Meat Wagons

Conceção e Análise de Algoritmos

2019/2020

Turma 3 – Grupo 1

André Gomes – up201806224@fe.up.pt

Gonçalo Alves – up201806451@fe.up.pt

Pedro Seixas – up201806227@fe.up.pt

23 de abril, 2020

Índice

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 sobrarem 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 OpenStreetMap;
- 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-nésimo elemento, cada um é caracterizado por:

- **ID** - Número identificador de prisioneiro
- **Destino** - Identificador 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, com número de dígitos variável, é correspondente ao ID do node de destino. Como exemplo: Um tribunal poderá ter um Destino de 1000001 enquanto que um estabelecimento prisional poderá ter um Destino de 2000001.

Fi: Lista de veículos da frota, sendo F(i) o i-nésimo 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 -> sentido da rua, Pesado -> 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 ⊆ 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** ∈ Vi - Vértice de destino

S ∈ Vi: Vértice inicial (Estabelecimento prisional)

T ⊆ Vi: 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-nésimo 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

- **∀ i ∈ [1; |Fi|], cap(Fi[i]) > 0**, dado que uma capacidade representa os assentos disponíveis
- **∀ v ∈ Vi, tag ≥ 0**
- **∀ e ∈ Ei, w > 0**, dado que o peso de uma aresta representa uma distância entre pontos de um mapa
- **∀ e ∈ Ei**, e deve ser utilizável pelo elemento da frota. Senão, não é incluída no grafo Gi

Sobre os Dados de Saída

No grafo Gf:

- **∀ vf ∈ Vf, ∃ vi ∈ Vi** tal que vi e vf têm os mesmos valores para todos os atributos
- **∀ ef ∈ Ef, ∃ ei ∈ Ei** 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: ∑ dos valores das arestas percorridas pelos elementos da frota
2. Diminuir o tempo de execução do cálculo das rotas (∑ tempos de execução dos algoritmos)
3. Diminuir o número de veículos da frota usados (h = |Ff|)

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 Gr)
- Segunda pesquisa em profundidade, em Gr, 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()
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

```
1. for each v in V do
2.  dist(v) <- INF
3.  path(v) <- nil
4. dist(s) <- 0
5. Q <- 0 // min-priority queue
6. INSERT(Q, (s, 0)) // inserts s with key 0
7. while Q != 0 do
8.  v <- EXTRACT-MIN(Q)
9.  for each w in 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 not in Q then
14.    INSERT(Q, (w, dist(w)))
15.   else
16.    DECREASE-KEY(Q, (w, dist(w)))
```

BIDIRECTIONAL DIJKSTRA (G, s)

```
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 <- Qi.GetFirst()
6.  if x = xg or x ∈ Qg
7.    return SUCCESS
8.  forall u ∈ U(x)
9.     x' <- 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' <- Q.GetFirst()
17.  if x' = xi or x' ∈ Qi
18.    return SUCCESS
19.  forall u^(-1) ∈ U^(-1)(x')
20.     x <- f^(-1)(x',u^(-1))
21.   if x not visited
22.    Mark x as visited
23.    Qg.Insert(x)
24.   else
25.    Resolve duplicate x
26. return FAILURE
```

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:

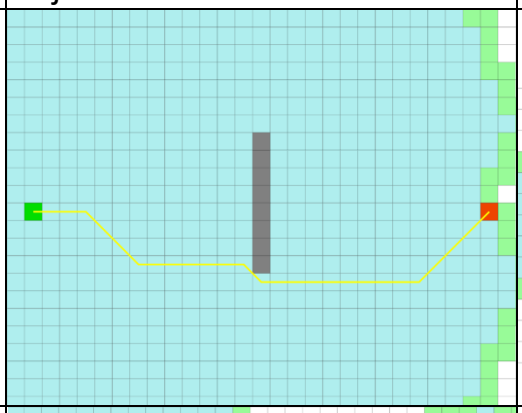
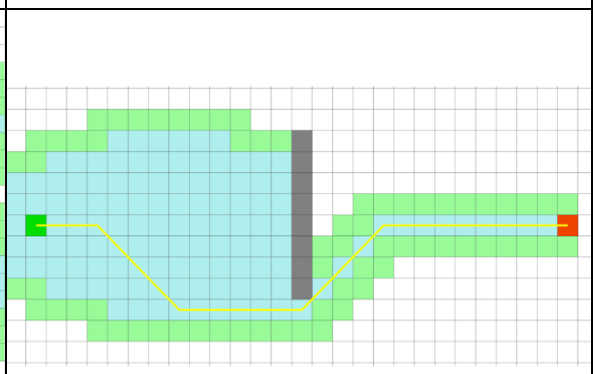
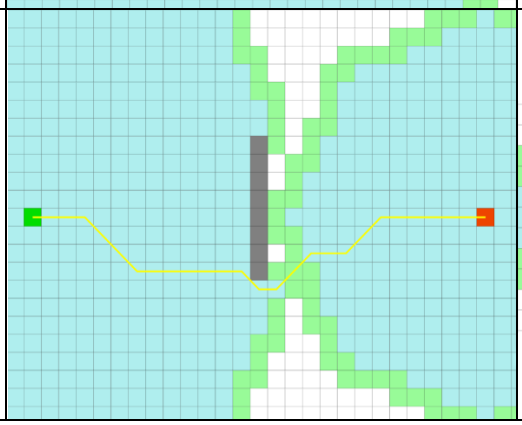
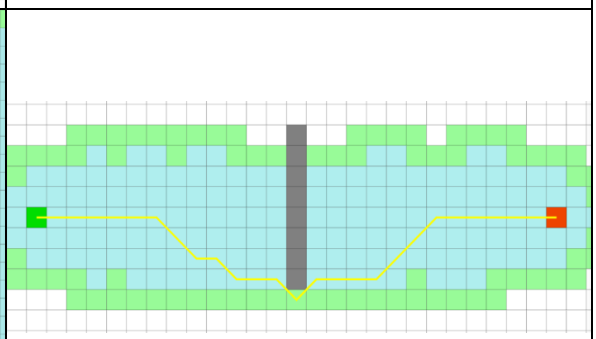
$f(v) = h(v) + g(v)$

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
```

	Dijkstra	A*
Normal		
Bidirecional		

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 selecionar o distrito que pretende explorar.
Após essa seleção, o utilizador poderá selecionar 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 seleção, 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 selecionada. Estes pontos poderão ser apresentados ao utilizador;
- **Edição da lista de prisioneiros**: caso não haja um caminho possível entre a origem selecionada 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 selecionar a opção de 'Cálculo do caminho ótimo'.
Numa fase ainda mais avançada, poderá ser dada a opção de o utilizador escolher um tipo de veículo, de modo a transportar um prisioneiro específico.

2ª Parte

Após um feedback sobre a primeira entrega do trabalho, foi necessário reestruturar o nosso trabalho visto que os objetivos do grupo e do professor divergiam.

Tal como descrito na primeira parte, acreditávamos que o objetivo deste projeto seria a procura de um caminho ótimo para o transporte de prisioneiros, com vários veículos à nossa disposição. Além disso, como pensávamos que conseguiríamos usar o mapa de Portugal inteiro, estaríamos à espera de usar a densidade populacional como fator para cálculo do caminho ótimo. Este fator não foi usado devido aos mapas fornecidos.

O nosso objetivo foi alterado pois o professor comentou que seria mais interessante em cumprir horários de entrega de prisioneiros. Ou seja, se anteriormente procurávamos o ponto de interesse mais próximo (destino de um prisioneiro) da origem, agora o nosso caminho "ótimo" seria em função do tempo de entrega.

Assim, foram adicionadas funcionalidades que anteriormente não foram previstas, que serão mencionadas na secção seguinte

Principais casos de uso implementados

Nesta secção serão descritos, em detalhe, todos os casos de uso implementados.

No primeiro menu é dada a opção de o utilizador escolher o mapa do Porto completo ou do mapa fortemente conexo. De seguida, é dito ao utilizador para selecionar um ponto de origem (este ponto faz parte dos pontos de interesse do grafo).

Após esta seleção, o utilizador tem três opções: Menu de Prisioneiros/Veículos; Visualização do Grafo; Comparação de Algoritmos;

No menu de Prisioneiros/Veículos, o utilizador pode:

- Adicionar e remover um prisioneiro/veículo;
- Visualizar os prisioneiros/veículos;
- Mudar o veículo em que um prisioneiro será transportado;
- Usar um setup para exemplo com: três prisioneiros, de destinos diferentes, e dois autocarros;

No menu de Visualização do Grafo, o utilizador pode:

- Trocar o tipo de mapa (completo ou conexo) e o ponto de origem;
- Visualizar uma lista dos Pontos de Interesse do Mapa;
- Visualizar o mapa;
- Visualizar o caminho mais curto: seguindo a lógica do Vizinho Mais Próximo; tendo em conta a hora a que o prisioneiro tem de chegar ao seu destino; tendo em conta as duas opções apresentadas anteriormente;
- Visualizar a conectividade do grafo (mais interessante no mapa completo): total, usando o algoritmo de pesquisa DFS; a partir do ponto de origem selecionado, usando o algoritmo de pesquisa BFS;

No menu de Comparação de Algoritmos, o utilizador pode:

- Correr uma comparação entre os algoritmos de Dijkstra e A*, sendo-lhe mostrado o tamanho do caminho (tempo demorado a percorrer em segundos) e o tempo de execução dos dois algoritmos;
- Correr uma comparação entre os algoritmos DFS e BFS, sendo-lhe mostrado o tamanho do grafo (número de nodes) e o tempo de execução dos dois algoritmos;
- Correr uma comparação entre os algoritmos A* e ALT, ou Dijkstra e ALT, sendo estas comparações não viáveis pois o algoritmo ALT não se encontra a funcionar devidamente.

Estruturas de dados utilizadas

Grafo

A representação de um **Grafo** foi feita com base nesta estrutura, definida em *Graph.h*. Esta classe tem como base aquela disponibilizada nas aulas práticas, com algumas modificações. A classe Graph contem: um vetor de apontadores de **Vertex**; um mapa com os ids dos **Vertex**, de modo a facilitar a procura de um; um vetor de **POI**; variáveis max e min para as coordenadas, de modo a ajudar na representação gráfica do **Grafo**; funções de adição e procura de **Vertex** e **POI**; funções que implementam algoritmos de pesquisa e de obtenção de caminho ótimo.

Os algoritmos associados ao **Grafo** são: **DFS**, **BFS**, **Dijkstra**, **A*** e **ALT**. Este último demonstrou-se extremamente eficiente no cálculo de um caminho, para os mapas "Grid", pois fazia um pré-processamento dos vértices do **Grafo**. No entanto, devido a um elevado tempo de pré-processamento, foi decidido não usar este algoritmo para os mapas de cidades.

As classes **Vertex** e **Edge**, também adaptadas das classes disponibilizadas, mantêm-se relativamente iguais, com exceção da adição de uma variável "tag" à classe **Vertex**, de modo a distinguir um **POI** de um **Vertex** normal.

Path

A representação de um caminho foi feita com base nesta estrutura, definida em *Path.h*. Esta classe é constituída por: um vetor de id's de **Vertex**; uma variável de comprimento do caminho e um mapa com os ids e o tempo necessário para chegar a um **POI**.

GraphViewer

Esta classe foi fornecida pelos professores e não sofreu qualquer tipo de modificação.

GUI

Esta classe, definida em *GUI.h*, é constituída por: um **Grafo**, um **GraphViewer** e a altura e comprimento da janela do **GraphViewer**. É responsável por fazer a representação gráfica do **Grafo** e dos caminhos ótimos.

Prisoner

Esta classe, definida em *Prisoner.h*, é constituída por: um id, um nome, uma idade, um destino (sob forma de um id) e um tempo de chegada ao destino. Foram criadas funções para obter os destinos de todos os prisioneiros, comparar o tempo de chegada ao destino e ordenar o vetor de prisioneiros pelo seu tempo.

Vehicle

Esta classe, definida em *Vehicle.h*, é constituída por: uma capacidade, uma velocidade máxima e um vetor de apontadores de **Prisoner**

Time

Time é uma classe auxiliar, definida em *Time.h*, que permite fazer operações mais facilmente sobre o tempo, sendo útil na gestão de tempo de entrega dos **Prisoner**.

Funções Auxiliares

Nos ficheiros *Parser.h* e *utils.h* há várias funções úteis para o desenvolvimento do trabalho, tais como: funções de parsing de ficheiros de nodes/edges/tags; funções de leitura de números/string; funções de comparação de algoritmos.

Algoritmos Efetivamente Implementados

DFS

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

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

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

O algoritmo DFS (Depth-First Search) explora todas as arestas, a partir do vértice mais recentemente descoberto. Este algoritmo é útil para a avaliação da conectividade de um grafo.

A **complexidade temporal** deste algoritmo é **O(|V| + |E|)**, ou seja, linear no tamanho total do grafo (|V| e |E| representam o número de vértices e arestas do grafo, respetivamente). Cada vértice é visitado, no máximo, uma vez e a pesquisa é realizada a partir de cada vértice visitado para todos os seus vértices adjacentes a partir das arestas que os unem. A inserção num vetor, no pior caso, é de complexidade **O(|V|)**, logo não altera a complexidade temporal. Quanto ao espaço, por ser um algoritmo recursivo, irá ter complexidade espacial **O(|V|)**, no seu pior caso. Assim, a complexidade espacial deste algoritmo é **O(|V|).**

BFS

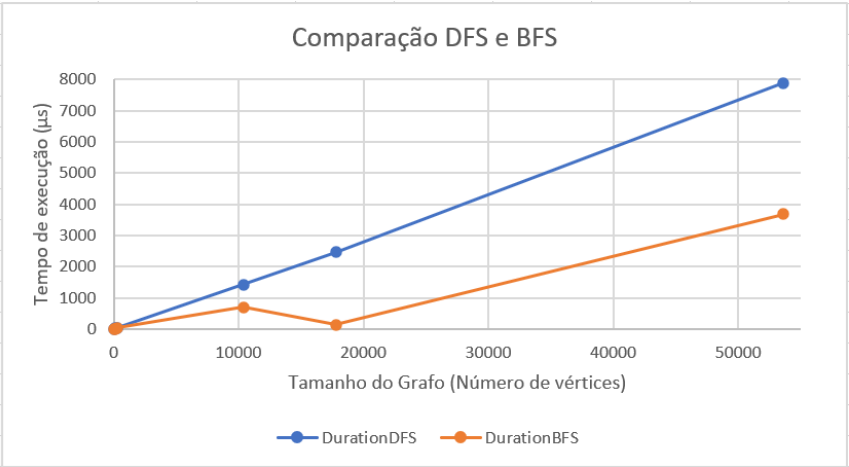
```
G = (V, E)
Adj(v) = {w | (v, w) ∈  E} (∀ v ∈ V)

BFS(G, s):
1.    visitedVertex{}
2.    for each v ∈  V
3.          visited(v) ← false
4.    v = s
5.    Q ← ∅
6.    ENQUEUE(Q, v)
7.    visited(v) ← true
8.    while Q ≠ ∅ do
9.          v ← DEQUEUE(Q)
10.         insertVisited(v)
11.         for each w ∈  Adj(v)
12.                   if not visited(w)
13.                         ENQUEUE(Q, w)
14.                         visited(w) ← true
15.   return visitedVertex
```

O algoritmo BFS (Breadth-First Search) explora todas as arestas, a partir do vértice definido como origem. Este algoritmo também é útil para a avaliação da conectividade de um grafo.

A **complexidade temporal** deste algoritmo é **O(|V| + |E|)**, ou seja, linear no tamanho total do grafo (|V| e |E| representam o número de vértices e arestas do grafo, respetivamente). Cada vértice é visitado, no máximo, uma vez e a pesquisa é realizada a partir de cada vértice visitado para todos os seus vértices adjacentes a partir das arestas que os unem. A inserção num vetor, no pior caso, é de complexidade **O(|V|)**, logo não altera a complexidade temporal. Quanto ao espaço, no pior caso, a fila terá **|V|** elementos. Assim, a complexidade espacial deste algoritmo é **O(|V|).**

De modo a confirmar esta análise teórica, foram feitos testes experimentais. Os grafos utilizados para os testes foram os mapas “Grid” e os mapas “Full”, disponibilizados pelos professores. Os resultados obtidos podem ser observados no gráfico seguinte:

Graph Size	DurationDFS	DurationBFS
25	14	12
81	18	13
289	42	34
10365	1417	691
17772	2461	146
53621	7886	3676



Dijkstra

```
Dijkstra(G, s, d)
   1. for each v in V do
   2.    dist(v) ← INF
   3.    path(v) ← nil
   4. dist(s) ← 0
   5. Q ← ∅ // min-priority quele
   6. INSERT(Q, (s, 0)) // inserts s with key 0
   7. while Q != ∅ do
   8.       v ←  EXTRACT-MIN(Q)
   9.       for each w in 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 not in Q then
   14.            INSERT(Q, (w, dist(w)))
   15.         else
   16.            DECREASE-KEY(Q, (w, dist(w)))
   17. p ← ∅
   18. INSERT(p,d)
   19. length ← 0
   20. while path(d) != ∅ do
   21.      length += cost(path(d))
   22.      d ← path(d)
   23.      INSERTBEGIN(p,id(d))
   24. return Path(length,p)
```

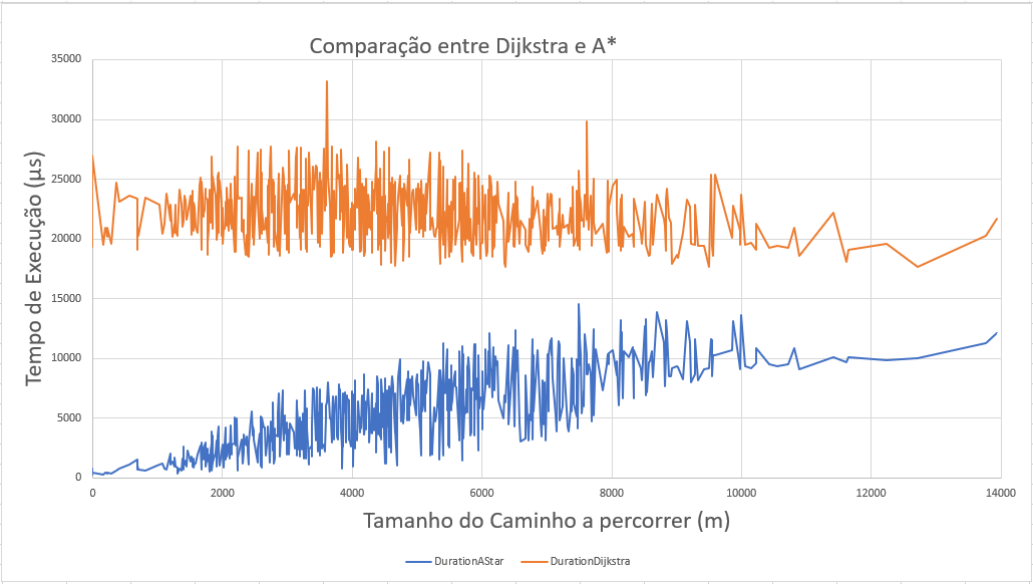
O algoritmo de Dijkstra calcula o melhor caminho entre dois vértices de um grafo. A primeira fase, de preparação, tem complexidade temporal **O(|V|)**. A segunda fase, do algoritmo em si, é feita em tempo logarítmico **O(log |V|)**. Assim, a complexidade temporal do algoritmo será **O((|V| + |E|) * log |V|)** e a sua complexidade espacial será **O(|V|)**.

A*

```
A*(G, s, d)
   1. for each v in V do
   2.  dist(v) ← INF
   3.  path(v) ← nil
   4. dist(s) ← euclidianDistance(s,d)
   5. Q ← ∅ // min-priority quele
   6. INSERT(Q, (s, 0)) // inserts s with key 0
   7. while Q != ∅ do
   8.       v ←  EXTRACT-MIN(Q)
   9.       for each w in Adj(v) do
   10.      if dist(w) > euclidianDistance(dest(w),d) then
   11.       dist(w) ← dist(v)+ weight(v,w)
   12.       path(w) ← v
   13.       if w not in Q then
   14.        INSERT(Q, (w, dist(w)))
   15.       else
   16.        DECREASE-KEY(Q, (w, dist(w)))
   17. p ← ∅
   18. INSERT(p,d)
   19. length ← 0
   20. while path(d) != ∅ do
   21.      length += cost(path(d))
   22.      d ←path(d)
   23.      INSERTBEGIN(p,id(d))
   24. return Path(length,p)
```

O algoritmo A*, muito semelhante ao de Dijkstra, calcula o melhor caminho entre dois vértices de um grafo. A primeira fase, de preparação, tem complexidade temporal **O(|V|)**. A segunda fase, do algoritmo em si, é feita em tempo logarítmico **O(log |V|)**. Assim, a **complexidade temporal** do algoritmo será **O((|V| + |E|) * log |V|)** e a sua **complexidade espacial** será **O(|V|)**.

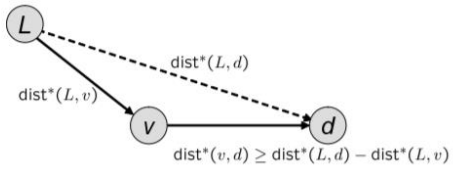
De modo a confirmar esta análise teórica, foram feitos testes experimentais. Os grafos utilizados para os testes foram os mapas "Strong", disponibilizados pelos professores, onde se utilizaram como pontos de origem e pontos de destino, todos os POI. Os resultados obtidos podem ser observados no gráfico seguinte:



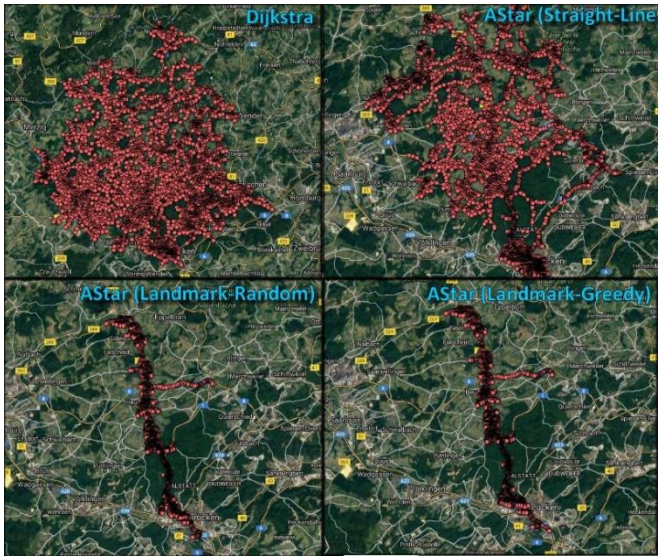
ALT

Após alguma pesquisa, descobrimos uma variação do algoritmo A*, o ALT (A*, Landmarks and Triangle inequality). Este algoritmo seria uma otimização no nosso cálculo de caminho mais curto pois tem em conta situações em que, por exemplo, é necessário dirigir-se a um node que está mais afastado do nosso destino, mas que será mais rápido. Um exemplo disto seria uma autoestrada, pois às vezes, é necessário percorrer o caminho contrário até chegar à autoestrada, no entanto quando se entra nela, o caminho será percorrido mais rapidamente.

O ALT usa um grupo de vértices chamados landmarks e calcula a distância (*v*,d) baseado na seguinte desigualdade de triângulos, onde L é o vértice landmark:



Esta função heurística permite uma maior eficiência na pesquisa de caminhos mais curtos em grafos que retratam estradas como mostra a seguinte imagem:



Como na fase do planeamento do projeto pensávamos em utilizar um mapa inteiro de Portugal, este algoritmo seria extremamente útil.

Após a tentativa de implementação o algoritmo não se demonstrou mais eficiente do que o A*, talvez devido ao modo como implementado, à escolha dos landmarks ou ao tipo de estruturas de dados usados e, como estávamos limitados a distritos, decidimos então não aplicar este algoritmo no cálculo do caminho ótimo.

Nearest Neighbour

```
NearestNeighbour(s, pois, p)
   1. if p = ∅ then
   2.       INSERT(p, s)
   3. if pois = ∅ then
   4.       return p
   5. next = nearestShortestPath(s, pois)
   6. JOIN(p,next)
   7. ERASE(pois, lastNode(p))
   8.
   9. return NearestNeighbour(lastNode(p), pois, p)

nearestShortestPath(s, pois)
   1. length(p) ← INF
   2. path(p) ← nil
   3.
   4. for POI in pois
   5.       newpath ← shortestPath(s, POI)
   6.       if length(newpath) < length(p) then
   7.             p ← newpath
   8. return path
```

O algoritmo de Nearest Neighbour é uma possível solução para o "Problema do Caixeiro Viajante", problema de deslocamento entre dois pontos passando por um conjunto de pontos de interesse. Este algoritmo percorre os POI's todos e calcula o caminho ótimo para aquele que estiver mais próximo. Após este cálculo, o ponto que estava mais próximo da origem tornar-se-á na nova origem e o algoritmo continua o cálculo do caminho ótimo para o próximo ponto. O cálculo de caminho ótimo termina quando se esgotarem todos os pontos de origem.

No caso do nosso trabalho, este algoritmo é especialmente útil, pois permite-nos ter um caminho ótimo para um percurso em que tenhamos de passar por vários POI's numa viagem só.

Fizemos uso do algoritmo A* para auxiliar o Nearest Neighbour, visto que este apresentava uma melhor performance relativamente ao algoritmo de Dijkstra. Este algoritmo tem **complexidade temporal O((|V| + |E|) * log |V| * |N|)**, pois necessita de percorrer todos elementos do vetor de pontos de interesse (N) e faz uso do algoritmo A*. A sua **complexidade espacial** é **O(|V*N|)**, pois estamos a recorrer a um algoritmo recursivo combinado com o algoritmo A*.

Conectividade dos grafos

No menu de Visualização do Grafo, é possível analisar a conectividade do Grafo de duas formas:

A opção “Show Connectivity”, que implementa o algoritmo DFS, percorre todos os vértices do grafo e mostra ao utilizador o mapa com os vértices que fazem parte da conectividade total do mapa.

A opção “Show Connectivity from Origin”, que implementa o algoritmo BFS, ao contrário da opção anterior, apenas percorre os vértices que são possíveis de atingir, partindo de um ponto de origem selecionado pelo utilizador.

Conclusão Final

Consideramos que o tema do trabalho é extremamente interessante, que tem uma componente prática muito útil para situações da vida real e além disso, tem um grau de dificuldade desafiante. Ao longo do desenvolvimento deste projeto, através da pesquisa de algoritmos, encontrámos várias soluções interessantes para: otimizações de algoritmos de pesquisa, otimizações de algoritmos de conectividade e possíveis soluções para o "Problema do Caixeiro Viajante", e como tal, aprofundámos o nosso conhecimento da unidade curricular e o nosso interesse pelos temas.

No entanto, a organização e o apoio ao trabalho deixam um bocado a desejar. O tempo dado para a execução deste é muito reduzido e, infelizmente, a entrega de mapas úteis para o trabalho (à exceção dos mapas "Grid") na semana anterior à entrega deste, dificultou a nossa tarefa. A alteração do objetivo do trabalho também trouxe novos desafios à implementação deste, no entanto conseguimos ultrapassá-los.
Quanto ao apoio dado, sentimos que a comunicação não foi a melhor.

Esforço dedicado: André Daniel (Organização do Git, frota de veículos) – 20%; Gonçalo Alves (implementação de algoritmos, visualização do grafo, relatório) – 40%; Pedro Seixas (implementação e pesquisa de algoritmos) – 40%

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
- Pseudocódigo de Dijkstra Bidirecional, http://planning.cs.uiuc.edu/node50.html
- Algoritmo A*, 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
- Programa de visualização de Algoritmos, *https://qiao.github.io/PathFinding.js/visual/*
- OpenStreetMap, https://www.openstreetmap.org/
- Ferramenta de obtenção de tags, https://overpass-turbo.eu/
- Algoritmo ALT, https://stackoverflow.com/questions/47485510/how-to-calculate-heuristic-value-in-a-algorithm ; https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/shortestpath/ALTAdmissibleHeuristic.java