

# Meat Wagons

Conceção e Análise de Algoritmos  
2019/2020

Turma 3 – Grupo 1

André Gomes – [up201806224@fe.up.pt](mailto:up201806224@fe.up.pt)

Gonçalo Alves – [up201806451@fe.up.pt](mailto:up201806451@fe.up.pt)

Pedro Seixas – [up201806227@fe.up.pt](mailto:up201806227@fe.up.pt)

# Índice

DESCRIÇÃO DO TEMA	2
Fase 1: Um veículo para todos os prisioneiros	2
Fase 2: Diferentes Veículos, capacidade ilimitada	2
Fase 3: Mais veículos, capacidade limitada	2
POSSÍVEIS PROBLEMAS A ENCONTRAR	3
FORMALIZAÇÃO DO PROBLEMA	4
Dados de Entrada	4
Dados de Saída	5
Restrições	5
Sobre os Dados de entrada	5
Sobre os Dados de Saída	5
PERSPETIVA DE SOLUÇÃO	6
Fase 1	6
1. Preparação dos ficheiros de entrada	6
2. Análise da Conectividade do Grafo	6
3. Criação de POI's	6
4. Identificação de Técnicas de Conceção	8
Fase 2	11
Hipótese 1 - Não tendo em conta o caminho	11
Hipótese 2 - Tendo em conta o caminho	11
Hipótese 3 - Apenas considerando os vértices onde o veículo pode transportar	11
Fase 3	11
CASOS DE UTILIZAÇÃO	12
CONCLUSÃO	13
BIBLIOGRAFIA	14

# Descrição do Tema

Um estabelecimento prisional necessita de gerir o transporte dos seus prisioneiros entre diferentes penitenciárias, tribunais, estabelecimentos prisionais ou esquadras policiais, tendo ao seu dispor diferentes veículos e sendo preciso ter em conta: prisioneiros com o mesmo destino, propriedades do destino para escolher o veículo e planeamento de rotas.

O objetivo é desenvolver um programa que calcule, para o dia, as rotas a tomar para cada veículo da frota tendo em conta os destinos do dia para a lista de prisioneiros.

Nesta fase inicial o projeto encontra-se dividido em 3 fases.

## **Fase 1: Um veículo para todos os prisioneiros**

Tendo uma lista dos prisioneiros e sabendo para cada um o seu destino, o objetivo desta primeira parte é ter um autocarro, com capacidade ilimitada, que passe por todos os POI designados para cada prisioneiro, tendo em conta o caminho mais curto que passe em todos os POI e que depois retorne para a origem.

Certas vias podem ser inacessíveis por razões diversas (obras, cortes de estrada, largura de rua não ser suficiente). Assim, durante o processamento do grafo, será necessário desprezar certas arestas.

## **Fase 2: Diferentes Veículos, capacidade ilimitada**

Nesta fase, cada vértice do grafo terá a informação extra sobre a sua Densidade Populacional (Cidade, Periferia ou Campo). Os prisioneiros serão previamente divididos pelos veículos, tendo em conta o seu destino, e cada veículo estará especializado para uma Densidade Populacional. Começaremos por ter uma camioneta (Campo e Periferia) e um carro (Cidade), ambos com capacidade ilimitada, para testar a divisão dos prisioneiros e a formulação de rotas.

## **Fase 3: Mais veículos, capacidade limitada**

Na fase final, teremos uma frota de veículos com capacidade limitada e um conjunto de prisioneiros com o seu destino. A leitura de dados inicial será para o dia, ou seja, ao executar o programa ficam em memória os prisioneiros, com o seu destino, e a frota disponível. Caso toda a frota esteja ocupada e ainda sobram prisioneiros, tendo em conta o seu destino, serão retornados para a origem os veículos necessários para refazer rotas e os transportarem.

# Possíveis Problemas a Encontrar

- Leitura dos mapas provenientes do OpenStreetMaps;
- Peso das arestas do grafo;
- Conversão dos dados de modo a possibilitar a utilização do GraphViewer;
- Cruzamento de ruas com o mesmo nome;
- Acentuação nos nomes das ruas;
- Geração de tags referentes a tribunais ou estabelecimentos prisionais;
- Distribuição da tag densidade populacional pelos vértices do grafo;

# Formalização do Problema

## Dados de Entrada

**Pi**: Lista de prisioneiros com destinos para o dia, sendo  $P(i)$  o  $i$ -ésimo elemento, cada um é caracterizado por:

- **ID** - Número identificador de prisioneiro
- **Destino** - ID do node de destino

O número identificador de destino terá duas partes: a primeira parte, de 1 dígito, é correspondente ao tipo de destino; a segunda parte, de 3 dígitos, é correspondente ao destino concreto de um certo tipo. Como exemplo: Um tribunal poderá ter um Destino de 1001 enquanto que um estabelecimento prisional poderá ter um Destino de 2001.

**Fi**: Lista de veículos da frota, sendo  $F(i)$  o  $i$ -ésimo elemento, cada um é caracterizado por:

- **ID** - Número identificador do veículo (tal como em Destino, será um número que terá implícito o tipo de veículo)
- **cap** - Número de assentos destinados a prisioneiros

**Gi = (Vi, Ei)**: Grafo Dirigido Pesado (Dirigido  $\rightarrow$  sentido da rua, Pesado  $\rightarrow$  Distância entre vértices)

- **V**: Vértices representativos de pontos da cidade com:
  - **ID do Node** - retirado do ficheiro de nós fornecidos pelos mapas
  - **tag** - Penitenciária, tribunal, esquadra policial ou estabelecimento prisional (cada um terá um ID igual a Destino, caso não seja nenhum desses, ID = 0)
  - **DP** - Informação sobre densidade populacional (Cidade, Periferia ou Campo)
  - **adj**  $\subseteq E$  - Conjunto de arestas que partem do vértice
- **E**: Arestas representativas das vias de comunicação
  - **w** - Peso da aresta (representa a distância entre os dois vértices que a delimitam)
  - **ID** - Identificador único da aresta
  - **dest**  $\in V_i$  - Vértice de destino

**S**  $\in V_i$ : Vértice inicial (Estabelecimento prisional)

**T**  $\subseteq V_i$ : Vértices finais (Destinos)

## Dados de Saída

**Gf = (Vf, Ef):** Grafo Dirigido Pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei.

**Ff:** Lista ordenada de todos os veículos usados, sendo Ff(i) o seu i-ésimo elemento. Cada um tem os seguintes valores:

- **Capacidade** - número de assentos utilizados
- **I** - Lista de prisioneiros que o veículo transportará por ordem de paragem
- **P = {e ∈ Ei | 1 ≤ j ≤ |P|}** - sequência ordenada (com repetidos) de arestas a visitar, sendo ej o seu j-ésimo elemento

## Restrições

### Sobre os Dados de entrada

- $\forall i \in [1; |F_i|], \text{cap}(F_i[i]) > 0$ , dado que uma capacidade representa os assentos disponíveis
- $\forall v \in V_i, \text{tag} \geq 0$
- $\forall e \in E_i, w > 0$ , dado que o peso de uma aresta representa uma distância entre pontos de um mapa
- $\forall e \in E_i$ , e deve ser utilizável pelo elemento da frota. Senão, não é incluída no grafo Gi

### Sobre os Dados de Saída

No grafo Gf:

- $\forall v_f \in V_f, \exists v_i \in V_i$  tal que vi e vf têm os mesmos valores para todos os atributos
- $\forall e_f \in E_f, \exists e_i \in E_i$  tal que ei e ef têm os mesmos valores para todos os atributos

## Funções Objetivo

É possível determinar três funções objetivo para o nosso projeto, as quais apresentamos por ordem de prioridade:

1. Diminuir a distância total percorrida pela frota, que será:  $\sum$  dos valores das arestas percorridas pelos elementos da frota
2. Diminuir o tempo de execução do cálculo das rotas ( $\sum$  tempos de execução dos algoritmos)
3. Diminuir o número de veículos da frota usados ( $h = |F_f|$ )

# Perspetiva de solução

## Fase 1

Esta primeira fase terá vários passos referentes à preparação do ambiente de trabalho, começando pela:

### 1. Preparação dos ficheiros de entrada

Serão utilizados os ficheiros de nodes e edges fornecidos pelos professores. A informação lida dos ficheiros será guardada num grafo  $G$ . Será criada uma tag para cada tipo de edifício de interesse (prisões, esquadras e tribunais) de forma a facilitar a identificação dos pontos de interesse.

### 2. Análise da Conectividade do Grafo

Para nos certificarmos que haverá um caminho de retorno para qualquer rota calculada o grafo terá de ter uma componente fortemente conexa. De modo a fazer essa avaliação, num grafo dirigido pesado será preciso seguir o método fornecido nas aulas teóricas:

- Pesquisa em profundidade no grafo  $G$  determina floresta de expansão, numerando vértices em pós-ordem (ordem inversa de numeração em pré-ordem)
- Inverter todas as arestas de  $G$  (grafo resultante é  $G_r$ )
- Segunda pesquisa em profundidade, em  $G_r$ , começando sempre pelo vértice de numeração mais alta ainda não visitado
- Cada árvore obtida é uma componente fortemente conexa, i.e., a partir de qualquer um dos nós pode chegar-se a todos os outros

Com recurso a uma Pesquisa em Profundidade, é possível verificar se de facto há um caminho de ida entre o ponto de origem e o destino e um caminho de volta para o estabelecimento prisional original.

### 3. Criação de POI's

Após a leitura dos ficheiros com os nodes e edges, serão lidos os ficheiros das tags de forma a identificar os pontos de interesse, alterando, para esse node, a sua variável `*tag*`, inicializada a 0, para o seu valor correspondente ao tipo de ponto de interesse.

No ficheiro de tags disponibilizado não existem tags referentes ao nosso tema, tendo isso em mente, será necessário obter os nós referentes a prisões, tribunais e esquadras.

Após alguma pesquisa sobre como obter essa informação, percebemos que a melhor forma seria através da ferramenta disponível no site <https://overpass-turbo.eu/>. Esta ferramenta permite o acesso à informação do OpenStreetMap em formato json.

Foram utilizadas as seguintes queries para esse acesso:

#### Prisões

```
[out:json];{{geocodeArea:Portugal}}->.searchArea;
(
  node[amenity=prison] (area.searchArea);>;
  way[amenity=prison] (area.searchArea);>;
  relation[amenity=prison] (area.searchArea);>;
);
out;
```

#### Tribunais

```
[out:json];{{geocodeArea:Portugal}}->.searchArea;
(
  node[amenity=courthouse] (area.searchArea);>;
  way[amenity=courthouse] (area.searchArea);>;
  relation[amenity=courthouse] (area.searchArea);>;
);
out;
```

#### Esquadras

```
[out:json];{{geocodeArea:Portugal}}->.searchArea;
(
  node[amenity=police] (area.searchArea);>;
  way[amenity=police] (area.searchArea);>;
  relation[amenity=police] (area.searchArea);>;
);
out;
```

Após obter os ficheiros json com a informação pretendida, será necessário fazer um script (será feito em python devido à maior facilidade com o tratamento de dados) com o objetivo de filtrar a informação e obter apenas os ID's dos nodes que correspondem aos POI's. O seguinte pseudocódigo mostra o procedimento necessário para isso:

```
1. loadPrisonJSON()
2.
3. for prison in data:
4.   for node in prison['nodes']:
5.     for locationFile in nodeFilesFolder:
6.       for availableNode in locationFile.read():
7.         if node.ID == availableNode.ID:
8.           write node.ID to file (location + 'tags.txt')
9.
10. loadCourtJSON()
11.
12.
13. for court in data:
14.   for node in court['nodes']:
15.     for locationFile in nodeFilesFolder:
16.       for availableNode in locationFile.read():
17.         if node.ID == availableNode.ID:
18.           write node.ID to file (location + 'tags.txt')
19.
20. loadPoliceJSON()
21.
22. for police in data:
23.   for node in police['nodes']:
24.     for locationFile in nodeFilesFolder:
25.       for availableNode in locationFile.read():
26.         if node.ID == availableNode.ID:
27.           write node.ID to file (location + 'tags.txt')
```



No fim deste script o resultado obtido será um conjunto de ficheiros de tags (um para cada cidade) com o conjunto de nodes correspondentes a cada uma das tags necessárias para o nosso trabalho. Estes ficheiros permitirão a identificação dos POI's no Grafo.

#### 4. Identificação de Técnicas de Conceção

Assim que a preparação estiver pronta, é possível seguir para a implementação de código. Nesta fase será necessário que o programa consiga criar 2 rotas, uma de ida e outra de volta.

Como primeira tentativa decidimos usar para a Rota de ida o algoritmo de Dijkstra:

DIJKSTRA (G, s): // $G = (V, E)$ , $s \in V$	BIDIRECTIONAL DIJKSTRA (G, s)
<pre> 1. for each v in V do 2.   dist(v) &lt;- INF 3.   path(v) &lt;- nil 4. dist(s) &lt;- 0 5. Q &lt;- 0 // min-priority queue 6. INSERT(Q, (s, 0)) // inserts s with key 0 7. while Q != 0 do 8.   v &lt;- EXTRACT-MIN(Q) 9.   for each w in Adj(v) do 10.    if dist(w) &gt; dist(v) + weight(v,w) then 11.      dist(w) &lt;- dist(v) + weight(v,w) 12.      path(w) &lt;- v 13.      if w not in Q then 14.        INSERT(Q, (w, dist(w))) 15.      else 16.        DECREASE-KEY(Q, (w, dist(w))) </pre>	<pre> 1. Qi.Insert(x1) and mark xi as visited 2. Qg.Insert(xg) and mark xg as visited 3. while Qi not empty and Qg not empty do 4.   if Qi not empty 5.     x &lt;- Qi.GetFirst() 6.     if x = xg or x ∈ Qg 7.       return SUCCESS 8.   forall u ∈ U(x) 9.     x' &lt;- f(x,u) 10.    if x' not visited 11.      Mark x' as visited 12.      Qi.Insert(x') 13.    else 14.      Resolve duplicate x' 15.   if Qg not empty 16.     x' &lt;- Qg.GetFirst() 17.     if x' = xi or x' ∈ Qi 18.       return SUCCESS 19.   forall u<sup>(-1)</sup> ∈ U<sup>(-1)</sup>(x') 20.     x &lt;- f<sup>(-1)</sup>(x', u<sup>(-1)</sup>) 21.     if x not visited 22.       Mark x as visited 23.       Qg.Insert(x) 24.     else 25.       Resolve duplicate x 26. return FAILURE </pre>

da seguinte forma: Começando no estabelecimento prisional, onde se encontram os prisioneiros, é usado o algoritmo até encontrar um vértice, que será uma paragem de um dos prisioneiros. Neste ponto é usado outra vez o algoritmo de Dijkstra, mas com o vértice encontrado a ser usado como vértice de início, para encontrar a próxima paragem. Assim que todos os prisioneiros estiverem distribuídos será necessário encontrar o caminho de volta. Para isso, é aplicado o algoritmo de Dijkstra Bidirecional, de modo a encontrar o caminho mais curto entre o Vértice final do passo anterior e o estabelecimento prisional inicial.

**ALGORITMO A\***

Após alguma reflexão sobre qual seria o melhor algoritmo para o cálculo mais eficiente das rotas percebemos que o algoritmo A\* seria melhor, quando comparado com o algoritmo de Dijkstra e o algoritmo Dijkstra Bidirecional. Este algoritmo funciona de forma semelhante aos dois previamente apresentados, mas com uma ligeira diferença. O cálculo dos pesos da aresta segue a função:

$$f(v) = h(v) + g(v)$$

sendo  $h(v)$  a função heurística.

O algoritmo de Dijkstra é uma variância deste algoritmo em que a função  $h(v) = 0$ . Utilizando uma função melhor, é possível otimizar o cálculo do custo de cada vértice e desta forma melhorar significativamente a eficiência do algoritmo.

Optamos então pela implementação deste algoritmo usando como função heurística a distância euclidiana ao destino, isto é, permite que o custo de cada vértice seja calculado tendo em conta, não só o seu custo, mas também se se aproxima ou não do destino.

Input: A Graph  $G(V, E)$  with source node start and goal node end.

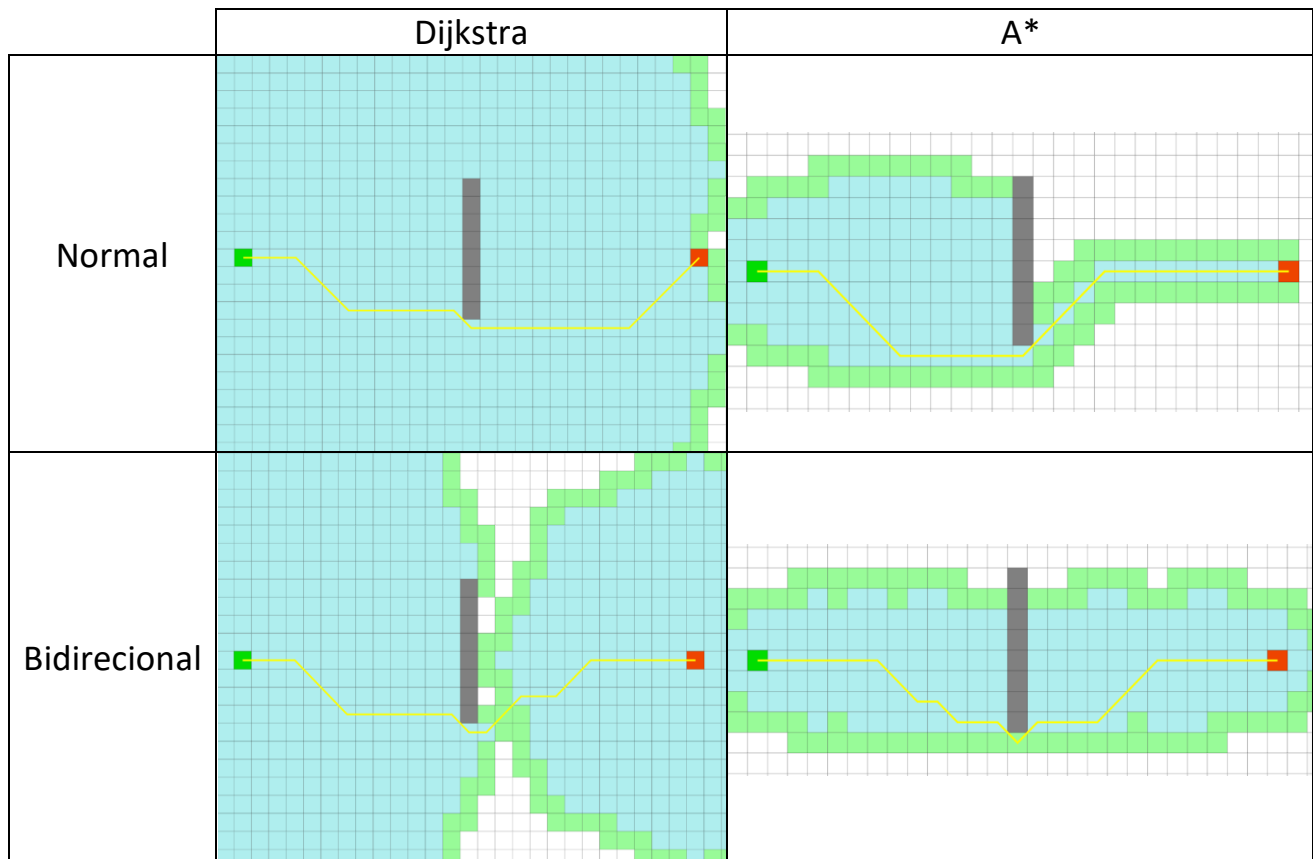
Output: Least cost path from start to end

Initializitation:

```

open_list = {start}                // List of nodes to be traversed
closed_list = {}                  // List of already traversed nodes
g(start) = 0                      // Cost from source node to a node
h(start) = heuristic_function(start,end) // Estimated cost from node to goal node
f(start) = g(start) + h(start)    // Total cost from source to goal node
while open_list is not empty
    m = Node on top of open_list with lowest f
    if m == end                    // if current note is end node the solution was found
        return success
    remove m from open_list
    add m to closed_list
    for each n in child(m)         // traverse the child nodes
        if n in closed_list
            continue
        cost = g(m) + distance(m,n) // cost of current child
        if n in open_list and cost < g(n)
            remove n from open_list as new path is better
        if n in closed_list and cost < g(n)
            remove n from closed_list
        if n not in open_list and n not in closed_list
            add n to open_list
        g(n) = cost
        h(n) = heuristic_function(n,end)
        f(n) = g(n) + h(n)
return failure

```



Imagens obtidas a partir de <https://qiao.github.io/PathFinding.js/visual/>

Como observações finais para esta fase temos a comparação e decisão dos algoritmos tendo em conta a informação revista: Já ficou provado que será melhor usar o algoritmo A\* invés dos algoritmos Dijkstra e Bidirecional Dijkstra. Restou-nos então a decisão entre A\* normal ou A\* Bidirecional.

Como, diferente do exemplo, num grafo que representa vias e estradas não haverá uma *parede* que perturbe de forma significativa o algoritmo A\*, causando, por exemplo, que este processe demasiados nós à volta do sítio bloqueado pela parede em vez de avançar de forma direta para o nó destino, juntamente com o facto de que o speedup ganho não é tão significativo entre os algoritmos A\* como entre os algoritmos de Dijkstra, decidimos que usaremos o algoritmo A\* normal.

## Fase 2

Na segunda fase teremos em conta o valor de densidade populacional (DP) dos vértices e 2 veículos, cada um especializado para os seus valores de DP. Isso leva-nos a 3 formas de encontrar o caminho mais curto, tendo em conta uma divisão prévia dos prisioneiros:

### Hipótese 1 - Não tendo em conta o caminho

Como primeira hipótese considera-se apenas a DP dos destinos de cada um dos prisioneiros. Tendo isso em conta é feita uma divisão em dois grupos baseado na DP do destino de cada prisioneiro. Um grupo será levado por um carro (com capacidade infinita) para destinos com DP de *cidade* enquanto que o outro grupo será levado por um autocarro (também com capacidade infinita) para destinos com DP de *periferia* ou *campo*.

Feita a divisão o problema simplifica-se a aplicar o método da fase 1 para cada veículo.

### Hipótese 2 - Tendo em conta o caminho

Nesta hipótese será feito, no início, o cálculo de uma rota para todos os prisioneiros. A partir do processamento da rota, cada prisioneiro ficará com o valor de cada DP dos vértices pela qual passou. Tendo em conta a DP máxima de cada prisioneiro fazem-se as divisões em 2 grupos e segue-se como na hipótese anterior para a divisão nos veículos e cálculo de rotas.

A contagem das DP para cada prisioneiro terá em conta apenas a rota inicial e não as rotas criadas pelos veículos aos quais ficaram designados, isto poderá não trazer os melhores resultados quanto à divisão dos prisioneiros entre veículos, mas é uma melhoria face à hipótese anterior.

### Hipótese 3 - Apenas considerando os vértices onde o veículo pode transportar

Nesta hipótese consideram-se diferentes tipos de veículo para transportar em *cidade* ou não. Quando são selecionados veículos que não estão aptos para transportar prisioneiros em cidades, o grafo será filtrado, removendo temporariamente todos os vértices em que a Densidade Populacional corresponde a uma cidade (não seguindo por arestas cujo destino é cidade), de forma a que encontre o caminho mais curto, apenas passando por vértices com DP de *campo* ou *periferia*. Desta forma, será descoberto o caminho mais curto para cada tipo de veículo.

## Fase 3

A diferença principal desta fase para a anterior é o limite em cada veículo. Com isso em conta, para esta fase, adotamos os mesmos passos da hipótese 2 da fase 2, até à divisão dos prisioneiros pelos veículos, exceto que neste caso teremos de fazer um cálculo do *resto* de prisioneiros, caso não haja transporte para todos. Caso haja transporte para todos, o problema torna-se igual à fase 2. Em caso contrário, faz-se um cálculo dos veículos que terão de retornar ao estabelecimento prisional inicial tendo em conta: a sua capacidade, o número de prisioneiros restantes e o grupo ao qual os prisioneiros restantes estão designados.

Assim que este cálculo for realizado repete-se este método, a partir do ponto em que se calcula o resto dos prisioneiros.

# Casos de utilização

A aplicação utilizará uma interface simples, de texto, de modo a interagir com o utilizador. Para isso, será usado um conjunto de menus com diversas opções.

Inicialmente, o programa apresentará um menu que permita ao utilizador seleccionar o distrito que pretende explorar.

Após essa selecção, o utilizador poderá seleccionar uma das seguintes opções:

- **Visualização do mapa:** através do GraphViewer;
- **Seleção da origem** (um dos estabelecimentos prisionais desse distrito). É de notar que, após esta selecção, o utilizador terá de criar pelo menos um prisioneiro, com informação de ID e Destino;
- **Conectividade da origem:** através dos algoritmos de Pesquisa em Largura/Profundidade, será calculada a quantidade de esquadras, prisões e tribunais que podem ser alcançados a partir da origem seleccionada. Estes pontos poderão ser apresentados ao utilizador;
- **Edição da lista de prisioneiros:** caso não haja um caminho possível entre a origem seleccionada ou caso o utilizador queira alterar o destino de um prisioneiro, será possível alterar a lista de prisioneiros;
- **Cálculo do caminho ótimo:** esta opção calculará o caminho ótimo através do algoritmo de Dijkstra/Dijkstra Bidirecional/A\* e apresentará o caminho ótimo sob forma de um grafo;

## **Notas:**

*Numa fase posterior, com a implementação de diferentes tipos de veículos serão apresentadas todas as rotas, quando o utilizador seleccionar a opção de 'Cálculo do caminho ótimo'.*

*Numa fase ainda mais avançada, poderá ser dada a opção do utilizador escolher um tipo de veículo, de modo a transportar um prisioneiro específico.*

# Conclusão

Visto que esta primeira entrega foi maioritariamente teórica, ainda não podemos fazer uma análise detalhada sobre aquilo que o projeto tem ou terá. No entanto, acreditamos que temos um bom plano para começar a implementar as várias funcionalidades mencionadas anteriormente.

Na altura da implementação, esperamos conseguir tratar de forma eficiente de todas as partes, principalmente, do uso da parte gráfica de criação de grafos, visto que será a primeira vez neste curso que faremos uso de algo semelhante.

Todos os membros contribuirão para o trabalho de forma igual, quer fosse a procurar e analisar algoritmos para implementar no trabalho quer fosse a pensar em funcionalidades a implementar.

# Bibliografia

- Apresentações das aulas teóricas de Conceção e Análise de Algoritmos 2019/20, da autoria do Professor Rosaldo Rossetti, Professora Liliana Ferreira, Professor Henrique Cardoso e Professor Francisco Xavier;
- Dijkstra e Problema do Caixeiro Viajante, [http://www.gitta.info/Accessibiliti/en/html/unit\\_Dijkstra.html](http://www.gitta.info/Accessibiliti/en/html/unit_Dijkstra.html)
- Pseudocódigo de Dijkstra Bidirecional, <http://planning.cs.uiuc.edu/node50.html>
- Algoritmo A\*, [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
- Pseudocódigo de A\*, [https://www.researchgate.net/profile/Peter\\_Hufnagl/publication/232085273/figure/fig8/AS:214001028997142@1428033229986/A-search-algorithm-Pseudocode-of-the-A-search-algorithm-operating-with-open-and-closed.png](https://www.researchgate.net/profile/Peter_Hufnagl/publication/232085273/figure/fig8/AS:214001028997142@1428033229986/A-search-algorithm-Pseudocode-of-the-A-search-algorithm-operating-with-open-and-closed.png)
- Programa de visualização de Algoritmos, <https://qiao.github.io/PathFinding.js/visual/>
- OpenStreetMaps, <https://www.openstreetmap.org/>
- Ferramenta de obtenção de tags, <https://overpass-turbo.eu/>