Meat Wagons - Transporte de Prisioneiros

Turma 2 Grupo 3

up201806250@fe.up.pt	Diogo Samuel Gonçalves Fernandes
up201806490@fe.up.pt	Hugo Miguel Monteiro Guimarães
up201806554@fe.up.pt	Telmo Alexandre Espirito Santo Baptista

25 de Maio de 2020

Projeto CAL - 2019/20 - MIEIC

Professor das Aulas Práticas: Rosaldo José Fernandes Rossetti

Índice

1 Descrição do Problema

Os transportes de prisioneiros entre diversos estabelecimentos como, por exemplo, as prisões, esquadras e tribunais são feitos utilizando veículos que se encontrem adaptados ao serviço. Estes veículos têm a necessidade de serem altamente resistentes, uma vez que é necessário garantir que os prisioneiros não conseguem escapar.

Para este projeto, queremos otimizar o percurso dos veículos de forma a recolher e entregar os prisioneiros nos pontos de interesse. De modo a cumprir o pretendido, é possível dividir o nosso projeto nas seguintes fases:

Primeira Iteração - Recolha de prisioneiros utilizando um único veículo

Inicialmente, consideramos que só existe um único veículo para realizar todos os serviços. Com a primeira iteração pretende-se que apenas um veículo recolha os prisioneiros numa dada localização. É necessário ter em consideração obras nas vias públicas, uma vez que estas podem tornar certas zonas inacessíveis, inviabilizando o transporte de prisioneiros.

É importante notar que a recolha só pode ser efetuada caso existam caminhos que liguem todos os pontos de interesse, ou seja, o grafo necessita de ser conexo.

Segunda Iteração - Recolha de prisioneiros utilizando vários veículos

Durante a segunda iteração ter-se-à em consideração o diverso número de veículos que a frota possui. Os veículos vão diferir uns dos outros conforme um determinado tipo. Nesta fase do projeto irão existir veículos específicos para transportar tipos específicos de prisioneiros.

Terceira Iteração - Recolha seletiva de prisioneiros utilizando um único veículo

Na terceira iteração será considerada a possibilidade de um veículo atender a diferentes pedidos de transporte de prisioneiros, com diversos pontos de interesse diferentes, desde que não afete consideravelmente o tempo de espera do pedido anterior e não ultrapasse a capacidade do veículo.

Quarta Iteração - Recolha seletiva de prisioneiros utilizando vários veículos

A quarta iteração assemelha-se à terceira iteração, mas considerando um número variável de veículos disponíveis, tentando otimizar também o número de veículos utilizados.

2 Formalização do Problema

2.1 Dados de Entrada

C_i - sequência de veículos, sendo $C_i(i)$ o seu i-ésimo elemento. Cada veículo é caraterizado por:

- *capacity* - número de prisioneiros que pode transportar
- *type* - tipo de veículo

R_i - sequência de pedidos de transporte de prisioneiros, sendo $R_i(i)$ o seu i-ésimo elemento. Cada pedido é caraterizado por:

- *pickup* - local de recolha dos prisioneiros
- *dest* - local de destino dos prisioneiros
- $numPris$ - número de prisioneiros a serem transportados
- *type* - tipo de prisioneiros
- p_d - peso da distância no trajeto a efetuar
- p_t - peso do tempo no trajeto a efetuar

$G_i = (V_i, E_i)$ - grafo dirigido pesado, composto por:

- V - vértices, representando pontos da rede viária, com:
  - ID - Identificador único do vértice
  - D - Densidade populacional no vértice
  - $Adj \subseteq E$ - arestas que saiem do vértice
  - $avg - speed$ - velocidade média na área em volta do vértice
  - $reachable$ - se o vértice é alcançável a partir da central
- E - arestas, representando conexão entre dois pontos da rede viária, com:
  - ID - Identificador único da aresta
  - W_d - peso da aresta em relação à distância (representa a distância entre os dois vértices)
  - W_t - peso da aresta em relação ao tempo (representa o tempo médio que demora a percorrer a distância entre os dois vértices, considerando o tráfego normal naquela conexão da rede viária)

- *open* - se a conexão entre os vértices está aberta, isto é, se a rua estiver cortada por alguma razão então não é possível utilizar esta conexão

S - vértice da central

2.2 Dados de Saída

$G_f = (V_f, E_f)$ - grafo dirigido pesado, tendo V_f e E_f os mesmos atributos que V_i e E_i, excluindo atributos específicos do algoritmo utilizado

C_f - sequência de veículos com os serviços a realizar, sendo $C_f(i)$ o seu i-ésimo elemento. Cada veículo é caraterizado por:

- S - sequência de serviços a realizar, sendo $S(i)$ o seu i-ésimo elemento. Cada serviço é caraterizado por:
  - *emptySeats* - número de lugares vazios
  - R_f - sequência de pedidos atendidos, sendo $R_f(i)$ o seu i-ésimo elemento. Cada pedido atendido é caraterizado por:
    - *pickupHour* - hora de chegada ao local de recolha
    - *destHour* - hora de chegada ao local de destino
    - p_d - peso da distância no trajeto a efetuar
    - p_t - peso do tempo no trajeto a efetuar
  - $P = e \in E_i$ - sequência de arestas a percorrer, sendo $P(i)$ o seu i-ésimo elemento
  - *dist* - distância percorrida no serviço
  - *startHour* - hora esperada de ínicio do serviço
  - *endHour* - hora esperada de termino do serviço

2.3 Restrições

Sobre os dados de entrada

- $\forall i \in [0, |C_i|[: capacity(C_i(i)) > 0$, uma vez que não faz sentido os veículos não poderem transportar prisioneiros
- $\forall r \in R_i, dest(r)$ deve pertencer ao mesmo componente fortemente conexo do grafo G_i que o vértice S, uma vez que o veículo tem de ser capaz de voltar à central
- $\forall r \in R_i, numPris(r) > 0$, uma vez que não faz sentido ter um pedido para transportar zero prisioneiros
- $\forall r \in R_i, p_d \geq 0 \wedge p_t \geq 0 \wedge (p_d \neq 0 \vee p_t \neq 0)$
- $\forall v \in V_i, avg\text{-}speed(v) > 0$
- $\forall e \in E_i, W_d(e) > 0 \wedge W_t(e) > 0$, uma vez que o peso da aresta representa a distância ou o tempo médio necessário para percorrer a aresta, se esta distância ou tempo forem zero estaremos num ciclo no mesmo vértice
- $\forall e \in E_i, e$ deve ser uma rua ao qual os veículos possam utilizar, ruas que os veículos não tenham permissão para entrar não são incluídas no grafo G_i
- $S \in V_i$, uma vez que a central é um vértice do grafo G_i

Sobre os dados de saída

- $|C_f| \leq |C_i|$ - não se pode usar mais veículos que os disponíveis
- $\forall v_f \in V_f, \exists v_i \in V_i$ tal que v_i e v_f têm os mesmos valores para todos os atributos, com exceção de atributos específicos aos algoritmos utilizados
- $\forall e_f \in E_f, \exists e_i \in E_i$ tal que e_i e e_f têm os mesmo valores para todos os atributos, com exceção de atributos específicos aos algoritmos utilizados
- $\forall r_f \in R_f, \exists r_i \in R_i$ tal que r_f e r_i têm os mesmo valores para os atributos p_d e p_t
- $\forall c \in C_f, \forall s \in S(c), 0 \leq emptySeats < capacity(c)$ pois cada serviço deve ter pelo menos um prisioneiro, e não pode haver sobrelotação do veículo
- $\forall c \in C_f, \forall s \in S(c), |R_f(s)| > 0$ uma vez que só faz sentido realizar um serviço se existir mais de um pedido de transporte de prisioneiros
- $\forall c \in C_f, \forall s \in S(c), endHour(s) > startHour(s)$
- $\forall c \in C_f, \forall s \in S(c), startHour(s) < pickupHour(\forall r \in R_f) < endHour(s) \wedge startHour(s) < destHour(\forall r \in R_f) \leq endHour(s)$
- $\forall c \in C_f, \forall s \in S(c), \forall r \in R_f(s), index(dest(r)) > index(pickup(r))$

2.4 Função objetivo

A solução ótima passa por minimizar a soma ponderada da distância percorrida e o tempo do serviço de um determinado veículo, que resulta na seguinte função:

$\sum_{c \in C_f} \sum_{s \in S} \sum_{e \in P} (W_d(e) * max(p_d(R_f(s))) + W_t(e) * max(p_t(R_f(s))$

- $max(p_d(R_f(s))$ - é o maior valor para o peso da distância numa determinada sequência de pedidos de um serviço de um veículo
- $max(p_t(R_f(s))$ - é o maior valor para o peso do tempo numa determinada sequência de pedidos de um serviço de um veículo

Deste modo, obtivemos a função objetivo para o nosso problema que se encontra acima.

3 Perspectiva de solução

3.1 Pré-processamento dos dados de entrada

Grafo

Partindo da central, todos os vértices que não forem alcançáveis têm a variável *reachable* definida como falsa.

Além disso, todas os vértices do grafo que não pertençam à componente fortemente conexa de origem devem ser marcados como inacessíveis (*reachable* é colocado a falso).

Pedidos de transporte de prisioneiros

Remover todos os pedidos de transporte de prisioneiros que não pertençam ao grafo pré-processado, isto é, remover aqueles que façam parte de vértices que têm a componente *reachable* definida como falsa.

Também devemos organizar os pedidos de transporte de prisioneiros por ordem decrescente do número de prisioneiros a transportar, facilitando, em seguida, o alocamento de veículos para o seu transporte.

Veículos para transporte de prisioneiros

Relativamente ao pré-processamento dos veículos de transporte, devemos organizá-los por ordem decrescente de capacidade. Deste modo, como também temos os pedidos de transporte de prisioneiros organizados por ordem decrescente do número de prisioneiros a transportar podemos potencialmente minizar o número de veículos utilizados.

3.2 Identificação do problema

A empresa de transporte de prisioneiros Meat Wagons necessita de transportar os prisioneiros de um ponto de recolha até um determinado destino. De modo a otimizar este transporte, a empresa optou por procurar o caminho mais eficiente para a efetuar a viagem.

Na primeira iteração, onde apenas está disponível um veículo, que realiza os pedidos de transporte um de cada vez, este problema trata-se do **caminho mais curto** entre a origem e o local de recolha seguido do **caminho mais curto** entre o local de recolha e o destino. A segunda iteração é semelhante à primeira iteração, variando apenas o número de veículos disponíveis para realizar os pedidos.

Na terceira iteração, um veículo poderá realizar vários pedidos simultâneamente, equiparando-se assim ao **Travelling Salesman Problem**, com restrições no vértice de origem, e na ordem de visita dos vértices.

Na quarta e última iteração, não só varia o número de veículos disponíveis, como também o número de pedidos de transporte que um veículo pode realizar num único serviço, equiparando-se ao problema designado por **Vehicle Routing Problem**, uma generalização do **Travelling Salesman Problem**, um problema *NP-díficil*.

Vale também realçar que os veículos devem retornar para a central no fim do serviço.

3.3 Caminho mais curto

Este é o problema referido na primeira e segunda iteração, e trata-se de encontrar o percurso mais curto e eficiente entre dois pontos, ou entre todos os pares de pontos do grafo.

Entre dois pontos

Entre os vários algoritmos que existem para calcular o caminho mais curto entre dois pontos destacam-se os seguintes algoritmos:

Algoritmo de Dijkstra

Este algoritmo foi concebido por Edsger W. Dijkstra e resolve problemas do caminho mais curto de uma única origem em grafos que possuam pesos não negativos.

Para poder aplicar este algoritmo é necessário que cada vértice guarde a seguinte informação:

- W - custo mínimo até ao local da origem (combinação linear da distância e tempo, como visto na função objetivo)
- $path$ - vértice antecessor no caminho mais curto

O algoritmo de Dijkstra pode utilizar uma *priority queue* ou um *array* para inserir os novos vértices. Este consiste em inicializar os vértices, o que se pode fazer em tempo linear $O(|V|)$. Seguidamente, inicializar a estrutura auxiliar, que neste caso consideramos a $priorityqueue$ devido a ter maior eficiência relativamente ao *array*, com o vértice origem.

Processam os vértices que se encontram na *queue* extraindo-os e seguidamente percorrendo cada aresta do vértice a ser processado. Posteriormente, se o custo relativo ao vértice de destino da aresta for maior do que o custo do caminho atual, terá que se atualizar o vértice de destino e inserindo na *priority queue* caso ele ainda não esteja na fila de processamento ou fazendo a operação $DECREASE - KEY$ caso este já esteja na fila de processamento.

As operações de inserção, extração e $DECREASE - KEY$ têm complexidade temporal $O(log(N))$. Dado que é necessário percorrer todos os vértices e arestas resulta numa complexidade de $O((|V| + |E|) * log(|V|))$.

Assim podemos concluir que o tempo de execução do algoritmo é $O((|V| + |E|) * log(|V|))$.

O pseudo-código para implementar este algoritmo é o seguinte:

```
FOR EACH v ∈ V DO
   COST(v) ← ∞
   PATH(v) ← NULL

COST(s) ← 0
Q ← ∅ // MIN PRIORITY QUEUE
INSERT(Q, (s, COST(s)))
WHILE Q ≠ ∅ DO
   v ← EXTRACT-MIN(Q)
   FOR EACH w ∈ Adj(v) DO
      IF COST(w) > COST(v) + WEIGHT(v, w) THEN
         COST(w) ← COST(v) + WEIGHT(v, w)
         PATH(w) ← v
         IF w ∉ Q THEN
            INSERT(Q, (w, COST(w)))
         ELSE
            DECREASE-KEY(Q, (w, COST(w)))
```

Este algoritmo destaca-se pela sua facilidade de implementação, porém o algoritmo pode explorar demasiados vértices desnecessários.

A ineficiência do algoritmo pode ser visto na imagem abaixo:

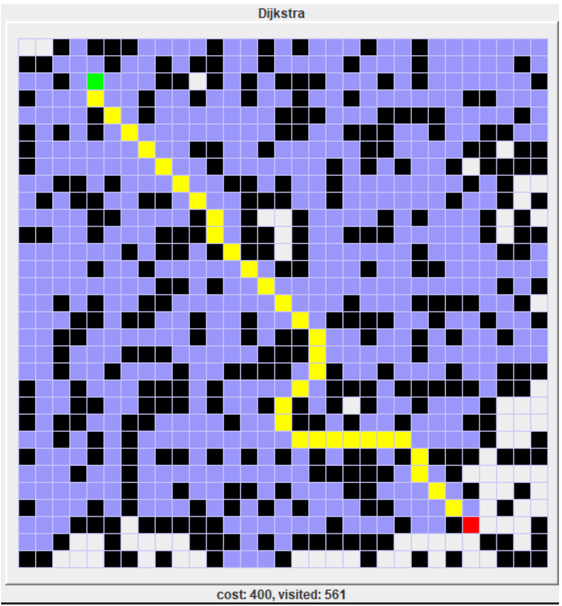


Figura 1: *Dijkstra's algorithm*

■ walls ■ visited
■ origin ■ destination
■ shortest path

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford corresponde a uma extensão do algoritmo de Dijkstra permitindo a existência de pesos negativos nas arestas, sendo mais lento que o de Dijkstra por esse mesmo motivo.

Uma vez que foi imposta a restrição de pesos não negativos nas arestas, este algoritmo não se vê útil, uma vez que não se vê necessário tratar pesos negativos.

Algoritmo A*

O algoritmo A*, desenvolvido por Peter Hart, Nils Nilsson e Bertram Raphael, pode ser visto como uma extensão do algoritmo de Dijkstra, usando heurística para guiar a sua pesquisa.

Em cada iteração, o algoritmo precisa decidir qual caminho processar, baseando-se no custo do caminho desde a origem até ao ponto atual e numa estimativa do custo do caminho desde o vértice adjacente a testar até ao destino, isto é o algoritmo visa minimizar a seguinte função

$$f(n) = g(n) + h(n) \tag{1}$$

onde n é o próximo vértice do caminho, $g(n)$ o custo desde a origem até n e $h(n)$ uma estimativa do custo mínimo desde n até ao destino.

Uma possível implementação do algoritmo está demonstrada no seguinte pseudo-código:

```
RECONSTRUCT_PATH(current)
  path ← {current}
  WHILE PATH(current) ≠ NULL
    current ← PATH(current)
    PREPEND(path, current)
  RETURN path

A_STAR(start, goal, heuristic)
  FOR EACH v ∈ V DO
    G_COST(v) ← ∞
    F_COST(v) ← ∞
    PATH(v) ← NULL

  G_COST(start) ← 0
  F_COST(start) ← heuristic(start) // G_COST(start)+heuristic(start)
  Q ← ∅ // MIN PRIORITY QUEUE
  INSERT(Q, (start, F_COST(start)))
  WHILE Q ≠ ∅ DO
    v ← EXTRACT-MIN(Q)
    IF V = GOAL
      RETURN RECONSTRUCT_PATH(v)

    FOR EACH w ∈ Adj(v) DO
```

```
IF G_COST(w) > G_COST(V) + WEIGHT(v, w) THEN
   G_COST(w) ← G_COST(v) + WEIGHT(v, w)
   PATH(w) ← v
   F_COST(w) ← G_COST(w) + heuristic(w)
   IF w ∉ Q THEN
      INSERT(Q, (w, F_COST(w)))
   ELSE
      DECREASE-KEY(Q, (w, F_COST(w)))
```

O algoritmo A* é um algoritmo de elevada eficiência e otimização, sendo usado em muitos contextos, como nos sistemas de encaminhamento de viagens que corresponde às duas primeiras iterações do nosso problema.

A eficiência deste algoritmo pode ser observada comparando o número de vértices explorados durante a pesquisa com o algoritmo de Dijkstra, como é demonstrado na imagem abaixo:

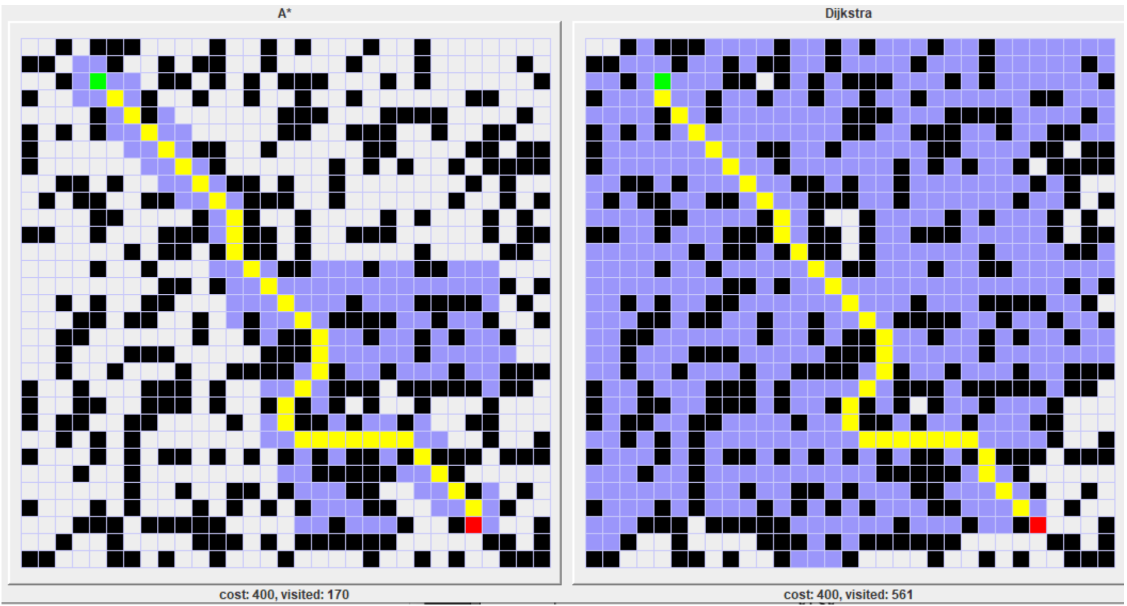


Figura 2: *A* algorithm **vs.** Dijkstra's algorithm*

- walls
- visited
- origin
- destination
- shortest path

Embora a eficiência do algoritmo seja maior, o algoritmo A* não garante a solução ótima para todos os casos, ao contrário de algoritmos como o de Dijkstra. Esta desvantagem pode ser observada na imagem abaixo:

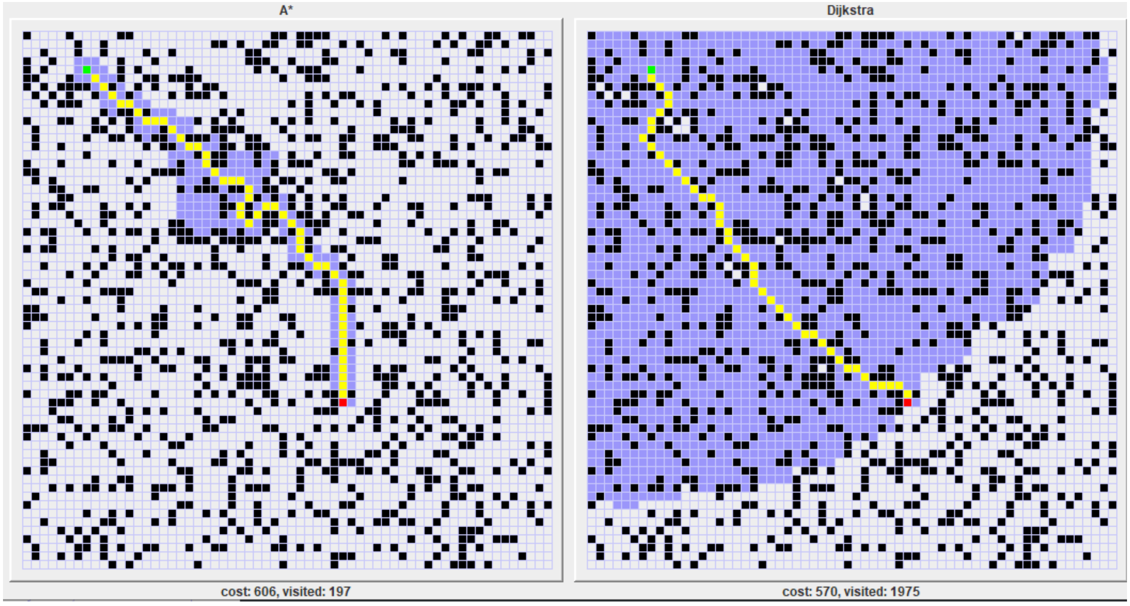


Figura 3: *A* algorithm* ***vs.*** *Dijkstra's algorithm*

■ walls ■ visited
■ origin ■ destination
■ shortest path

Analisando os resultados obtidos, é possível constatar que o algoritmo de Dijkstra visitou aproximadamente dez vezes mais vértices que o algoritmo A* (1975 **vs.** 197). Porém, o caminho mais curto encontrado pelo algoritmo A* não corresponde ao caminho com menor custo, uma vez que o caminho encontrado pelo algoritmo de Dijkstra possui um custo menor que o algoritmo de A* (570 **vs.** 606).

Entre todos os pares de vértices

É possível calcular o caminho entre todos os pares de vértices através de algoritmos, como a aplicação repetida do algoritmo de Dijkstra ou a utilização do algoritmo de Floyd-Warshall.

Estes algoritmos são bastante utilizados para pré-processamento de mapas de estradas, porém no nosso problema, como os pesos para a distância e o para o tempo variam de pedido para pedido, o pré-processamento dos caminhos mais curtos para todos os pares de vértices não traria nenhuma vantagem, apenas uma diminuição na eficiência do programa.

3.4 Caminho mais curto com vários pedidos

Dada a possibilidade de um veículo realizar vários pedidos num único serviço, existirá um conjunto de locais de recolha e locais de destino a serem percorridos.

Deparámo-nos então com um problema similar ao **Travelling Salesman Problem**, um problema NP-difícil. Como se trata de um grafo dirigido é a versão assimétrica do problema **Travelling Salesman Problem**.

As soluções deste problema podem dividir-se em duas categorias:

- **Soluções Exatas** - algoritmos que encontram a solução exata do problema
- **Soluções Aproximadas** - algoritmos que aproximam a solução do problema através de heurísticas e aproximações

Soluções Exatas

Brute-force

O método brute-force testa todas as permutações possíveis para o percurso, atualizando o caminho ótimo sempre que encontra um custo menor ao atual, resultando assim numa complexidade $O(n!)$, sendo n o número de vértices a percorrer.

Held-Karp

O algoritmo de Held-Karp é um algoritmo de programação dinâmica que tem como objetivo resolver o **Travelling Salesman Problem**, utilizando formulas recursivas para dividir o problema.

O algoritmo apresenta uma complexidade temporal elevada, $O(2^n n^2)$, requirindo, assim, muito poder computacional para obter a solução ótima.

Embora este algoritmo obtenha a solução ótima para o problema, a sua implementação, dada as restrições impostas, pode ser muito complexa, pelo que serão priorizados os algoritmos de soluções aproximadas do problema, dado à sua simplicidade e flexibilidade.

Analisando as complexidades dos algoritmos apresentados podemos verificar que o método de brute-force é mais eficiente para valores de n menores que sete, sendo o algoritmo de Held-Karp mais eficiente para os restantes valores de n, sendo n o número de vértices a percorrer, assim como se pode observar no gráfico seguinte:

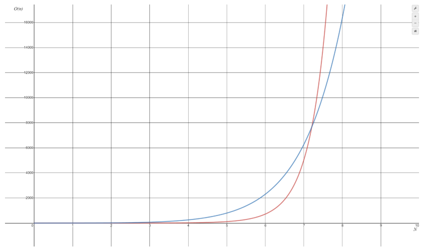


Figura 4: *Brute-force* ***vs.*** *Held-Karp algorithm: Complexities*

— Brute-force ($O(n!)$) — Held-Karp ($O(n^2 2^n)$)

Na implementação do cálculo da solução exata alternaríamos o método utilizado conforme o número de vértices a percorrer, usando brute-force para $n \leq 7$ e o algoritmo Held-Karp para $n > 7$.

Soluções Aproximadas

Nearest Neighbour

O algoritmo de **nearest neighbour** consiste em escolher um vértice aleatório para o início, e de seguida escolher o vértice mais próximo como próximo vértice a percorrer repetindo este passo até visitar todos os vértices a serem percorridos. Trata-se assim de algoritmo ganancioso que encontra uma solução aproximada em tempo reduzido, no entanto esta solução não é garantidamente a solução ótima.

O pseudo-código deste algoritmo é o seguinte:

```
FOR EACH v ∈ V DO
   VISITED(v) ← false
   PATH(v) ← NULL

v ← RANDOM_VERTEX(V) // choose starting point
VISITED(v) ← true

WHILE NOT ALL_VISITED(V) DO
   w ← CLOSEST_VERTEX(V, v) // get closest vertex to v
   VISITED(w) ← v
   PATH(w) ← v
   v ← w
```

No nosso problema, o ponto inicial é fixo, sendo este a central, retirando assim a aleatoriedade do ponto inicial do algoritmo.

Algoritmo Genético

Algoritmos genéticos são algoritmos baseados em heurísticas que simulam o processo de evolução de espécies, a *seleção natural*, selecionando os melhores espécimes de cada geração.

Os algoritmos genéticos podem ser divididos em cinco fases:

1. Gerar a população
2. Calcular a aptidão de cada indivíduo da população
3. Escolher os indivíduos mais aptos
4. Reproduzir os indivíduos escolhidos (por replicação ou *crossover*)
5. Mutação dos indivíduos de modo a introduzir pequenas variações na população

```
// calculate  fitness
CALCULATE_FITNESS(I)
  fitness ← 0
  FOR i ← 1 TO |VERTICES(I)|
    // add cost of going from vertex i−1 to vertex i
    fitness ← fitness + COST(VERTICES[i−1], VERTICES[i])
  FITNESS(I) ← fitness

// Choose the n best individuals
CULL_POPULATION(P, n)
  sorted ← SORT_BY_FITNESS(P) // sort by descending order of fitness
  best ← ∅
  FOR i ← 0 TO n
    INSERT(best, sorted(i))
  RETURN best

// replicate individual
REPLICATE(I)
  return EXACT_COPY(I)

// create new individual from two parents
CROSSOVER(parent_A, parent_B)
  child ← NEW_INDIVIDUAL()
  // being N the number of vertices to visit
  // random integer in [0, N[
  section_start ← RANDOM_INT(0, N)
  // random integer in ]section_start, N[
  section_end ← RANDOM_INT(section_start + 1, N)
  // copy random section from parent A
  FOR i ← section_start TO section_end DO
    VERTICES(child) AT (i) ← VERTICES(parent_A) AT (i)
```

```
  // fill remaining empty sections with genes from parent B
  FOR i ← 0 TO N DO
    IF VERTICES(child) AT (i) = NULL
      VERTICES(child) AT (i) ← VERTICES(parent_B) AT (i)
  RETURN CHILD

// mutate individual
MUTATE(I)
  v ← RANDOM_VERTEX(VERTICES(I)) // choose random vertex
  w ← RANDOM_VERTEX(VERTICES(I)) // choose another random vertex
  SWAP(v, w)

// using crossover to reproduce (can be done with replication)
// reproduces population P
REPRODUCE_POPULATION(P)
  NEW_P ← ∅
  FOR i ← 0 TO POPULATION_SIZE DO
    // choose parents (can be tested to be different parents)
    parent_A ← RANDOM_INDIVIDUAL(P)
    parent_B ← RANDOM_INDIVIDUAL(P)
    I ← CROSSOVER(parent_A, parent_B)
    random ← RANDOM_FLOAT(0, 1) // random number between 0 and 1
    IF random < MUTATION_RATE THEN
      MUTATE(I)
    INSERT(NEW_P, I)
  RETURN NEW_P


// generate random population (random order of vertices to visit)
P ← GENERATE_RANDOM_POPULATION(V)

WHILE ... // decide stopping criteria
  FOR EACH individual ∈ P
    CALCULATE_FITNESS(individual)

  best ← CULL_POPULATION(P, n)
  P ← REPRODUCE_POPULATION(best) // reproduce best individuals
```

De modo a garantir a restrição imposta sobre a ordem de visita dos locais de interesse, isto é, deve ser visitado o local de recolha antes do local de destino, podemos atribuir a aptidão mínima a todos os indíviduos que não respeitem tal restrição.

4 Funcionalidades a implementar

4.1 Pré-processamento dos dados de entrada

Grafo

De modo a marcar todas as arestas alcançáveis a partir do vértice da central pode ser utilizada uma estratégia semelhante à procura em profundidade ($Depth-FirstSearch$), começando a visita na central, e marcar todos os vértices que forem visitados como *open*.

O pseudo-código para esta estratégia é o seguinte:

```
VISIT(node)
  reachable(node) ← true
  FOR w ∈ Adj(node) DO
    IF NOT reachable(w) THEN
      VISIT(w)

// G - graph
// source - starting point
VISIT_FROM_SOURCE(G, source)
  FOR v ∈ VERTICES(G) DO
    reachable(v) ← false

  VISIT(source)
```

Para a identificação dos vértices do grafo que pertençam à componente fortemente conexa do vértice da central, será necessário analisar a conetividade do grafo e a construção do componente fortemente conexo, para o qual se destacam os algoritmos de Kosaraju e de Tarjan.

Algoritmo de Kosaraju

O algoritmo de Kosaraju consiste nos seguintes passos:

1. Realizar uma pesquisa em profundidade no grafo colocando os vértices numa stack após visitar o vértice, obtendo assim os vértices em pós-ordem
2. Transpor o grafo (inverter o sentido de todas as arestas)
3. Fazer uma pesquisa em profundidade nos vértices pela ordem que estão definidos na stack. Depois de ser feita a pesquisa obtém-se a componente fortemente conexa a que esse vértice pertence

No entanto, como no nosso caso só interessa saber a componente fortemente conexa relativa à central, podemos apenas percorrer o grafo transposto a partir desse mesmo vértice.

O pseudo-código para o algoritmo é, então, o seguinte:

```
// G - graph
// C - container to store the vertices of the SCC
DFS_VISIT(G, node, C)
  visited(node) ← true
  FOR w ∈ Adj(v) DO
    IF not visited(w) THEN
      DFS_VISIT(w)
  INSERT(C, node)

  GT ← TRANSPOSE(G)

  SCC ← ∅

  DFS_VISIT(GT, source, SCC)
```

A complexidade temporal de uma pesquisa em profundidade, assim como, a complexidade de inverter todas as arestas é proporcional ao tamanho do grafo isto é $O(|V| + |E|)$ sendo $|V|$ o número de vértices e $|E|$ o número de arestas.

Como o algoritmo de Kosaraju se baseia em duas pesquisas em profundidade e numa inversão do grafo realizadas sequencialmente a complexidade do algoritmo também é $O(|V|+|E|)$ (uma vez que a inserção, deleção e a obtenção do topo da stack são realizadas em tempo constante, $O(1)$, não afetando a complexidade temporal do algoritmo).

Algoritmo de Tarjan

O Tarjan é uma versão mais eficiente do algoritmo de Kosaraju, precisando apenas de realizar uma única pesquisa em profundidade para obter o grafo fortemente conexo.

Similarmente ao algoritmo de Kosaraju, o algoritmo de Tarjan também executa em tempo linear, $O(|V|+|E|)$, porém, este último baseia-se numa única pesquisa em profundidade, sendo assim mais eficiente.

Como no nosso caso apenas interessa saber a componente fortemente conexa a partir da central, este algoritmo não irá trazer muitas vantagens relativamente ao algoritmo visto anteriormente. Deste modo, a sua implementação não será uma prioridade, podendo ser considerada numa fase futura.

4.2 Casos de Implementação

Perante a organização dos caminhos a percorrer pelos veículos, é necessário ter em consideração os seguintes aspetos:

- Escolha do melhor percurso para um veículo
- Escolha dos pedidos de transporte de prisioneiros para cada veículo
- Agrupar os pedidos de prisioneiros
- Agrupar os veículos

Deve ter-se como objetivo a atribuição de uma carrinha a um serviço, tendo atenção aos prisioneiros que se precisam de transportar.

Devem ser, portanto, seguidos os seguintes passos quando é recebido um novo pedido de transporte de prisioneiros:

- Ordenação dos serviços de modo a que os pedidos de transporte de prisioneiros mais antigos sejam analisados primeiro
- Escolha do veículo disponível que possa efetuar os pedidos que foram recebidos num determinado período de tempo
- Verificar se veículos que estão a executar algum pedido estão aptos à existência de novos pedidos de transporte de prisioneiros. Se um veículo puder efetuar esse pedido sem alterar o seu percurso, então o pedido deve ser sempre aceite

4.3 Casos de Utilização

A aplicação a implementar irá incluir um menu de interação com o utilizador (GUI) capaz de navegar entre vários submenus, possibilitando o acesso às seguintes funcionalidades:

- Visualização do grafo que contém o mapa disponibilizado (utilizando GraphViewer)
- Cálculo otimizado do percurso entre dois pontos
- Adição e visualização do número de veículos
- Adição de novos pedidos
- Atribuição dos serviços tendo em conta os pedidos e as carrinhas disponíveis

Para adicionar novos pedidos, o utilizador terá que indicar o **local de recolha** e **destino dos prisioneiros**, o **número de prisioneiros**, **tipo de prisioneiros**, **peso da distância no trajeto** e o **peso do tempo no trajeto**.

Deste modo, a aplicação terá capacidade de gerir de uma forma otimizada e segura o transporte de passageiros entre diversos estabelecimentos, para além de armazenar a informação relativa aos percursos, permitindo a sua visualização.

5 Estruturas de dados utilizadas

5.1 Grafo

O grafo que fora utilizado no desenvolvimento deste projeto é uma adaptação de um grafo fornecido previamente nesta mesma unidade curricular, porém adaptado ao nosso problema. Para guardar os vértices e as arestas foi utilizado a estrutura de dados *vector* que contém um pointers para cada elemento. Encontra-se na classe *Graph*.

Vértice

Possuem um ID que permite a sua identificação, a posição em que o mesmo é desenhado, as arestas que partem dele e campos auxiliares que são utilizados na execução de diversos algoritmos. Encontra-se na classe $Vertex$.

Aresta

Possuem um ID que permite a sua identificação, assim como um vértice de destino, o peso do tempo e o da distância, além de campos auxiliares para diversos algoritmos. Encontra-se na classe $Edge$.

5.2 Departamento

A classe *Departament* possui o grafo, assim como um vetor que permite acessar todos os veículos disponíveis e todos os pedidos a que este mesmo departamento tem de responder.

Pedidos

Os pedidos encontram-se na classe *Request* e são caracterizados pelo número de prisioneiros e o tipo de transporte que vai ser necessário efetuar. Além disso, o pedido de transporte também possui o local de recolha e o ponto de destino, assim como a importância (peso) da distância e peso do tempo no pedido.

Veículos

Os veículos encontram-se na classe $Waggon$ e possuem uma capacidade máxima, que corresponde ao número de prisioneiros que a mesma consegue transportar. Possuem também uma lista com os serviços que a mesma irá efetuar.

Serviço

Os serviços possuem a distância percorrida e os pedidos que irão ser atendidos assim como uma hora de partida e de chegada, encontrando-se implementado na classe $Service$.

5.3 Leitura e desenho de mapas

Os mapas são lidos através de ficheiros de texto que possuem informação sobre os multiplos vértices e as arestas. Existem também ficheiros que possuem informação sobre a localização da central e sobre os pedidos de transporte de prisioneiros. Esta informação é lida em métodos da classe *GraphReader*. Posteriormente, o grafo é desenhado na classe *GraphDrawer*, através do *GraphViewer*.

É também fornecida uma funcionalidade para introduzir atrasos no desenho do caminho durante os serviços, de modo a facilitar a visualização do percurso efetuado pelo veículo, para tal deve colocar o último argumento da linha de comandos (*delayed*) com o valor *true*.

6 Algoritmos implementados e complexidade

6.1 Pré-processamento

Antes de fazer pré-processamento no nosso grafo, verificamos que existe pelo menos um departamento no mapa a analisar. Caso exista, aplicamos o algoritmo de Kosaraju a partir do departamento. Logo após a aplicação do algoritmo, obtemos a parte fortemente conexa a que o nosso departamento pertence, do qual irão sair todos os veículos de transporte de prisioneiros irão partir.

O pseudo-código encontra-se já descrito no capitulo de funcionalidades a implementar.

Análise teórica

Análise temporal empírica

6.2 Djisktra

Para a implementação do algoritmo de Djisktra, utilizamos uma $MutablePriorityQueue$ que foi fornecida previamente para a resolução de exercícios nas aulas teórico-práticas. A utilização desta estrutura de dados demonstra ser vantajosa para a implementação do algoritmo uma vez que permite alterar a *key* de um elemento sem precisar de o remover e voltar a inserir na $PriorityQueue$.

O pseudo-código encontra-se já descrito no capitulo que aborda a perspectiva de solução, assim como foi efetuada a sua análise temporal.

Análise teórica

Segundo a nossa implementação de Dijkstra, o algoritmo começa por pré-processar todos os vértices do grafo, tendo então uma complexidade linear ($O(|V|)$), sendo V os vértices do grafo.

De seguida é efetuada uma pesquisa do vértice inicial, que no pior caso, também é linear, tendo de percorrer todos os vértices para encontrar, não aumentando assim a complexidade total do algoritmo (mantém-se linear).

Após a pesquisa é criado a $MutablePriorityQueue$ e é inserido o vértice inicial, tratando-se de uma inserção numa *priority queue* a complexidade temporal desta operação é logarítmica, porém como a *queue* está inicialmente vazia, a operação é constante $O(1)$.

Por fim é feita um varrimento pela *queue* de vértices a visitar, sendo o pior caso possível a visita de todos os vértices do grafo.

Dentro do varrimento são executadas várias operações de complexidade constante, não sendo assim relevantes para a análise, porém ocorre um varrimento das arestas do vértice, mas vale notar que uma aresta não pode pertencer a mais de um vértice, tornando-se assim em conjuntos únicos. Isto implica que no final dos varrimentos cada vértice e cada aresta são apenas percorridos uma única vez, resultando assim num complexidade temporal de $O(|V|+|E|)$, sendo E as arestas do grafo.

Ainda dentro do varrimento são executadas as operações de $INSERT$ e $DECREASE_KEY$ na $MutablePriorityQueue$, em que ambas as operações possuem complexidade logarítmica, e são executadas para inserir ou alterar os vértices guardados na queue, sendo assim possuem complexidade de $O(log(|V|))$

Concluindo então, obtemos uma complexidade total de:

$$O(|V| + |V| + |E| + |V| * log(|V|)) \sim O(|E| + |V|log(|V|)) \tag{2}$$

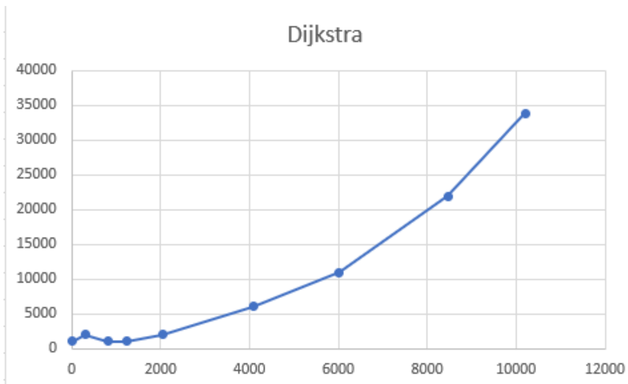
Quanto à complexidade espacial, o algoritmo utiliza o grafo em si, tendo então uma complexidade de $O(|V| + |E|)$ e a $MutablePriorityQueue$ para guardar os vértices a visitar, dado que todas as outras variáveis possuem todas complexidade constante (inteiros, pointers, entre outros).

Assim a complexidade espacial total do algoritmo é:

$$O(|V| + |E| + |V|) \sim O(|V| + |E|) \tag{3}$$

Análise temporal empírica

Uma vez que não é possível aumentar o tamanho do input do algoritmo de Dijkstra, já que apenas recebe dois pontos, a origem e o destino, foi feita uma análise variando a distância entre os dois pontos, obtendo-se assim os seguintes resultados num *grid graph* 100x100:



O gráfico resultante assemelha-se ao esperado de uma função complexidade semelhante a $n * log(n)$.

6.3 A-Star

O algoritmo $A-Star$ é semelhante ao de Djisktra utilizando também uma $MutablePriorityQueue$. No entanto, além desta estrutura de dados, o algoritmo possui uma heurística que permite fazer uma estimativa do vértice que se poderá encontrar mais próximo do local de destino.

O pseudo-código encontra-se já descrito no capitulo que aborda a perspectiva de solução.

Análise teórica

Assim como mencionado na perspetiva de solução, a complexidade temporal do algoritmo $A*$ é no geral similar à do algoritmo de $Dijkstra$, possuindo no pior caso uma complexidade de $O(|E| + |V|log(|V|))$, sendo V os vértices do grafo e $|E|$ as arestas do grafo.

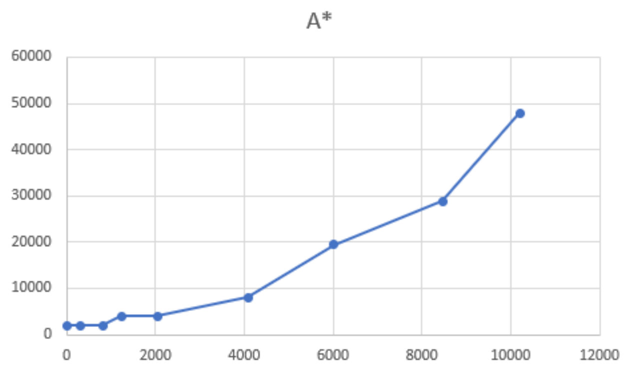
Na nossa implementação do $A*$ este principio mantém-se, podendo notar-se até que o código usado é basicamente igual. A única diferença do $A*$ é usar uma heuristíca de modo a tentar aproximar-se mais rapidamente do destino, convergindo mais rápido do que $Dijkstra$ na maioria dos casos, mas no pior caso terá a mesma complexidade que o de $Dijkstra$ como já mencionado.

Assim como no algoritmo anterior, o $A*$ utiliza o próprio grafo e uma $MutablePriorityQueue$ auxiliar, sendo assim possui uma complexidade espacial também de

$$O(|V| + |E|) \tag{4}$$

Análise temporal empírica

Assim como no algoritmo anterior, não tem como aumentar o número de inputs, portanto, aumentou-se a distância entre os dois pontos de input de modo a verificar o aumento do tempo conforme a distância entre estes, os resultados obtidos num *grid graph* 100x100 foram os seguintes:



Como era expectável, o gráfico assemelha-se ao do algoritmo de $Dijkstra$ apenas observando-se desvios em comparação com o gráfico de $Dijkstra$, que realçam o funcionamento da heurística do $A*$, levando a convergências mais rápidas em alguns casos, ou mais lentas noutros.

6.4 Nearest Neighbour

O algoritmo $NearestNeighbour$ faz uso de $MultiMaps$ e $Sets$. Uma vez que é possível que vários pedidos tenham o mesmo ponto de recolha, a utilização de multimaps permite a existência de $Keys$ iguais, mas com pontos de destino diferentes.

Análise teórica

Segundo a nossa implementação do algoritmo $NearestNeighbour$, o algoritmo começa por pré-processar todos os vértices do grafo, tendo então uma complexidade linear ($O(|V|)$), sendo V os vértices do grafo.

De seguida é efetuada uma pesquisa do vértice inicial, que no pior caso, também é linear, tendo de percorrer todos os vértices para encontrar, não aumentando assim a complexidade total do algoritmo (mantém-se linear).

Após a pesquisa, o algoritmo necessita inicializar o *set* de vértices a visitar com todos os pontos de recolha, iterando assim pelo *multimap* recebido no input. Para cada ponto de recolha no *multimap*, ocorre uma pesquisa desse vértice e uma inserção no *set*. A primeira operação é linear como já vista anteriormente, a segunda é logarítmica no geral, mas dado uma dica da posição a inserir, este inserção pode ser otimizada para complexidade constante amortizada. Assim, esta fase de inicialização reduz-se a:

$$O(|PD| * |V|) \quad amortizado \tag{5}$$

sendo $|PD|$ o número de pares de pontos a visitar.

Após a inicialização, é necessário varrer os pontos a visitar, que são no pior caso todos listado no mapa (pode não percorrer todos devido ao uso do *set*, podendo assim acumular o mesmo lugar a visitar não tendo que o repetir múltiplas vezes).

Em cada varrimento deve ser escolhido o ponto mais próximo do atual, operação feita linearmente no tamanho dos vértices a visitar, $O(|PD|)$, uma vez que se usa heurística para esta seleção. Deve-se também remover o elemento escolhido dos pontos a visitar, sendo assim, uma operação de remoção num *set* tem complexidade logarítmica no tamanho do set, isto é, $O(log(|PD|))$.

Para cada ponto escolhido, em caso de ser um ponto de recolha deve-se então fazer uma pesquisa no *multimap* para obter todos os pontos de destino a adicionar no *set* de pontos a visitar, removendo posteriormente todos esses pares de valores do *multimap*. Esta operação vai ser realizada o mesmo número de vezes de pontos de recolha dados, uma vez que esta operação é realizada para cada ponto de recolha. Dado que a pesquisa no *multimap* usando *equal_range* e a remoção são de complexidade logarítmica, a pesquisa do ponto a adicionar ao *set* é linear no número de vértices do grafo, e a inserção no *set*, desta vez não otimizada, é de complexidade também logarítmica. Assim, esta operação tem uma complexidade de $O(log(|PD|) + log(|PD|) + |V| + log(|PD|) \sim O(log(|PD|) + |V|)$.

Após esta operação, tendo já o próximo ponto escolhido, é necessário obter o caminho mais curto entre esses pontos (dado que não são adjacentes, no caso geral), para tal foi utilizado o algoritmo $A*$, que já analisado anteriormente possui uma complexidade de $O(|E|+|V|log(|V|)$.

Após a obtenção do caminho, é necessário apenas transferir os vértices da estrutura auxiliar para o output, e uma vez que o algoritmo $A*$ apenas visita um vértice no máximo uma vez, o pior caso possível trata-se de visitar todos os vértices, sendo assim, esta transferência adiciona uma complexidade linear ao algoritmo no número de vértices do grafo.

Concluindo assim, a versão do algoritmo $NearestNeighbour$ implementada tem uma complexidade total de:

$$O(|V| + |V| + |V| + |PD| * |V| + |PD| * (|PD|) \tag{6}$$

Semelhante aos algortimos já analisados, o $NearestNeighbour$ também utiliza o próprio grafo, adicionando assim uma complexidade espacial de $O(|V| + |E|)$.

Além dessa estrutura, o algoritmo também recebe pelo seu input um *multimap* em que o seu tamanho depende do número de pares de pontos a visitar, adicionando assim uma complexidade espacial de $O(|PD|)$ sendo PD o número de pares de pontos a visitar.

O algoritmo também recebe um *vector* onde irá guardar o seu resultado final, porém este não será contabilizado para a complexidade uma vez que isso dificultaria a distinção entre problemas complexos dado que produzem um output grande de problemas complexos onde calcular um único output já é bastante complicado.

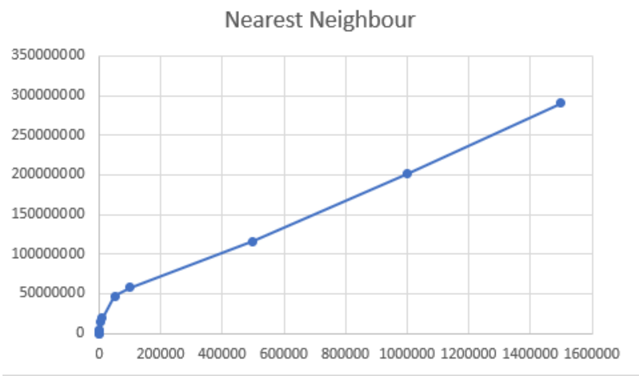
Como estrutura auxiliar, o algoritmo utiliza um *set* para guardar os vértices a visitar, esta estrutura otimiza o espaço e o tempo requirido pelo algoritmo dado que é possível visitar o mesmo ponto várias vezes durante o percurso todo. Sendo um *set*, a complexidade espacial introduzida é de $O(|V|)$.

É também utilizada outra estrutura auxiliar com função de ser a intermediária entre o algoritmo $A*$ e o path final. Uma vez que numa iteração do $A*$ o máximo de vértices visitados é o número de vértices do grafo, e esta estrutura é destruida após o seu uso, temos uma complexidade espacial adicional de $O(|V|)$.

Concluindo, o algoritmo $NearestNeighbour$ tem então uma complexidade espacial total de:

$$O(|V| + |E| + |PD| + |V| + |V|) \sim O(|V| + |E| + |PD|) \tag{7}$$

Análise temporal empírica



6.5 Processamento de pedidos

Para distribuir os pedidos pelos diferentes veículos fizemos um algoritmo que distribui os pedidos pelas várias carrinhas, tentando fazer com que estas realizem o menor número de viagens possível.

Pseudo-código

Análise teórica

Análise temporal empírica

7 Conectividade dos grafos utilizados

Conforme foi referido anteriormente, antes de começar a procura do caminho mais curto é necessário verificar que todos os pontos de interesse a serem percorridos encontram-se na parte fortemente conexa do grafo. Para garantir isto foi utilizado o algoritmo de $Kosaraju$ que permite obter a parte fortemente conexa do grafo. Depois deste algoritmo ser aplicado é possível distinguir através do $GraphViewer$ as componentes do grafo que se encontram conexas dos componentes que não se encontram conexos atráves da sua cor. Os vértices e arestas que se encontram representados com a cor vermelha, correspondem àqueles que se encontram na parte não conexa do grafo. Respetivamente, aqueles elementos que se encontram com cor verde correspondem à parte conexa do grafo.

7.1 Grafos pouco conexos

Os grafos pouco conexos foram utilizados para verificar que o algoritmo de $Kosaraju$ encontrava-se, de facto, bem implementado da nossa parte. Não realizamos nenhuma iteração do nosso problema nestes grafos uma vez que a parte que se encontrava conexa era bastante reduzida e por esse mesmo motivo não era profícuo testar com estes mapas. O mapa pouco conexo que fora utilizado foi o do *Porto*.

7.2 Grafos muito conexos

Os grafos conexos foram utilizados para testar as diversas iterações. Uma vez que estes permitem visitar quase todos os vértices, constituem o melhor tipo de mapas para encontrar o caminho mais curto entre vários pontos. Foram utilizados os $GridMaps$ $8x8$ e $16x16$ com ligeiras alterações, assim como o mapa de *Cabeceiras* que foi criado com o âmbito de possuir outro mapa conexo sem ser $GridMaps$.

8 Conclusão

O objetivo deste trabalho foi o desenvolvimento de uma estratégia responsável pela atribuição de veículos e rotas, através da criação de um modelo capaz de otimizar a resolução deste problema.

O modelo foi dividido em três iterações, sendo as primeiras duas problemas do tipo **caminho mais curto entre dois vértices**. A terceira e a última iteração são equiparadas aos problemas: **Travelling Salesman Problem** e **Vehicle Routing Problem**.

Foram utilizados múltiplos algoritmos no sentido de resolver estes problemas, nomeadamente **Dijkstra**, **A***, **Held-Karp**, **Nearest Neighbour**,**Genético**, **Kosaraju** e **Tarjan**.

Estes algoritmos envolvem conhecimentos em várias áreas da programação, transpondo os conceitos abordados na cadeira, tais como: **bruteforce**, **recursão**, **programação dinâmica**, **algoritmos gananciosos** (**Dijkstra**, **A***, **Nearest Neighbour**), **heurísticas**, vários conceitos associados a algoritmos genéticos, tais como, **mutação**, **crossover**, **seleção** e outros tópicos associados a grafos, nomeadamente, **conetividade** e **ordem topológica**.

Com a realização deste trabalho, descobrimos novas maneiras de implementar algoritmos já conhecidos de maneira eficiente e eficaz. A realização da análise empírica também permitiu comparar a eficácia dos vários algoritmos aplicados ao longo deste projeto.

Cada membro do grupo foi responsável por uma igual parte do trabalho, sendo cada um responsável pelas seguintes partes do projeto:

- *Diogo Samuel Fernandes* - Descrição do Problema, Formalização do Problema, Perspetiva de solução, Funcionalidades a implementar, Estruturas de dados utilizadas, Algoritmos implementados e complexidade, Conectividade dos grafos utilizados, Conclusão
- *Hugo Guimarães* - Descrição do Problema, Formalização do Problema, Perspetiva de solução, Funcionalidades a implementar, Estruturas de dados utilizadas, Algoritmos implementados e complexidade, Conectividade dos grafos utilizados, Conclusão
- *Telmo Baptista* - Descrição do Problema, Formalização do Problema, Perspetiva de solução, Funcionalidades a implementar, Estruturas de dados utilizadas, Algoritmos implementados e complexidade, Conectividade dos grafos utilizados, Conclusão

9 Bibliografia

- Apresentações fornecidas pelo professor Rosaldo José Fernandes Rossetti nas aulas téoricas da cadeira Conceção e Análise de Algoritmos
- Shortest Path Problem,
  https://en.wikipedia.org/wiki/Shortest_path_problem
- Dijkstra's Algorithm,
  https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Bellman-Ford Algorithm,
  https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
- A* algorithm,
  https://en.wikipedia.org/wiki/A*_search_algorithm
- Admissible heuristic,
  https://en.wikipedia.org/wiki/Admissible_heuristic
- Traveling Salesman Problem,
  https://en.wikipedia.org/wiki/Travelling_salesman_problem
- Vehicle Routing Problem,
  https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Held-Karp algorithm,
  https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
- Nearest neighbour algorithm,
  https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- Genetic Algorithm,
  https://en.wikipedia.org/wiki/Genetic_algorithm
- Natural selection,
  https://en.wikipedia.org/wiki/Natural_selection
- DNA replication,
  https://en.wikipedia.org/wiki/DNA_replication
- Chromosomal crossover,
  https://en.wikipedia.org/wiki/Chromosomal_crossover
- Mutation,
  https://en.wikipedia.org/wiki/Mutation
- GeeksForGeeks - Strongly connected components,
  https://www.geeksforgeeks.org/strongly-connected-components
- Kosaraju's algorithm,
  https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

- Tarjan's algorithm,
  https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- GeeksForGeeks - Tarjan's algorithm,
  https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components
- Desmos Graphing Tool,
  https://www.desmos.com/calculator
- Path Finder Visualization Program,
  https://github.com/kevinwang1975/PathFinder
- Branch and Bound,
  https://en.wikipedia.org/wiki/Branch_and_bound
- Spacial Complexity Analysis,
  https://redirect.is/space_complexity