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

21 de Abril de 2020

Projeto CAL - 2019/20 - MIEIC

Professor das Aulas Práticas: Rosaldo José Fernandes Rossetti

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

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 optimizar o percurso dos veículos de forma a recolher e entregar os prisioneiros nos pontos de interesse. De modo a cumprir o pretendido, é possivel dividir 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 vá recolher os prisioneiros a uma dada localização.

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

Algumas vezes, obras nas vias públicas podem fazer com que certas zonas tornem-se inacessíveis, inviabilizando o acesso ao destino de alguns prisioneiros.

Segunda Iteração - Recolha de prisioneiros utilizando vários veículos

Durante a segunda fase vai-se ter em consideração o diverso número de veículos que a frota possui. Alguns veículos vão diferir de outros, tendo cada veículo uma determinada função. Vão existir veículos específicos para transportar prisioneiros até aos aeroportos e linhas de comboio.

Terceira Iteração - Recolha seletiva de prisioneiros utilizando vários veículos

Na terceira e última iteração vamos ter em consideração não só o número de veículos disponíveis, como também o número de pedidos de transporte de prisioneiros que um veículo pode realizar num único serviço. Um único veículo pode sair da central e transportar vários prisioneiros, desde que o número de prisioneiros não ultrapasse a capacidade do veículo.

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 especificos 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 especificos 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
- $\forall c \in C_f, \forall s \in S(c), endHour(s) > startHour(s)$
- $\forall c \in C_f, \forall s \in S(c), startHour(s) < pickupHour(\forall r \in R_f) < endHour(s) \wedge startHour(s) < destHour(\forall r \in R_f) \leq endHour(s)$

2.4 Função objetivo

A solução ótima passa por minizar 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 depois no 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. Assim, 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 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 problema do **Travelling Salesman Problem**, um problema NP-díficil.

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

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 *priorityqueue* 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 *priorityqueue* 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:

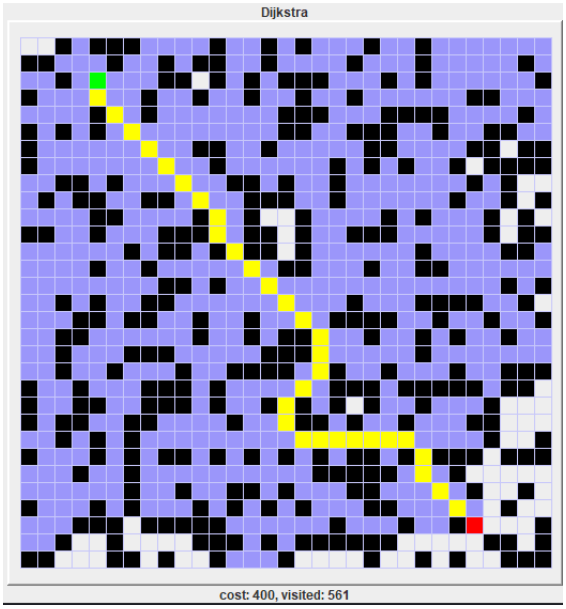

Figura 1: *Dijkstra's algorithm*

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)
    path.PUSH_FRONT(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, 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, COST(w)))
```

```
        ELSE
            DECREASE–KEY(Q, (w, 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:

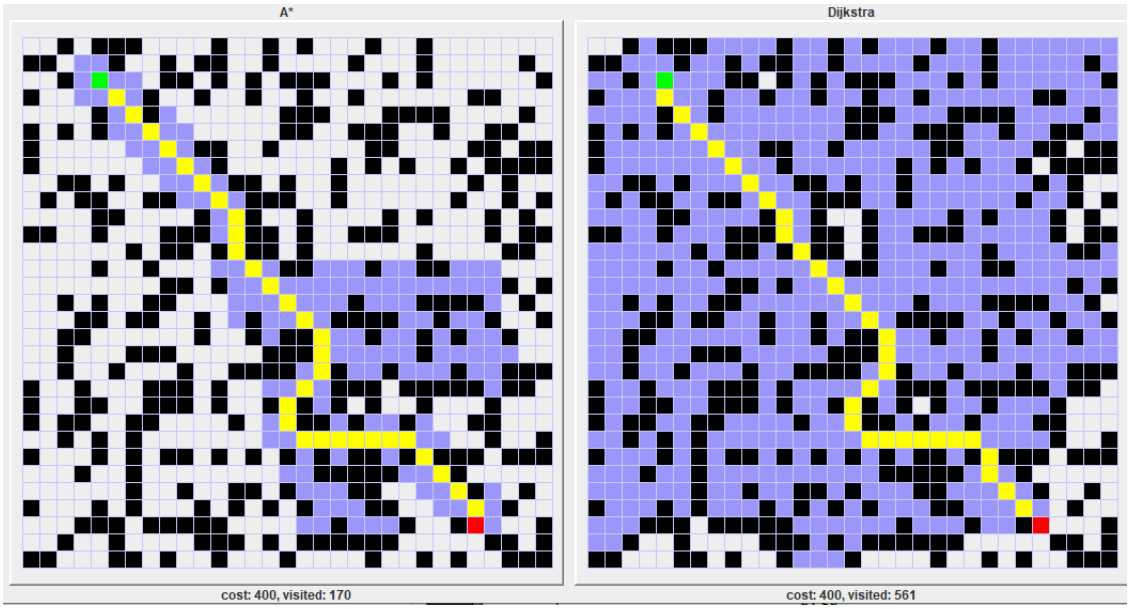

Figura 2: *A* algorithm* ***vs.*** *Dijkstra's algorithm*

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:

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.

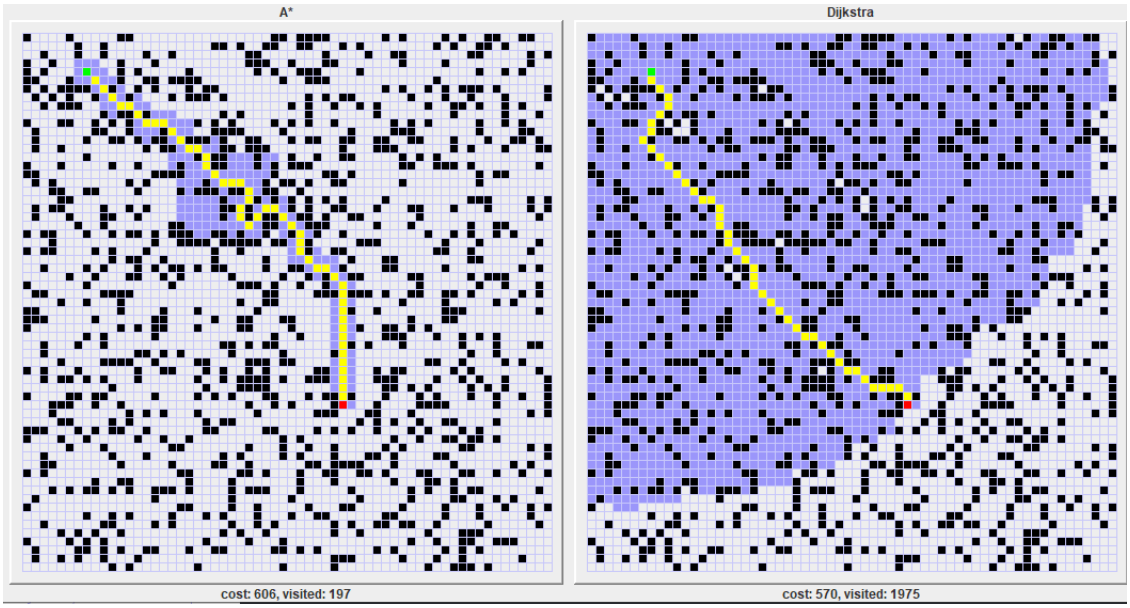

Figura 3: *A* algorithm* ***vs.*** *Dijkstra's algorithm*

walls visited
origin destination
shortest path

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.

Deparamo-nos então com um problema similar ao **Travelling Salesman Problem**, um problema NP-díficil. 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

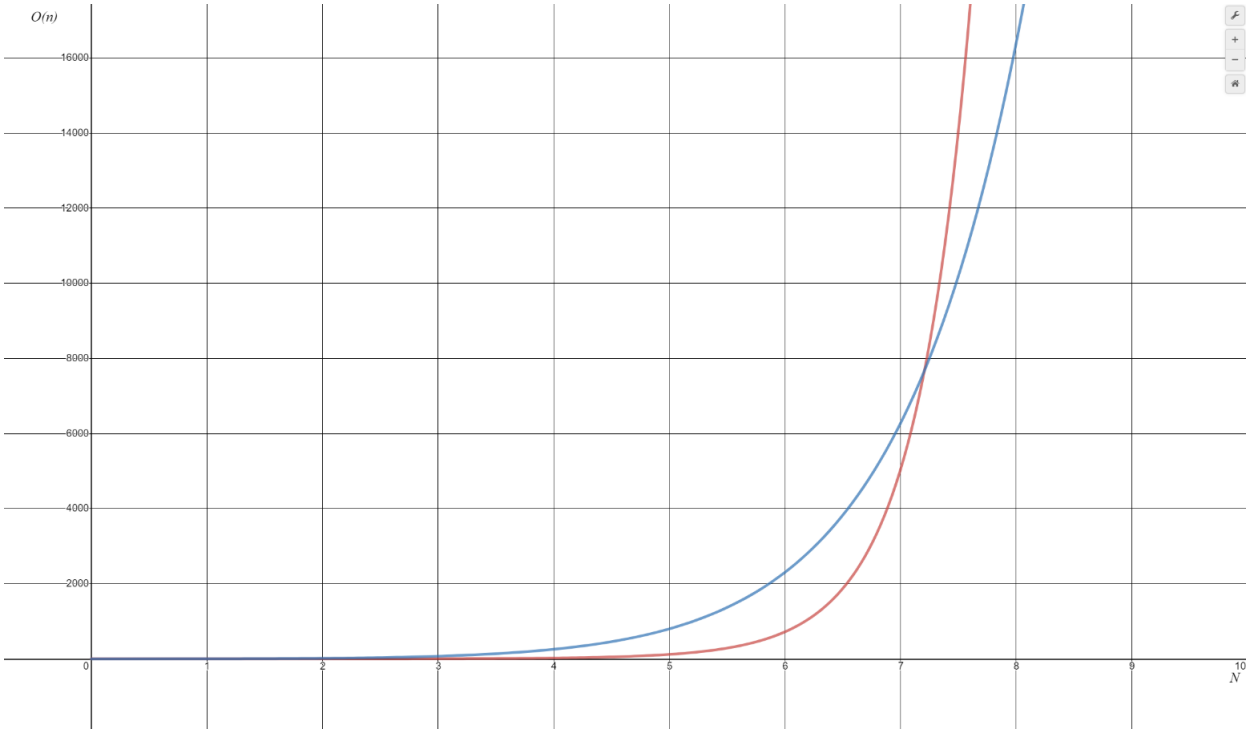

Figura 4: *Brute-force* ***vs.*** *Held-Karp algorithm: Complexities*

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

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

```
Pseudo Codigo
asdadssda
```

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:

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 ínicio, 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 \in 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
```

Algoritmo Genético

Algoritmos genéticos são algoritmos baseados em heurísticas que simulam o processo de evolução de espécies, o processo de *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
```

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.

```
// 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 de 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
- Agrupamento dos veículos

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

É necessário ter em consideração a urgência do transporte de um prisioneiro, pelo que se devem agrupar os prisioneiros cujos locais de partida e chegada sejam pouco distantes, priorizando sempre o transporte dos prisioneiros mais urgentes.

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

5 Conclusão

6 Bibliografia

- Apresentações fornecidas pelo professor Rosaldo José Fernandes Rossetti nas aulas téoricas da cadeira Conceção e Análise de Algoritmos
- Shortest Path Problem
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- A* algorithm
- Admissible heuristic
- Traveling Salesman Problem
- Vehicle Routing Problem
- Held-Karp algorithm
- Nearest neighbour algorithm
- Genetic Algorithm
- Natural selection
- DNA replication
- Chromosomal crossover
- Mutation
- GeeksForGeeks - Strongly connected components
- Kosaraju's algorithm
- Tarjan's algorithm
- GeeksForGeeks - Tarjan's algorithm
- Desmos Graphing Tool
- Path Finder Visualization Program