U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto de

# Conceção e Análise de Algoritmos

MIEIC - 2º Ano

# Meat Wagons

**Transporte de prisioneiros**

2ª Entrega

Ricardo Fontão - up201806317@fe.up.pt
Davide Castro - up201806512@fe.up.pt
Henrique Ribeiro - up201806529@fe.up.pt

Turma 4 - Grupo 6

# Índice

# 1. Identificação e descrição do problema do projeto

Uma equipa de transporte de prisioneiros necessita de um programa que calcule a distância mínima possível entre vários locais de interesse, como tribunais, prisões ou esquadras da polícia, de forma a maximizar a eficácia do transporte e da recolha dos prisioneiros.

# 2. Formalização do problema

## 2.1 - Dados de Entrada

**prisioneiros -** conjunto de prisioneiros que contêm o local onde se encontra, onde é para o levar e o seu grau de perigo.

**camionetas** - array com as camionetas que serão usados no deslocamento dos prisioneiros. Cada um é caracterizado por:

-cap : capacidade máxima do meio de transporte

-tipo: tipo de veículo tendo em conta os tipos especializados para locais de diferentes naturezas.

**Gi = (Vi, Ei)** - grafo dirigido pesado composto por:

-V : vértices, que representam pontos do mapa, por sua vez contêm:

-adj : conjunto de arestas que saem do vértice.

-E : arestas, que representam vias de trânsito, por sua vez contêm:

-dest : pointer para a aresta de destino;

-weight : peso da aresta;

-id : identificador da aresta.

**prisonLocation** - vértice onde se inicia a pesquisa.

## 2.2 - Dados de Saída

**Gf = (Vf, Ef)** - grafo dirigido pesado, sendo que Vf e Ef são semelhantes a Vi e Ei (dados de entrada).

**results** - conjunto do caminho que as camionetas irão realizar.

## 2.3 - Restrições

A. Sobre os dados de entrada.

-O conjunto de prisioneiros não pode estar vazio, e o seu grau de perigo tem que estar entre 0 e 5 (∀ p ∈ prisioneiros, 0 ≤ grauPerigo ≤ 5).
-O atributo “maxCapacity” das camionetas tem que ser superior a 0 e inferior a 50 (∀ camioneta ∈ camionetas, 0 ≤ cap ≤ 50) dado que representa uma unidade de medida arbitrária usada para a medição da capacidade das camionetas.
-O conjunto de camionetas nao pode estar vazio.
-O grafo não pode estar vazio.
-∀ e ∈ E, weight>0.
-∀ e ∈ E, o seu vértice origem e destino tem que existir em V.
-O valor de start tem que existir em V.
-Seja F= {v ∈ Vi | v = S V weight(v) > 0} .Todos os vértices pertencentes a F têm que estar fortemente conexos de forma a garantir o bom funcionamento do programa.

B. Sobre os dados de saída

**no grafo:**

-∀ vf ∈ Vf, ∃ vi ∈ Vi tal que vf = vi, exceto no número de prisioneiros que possivelmente será alterado.

-∀ ef ∈ Ef, ∃ ei ∈ Ei tal que ef = ei para todos os parâmetros.

**no conjunto de camionetas:**

-∀ camioneta ∈ camionetasFim, cap = capMax, ou seja, no fim do seu percurso a camioneta tem que ter deixado todos os prisioneiros no local respetivo.

-|camionetasFim| <= |camionetas|.

## 2.4 - Função objetivo

A solução ótima no nosso problema passa por minimizar o número de camionetas utilizados na recolha e entrega dos prisioneiros, tal como obter as rotas necessárias para cada um com a mínima distância percorrida possível. Temos como objetivo então minimizar as seguintes funções:

a = Nº de camionetas ( |camionetas| )

b = Distância percorrida (Soma de todos os pesos de todas as arestas dos paths)

# 3. Perspetiva de solução

## 3.1- Algoritmos de resolução

Para este problema, chegamos à conclusão de que será mais favorável usar o algoritmo de Dijkstra em vez do de Floyd-Warshall pois, neste caso, não é necessário fazer várias vezes o algoritmos de dijkstra, o que faz com que a eficácia deste seja maior (seria necessário realizar por volta de 160000 vezes o algoritmo de dijkstra para igualar o tempo de execução de Floyd-Warshall).

Para complementar o nosso algoritmo de dijkstra foi também implementado um algoritmo de "nearest neighbor" alterado. A alteração consiste em não só a camioneta se dirigir para o vértice mais próximo mas, antes de decidir o caminho, procura pelas restantes camionetas se existe alguma que se encontra mais próxima do vértice que este se pretende dirigir, ou se alguma camioneta, após entregar algum prisioneiro que está a transportar, e que seja do tipo correto para o prisioneiro, se encontrará numa melhor posição para o ir levantar. Caso isto aconteça, o prisioneiro que atualmente está a ser testado será colocado num vetor auxiliar (ignoredVertexes), que são repostos no final da iteração para verificar se estes realmente foram levantados ou se será necessário passar pelos vértices destes para os recolher.

Pseudocódigo de Floyd-Warshall:

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction() is
    for each edge (u, v) do
        dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
        next[u][v] ← v
    for each vertex v do
        dist[v][v] ← 0
        next[v][v] ← v
    for k from 1 to |V| do // standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j]
                    next[i][j] ← next[i][k]
```

```
procedure Path(u, v)
    if next[u][v] = null then
        return []
    path = [u]
    while u ≠ v
        u ← next[u][v]
        path.append(u)
    return path
```

Pseudocódigo de Dijkstra:

```
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)))
```

Tempo de execução:
O((|V|+|E|) * log |V|)

## 3.2- Diferentes passos/iterações na resolução

### 3.2.1- 1º Passo: capacidade ilimitada, um tipo de prisioneiro, um autocarro e um tipo de autocarro

Numa primeira fase, despreza-se o limite de capacidade de um autocarro. Assim, o objetivo trata-se simplesmente de determinar a rota mais curta que começa no estabelecimento prisional, passe em todos os pontos de recolha e entrega planeados dos prisioneiros e termine no sítio onde começou ou noutro estabelecimento prisional. Começamos também por usar apenas um autocarro que realiza todos os transportes.

Pode-se salientar que a recolha e a entrega só pode ser efetuada se existirem caminhos que liguem todos os pontos de interesse (estabelecimento prisional, pontos de recolha e pontos de entrega de prisioneiros). Por outras palavras, todos os pontos de interesse devem fazer parte do mesmo componente fortemente conexo do grafo. Assim, podemos concluir que é necessário efetuar uma análise da conectividade do grafo.

No caso de algumas vias de circulação se encontrarem inacessíveis temos de ignorá-las quando for feito o processamento do grafo.

### 3.2.2- 2º Passo: capacidade ilimitada, um tipo de prisioneiro, vários autocarros e um tipo de autocarro

Na segunda fase do projeto, teremos em conta que passa a existir uma multitude de autocarros disponíveis para o transporte, em vez de apenas um. Temos desta forma de calcular mais que uma rota, uma para cada autocarro. Em princípio cada autocarro termina o seu caminho no último ponto de paragem. Em casos mais complexos poderemos implementar de forma que o autocarro volte ao ponto onde começou.

### 3.2.3- 3º Passo: capacidade ilimitada, um tipo de prisioneiro, vários autocarros e vários tipos de autocarros

Já nesta fase, incluiremos os diferentes tipos de autocarro, usados para diferentes locais de interesse. Como por exemplo, transporte especial para recolha e entrega em aeroportos ou comboios.

### 3.2.4- 4º Passo: capacidade limitada, um tipo de prisioneiro, vários autocarros e vários tipos de autocarros

Nesta fase, restringiremos a capacidade dos autocarros de forma a que possa ser preciso mais de um autocarro do mesmo tipo. Neste caso teremos de minimizar a distância percorrida e a quantidade de autocarros usados.

### 3.2.5- 5º Passo: capacidade limitada, vários tipos de prisioneiros, vários autocarros e vários tipos de autocarros

Na quinta iteração do projeto, vamos considerar que existem múltiplos tipos de prisioneiros, ou seja, vários graus de perigo. Para prisioneiros com maior grau de perigo é necessário mais espaço no autocarro, devido à distância a que deve estar dos restantes e da capacidade ocupada pelos seguranças adicionais.

No fundo, sempre que um prisioneiro tiver que ser recolhido é necessário averiguar se existe esse espaço disponível tendo em conta o grau de perigo. Cada grau de perigo terá uma capacidade mínima por prisioneiro associado.

## 4. Identificação dos casos de utilização a serem suportados, e respectivas funcionalidades a serem implementadas

- Leitura de ficheiros (para inicialização dos vértices e arestas do grafo).
- Menu e visualização do grafo de forma simples e intuitiva.
- Descoberta do caminho mais curto entre o ponto inicial e o ponto final passando por pontos intermédios desejados.
- Consideração de várias variáveis, tal como a capacidade máxima das camionetas e o nível de perigo dos prisioneiro

## 5. Casos de uso implementados

- Um só autocarro com capacidade limitada à escolha e um/vários prisioneiros com nível de perigo (unidades de espaço no autocarro) também à escolha;
- Vários autocarros com capacidade limitada partindo do mesmo destino (prisão), recolhendo qualquer prisioneiro (tipo “Any”)
- Vários autocarros com capacidade limitada partindo do mesmo destino podendo recolher prisioneiros de acordo com o seu tipo especializado (airports, trains, regular) e o destino dos prisioneiros. Se não houver autocarro do tipo certo para um certo prisioneiro este pode ser levado por qualquer um dos outros

## 6. Estrutura de dados utilizados

- Heap usando um vetor como container: usado para ordenar os vértices importantes que cada camioneta terá que passar.
- Vetor de Vértices- ignoredVertexes: vetor utilizado para guardar os vértices ignorados pelo algoritmo na iteração atual de forma a ser possível repô-los no final desta uma vez que ainda são vértices que a camioneta poderá ter que visitar.
- Bus declarado na classe busManager: estrutura criada para o nosso projeto. Contém toda a informação da camioneta como a capacidade máxima/atual, o seu tipo e a última localização visitada.
- Prisoner declarado na classe busManager: estrutura criada para o nosso projeto. Contém toda a informação do prisioneiro como o seu ponto de partida/destino, se já foi levantado/entregue, entre outras.
- Vetor de camionetas - buses: vetor que contém as camionetas.
- Vetor de prisioneiro - prisoners: vetor que contém os prisioneiros.
- Vetor bidimensional de inteiros - results: vetor utilizado para armazenar o caminho percorrido por cada camioneta.
- Enum - busType: enum usada para identificar o tipo de camioneta que está a ser usada ou que tipo de camioneta correspondente a cada prisioneiro

# 7. Algoritmos implementados

**Algorithm 1** Calculate best paths for buses

```
1: Input: prisioners, buses, graph
2: Output: paths of all buses
3:
4: results ← [ ]
5: for all prisoners do
6:   if !findBusType(prisoner) then
7:     setType(prisoner, 'Any')
8:   end if
9: end for
10:
11: for all buses do
12:   destinations ← [ ]
13:   for all prisoners do
14:     if hasPath(bus,prisoner) then
15:       if checkType(bus,prisoner) then
16:         add(destinations, prisoner)
17:       end if
18:     else
19:       erase(prisoner)
20:     end if
21:   end for
22:   bus.destinations ← destinations
23: end for
24:
25: while any bus destinations not empty do
26:   ignoredVertexes ← [ ]
27:   for all buses do
28:     if !empty(bus.destinations) then
29:       hadAction ← false
30:       dijkstra(bus.location)
31:       nextDest ← pop(bus.destinations)
32:       pathToNext ← getPath(bus.location, nextDest)
33:       currentLocation ← bus.location
34:       bus.location ← nextDest
35:       prisonerLoop:
36:       for all prisoners do
37:         if prisioner.weight > bus.capacity then
38:           go to prisonerLoop
39:         end if
40:         if prisioner.start = bus.location and !pickedUp(prisoner) and
    checkType(bus,prisoner) then
41:           currentNextDist ← dist(nextDest)
42:           for all buses as bus2 do
43:             djikstra(bus2.location)
44:             nextDist ← dist(nextDest)
45:             ignoreDest ← false
46:             for all prisoners as prisoner1 do
47:               if bus!=bus2 and pickedUp(prisoner1) and
    !delivered(prisoner1) and getBus(prisoner1)=bus2 and
    checkType(bus2,prisoner1) then
48:                 dijkstra(destination(prisoner1))
49:                 if dist(nextDest)< nextDist then
50:                   add(ignoredVertexes, nextDest)
51:                   ignoreDest ← true
52:                   break loop
53:                 end if
54:               end if
55:             end for
56:             if ignoreDest then
57:               break loop
58:             end if
59:             if bus!=bus2 and currentNextDist > nextDist and
    checkType(bus2,prisoner) then
60:               add(ignoredVertexes,nextDest)
61:               break loop
62:             else if bus2 = lastBus then
63:               if canFit(bus, prisoner) then
64:                 pickUp(prisoner, bus)
65:                 add(bus.destinations, destination(prisoner))
66:               end if
67:             end if
68:           end for
69:         else if destination(prisoner)=bus.location and
    pickedUp(prisoner) and getBus(prisoner)=bus then
70:           deliver(prisoner, bus)
71:           hadAction ← true
72:         end if
73:       end for
74:       if hadAction or empty(bus.destinations) then
75:         if !hadAction then
76:           bus.location ← currentLocation
77:         end if
78:         for all ignoredVertexes do
79:           addDestination(bus, ignoredVertex)
80:         end for
81:         ignoredVertexes ← [ ]
82:       else
83:         bus.location ← currentLocation
84:         repeat loop iteration
85:       end if
86:       if hadAction then
87:         results[bus] ← pathToNext
88:       end if
89:     end if
90:   end for
91: end while
92:
93: returns results
```

## Algumas considerações no código:

```
4: results ← [ ]
5: for all prisoners do
6:   if !findBusType(prisoner) then
7:     setType(prisoner, 'Any')
8:   end if
9: end for
10:
11: for all buses do
12:   destinations ← [ ]
13:   for all prisoners do
14:     if hasPath(bus,prisoner) then
15:       if checkType(bus,prisoner) then
16:         add(destinations, prisoner)
17:       end if
18:     else
19:       erase(prisoner)
20:     end if
21:   end for
22:   bus.destinations ← destinations
23: end for
```

Inicialmente, executamos um pré-processamento dos dados. Neste caso, primeiro verificamos se existe pelo menos um autocarro para o tipo de todos os prisioneiros (se vai para um aeroporto, comboio, etc.). Se não existir o prisioneiro passa a ter o tipo 'Any', ou seja, qualquer autocarro o pode transportar.
Seguidamente, adicionamos ao vetor de destinos de cada autocarro as start locations de todos os prisioneiros que podem ser recolhidos por este, isto é, existe um caminho para o prisioneiro. Se não houver caminho, o prisioneiro é ignorado.

```
36:     for all prisoners do
37:       if prisioner.weight > bus.capacity then
38:         go to prisonerLoop
39:       end if
40:       if prisioner.start = bus.location and !pickedUp(prisoner) and
    checkType(bus,prisoner) then
41:         currentNextDist ← dist(nextDest)
42:         for all buses as bus2 do
43:           djikstra(bus2.location)
44:           nextDist ← dist(nextDest)
45:           ignoreDest ← false
46:           for all prisoners as prisoner1 do
47:             if bus!=bus2 and pickedUp(prisoner1) and
    !delivered(prisoner1) and getBus(prisoner1)=bus2 and
    checkType(bus2,prisoner1) then
48:               dijkstra(destination(prisoner1))
49:               if dist(nextDest)< nextDist then
50:                 add(ignoredVertexes, nextDest)
51:                 ignoreDest ← true
52:                 break loop
53:               end if
54:             end if
55:           end for
56:           if ignoreDest then
57:             break loop
58:           end if
59:           if bus!=bus2 and currentNextDist > nextDist and
    checkType(bus2,prisoner) then
60:             add(ignoredVertexes,nextDest)
61:             break loop
62:           else if bus2 = lastBus then
63:             if canFit(bus, prisoner) then
64:               pickUp(prisoner, bus)
65:               add(bus.destinations, destination(prisoner))
66:             end if
67:           end if
68:         end for
69:       else if destination(prisoner)=bus.location and
    pickedUp(prisoner) and getBus(prisoner)=bus then
70:         deliver(prisoner, bus)
71:         hadAction ← true
72:       end if
73:     end for
```

Nesta secção do algoritmo, percorremos os prisioneiros em cada autocarro. Se não tiver sido recolhido e seja compatível com ele, é executado um loop pelos outros autocarros, para verificar se algum deles está a transportar um prisioneiro com destino próximo de nextDest. Neste caso, o autocarro em questão ignora temporariamente o nextDest, pois é provável que o outro consiga melhores resultados. Também acontece isto caso o outro autocarro em si esteja localizado mais perto do nextDest do que este.
Com estas duas situações em mente é possível minimizar o tamanho das viajens dos autocarros, pois têm em consideração os outros autocarros e os caminhos que eles estão previstos a tomar para tomar a decisão de onde ir a seguir.

# 8. Análise de complexidade

## 8.1. Análise temporal

O algoritmo usado tem complexidade:

$$O( P^3B^2RD^2 + PBRD + PB^2R + PRBlog(B) + 3PB + R^2 )$$

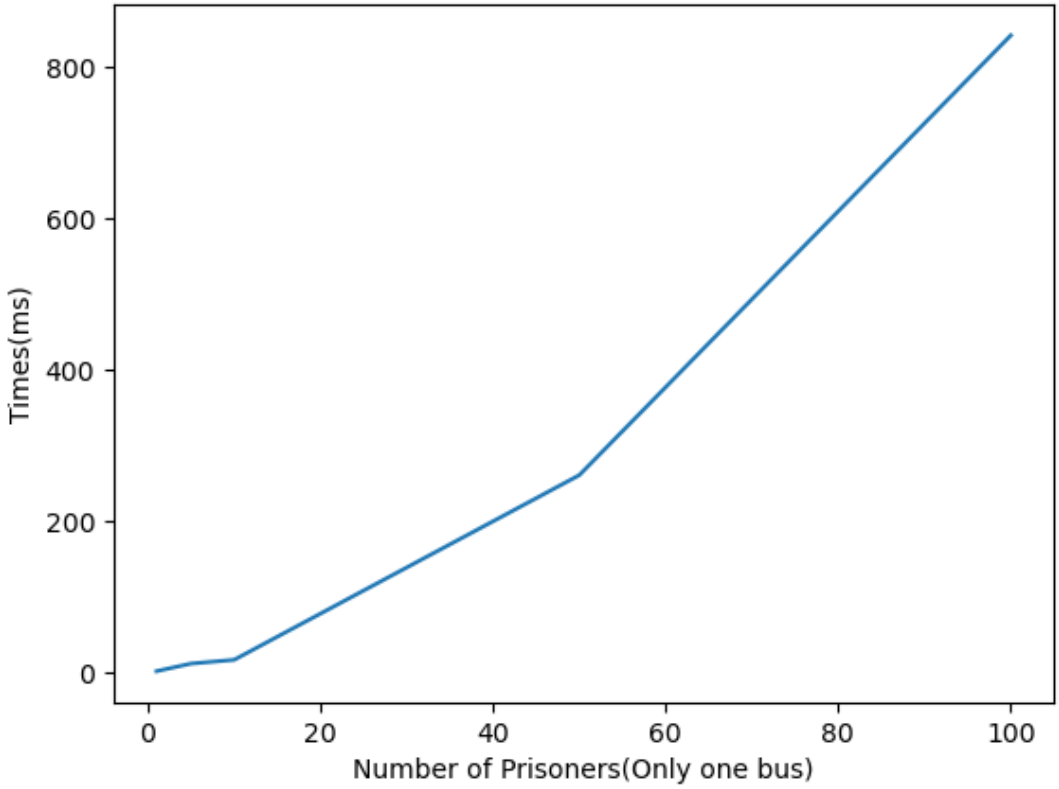
sendo:

P - Número de prisioneiros

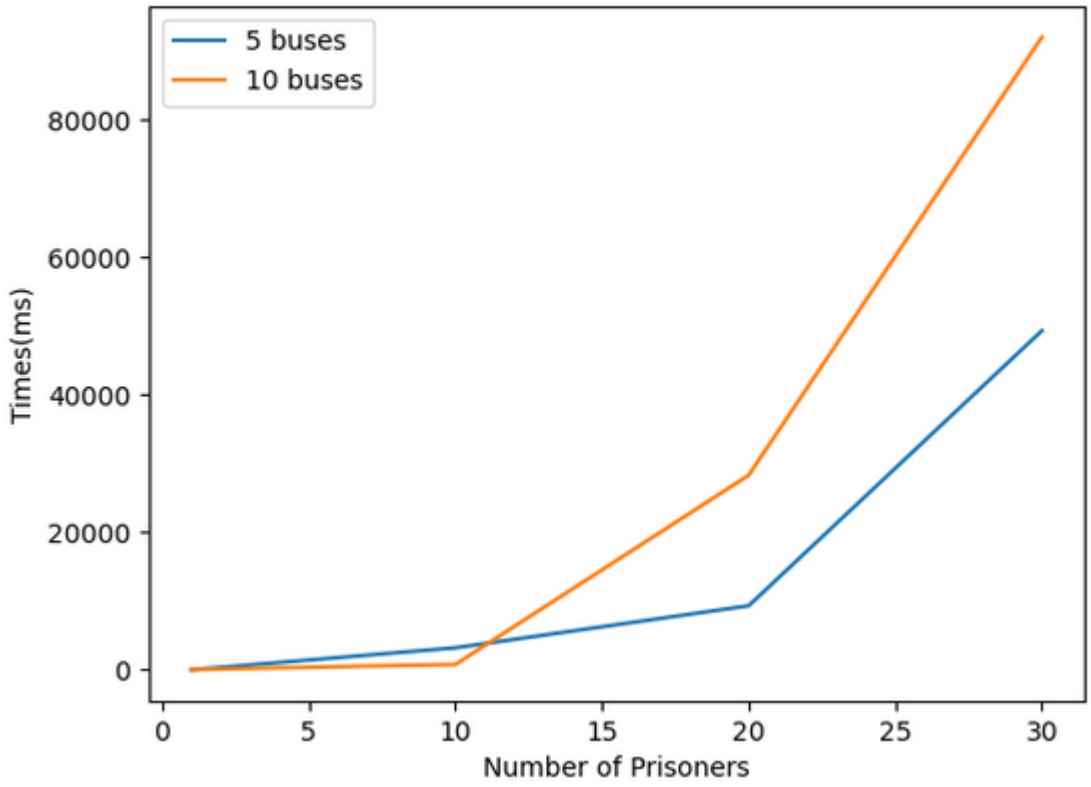
B - Número de camionetas

R - Tamanho do vetor de resultado

D - Complexidade de Dijkstra




Tempo total do algoritmo com apenas um autocarro




Tempo total do algoritmo com 5 e 10 autocarros

## 8.2 Análise espacial

-Buses : Vetor com as camionetas(O(B) sendo B o número de camionetas)

-Bus: Camioneta com um vetor de vértices de destino e um vetor de pares de ints, strings(O(D + IS) sendo D o número de vértices de destino e IS o número de pares)

-Prisoners : Vetor de prisioneiros(O(P) sendo P o número de prisioneiros)

−Results: Vetor bidimensional de inteiros onde são guardados os índices dos vértices dos paths (O($\sum_B$ BP) sendo B o número de camionetas disponíveis e BP o path de cada uma).

-ignoredVertexes: Vetor com vertices ignorados durante a iteração(O(V) sendo V o número de vértices)

-pathToNext: Vetor que contém os índices dos vértices que formam o caminho deste o vértice atual da camioneta até ao seu destino(O(I) sendo I o número de índices de vértices do caminho)

Desta forma a complexidade espacial do algoritmo é :

$O(B + (D + IS) + P + \sum_B BP + V + I)$

## 9. Conectividade do grafo usado

No nosso caso, utilizamos um grafo fortemente conexo, de forma a evitar que um prisioneiro inserido num ponto que não teria caminho possível seja ignorado pelo algoritmo, pois é feita uma verificação no início do algoritmo que filtra os prisioneiros que causariam uma situação em que o autocarro não teria caminho para o ir recolher ou caminho para regressar depois da recolha.

# Conclusão

Em suma, conseguimos completar todos os objetivos que tínhamos proposto inicialmente, obtendo as várias iterações com sucesso. O algoritmo conseguido poderia ser mais otimizado, em particular no caso da complexidade temporal, se houvesse mais tempo para efetuar melhorias, mas, no geral, consideramos que, com este algoritmo, obtêm-se soluções boas face ao problema apresentado.

- ## Esforço por cada elemento

**Davide**:

- Menus, gestão de prisioneiros e autocarros, mostrar caminhos,
- Leitura de dados dos ficheiros,
- Implementação da primeira versão com heap(priority_queue) para os destinos mais próximos de cada autocarro,
- Implementação de vários tipos de autocarros,
- Colaboração nas tags,
- Implementação da otimização no caso de autocarros que têm prisioneiros com destinos mais próximos que o atual,
- Implementação da classe Bus,
- Colaboração geral na produção, melhorias e correções do código e do algoritmo,
- Relatório: Pseudocódigo, casos de uso, conectividade.

**Henrique**:

- Correção da primeira parte, ponto 6 e ponto 8.2 do relatório.
- Implementado o algoritmos de múltiplas camionetas com capacidade infinita.
- Implementado o algoritmo onde a posição atual do autocarro é comparada com a dos restantes para tentar optimizar o caminho mínimo.
- Implementado a capacidade limitada nos autocarros.
- Ajuda geral no código.
- Relatório:

**Ricardo**:

- Implementação inicial do pathfinding(prisão->destino)
- Implementação inicial das class Renderer e BusManager
- Adaptação do mapa de Espinho do OpenStreetMaps
- Implementação da importação e display de mapas
- Implementação do perigo de cada prisioneiro
- Relatório: Análise temporal (gráficos e complexidade)

- Implementação inicial de tags
- Ajuda geral no código

**Em conjunto**: Maior parte do algoritmo principal teve a colaboração conjunta de todos os elementos a trabalhar em grupo para as diversas iterações.

## Webgrafia/Bibliografia


- Wikipedia contributors, "Floyd–Warshall algorithm," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Floyd%E2%80%93Warshall_algorithm&oldid=948011949 (acedido a 18 de abril, 2020)

- Comparison of Dijkstra's and Floyd–Warshall algorithms, GeeksforGeeks, https://www.geeksforgeeks.org/comparison-dijkstras-floyd-warshall-algorithms/ (acedido a 18 de abril, 2020)

- "Introduction to Algorithms", 3rd Edition, T.H. Cormen,C. E. Leiserson,R. L. Rivest , C. Stein., MIT Press, 2009 - Capítulo 24 (Single-Source Shortest Paths)