular o conjunto de vértices O com grau ímpar em T.
3. Formar o subgrafo M de G usando apenas os vértices de O.
4. Construir um acoplamento perfeito de peso mínimo no subgrafo M.
5. Unir T com M.
6. Calcular o ciclo euleriano.
7. Remover vértices repetidos.

Grau de um vértice: número de arestas adjacentes a esse vértice.
Acoplamento perfeito: conjunto de arestas sem vértices em comum.
Ciclo euleriano: caminho que visita todas as arestas uma só vez, regressando ao ponto de partida.

Uma árvore de expansão mínima - obtida no passo 1 - é uma árvore que liga todos os vértices do grafo usando arestas com um custo total mínimo, resultando num grafo acíclico, conexo e não dirigido, em que o número de arestas é $V-1$. É possível obter uma árvore de expansão mínima usando o *algoritmo de Prim* e o *algoritmo de Kruskal*.

O ***algoritmo de Prim*** é um algoritmo ganancioso, que parte de um vértice e vai expandido a árvore, escolhendo sempre a aresta de melhor de menor custo (v_1, v_2), sabendo que v_1 pertence à árvore e v_2 ainda não. Na verdade, o *algoritmo de Prim* é bastante semelhante ao *algoritmo de Dijkstra*.

Algoritmo 6: Algoritmo de Prim

foreach $v \in V$ **do**
    $dist(v) \leftarrow \infty$;
    $path(v) \leftarrow nil$;
    $visited(v) \leftarrow false$;
end foreach
$Q \leftarrow \emptyset$;
$INSERT(Q, (s, 0))$;
while $Q \neq \emptyset$ **do**
    $v \leftarrow EXTRACT\_MIN(Q)$;
    $visited(v) \leftarrow true$;
    foreach $w \in Adj(v)$ **do**
        $v_{dest} \leftarrow dest(w)$;
        $oldDist \leftarrow dist(v_{dest})$;
        if *not* $visited(v_{dest}) \wedge dist(v_{dest}) > weight(w)$ **then**
            $path(v_{dest}) \leftarrow v$;
            $dist(v_{dest}) \leftarrow weight(w)$;
            if $oldDist = \infty$ **then**
                $INSERT(Q, (w, dist(w)))$;
            end if
            else
                $DECREASE\_KEY(Q, (w, dist(w)))$;
            end if
        end if
    end foreach
end while

A complexidade temporal do *algoritmo de Prim* depende da estrutura de dados utilizada:

Sem fila de prioridade: $O(V^2)$
Com fila de prioridade: $Elog(V)$

O ***algoritmo de Kruskal*** é outro algoritmo cujo resultado é uma árvore de expansão mínima. Este algoritmo, tal como o anterior, é um algoritmo ganancioso, pois analisa as arestas por ordem crescente de peso e aceita as que não provocarem ciclos (isto é, que pertençam a um conjunto disjunto diferente).

A complexidade temporal do *algoritmo de Kruskal* é $|E|log(|E|)$, no pior dos casos. Sabendo que $|E| \leq |V|^2 \iff log(|E|)2log(|V|)$, a complexidade temporal simplifica para $O(|E|log|V|)$.

Algoritmo 7: Algoritmo de Kruskal

```
foreach v ∈ V do
|  MAKESET(v)
end foreach
SORT_BY_WEIGHT(E);
Q ← ∅;
foreach e ∈ E do
|  if FIND_SET(orig(e)) ≠ FIND_SET(dest(e)) then
|  |  INSERT(Q, e);
|  |  UNION(orig(e), dest(e));
|  end if
end foreach
```

Algoritmos exatos

Pesquisa exaustiva

A solução mais evidente passa por encontrar todas as combinações de caminhos válidos e extrair o mais curto, que resultaria numa complexidade temporal muito significativa, de $O((|C| - 1)!)$.

Algoritmo de Held-Karp

Como forma de melhorar a simples pesquisa exaustiva, é possível aplicar técnicas de programação dinâmica. No caso do problema do caixeiro-viajante, existe subestrutura ótima: [9]

Cada subcaminho de um caminho mais curto é também o mais curto.

Desta propriedade vem a fórmula recursiva:

Sejam:

- $cost(V_1, V_2, V_3)$: distância do vértice V_1 ao vértice V_2, visitando todos os outros vértices exceto V_3
- $dist(V_1, V_2)$: distância do vértice V_1 ao vértice V_2

$cost(a, c, \{\}) = min(dist(a, b, c) + dist(b, c))$

A complexidade temporal deste algoritmo está na ordem de $O(C^2 2^C)$ [9], que representa uma melhoria significativa face à pesquisa exaustiva, mesmo com um número reduzido de cidades a visitar:

Número de cidades	Pesquisa exaustiva	Algoritmo de Held-Karp
5	120	800
10	3628800	102400
15	1307674368000	7372800
20	2432902008176640000	419430400

5. Funcionalidades implementadas

Na lista abaixo, apresentam-se as funcionalidades que se consideraram essenciais para a utilização do programa em contexto real, na perspetiva do condutor, e foram implementadas no menu inicial que serve de demonstração do programa.

1. **Importar um mapa**: selecionar o ficheiro com a informação do mapa da cidade onde se deseja calcular o percurso.

2. **Analisar a conectividade do grafo**: verificar se, para cada ponto do mapa, todos os outros são alcançáveis de modo a garantir que o percurso calculado pelos algoritmos é válido.

3. **Definir o ponto de partida**: escolher/alterar o ponto inicial.

4. **Definir o ponto de chegada**: escolher/alterar o destino.

5. **Assinalar obras na via pública**: marcar uma estrada (conjunto de arestas) como intransitável.

6. **Adicionar uma paragem ao percurso**: acrescentar uma tarefa, indicando se é necessário estacionar (por exemplo, ir a um restaurante) ou basta passar lá (por exemplo, passar num drive through para comprar comida), ao conjunto de tarefas a realizar durante o percurso.

7. **Remover uma paragem do percurso**: retirar uma tarefa ao conjunto de tarefas a executar durante o percurso.

8. **Calcular o percurso**: escolher o melhor percurso, dando prioridade a um dos três critérios anteriormente abordados.

6. Algoritmos implementados

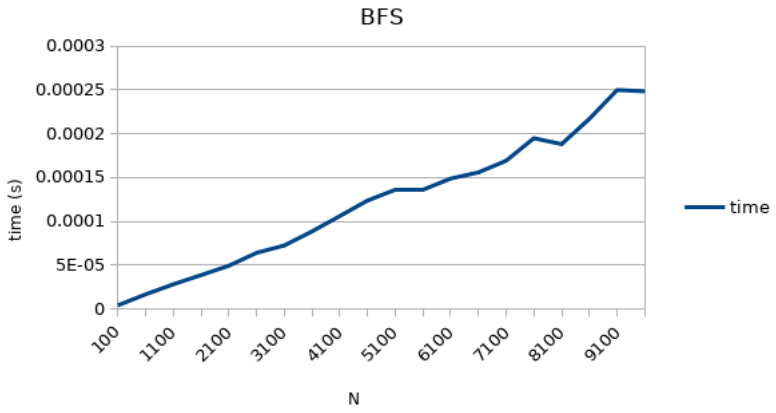
Tendo uma perspetiva formal dos algoritmos conhecidos para a resolução de cada fase do problema em mãos, optou-se por implementar os seguintes algoritmos:

1. **Análise da conectividade do grafo** - *algoritmo de Kosaraju*.
2. **Procura de parques de estacionamento** - restrição por distância euclidiana.
3. **Cálculo dos caminhos a pé** - algoritmo de Dijkstra.
4. **Cálculo dos caminhos entre pontos de paragem** - repetição do *algoritmo de Dijkstra*, dado que o *algoritmo de Floyd-Warshall* se mostrou ineficiente, por restrições de espaço da matriz auxiliar.
5. **Cálculo da travessia de carro** - algoritmo do vizinho mais próximo.

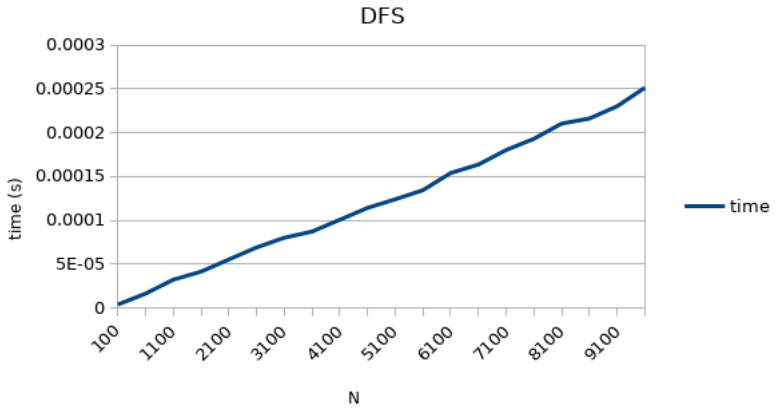
7. Análise da performance

Os algoritmos implementados foram testados em larga escala, com grafos gerados de forma aleatória e de tamanho crescente, para testar a sua viabilidade no contexto do problema. Abaixo apresentamos o resultado das experiências realizadas, sob a forma de tempos de execução, que demonstram a aproximação da complexidade temporal de cada um destes àquela que é conhecida.

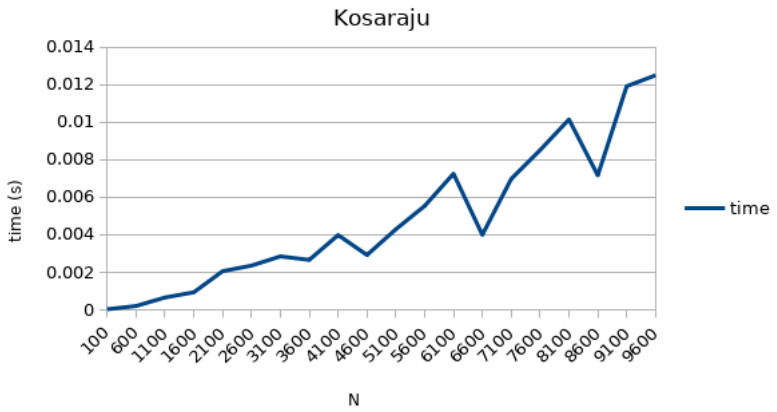
BFS: Complexidade teórica de O(V + E):



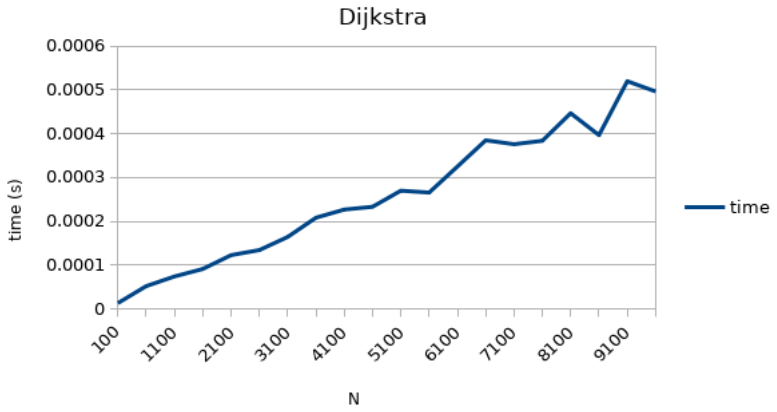
DFS: Complexidade teórica de O(V + E):



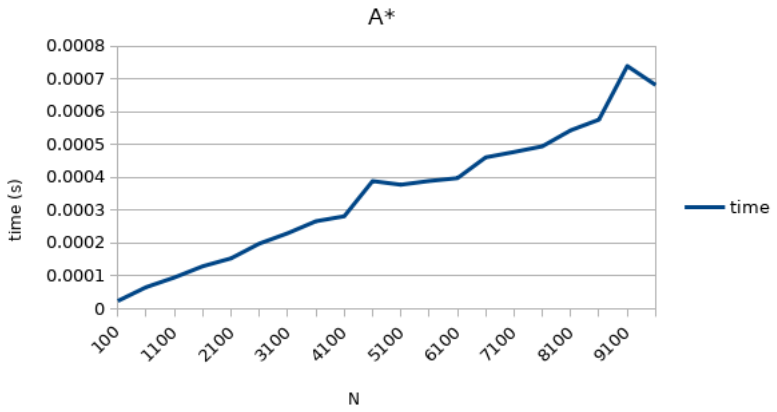
Kosaraju: Complexidade teórica de O(V + E):



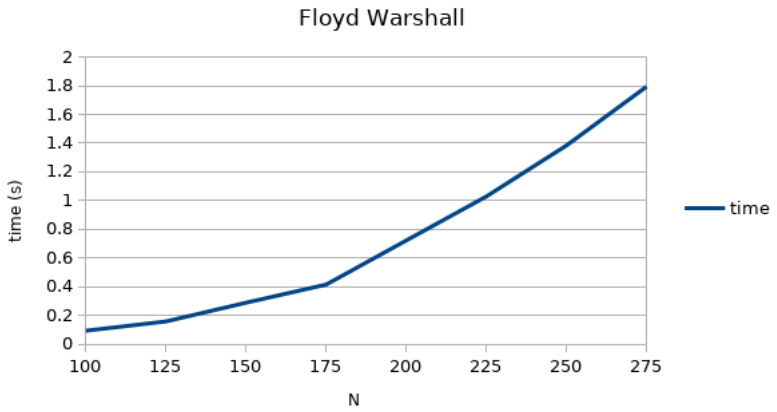
Dijkstra: Complexidade teórica de O(E * log(V)):



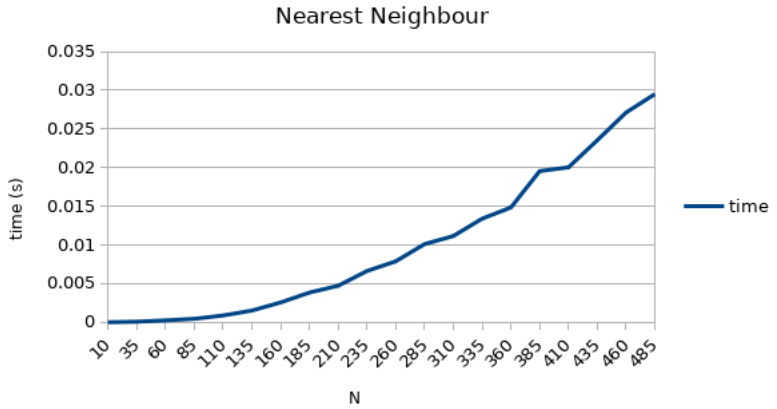
A*: Complexidade teórica de O(|E|), no pior caso, mas melhora com heurísticas aplicadas:



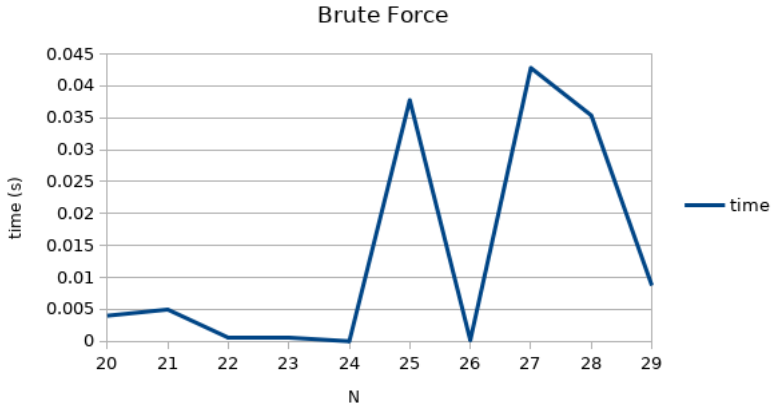
Floyd Warshall: Complexidade teórica de O(V^3):



Vizinho Mais Próximo: Complexidade teórica de O(C), não contando com o algoritmo de caminho mais curto utilizado



Brute Force: Complexidade teórica de O(N!)



8. Conclusão

O problema do condutor citadino, enquanto derivado do problema do caixeiro-viajante, trouxe as dificuldades habituais de resolução de um problema NP-completo. Na implementação, procurou-se escolher um leque de algoritmos robusto e eficiente, abordado anteriormente de um ponto de vista teórico, com vista a aproximar uma solução utilizável num contexto real.

8.1 Contribuição

- Adelaide Santos: 1/3
- Bruno Mendes: 1/3
- Rita Mendes: 1/3

Bibliografia

[1] Dorit S. Hochbaum. *The Pseudoflow Algorithm Graph Algorithms and Network Flows*. URL: https://hochbaum.ieor.berkeley.edu/files/ieor266-2012.pdf (ver p. 8).

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest e Clifford Stein. *Introduction to Algorithms*. 2009. URL: https://moodle.up.pt/pluginfile.php/164125/mod_label/intro/Introduction%20to%20algorithms.pdf (ver p. 12).

[3] Mareike Hedderich, Ulrich Fastenrath e Klaus Bogenberger. *Optimization of a Park Spot Route based on the A* Algorithm*. 2018. URL: https://ieeexplore.ieee.org/document/8569376 (ver p. 13).

[4] Amit Patel. *Heuristics - Amit's Thoughts on Pathfinding - Red Blob Games*. 2020. URL: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (ver p. 13).

[5] *A* Search - Brilliant Wiki*. URL: https://brilliant.org/wiki/a-star-search/ (ver p. 13).

[6] Wim Pijls e Henk Post. *Yet another bidirectional algorithm for shortest paths*. 2009. URL: http://repub.eur.nl/pub/16100/ei2009-10.pdf (ver p. 13).

[7] Vaidehi Joshi. *The Trials And Tribulations Of The Traveling Salesman*. 2017. URL: https://medium.com/basecs/the-trials-and-tribulations-of-the-traveling-salesman-56048d6709d (ver p. 15).

[8] Michael Goodrich e Roberto Tamassia. *Algorithm Design and Applications*. 2015. URL: http://canvas.projekti.info/ebooks/Algorithm%20Design%20and%20Applications%5BA4%5D.pdf (ver p. 16).

[9] *Graphs and Graph Algorithms - University of Manchester*. URL: http://www.cs.man.ac.uk/~david/algorithms/graphs.pdf (ver pp. 18, 19).