U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto de Concepção e Análise de Algoritmos

PapaRica: Distribuição de refeições prontas

André de Jesus Fernandes Flores – up201907001

Diogo Luís Araújo de Faria – up201907014

Tiago André Batista Rodrigues – up201906807

Índice

Primeira Parte

1. Descrição do tema

A **PapaRica** é uma empresa de confeção e distribuição de refeições prontas a consumir. Para tal, existe uma frota de veículos encarregue das entregas ao cliente, sendo que as recolhe diariamente em Vila do Conde para as distribuir na área metropolitana do Porto.

As encomendas são distribuídas em cabazes, sendo identificadas com a informação relevante acerca do destino e conteúdo.

O trabalho consiste em implementar um sistema que calcule os trajetos ótimos para a realização das entregas.

Numa primeira fase vai-se considerar a existência de um único veículo de entrega com capacidade ilimitada, sendo que se expande numa segunda fase a uma frota de veículos de diferentes capacidades e tipos.

Um exemplo de um trajeto de um veículo pode ser:

Sede -> Cliente 1 -> … -> Cliente n -> Sede

Para uma entrega poder ser realizada tem de existir pelo menos um trajeto possível que permita sair e retornar à Sede passando por todos os Clientes, tendo em consideração a existência de obras públicas que podem, em certas situações, impossibilitar a entrega de certas encomendas.

2. Formalização do problema

2.1. Dados de entrada

- Vi – Sequência de veículos disponíveis para utilização, sendo Vi[n] o n-ésimo elemento e caracterizado por:
  - type – Tipo de veículo (numa fase inicial = 'heavy');
  - cap – Capacidade (numa fase inicial = ∞).
- Ci – Sequência de cabazes para distribuição, sendo Ci[n] o n-ésimo elemento e caracterizado por:
  - clientName – Nome do destinatário;
  - packageNumber – Número de embalagens contidas no cabaz;
  - invoiceNumber – Número da fatura;
  - destAddress – Vértice de destino.
- Gi = (Ni, Ei) – Grafo dirigido pesado inicial, composto por:
  - N – Vértices do grafo, que representam pontos da cidade, caracterizados por:
    - type – Tipo de vértice;
    - address – Endereço;
    - adj ⊆ E – Conjunto de arestas que se iniciam do vértice.
  - E – Arestas do grafo, que representam o caminho entre 2 vértices, caracterizados por:
    - weight – Peso da aresta, que representa a distância;
    - dest ∈ Ni – Vértice de destino;
    - ID – Identificador de aresta.
- S ∈ Ni – Vértice que representa a Sede, de onde o veículo sai e retorna.

2.2. Dados de Saída

- Gf = (Nf, Ef) – Grafo dirigido pesado final, sendo que Nf = Ni e Ef = Ei.
- Vf – Sequência de veículos utilizados em entregas, sendo Vf[n] o n-ésimo elemento. Cada um é caracterizado por:
  - T – Sequência de vértices ordenados por ordem de passagem (com possibilidade de repetidos);
  - B – Sequência de cabazes para entrega pelo veículo ordenada pela ordem de entrega;
  - cap – Capacidade ocupada do veículo.

2.3. Restrições

2.3.1. Restrições sobre os dados de entrada

- ∀ n ∈ [0, |Vi| - 1]:
  - type(Vi[n]) = 'light' ∨ 'heavy' ∨ 'motorcycles' – tipos de veículos têm de ser carros ligeiros ('light'), carros pesados ('heavy') ou motociclos ('motorcycles');
  - cap(Vi[n]) >= 0 – capacidade têm de ser maior ou igual a zero, visto que se trata de uma quantidade de embalagens.
- ∀ n ∈ [0, |Ci| - 1]:
  - packageNumber(Ci[n]) >= 1 – número de embalagens tem de ser positivo, visto que um cabaz tem de ter pelo menos uma embalagem;

    - invoiceNumber(Ci[n]) >= 1 – número de fatura tem de ser maior ou igual a um;
    - destAddress(Ci[n]) ∈ Ni – vértice de destino tem de existir no conjunto de vértices do grafo.
- ∀ n ∈ Ni, type(n) = 'HQ' ∨ 'destiny' ∨ 'intermediate'.
- ∀ e ∈ Ei:
    - weight(e) >= 0 – arestas têm de ter um peso igual ou maior que zero visto que este corresponde à distância;
    - ID(e) >= 0 ∧ $\{\forall e1, e2 \in Ei \mid ID(e1) = ID(e2) \rightarrow e1 = e2\}$ – identificador de uma aresta tem de ser maior ou igual a zero e único para cada uma;
    - Deve ser utilizável por veículos.
- type(S) = 'HQ' – tipo de sede deve ser 'HQ'.

2.3.2. Restrições sobre os dados de saída

- $\forall vi \in Vi, \exists vf \in Vf \mid vi = vf$ – os vértices iniciais e finais vão ser iguais.
- $\forall ei \in Ei, \exists ef \in Ef \mid ei = ef$ – as arestas iniciais e finais vão ser iguais.
- Vf ⊆ Vi – os veículos utilizados têm de ser parte do conjunto de veículos disponíveis.
- ∀ v ∈ Vf:
    - T(v) ⊆ Vi – os vértices têm de fazer parte do conjunto de vertices iniciais;
    - B(v) ⊆ Ci – o conjunto de cabazes para entrega tem de fazer parte do conjunto de cabazes inicial;
    - cap(v) = $\sum_{k=0}^{|B|-1} packageNumber(B[k])$ – a capacidade utilizada de um veículo tem de ser igual ao número de embalagens totais em cabazes;
    - Capacidade utilizada do veículo tem ser menor ou igual à capacidade disponível do veículo.
- T[0] ∈ adj(S) ∧ dest(T[|T| - 1]) = S – o trajeto de todos os veículos começa e acaba na sede.

2.4. Funções objetivo

O objetivo do trabalho é encontrar o mínimo número de veículos para entregar as encomendas e o menor trajeto para cada veículo possível. Para tal é necessária a minimização de duas funções, sendo que uma se refere ao menor número de veículos e a outra ao menor trajeto possível:

- f = |Vf|
- g = $\sum_{v \in Vf} \sum_{e \in T} weight(e)$

Numa fase inicial devido à consideração de uma capacidade infinita de veículos, a minimização da função f é desnecessária, sendo que na fase seguinte se prioriza a sua minimização à função g.

3. Perspetiva de Solução

1.1. Primeira fase

Inicialmente vai ser considerada a existência de apenas um veículo com capacidade infinita para a distribuição das encomendas, sendo então apenas necessário encontrar o trajeto ótimo para o mesmo. Para tal, seguem-se as próximas etapas:

1. Remoção das arestas indesejáveis, que vão ser as inutilizáveis pelos veículos.
2. Pré-processamento em que se calculam as distâncias entre todos os pares de pontos.
3. Verificar se os pontos de interesse (Sede e clientes) pertencem a uma componente fortemente conexa.
4. Ordenação dos pontos de interesse.
5. Calcular a sequência de vértices a percorrer no trajeto.

1.2. Segunda fase

Numa segunda fase, já vão ser considerados vários veículos de diferentes tipos e capacidades, pelo que é necessária uma distribuição ótima das encomendas pelos veículos disponíveis antes do cálculo do trajeto ótimo. Para tal, são necessárias as seguintes etapas:

1. Remoção das arestas indesejáveis, que vão ser as inutilizáveis pelos veículos.
2. Pré-processamento em que se calculam as distâncias entre todos os pares de pontos.
3. Distribuir as encomendas pelos veículos disponíveis.
4. Verificar se os pontos de interesse (Sede e clientes) pertencem a uma componente fortemente conexa.
5. Ordenação de pontos de interesse para cada veículo, considerando os pontos de entrega específicos do trajeto.
6. Calcular a sequência de vértices a percorrer no trajeto de cada veículo.

1.3. Remoção das arestas indesejáveis

A remoção vai consistir em encontrar as arestas inutilizáveis por veículos, atribuindo ao seu peso o valor de infinito, utilizando uma pesquisa em profundidade, que ao percorrer todos os vértices os assinala como visitados e nos permite no final encontrar aqueles que tal não é possível.

Eficiência temporal: O(|N| + |E|); Eficiência espacial: O(|N|)

1.4. Distribuição das encomendas

Esta etapa vai ser realizado utilizando o algoritmo “First Fit Decreasing”[1], que ordena a sequência de cabazes por número de embalagens crescente e atribui sucessivamente a veículos, já ordenados pela sua capacidade, até que não existam mais cabazes para distribuição.

Eficiência temporal: O(|C| * log(|C|)); Eficiência espacial: O(|C|); Sendo |C| o número de cabazes;

1.5. Pré-processamento

Nesta etapa considerou-se a utilização de 2 diferentes algoritmos dependendo da densidade do grafo utilizado. Para um grafo esparso, utilizar-se-ia o algoritmo de Dijkstra repetidamente de forma a se obterem as distâncias entre cada par de ponto, enquanto que para um grafo denso, utilizar-se-ia o algoritmo de Floyd-Warshall.

Neste caso, considera-se que o grafo a utilizar da cidade metropolitana do Porto é um grafo denso e, por isso, utiliza-se o algoritmo de Floyd-Warshall.

Algorithm 1 Floyd-Warshall with path reconstruction

```
1:  dist ← |V| × |V| length array of minimum distances initialized to ∞
2:  next ← |V| × |V| length array of vertex indices initialized to null
3:  procedure FLOYD-WARSHALL(PATH RECONSTRUCTION)
4:      for each edge(u, v) do
5:          dist[u][v] ← w(u, v)
6:          next[u][v] ← v
7:      for k = 1 to |V| do
8:          for i = 1 to |V| do
9:              for j = 1 to |V| do
10:                 if dist[u][v] > dist[i][k] + dist[k][j] then
11:                     dist[u][v] ← dist[i][k] + dist[k][j]
12:                     next[i][j] ← next[i][k]
13: procedure GETPATH(u, v)
14:     if next[u][v] = null then
15:         return []
16:     path = [u]
17:     while u ≠ v do
18:         u ← next[u][v]
19:         path.append(u)
20:     return path
```

Eficiência temporal: $O(|N|^3)$; Eficiência espacial: $O(|N|^2)$;

1.6. Verificar se os pontos pertencem a um CFC

Este passo vai ser realizado utilizando o método lecionado na cadeira:

- Pesquisa em profundidade no grafo G determina floresta de expansão, numerando vértices em pós-ordem;
- Inverter todas as arestas de G;
- Segunda pesquisa em profundidade, em Gr, começando sempre pelo vértice de numeração mais alta ainda não visitado;
- Cada árvore obtida é um componente fortemente conexo.
- Percorrer cada CFC e verificar se têm todos os vértices do trajeto (Sede e clientes), sendo que se nenhum CFC for encontrado não vai existir um trajeto possível.

1.7. Ordenação dos pontos de interesse do trajeto

Para tal, utiliza-se o algoritmo de Nearest Neighbour[2] que, para cada vértice se calcula o vértice que se encontra a menor distância sucessivamente até se encontrar o trajeto final.

Este algoritmo é um método de construção heurística que apresenta um caminho aproximadamente 25% mais longo do que o caminho de menor distância exato. Tal facto é contrariado pelo facto de ser um algoritmo com boa eficiência temporal quando comparado com métodos exatos.

Algorithm 2 TSP by Nearest Neighbour

```
1: V ← vertices representing all destinations in the route
2: procedure NEAREST NEIGHBOUR(Vertex P)
3:     sortedVertices ← V \ {P}
4:     result = [P]
5:     while |sortedVertices| > 0 do
6:         sortedVertices.sortRelativeTo(P)
7:         P ← sortedVertices[0]
8:         sortedVertices.remove(P)
9:         result.append(P)
10:    result.append(result[0])
11:    return result
```

Eficiência temporal: O(|A^2|); Eficiência espacial: O(|A|); Sendo |A| o número de clientes

1.8. Calcular a sequência de vértices a percorrer

De forma a calcular esta sequência, utiliza-se sucessivamente um algoritmo, GetPath mencionado no algoritmo 1, para encontrar o caminho entre dois pontos sucessivamente entre cada ponto sucessivo dos vértices principais utilizando a array de vértices “next” calculada no algoritmo de Floyd-Warshall.

Eficiência temporal: O(|T|); Eficiência Espacial: O(|T|); Sendo |T| o número de vértices a percorrer no trajeto.

4. Conclusão

Com a preparação necessária para a elaboração deste relatório e consequente planeamento do projeto, adquirimos uma melhor compreensão do uso de grafos e algoritmos relacionados com os mesmos.

O trabalho foi dividido igualmente pelos 3 elementos do grupo.

Segunda Parte

1. Diferenças entre planeamento & implementação

O primeiro passo que foi dado no planeamento foi a remoção das arestas indesejáveis que, como explicado, consistia numa pesquisa em profundidade em começo no vértice da Sede e que marcava as arestas que chegassem a vértices impossíveis de alcançar com o valor de infinito.

No entanto, na implementação nós utilizamos, de forma a diminuir o grande número de vértices e, consequentemente, arestas, do grafo, um cálculo da maior componente fortemente conexa, considerando-se como Sede um vértice dessa mesma.

Assim, torna-se desnecessário a pesquisa em profundidade, visto que devido ao facto de os vértices pertencerem à componente fortemente conexa, é implícito que são vértices alcançáveis.

Por sua vez, este passo também faz com que a verificação de se a Sede e Clientes estão numa componente fortemente conexa seja desnecessário, visto que necessariamente têm de estar e, qualquer tentativa de se adicionar algum Cliente já vai verificar o mesmo também.

Uma outra diferença foi que se adicionou a implementação do algoritmo A* de forma a substituir o Floyd-Warshall, pelo que no implementação final existe a possibilidade de correr o algoritmo utilizando um dos dois algoritmos.

2. Funcionalidades implementadas

Quando se corre o programa, a seguinte mensagem aparece:

```
Do you want to add anything before starting?
[0] Add nothing.
[1] Add a Vehicle.
[2] Add an Order.
```

Opção 0

A opção 0 é simplesmente para correr o programa e receber os resultados, enquanto que as outras duas permitem adicionar mais coisas pelo utilizados.

Ao escolher 0 é nos dada a opção de escolher que algoritmo vai ser utilizado na escolha da trajetória:

```
With which algorithm do you want to run the program with?
[0] A*
[1] Floyd-Warshall
```

Escolhendo a opção 0, utiliza-se o A* na escolha do caminho, enquanto que se escolher o Floyd-Warshall tem de se realizar um pré-processamento primeiro antes de percorrer para o resto.

Opção 1

Ao escolher esta opção pode-se adicionar um novo veículo à empresa para que possa também efetuar entregas.

De seguida seleciona-se o tipo de veículo:

```
Select vehicle type:
[0] Heavy
[1] Light
[2] Motorcycle
[3] Go to the main menu
```

Escolhendo, pode-se também especificar a capacidade ou deixar como infinito, se preferível.

```
Input vehicle capacity (Write 0 if infinite or -1 to go back to the main menu):
```

Opção 2

Com esta opção pode-se adicionar outra ordem para entrega.

```
How many meals in the basket? (Write 0 to go back to the main menu)
```

```
Write your name 'First Last' (Write 0 to go back to the main menu):

Write the address in terms of x and y coordinates (Write 'a' to go back to the main menu):
x:

y:
```

Estas opções permitem criar uma ordem, dando o número de refeições por cabaz, seguido pelo nome da pessoa e por fim pela localização, dada em termos de x e y.

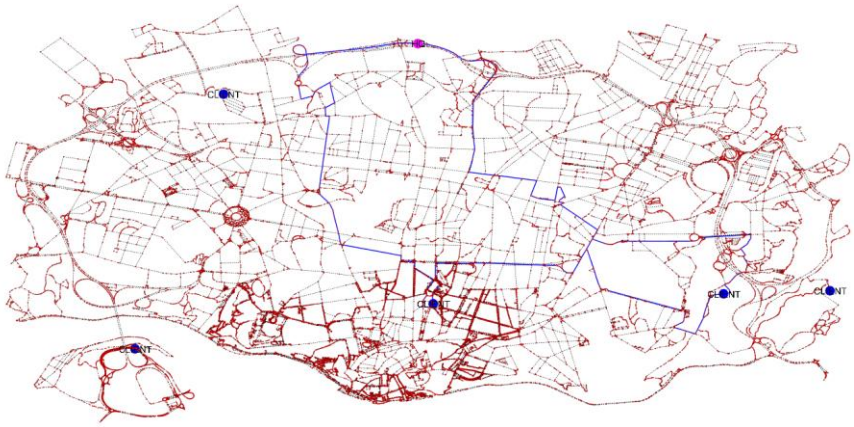
A localização é verificada e se não existir no grafo ou não for uma localização a que seja possível chegar a seguinte mensagem aparece e é dada a oportunidade de escrever de novo a localização, por motivos de engano, ou voltar para o inicio sem adicionar a ordem.

```
 Node doesn't belong to graph or isn't reachable.
Do you want to try again?
[0] Yes
[1] Go back to the main menu
```

Correr algoritmo

Depois de selecionar as opções pretendidas anteriormente, o algoritmo corre, sendo escrito na consola o número diferente de trajetos a percorrer pelos diferentes veículos.

Cada veículo diferente vai resultar num trajeto diferente e, consequentemente, numa janela com um grafo, em que está de cor diferente o trajeto a percorrer, diferente.



Esta imagem mostra um exemplo, em que o vértice a cor-de-rosa e a dizer “HQ” representa a sede e os vértices a azul e a dizer “Client” representam os clientes. As arestas a cor azul representam as arestas percorridas no caminho pelo veículo.

Fechando a janela, vai ser calculado o caminho para o veículo seguinte que, no final, vai ser apresentado da mesma forma e assim sucessivamente.

3. Estruturas de dados

Graph, Node & Edge

Estas estruturas são muito semelhantes às utilizados no decorrer do semestre, com apenas algumas funções ou atributos, como tags de forma a proceder à identificação de Clientes ou Sede.

MealBasket

Esta classe representa um cabaz, sendo caracterizado por:

```
class MealBasket {
private:
    static unsigned int counter;                        //Used to assign different invoice numbers
    unsigned int packageNumber;                         //Number of packages
    unsigned int invoiceNumber;                         //Unique invoice number
    std::string clientName;                             //Client Name
    std::pair<long double, long double> destAddress;    //Destination address in x and y coordinates
```

O atributo counter é incrementado por cada cabaz adicionado, começando em 0, e o atributo destAddress tem de ter coordenadas que correspondem a um vértice que seja alcançável.

Vehicle & vehicle_type

A classe Vehicle representa um veículo específico, enquanto que vehicle_type é um enum que representa os tipos de veículos. São caracterizados por:

```
enum vehicle_type {HEAVY, LIGHT, MOTORCYCLE};

class Vehicle {
    vector<MealBasket> meals;           //Meals to deliver
    vehicle_type type;                  //Vehicle type
    unsigned int cap;                   //Capacity in terms of number of packages it can hold
    unsigned int used;                  //Used capacity
```

O atributo meals, começa vazio visto que ainda não foram escolhidas as encomendas a entregar, o atributo type tem como valor default 'HEAVY', a capacidade tem como defaul 'INT_MAX' e o atributo used é inicializado com 0;

PapaRica

Esta classe é utilizada para a interação entre utilizador e aplicação e também para a aplicação dos algoritmos de distribuição de cabazes e consequente criação de caminho para cada veículo.

4. Algoritmos implementados & análise de complexidade

Kosaraju

Este algoritmo retorna a maior, que tem mais vértices, componente fortemente conexa calculável no grafo dado fazendo o cálculo de várias diferentes componentes e escolher a maior.

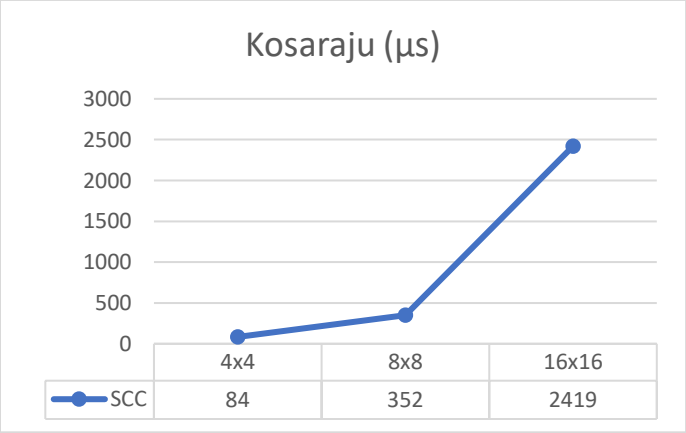
Algorithm 4 Get Largest Strongly Connected Component

```
1:  graph = (V, E)
2:  procedure GetSCCsByKosaraju
3:      result ← vector of vectors of nodes
4:      nodeStack ← stack of nodes
5:      setAllNodesToNotVisited(graph)
6:      for each node ∈ graph.getNodes() do
7:          SCCVisit(node, nodeStack)
8:      transpose ← graph.getTranspose()
9:      setAllNodesToNotVisited(graph)
10:     while not nodeStack.empty() do
11:         node ← nodeStack.top()
12:         nodeStack.pop()
13:         if not node.isVisited() then
14:             result.add(transpose.DFS(node))
15:     return result
16:
17: procedure SCCVisit(node, nodeStack)
18:     node.setVisitedToTrue()
19:     for each edge ∈ node.getEdges() do
20:         if not edge.getDestination().isVisited() then
21:             SSCVisit(edge.getDestination(), nodeStack)
22:     nodeStack.push(node)
23:
24: procedure GetLargestSCC
25:     SCCs ← GetSCCsByKosaraju
26:     nodes ← maxVectorBySize(SCCs)
27:     return nodes
```

Complexidade temporal: O(|N| + |E|)

Complexidade espacial: O(|N|)

Análise empírica temporal:



Floyd-Warshall

Numa primeira fase, nós tínhamos ideia que o grafo iria ser mais denso do que o que realmente seria, sendo que se procedeu à implementação deste algoritmo de forma a serem calculadas as distâncias entre todos os pontos de forma a facilitar os cálculos futuros no caminho mais curto e também a reconstrução de caminho necessário no fim.

Algorithm 1 Floyd-Warshall with path reconstruction

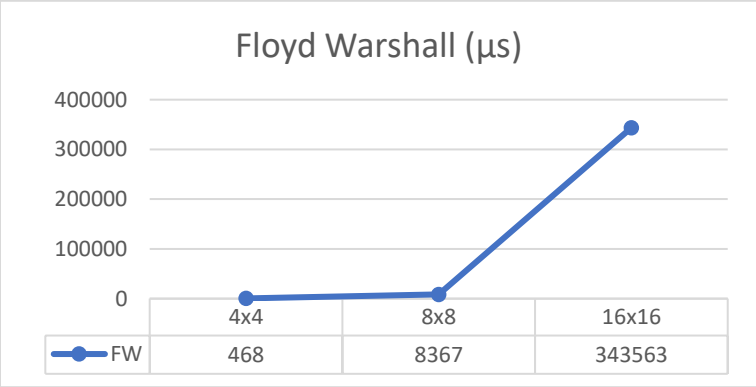
```
1:  dist ← |V| × |V| length array of minimum distances initialized to ∞
2:  next ← |V| × |V| length array of vertex indices initialized to null
3:  procedure FLOYD-WARSHALL(PATH RECONSTRUCTION)
4:      for each edge(u, v) do
5:          dist[u][v] ← w(u, v)
6:          next[u][v] ← v
7:      for k = 1 to |V| do
8:          for i = 1 to |V| do
9:              for j = 1 to |V| do
10:                 if dist[u][v] > dist[i][k] + dist[k][j] then
11:                     dist[u][v] ← dist[i][k] + dist[k][j]
12:                     next[i][j] ← next[i][k]
13: procedure GETPATH(u, v)
14:     if next[u][v] = null then
15:         return []
16:     path = [u]
17:     while u ≠ v do
18:         u ← next[u][v]
19:         path.append(u)
20:     return path
```

Visto que este algoritmo é efetuado numa componente fortemente conexa, e não no grafo como um todo, |N| e |E| são o número de vértices e arestas, respetivamente, nesta componente.

Complexidade temporal: $O(|N|^3)$

Complexidade espacial: $O(|N|^2)$

Análise empírica temporal:



A*

Com a informação de que o grafo acaba por ser mais esparso, em que o número de vértices e arestas difere por pouco, também implementamos o algoritmo A*.

Algorithm 5 Shortest Path Between Two Points By A* Algorithm

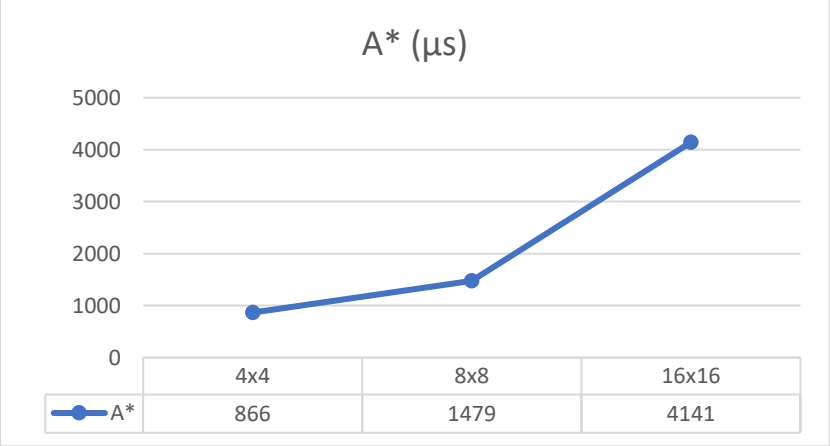
```
1:  procedure distMin(node1, node2)
2:      return staright line distance between node1 and node2 based on coordinates
3:
4:  procedure GetAStarPath(G = (V, E), srcNode, destNode)
5:      for each v ∈ V do
6:          dist(v) ← INF → Distance from the source node
7:          path(v) ← null → Path from the source node
8:          visited(v) ← false → Check if it is visited
9:      pqueue → Priority Queue of pairs of distances and nodes by ascending order of distance
10:     dist(srcNode) ← 0.0
11:     path(srcNode).push(srcNode)
12:     ENQUEUE(pqueue, pair(0.0, srcNode))
13:     while not pqueue.isEmptty() do
14:         node ← DEQUEUE(pqueue)
15:         visited(node) ← true
16:         if node = destNode then
17:             break
18:         for each edge ∈ node.getEdges() do
19:             dest ← edge.getDestination()
20:             weight ← edge.getWeight()
21:             distNext ← distMin(dest, destNode)
22:             distCurrent ← distMin(node, destNode)
23:             aStarHeuristic ← weight + distNext - distCurrent
24:             if not visited(dest) and dist(dest) ¿ dist(node) + weight + aStarHeuristic then
25:                 dist(dest) ← dist(node) + weight + aStarHeuristic
26:                 ENQUEUE(pqueue, pair(dist(dest), dest))
27:                 path(dest) = path(node)
28:                 path(dest).push(dest)
29:     return path(nodeDest)
```

Sendo B o número de arestas sucessoras e D o quão a função tem de pesquisar:

Complexidade temporal: O(B ^ D)

Complexidade espacial: O(B ^ D)

Análise temporal empírica (desde ponto 0 até ponto maior nos diferentes grafos):



First Fit Decreasing

De forma a ser realizada a distribuição das encomendas pelos veículos diferentes, visto que se fez a implementação da 2 fase que se falou, utiliza-se este algoritmo, que distribui as encomendas ordenadas por ordem crescente pelos veículos disponíveis, sendo que estes também estão ordenados pela sua capacidade.

Algorithm 3 First Fit Decreasing

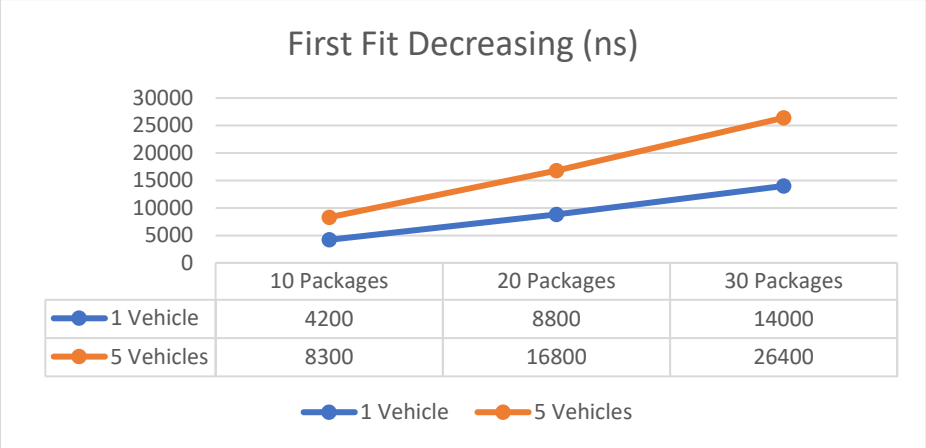
```
1: Vehicles ← all available vehicles
2: Orders ← all orders to distribute
3: procedure FFD
4:     sortVehiclesByCapacity(Vehicles)
5:     sortOrdersByCapacityDescending(Orders)
6:     for each order ∈ Orders do
7:         for each vehicle ∈ Vehicles do
8:             if vehicle.usedCapacity + order.packageNumber <= vehicle.maxCapacity then
9:                 vehicle.addOrder(order)
10:                break
11:    Orders.clear()
```

Sendo |C| o número de cabazes e |V| o número de veículos:

Complexidade temporal: O(|C| * |V|)

Complexidade espacial: O(|C|)

Análise temporal empírica:



Nearest Neighbour

Utilizamos este algoritmo para encontrar o caminho que um veículo tem de percorrer, iniciando o seu caminho no vértice da Sede, passando por todos os clientes e acabando na Sede. Apesar deste algoritmo apresentar um caminho aproximadamente 25% mais longo do ótimo, ele apresenta uma melhor complexidade temporal do que os algoritmos para caminhos ótimos.

Para este algoritmo foram criadas duas diferentes implementações que apenas diferem na forma de obter distância entre dois vértices diferentes na altura em que se realiza um "sort" relativo a um vértice especifico.

Algorithm 2 TSP by Nearest Neighbour

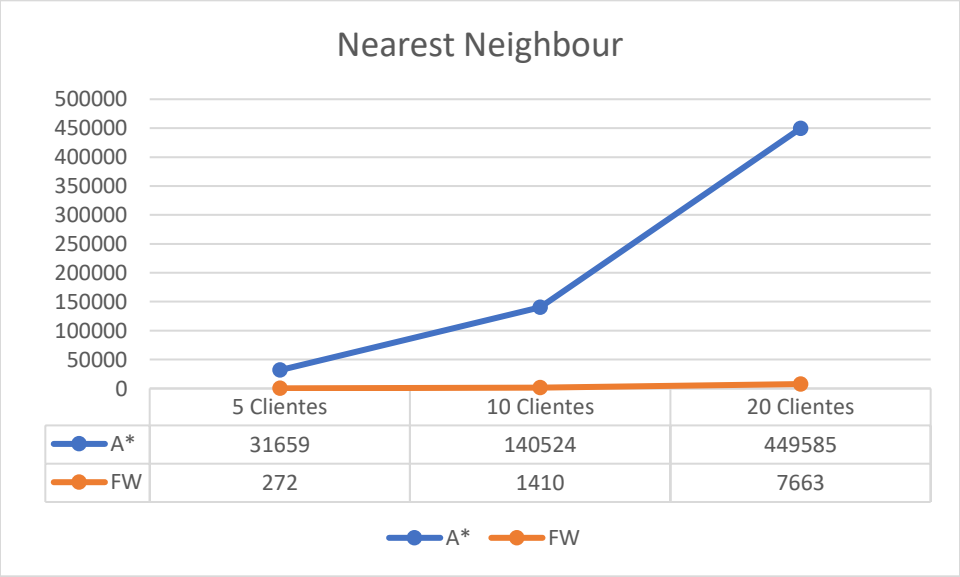
```
1:  V ← vertices representing all destinations in the route
2:  procedure NEAREST NEIGHBOUR(Vertex P)
3:      sortedVertices ← V \ {P}
4:      result = [P]
5:      while |sortedVertices| > 0 do
6:          sortedVertices.sortRelativeTo(P)
7:          P ← sortedVertices[0]
8:          sortedVertices.remove(P)
9:          result.append(P)
10:     result.append(result[0])
11:     return result
```

Sendo |A| o número de vértices de clientes:

Complexidade temporal: $O(|A^2|)$

Complexidade espacial: $O(|A|)$

Análise temporal empírica:



5. Análise da Conectividade

A conectividade de um grafo tem um impacto enorme nos nossos algoritmos, pelo que a utilização de vértices e arestas desnecessárias resulta num algoritmo significativamente mais lento do que apenas utilizando os necessários.

Por esta mesma razão, utilizou-se o algoritmo de Kosaraju mencionado anteriormente no início da execução do programa de forma a reduzir o número de vértices e arestas, calculando a maior componente fortemente conexa, da qual a Sede vai pertencer.

Assim, visto que qualquer trajeto vai iniciar e acabar no mesmo vértice, Sede, todos os clientes têm de necessariamente pertencer a uma componente fortemente conexa em que a mesma esteja inserida, logo é possível descartar os vértices, e por consequência arestas, que existem fora do grafo originado pelo algoritmo.

6. Conclusão

Concluindo, no decorrer deste projeto foi necessário o estudo de vários algoritmos, vários que não foram implementados pela sua complexidade ou pelo facto de se encontrarem melhores, permitindo que o nosso conhecimento deste tipo de problemas melhorasse imenso.

Foram realizadas duas implementações diferentes, diferindo no uso do algoritmo de Floyd-Warshall e do algoritmo de A*, sendo que, como esperado, o A*, apesar de apresentar tempos extremamente superiores ao do Floyd-Warshall na sua utilização durante o Nearest Neighbour, acaba por resultar num programa mais rápido, visto que não necessita de um expendioso pré processamento.

Tal conclusão pode não ser tão realizada pelos gráficos apresentados pois eles foram realizados em grafos em rede que são densos, ao contrário do grafo da cidade do Porto que é mais esparso. No entanto, neste grafo não foi possível obter resultados utilizando o algoritmo de Floyd-Warshall pois demorava demasiado tempo a correr.

Esta parte, tal como a primeira, foi divida igualmente pelos 3 elementos do grupo.

Bibliografia

[1] Dósa, György. (2007). The tight bound of first fit decreasing bin-packing algorithm is FFD(I) ≤ 11/9 OPT(I) + 6/9. Lect Notes Comput Sci. 4614. 1-11. 10.1007/978-3-540-74450-4_1.

[2] Johnson, D. S.; McGeoch, L. A. (1997). “The Traveling Salesman Problem: A Case Study in Local Optimization” (PDF). In Aarts, E. H. L.; Lenstra, J.K. *Local Search In Combinatorial Optimization*. London: John Wiley and Sons Ltd. pp. 215-310