U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Empresa de distribuição de frutas e vegetais

Projeto - 1ª Relatório
Conceção e Análise de Algoritmos EIC0110-2019/2020-2S
Professor Regente Rosaldo Rossetti
Turma 4 - Grupo 9

Deborah Marques Lago	up201806102	up201806102@fe.up.pt
João Paulo Silva da Rocha	up201806261	up201806261@fe.up.pt
Ricardo Amaral Nunes	up201706860	up201706860@fe.up.pt

Abril/2020

Lista de Símbolos e Definições

G - Grafo

E - Conjunto de arestas de um grafo G

V - Conjunto de vértices de um grafo G

|E| - Número de arestas de um grafo G

|V| - Número de vértices de um grafo G

Vi - Vértice inicial

Vf - Vértice final

O - Complexidade Temporal

1. Introdução

O objetivo do presente trabalho é desenvolver um sistema eficaz que permita a gestão de frotas e de entregas de produtos frescos aos clientes, como resultado parcial da Unidade Curricular Concepção e Análise de Algoritmos. O sistema proposto visa reduzir o tempo de entrega assim como investir nos caminhos mais curtos e otimizados entre o local de origem e os de destino. Serão utilizados, portanto, os algoritmos apropriados à resolução de cada problema. Será avaliada também a conectividade do grafo produzido para esta rede de entregas, de forma a evitar que sejam aceitas entregas em zonas inacessíveis da rede. Outro obstáculo a ser tratado são possíveis obras nas vias públicas que podem fazer com que certas zonas se tornem inacessíveis temporariamente, consequentemente inviabilizando as entregas.

2. Descrição do Problema

Uma empresa de logística e distribuição de frutas e vegetais do distrito da Guarda, denominada FarmFresh2U, tem camiões que recolhem diariamente os produtos frescos da quinta do produtor e entregam-nos nas casas dos clientes. Depois de as entregas estarem concluídas, os camiões recolhem-se à garagem, que pode estar estar localizada na quinta do produtor ou noutro local.

Este tipo de transporte de mercadorias frescas, de elevada qualidade, é cada vez mais frequente em zonas urbanas, permitindo o acesso a produtos da época, que não andaram a passear congelados entre continentes, e que foram cultivados de forma natural, cujo sabor, no prato, se faz notar. Portanto, deve-se levar em consideração que quão mais otimizado for o sistema de entregas, mais fiel ao princípio da empresa essa solução será.

3. Identificação do Problema

Pretende-se implementar um sistema que permita à empresa gerir a sua frota e a sua carteira de entregas. Os produtos são recolhidos diariamente na quinta do produtor onde se encontram já distribuídos por cabazes, os quais estão identificados com o nome do destinatário, peso do cabaz, destino, número de fatura, entre outras informações.

Este problema pode ser decomposto em quatro fases:

3.1. 1ª Fase

No início da implementação, durante esta fase inicial, a empresa tem apenas um camião que fará uma entrega. O objetivo, portanto, será encontrar o caminho mínimo entre a quinta do produtor, a morada do único cliente e o retorno à garagem (que pode ser a própria quinta).

3.2. 2ª Fase

A empresa, aqui, continua a ter apenas um camião, mas este é capaz de realizar a entrega de todos os cabazes a aguardar transporte. Podemos ignorar, a priori, a capacidade do camião referido. Portanto, tem-se ainda como meta encontrar o caminho mais curto entre a quinta do produtor e o destino final, sendo que o camião, agora, passa por todas as casas dos clientes que se encontram na lista de entregas do dia. No final das tarefas, o camião recolhe-se a uma garagem ou à própria quinta.

Será levada em consideração que cada cabaz a ser entregue está identificado com o nome do destinatário, peso do cabaz, número de factura e, principalmente, a localização do destino que será utilizada para criar o grafo de entregas, sendo necessário analisar a conectividade entre a origem e todos os pontos de interesse do dia, ou seja, as moradas dos clientes e a garagem.

3.3. 3ª Fase

A empresa conta com uma frota de camiões, de diferentes tipos e capacidades. É necessário agrupar itens de forma a não exceder a capacidade de carga do camião, tendo em conta a distância do camião ao destino. O objetivo será minimizar o caminho gerado, desde o produtor até à garagem, passando pelos pontos de entrega, assim como a menor quantidade de camiões utilizada para efetuar todas as entregas.

4. Formalização do Problema

4.1. Dados de Entrada

Ci - Sequência de camiões da empresa. Ci(i) é o seu i-ésimo elemento. Cada camião é caracterizado por:

- **cap** - Capacidade total do camião[1] (medida metros cúbicos)
- **type** - Tipo do camião

Bi - Sequência de cabazes (baskets) da empresa. Bi(i) é o seu i-ésimo elemento. Cada cabaz que é caracterizado por:

- **dest** - Destino do cabaz (id do vértice)
- **client** - Cliente que encomendou o cabaz
- **weight** - Peso do cabaz
- **volume** - Volume ocupado pelo cabaz (em metros cúbicos)
- **invoice** - Fatura

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

- **V** - Vértices (que representam pontos da cidade) com:
  - **type** - Tipos de vértices[2]
  - **coordinates** - coordenadas do ponto no mapa
  - **adj ⊆ E** - Conjunto de arestas que partem do vértice
- **E** - Arestas (que representam vias) com:
  - **weight** - peso da aresta (representa a distância entre os dois vértices que a delimitam)
  - **ID** - identificador único de uma aresta
  - **dest ∈** V - Vértice de destino da aresta

Ra - representa a distância máxima entre a quinta e local de entrega (raio de ação)

4.2. Dados de Saída

Gf = (Vf, Ef) - Grafo dirigido pesado, sendo que Vf e Ef possuem os mesmos atributos que Vi e Ei (com exceção de alguns atributos específicos relacionados ao algoritmo utilizado)

Cf - Sequência ordenada de todos os camiões usados, sendo Cf(i) o seu i-ésimo elemento[3]. Cada um tem os seguintes valores:

- **cap** - Capacidade total do camião (medida metros cúbicos)

[1] Na 1ª e 2ª Fases, a capacidade é considerada infinita (**cap** = ∞), enquanto que na 4ª Fase a capacidade é variável.

[2] Podem ser dos seguintes tipos: Vértice inicial (quinta do produtor), Vértices intermédios (Destinos dos cabazes, ou seja, a localização da casa de cada cliente), Vértice final (garagem, pode ser igual ou não ao vértice inicial) ou Vértice de Intersecção de Ruas

[3] Na 1ª Fase **Cf** apenas terá um camião.

- **P = { v ∈ Vf | 1 ≤ j ≤ |P| }** - Sequência ordenada de vértices a visitar, sendo P(j) o seu j-ésimo elemento
- **Bf** - Sequência ordenada de cabazes a entregar pelo camião (contém os mesmos atributos que Bi)
- **volumeAtual** - volume total dos cabazes que o camião carrega
- **totalPercorrido -** distância percorrida por um camião

4.3. Restrições

4.3.1. Sobre Dados de Entrada

∀ c ∈ Ci, cap > 0, dado que uma capacidade não é negativa nem nula.

∀ b ∈ Bi, weight > 0, uma vez que não existem pesos negativos e não se transportam cabazes vazios

∀ b ∈ Bi, invoice > 0. A fatura deve ser não nula e positiva

∀ b ∈ Bi, dist < Ra. Não se efectuam entregas localizadas fora do raio de ação

∀ e ∈ Ei, weight > 0, visto que os pesos das arestas representam distâncias. Não se consideram, portanto, distâncias nulas.

4.3.2. Sobre Dados de Saída

| Ci | = | Cf |

Σ| *Bf* | = | Bi | , ou seja, a soma do número de cabazes associado a cada camião é igual à quantidade total inicial

Seja wt(c) o volume total dos cabazes de um camião c. Então, ∀ c ∈ Cf, wt(c) ≤ cap. O peso dos cabazes associados a um camião não pode ser maior que a capacidade do mesmo.

4.4. Função Objetivo

O pretendido neste trabalho é minimizar o caminho percorrido pelos camiões, sendo a distância final calculada a partir do somatório da distância percorrida por todos os camiões.

$$f = \sum_{c \in Cf} (totalPercorrido)$$

A função f, será, então, a função objetivo.

5. Estrutura de Dados e Classes Utilizadas

5.1. Grafo

Para representar o grafo, utilizou-se a classe fornecida nas aulas práticas (situada no ficheiro Graph.h). Neste grafo, os apontadores para os vértices são guardados num vector vertexSet. Na secção pública desta classe, adicionou-se algumas funções necessárias para a implementação do projeto, que serão descritas posteriormente.

5.2. Vértice

Assim como para o grafo, utilizou-se para o vértice a classe fornecida nas aulas práticas (situada no ficheiro Graph.h). O ID do nó é guardado no atributo info. Foram adicionados alguns atributos auxiliares devido à implementação de certos algoritmos (como por exemplo, o atributo **heuristic** e **gValue**). Ao descrever a implementação dos algoritmos, será explicado o uso dos respetivos campos auxiliares.

5.3. Aresta

Similarmente ao grafo e ao vértice, utilizou-se, para a aresta, a classe fornecida nas aulas práticas (situada no ficheiro Graph.h). Nesta classe não se alterou nada, sendo assim idêntica à fornecida nas aulas práticas

5.4. FarmFresh2U

5.4.1. Basket

Na classe Basket, os atributos utilizados para descrever cada cabaz são o destino dest (ou seja, o índice do vértice que corresponde à morada do cliente), o idClient que é único para cada cliente, assim como o invoice, que refere ao pedido. Cada cabaz tem um peso (weight) e um volume.

5.4.2. Truck

Na classe Truck, definimos cada camião com sua capacidade volúmica cap, e seu tipo type.

5.4.3. FarmFresh2You

Esta classe contém dois atributos: o atributo **farm**, do tipo inteiro, que contém o id do vértice onde se situa a quinta, e o atributo **garage,** também do tipo inteiro, que contém o id do vértice onde se situa a garagem

5.5. Funções auxiliares

5.5.1. Função readBasketsFromFile

A função readBasketsFromFile lê de um ficheiro de nome filename os cabazes a entregar e guarda-os num vetor. Os cabazes, no ficheiro, estão formatados da seguinte forma:

```
idClient    <---- id do cliente
dest        <---- destino do cabaz
weight      <---- peso do cabaz
volume      <---- volume do cabaz
invoice     <---- fatura
:::::       <---- separador de cabazes
```

5.5.2. Função readTrucksFromFile

A função readTrucksFromFile lê de um ficheiro de nome filename os trucks pertencentes À frota da empresa e os guarda num vetor. Os camiões, no ficheiro, estão formatados da seguinte forma:

```
cap         <---- truck max capacity in cubic meters (possible values: 10.0<=cap<=50.0)
type        <---- truck type (van: cap <= 20.0 ; box truck: 20.1<=cap<=50.0)
:::::       <---- truck separator
```

5.5.3. Função readCompanyFromFile

A função readCompanyFromFile lê de um ficheiro com nome filename a companhia e retorna um objeto do tipo FarmFresh2You (descrito no capítulo 5.4.3). A companhia, no ficheiro, está formatada na seguinte forma:

```
farm    <---- id do vértice inicial (quinta)
garage  <---- id do vértice final (garagem)
```

5.5.4. readGraph

A função **readGraph** lê de um ficheiro com nome **nodesFilename** os nós e de outro ficheiro com nome **edgesFilename** e retorna um objeto do tipo **Graph<int>.** Para ler os ficheiros dos nós e das arestas, esta função recorre a duas outras funções auxiliares, designadas de **getNodesFromFile** e de **getEdgesFromFile**, que recebem o nome do ficheiro dos nós (no caso da função **getNodesFromFile)** ou das arestas (no caso da função **getEdgesFromFile).** Ambas as funções recebem também um grafo **graph** (descrito no capítulo 5.1) e retornam void. Os ficheiros dos nós e arestas são os que foram disponibilizados pelo docente da cadeira.

6. Perspectivas de Solução

Neste capítulo, apresenta-se os principais algoritmos analisados à luz dos problemas emergentes da Empresa FarmFresh2U.

6.1. Pré-processamento dos dados de entrada

O pré-processamento dos dados de entrada servem para melhorar o desempenho temporal e espacial dos algoritmos a utilizar.

6.1.1. Grafo

Deve-se eliminar arestas e vértices inacessíveis devido a condições externas (e.g. obras em vias públicas, que deixam alguns vértices inacessíveis). Verifica-se também quais os vértices localizados fora do raio de ação, removendo-os.

A seleção dos vértices a eliminar será realizado a partir da análise de conectividade no capítulo 5.6.

6.1.2. Cabazes

Como consequência do pré-processamento do grafo, removem-se todos os cabazes associados aos vértices removidos. De seguida, calcula-se a distância dos cabazes à quinta do agricultor, que será a heurística associada ao algoritmo A*.

6.2. Shortest-Path Problem (Entre dois pontos)

O problema da 1ª fase passa por encontrar a distância mínima entre 2 vértices do grafo (primeiro, entre a quinta e o ponto da entrega, e em seguida, entre o ponto de entrega e a garagem). Para este efeito, irá-se analisar alguns algoritmos como A*, Dijkstra, Bellman-Ford e Floyd-Warshall.

"Um caminho C num grafo é mínimo se não existe outro caminho que tenha a mesma origem e o mesmo término que C, mas comprimento menor que o de C. (Vale lembrar que o comprimento é o número de arcos do caminho)." (Retirado do link referente às aulas do IME-Br)

6.2.1. Algoritmo A*

O algoritmo A* foi criado como parte do projeto Shakey, um projeto que visava a construção de um robô que conseguia planear as suas próprias ações. Atualmente, este algoritmo é maioritariamente usado para descobrir o caminho mais curto entre dois objetos.

Este algoritmo seleciona o próximo vértice a visitar pela seguinte fórmula:

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

em que n é o próximo vértice a visitar, g(n) é o custo para alcançar n a partir do vértice de início (considerando o caminho seguido), e h(n) é uma heurística que estima o custo do percurso de n até ao vértice de destino. A* seleciona sempre o vértice com menor valor f.

Apesar da sua elevada eficiência temporal (no pior dos casos, O(|E|)), o presente algoritmo têm algumas desvantagens. Para além de não garantir a produção do caminho ótimo, pode requerer um pré-cálculo pesado para os valores heurísticos.

Contudo, consegue-se garantir um caminho ótimo se a heurística for admissível (é sempre menor do que o custo real para chegar ao vértice de destino) e consistente (o valor absoluto da diferença entre as heurísticas de dois vértices é menor do que o custo real para ir de um desses vértices para o outro). Por isto mesmo, e como o grafo associado a este tema é um mapa, utilizar-se-á a heurística da distância euclidiana.

Devido à sua elevada eficácia temporal, o algoritmo A* será o escolhido para lidar com este problema.

6.2.2. Dijkstra e Dijkstra-Bidirecional

O algoritmo de Dijkstra foi criado pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959 e visava solucionar o problema do caminho mais curto entre dois pontos. Em termos de aplicação, o grafo pode ser tanto dirigido quanto não dirigido, porém apenas com arestas de peso positivo. O algoritmo em si é simples e sua performance razoavelmente boa, mas possui algumas complicações vistas a seguir. Dijkstra não garante, contudo, a exatidão da solução caso haja a presença de arcos com valores negativos.

Dijkstra considera um conjunto S de potenciais menores caminhos, iniciado com um vértice inicial Vi. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a Vi e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por Vi estejam em S. Arestas que ligam vértices já pertencentes a S são desconsideradas.

Inicialmente, todos os pesos das arestas entre o vértice de saída e o vértice em questão v assumem, como uma espécie de pior caso, valor infinito ($d[v] \leftarrow \infty$), mas que serão atualizados à medida que são analisados pelo algoritmo. Esses valores ficam armazenados em uma estrutura, neste caso um vetor, de distâncias. A denotação $\pi[v]$ é o vértice de saída, ou seja, a origem do caminho até v, de forma a criar o caminho mínimo. Também conhecido como predecessor, o mesmo possui, no primeiro passo, o valor -1.

Depois, considera-se um novo conjunto Q (que pode utilizar a estrutura Heap-Min ou Lista) de vértices que ainda não foram atualizados com o "custo" do caminho determinado

até então. A partir disso, realiza-se o processo de 'relaxamento' das arestas todas, a cada iteração, enquanto Q não estiver vazio.

Para tanto, é necessária uma função extract_min(Q) que recebe Q, extrai e devolve o vértice u com menor valor d[u]. Então, em um ciclo, para cada vértice v adjacente a u, há duas condições a serem consideradas para o próximo passo, ambas retiradas do princípio matemático da Desigualdade Triangular de Euclides.

Se d[v] for maior que d[u] + peso(u,v), então o valor de d[v] deve ser atualizado com o resultado da soma anterior, onde peso(u,v) é o peso da aresta entre u e v. Deve-se, também, atualizar o predecessor de v, que agora será u ($\pi[v] \leftarrow u$). No final deste algoritmo, obtém-se, em S, o menor caminho entre Vi e qualquer outro vértice v do grafo.

O tempo computacional O(|E| + |V|*log|V|) onde |E| é o número de arestas e |V| é o número de vértices, caso seja usado um heap de Fibonacci. Se considerarmos a utilização de um heap binário, o tempo passa a O(|E|*log|V|) e $O(|V|^2)$ caso seja usado um vetor para armazenar Q. Dijkstra é um exemplo de algoritmo greedy que gera a solução ótima em tempo polinomial.

Dijkstra-Bidirecional, por outro lado, procura simultaneamente a partir de Vi e de Vf, na direção oposta, ou seja, o processamento pára quando as duas procuras encontram-se no meio, gerando assim o caminho mais curto em potencial, a ser verificado. Geralmente é utilizada uma Binary Heap com implementação com fila de prioridade, reduzindo a complexidade de Dijkstra pela metade.

Uma limitação destes algoritmos é não conseguir encontrar o menor caminho em um grafo com pesos negativos. Para esse propósito, pode-se usar os algoritmos citados posteriormente, como Floyd-Warshall, ou Bellman-Ford.

No caso da 1ª Fase do problema, não utilizaremos Dijkstra, por apresentar uma pior performance em termos de complexidade temporal do que o A*.

6.2.3. Bellman-Ford

Bellman-Ford foi primeiramente proposto por Shimbel em 1955, mas foi nomeado em homenagem a Richard Bellman e Lester Ford Jr, que publicaram o algoritmo em 1958 e 1956, respectivamente. Edward F. Moore publicou o mesmo em 1957, portanto o algoritmo pode eventualmente ser chamado de Bellman-Ford-Moore.

O Algoritmo de Bellman-Ford busca o caminho mínimo em um grafo dirigido e cujas arestas têm peso, inclusive negativo. O Algoritmo de Dijkstra resolve o mesmo problema, num tempo menor, porém exige que todas as arestas tenham pesos positivos. Portanto, o algoritmo de Bellman-Ford é normalmente usado apenas quando o grafo possui arestas de peso negativo.

Assim como o algoritmo de Dijkstra, o algoritmo de Bellman-Ford utiliza a técnica de relaxamento, ou seja, realiza sucessivas aproximações das distâncias até finalmente chegar na solução. A principal diferença entre Dijkstra e Bellman-Ford é que no algoritmo de Dijkstra é geralmente utilizada uma fila de prioridades para selecionar os vértices a serem relaxados, enquanto que o algoritmo de Bellman-Ford simplesmente relaxa todas as arestas.

Primeiramente, o algoritmo inicializa, para todos os vértices pertencentes ao grafo G, o valor do peso de cada aresta a infinito ($d[v] \leftarrow \infty$) e o predecessor de cada v como nulo ($\pi[v] \leftarrow null$), seguido da atualização do peso do vértice inicial Vi a zero ($d[Vi] \leftarrow 0$). Depois, há dois ciclos intrínsecos, sendo o primeiro com $|V|-1$ iterações e o segundo que percorre todas as arestas (vértice anterior u, vértice atual v) pertencentes ao conjunto E. A ação a ser tomada depende da condição "$d[v] > d[u] + peso(u,v)$", obtida através da Desigualdade de Triângulos. Caso se mostre verdadeira a condição, realiza-se o relaxamento, ou seja, atualiza-se o valor de $d[v]$ para $d[u] + peso(u,v)$ e $\pi[v]$ para u.

Outro aspecto que destaca o algoritmo de Bellman-Ford é a capacidade de detectar se há um ciclo negativo no grafo, pois caso haja, não é possível determinar o caminho mais curto. Este algoritmo não necessita de receber um vértice final pois o mesmo calcula o caminho mais curto para todos os vértices alcançáveis a partir de Vi. Bellman-Ford propõe resolução nos casos em que pode ser escolhida uma má ordem de relaxamentos e leva o tempo computacional a exponencial. Enquanto Dijkstra é greedy e seleciona o vértice mais próximo, Bellman-Ford relaxa todos os vértices, sendo esta a sua maior contribuição.

Bellman-Ford executa $|E|*|E|$ relaxamentos para cada iteração, sendo que acontecem $|V| - 1$ iterações. Portanto, o pior caso possível é que o mesmo tenha tempo computacional de $O(|V|*|E|)$.

Pode ser feito, porém em menos iterações. Para alguns grafos, apenas uma iteração é necessária, seja este o melhor caso possível, em tempo de $O(|E|)$. Isso apenas pode acontecer em situações, por exemplo, onde cada vértice apenas se conecta a um próximo, de forma linear.

Para resolução do problema apresentado neste trabalho não há possibilidade de alcançar esta solução ideal, já que não se garante que os grafos sejam lineares. Além disso, não se utilizarão grafos com pesos negativos aqui, então o algoritmo Bellman-Ford acaba por não ser a melhor opção.

6.2.4. Floyd-Warshall

O algoritmo de Floyd-Warshall, publicado por Robert Floyd em 1962, utiliza programação dinâmica e tem como objetivo calcular as distâncias mínimas entre todos os pares de vértices de um grafo dirigido.

Este algoritmo recebe como entrada uma matriz de adjacência que representa um grafo G. Necessita-se de utilizar duas matrizes: a matriz de distâncias mínimas D[i, j] e a

matriz de predecessores P[i, j]. Inicializa-se a matriz D igual à matriz de pesos das arestas (representando as arestas não existentes por infinito). De seguida, num ciclo de um até o número de vértices, efectua-se o seguinte relaxamento: D[i, j](k) = min(D[i, j](k - 1), D[i, k](k - 1) + D[k, j](k - 1)). Em paralelo, atualiza-se a matriz P com o vértice escolhido.

Tem uma complexidade temporal igual a $O(|V|^3)$. Logo, é completamente dependente do número de vértices do grafo. Com isto, o dado algoritmo é mais indicado para grafos densos porque não depende do número de arestas. Se o número de arestas do grafo for baixo, este algoritmo perde o seu maior benefício. Como no algoritmo de Bellman-Ford, este algoritmo também pode ter arestas com pesos negativos, mas é menos eficiente.

6.3. Travelling Salesman Problem (Caminho mais curto entre dois pontos com várias paragens)

O problema da 2ª fase passa por encontrar o percurso mínimo entre vários vértices do grafo. O percurso tem que começar na quinta do produtor, passar por todos os pontos de entrega e acabar na garagem.

Há, aqui, um problema semelhante ao do caixeiro-viajante (Travelling Salesman Problem/TSP) cuja complexidade é elevada. No decorrer da resolução deste problema, é possível alcançar soluções aproximadas ou exatas, a depender de qual algoritmo for utilizado.

6.3.1. Algoritmo Held–Karp

Um dos algoritmos que se destaca é algoritmo de Held-Karp, proposto em 1962 por Richard E. Bellman, Held e Richard M. Karp, que usa programação dinámica.

O TSP goza da seguinte propriedade :

> "Every subpath of a path of minimum distance is itself of minimum distance."

O algoritmo calcula todas as soluções para os subproblemas e guarda-as. Assim, para fazer o cálculo de problemas de ordem superior, podemos utilizar os valores guardados. Este algoritmo baseia-se intrinsecamente em programação dinâmica.

Held-Karp tem uma complexidade temporal exponencial igual a $O(2^n n^2)$ e uma complexidade espacial igual a $O(2^n n)$.

n	1	2	3	4	5	6	7	8
$2^n n^2$	2	16	72	256	800	2304	6272	16384
n!	1	2	6	24	120	720	5040	40320

Tabela 1 - Comparação entre as complexidades temporais do algoritmo de Held-Karp e do bruteforce.

Tal como podemos observar na tabela, a complexidade temporal do algoritmo de Held-Karp só é menor (mais eficiente) que a complexidade temporal do bruteforce para $n \geq 8$.

Apesar do mesmo possuir complexidades temporal e espacial exponenciais, utilizaremos este algoritmo por produzir sempre uma solução ótima.

6.3.2. Utilizando Árvore de Expansão Mínima (MST)

Um dos algoritmos que obtém soluções aproximadas é usando o algoritmo Minimum Spanning Tree de Prim. Foi desenvolvido em 1930 por Vojtěch Jarník e mais tarde publicado por Robert C. Prim em 1957 e por Edsger W. Dijkstra em 1959.

Este algoritmo apenas funciona se a instância do problema satisfizer a desigualdade triangular (o caminho mais curto para se chegar a um vértice j através de outro vértice i é diretamente de j a i, e não por um outro vértice k. Matematicamente, dist(i, j) < dist(i, k) + dist(k, j)).

No presente projecto verifica-se esta desigualdade, o caminho de um ponto a outro (ambos os pontos dentro da cidade) é mais curto se for direto. Se for necessário passar num ponto intermédio, o novo caminho será maior do que o anterior.

O algoritmo consiste em construir uma árvore de expansão mínima de Prim com origem no ponto inicial. De seguida, listam-se os vértices visitados em pré-ordem. Para finalizar basta adicionar o ponto final.

Este algoritmo de Prim tem uma complexidade temporal igual a $O(|V|^2)$ se for utilizado com uma matriz de adjacência, ou seja, escala quadraticamente de acordo com o número de vértices do grafo.

Como este algoritmo não produz uma solução ótima, será utilizada o algoritmo de Held-Karpesta, apesar desta última ter uma complexidade temporal maior que MST. É necessário ter em consideração que o problema é NP-hard, ou seja, não se conhece um algoritmo que produz uma solução ótima em tempo polinomial.

6.4. Vehicle Routing Problem (Caminho mais curto entre dois pontos com múltiplas paradas e com mais de um camião)

O problema da 3ª fase tem como objetivo resolver a questão "Qual é o conjunto de percursos ótimo para várias carrinhas conseguirem entregar todos os cabazes?"

Tem-se, portanto, um VRP (Vehicle Routing Problem). O VRP é NP-hard (non-deterministic polynomial-time hardness), logo o número de problemas que podem ser resolvidos usando programação é limitado.

Concebeu-se teoricamente um algoritmo a fim de resolver o referido problema.

6.4.1. Algoritmo criado

Ao introduzir vários camiões ao problema, surge-se outro: Como se garante a distância mínima percorrida por todos os camiões?

Depois de muita consideração, decidiu-se resolver o problema da seguinte maneira: todos os cabazes serão distribuídos pelos camiões, a fim de minimizar o número de camiões usados. Assim, o problema transforma-se no TSP, já discutido e solucionado anteriormente.

6.5. Análise da conectividade

A análise de conectividade de um grafo é extremamente importante, uma vez que permite verificar se este é conexo e resistente a falhas (ou seja, se continua conexo ao retirar arestas). Em relação ao grafo deste tema, é necessário verificar qual a componente conexa a que o vértice de partida pertence (uma vez que num mapa rodoviário é relativamente fácil de ruas tornarem-se inacessíveis devido a obras, etc). Esta análise permite efectuar o pré-processamento do grafo (por outras palavras, irá se eliminar os vértices que não pertençam à componente conexa do vértice de partida)

Para este efeito, irá se considerar três algoritmos: o algoritmo de Kosaraju, o algoritmo de Tarjan e uma simples pesquisa em profundidade

6.5.1. Algoritmo de Kosaraju

O algoritmo de Kosaraju foi desenvolvido por S. Rao Kosaraju em 1978 mas só foi publicado em 1981 por Micha Sharir.

Este algoritmo começa por realizar uma pesquisa em profundidade, numerando os vértices em pós-ordem. De seguida, efectua-se a transposição do grafo (ou seja, inverte-se o sentido das arestas). Por último, volta-se a fazer uma pesquisa em profundidade, começando

sempre pelo vértice de numeração mais alta ainda não visitado, e acabando quando todos os vértices forem visitados. Cada árvore obtida corresponde a um componente fortemente conexo do grafo.

Tem uma complexidade temporal de O(|V| + |E|). Contudo, na prática, é menos eficiente do que o algoritmo de Tarjan, que só necessita realizar uma pesquisa de profundidade.

6.5.2. Algoritmo de Tarjan

O algoritmo de Tarjan, criado por Robert Tarjan, é utilizado para descobrir os componentes fortemente conexos de um grafo dirigido.

O presente algoritmo faz uma pesquisa em profundidade, numerando os vértices em pré-ordem. Concorrentemente a esta pesquisa, atribui a cada vértice um valor low value, que representa o vértice com menor número que se atinge com zero ou mais arestas e, no máximo, com uma aresta de retorno. No final da pesquisa, os vértices que possuem igual low value pertencem ao mesmo componente fortemente conexo do grafo.

Este algoritmo tem complexidade linear, O(|V| + |E|).

6.5.3. Pesquisa em profundidade (Depth-First algorithm)

O algoritmo de pesquisa em profundidade foi criado por Charles Pierre Trémaux, matemático francês do século XIX.

Este algoritmo começa por escolher um vértice aleatório. De seguida, verifica qual o próximo vértice adjacente não visitado e visita-o. Depois, continua neste loop até não haver mais vértices adjacentes não visitados, ponto em que retorna para o vértice anteriormente visitado, e repete o processo para os outros vértices adjacentes não visitados. Finaliza quando já não existirem mais vértices a visitar.

Este algoritmo tem uma complexidade média de O(|V| +|E|). Por esta razão, e pelo facto de ser mais simples, será este o algoritmo a implementar.

7. Identificação de Casos de Utilização e Funcionalidades

7.1. Menus Interativos

A fim de existir uma boa interação com o utilizador, implementou-se menus interativos. Estes menus permitem, quando necessário, que o utilizador introduza dados, que serão de seguida processados pelo programa.

7.2. Rotina de Setup

Antes de entrar nos menus interativos, o programa pergunta ao utilizador qual o nome do ficheiro da companhia, para a importar. De seguida, pede pelos ficheiros dos vértices e arestas, com o propósito de importar o grafo pretendido.

7.3. Menu Principal

7.3.1. Mostrar grafo (utilizando o GraphViewer)

Este menu tem a finalidade de mostrar ao utilizador o grafo que importou

7.3.2. Calcular primeira fase

Este menu começa por pedir o ficheiro onde estão guardados os cabazes. De seguida, pede ao utilizador o nome do ficheiro para onde quer exportar os resultados. Com esta informação, realiza o algoritmo da primeira fase.

7.3.3. Calcular segunda fase

Este menu é muito semelhante ao anterior. Primeiro pede ao utilizador o nome do ficheiro onde estão guardados os cabazes. Depois pede o nome ficheiro para onde irão ser exportados os resultados e finalmente executa o algoritmo associado à segunda fase.

7.3.4. Calcular terceira fase

Este menu apenas permite correr a função referente à terceira fase, ou seja, de preencher os camiões com os cabazes e encontrar o menor caminho para cada camião. Primeiro, o programa solicita ao utilizador o ficheiro onde estão guardados os cabazes, depois onde estão guardados os camiões, o ficheiro para o qual gostaria que fossem enviados os resultados e por último executa o algoritmo referente à terceira fase.

8. Algoritmos Implementados

8.1. Análise da Conectividade

8.1.1. Pseudo-código

```
Vertice -> vértice de começo


dfsVisit(Vertice):
    SetVisited(Vertice);
    For (neighbor vertices of Vertice):
        dfsVisit(neighbor);
```

```
idVertice -> id do vértice de começo
Vertice -> vértice de começo

dfsRemoveUnvisited(idVertice):

   Vertice = InitializeAllVertices(idVertice);

   dfsVisit(Vertice);

   EraseAllUnvisitedVertices();

   EraseAllEdgesAssociated();
```

Figuras 8.1 e 8.2: Algoritmo DFS e algoritmo de análise de conectividade

8.1.2 Análise teórica da complexidade

O algoritmo representado pela função **dfsVisit** faz uma pesquisa em profundidade num vértice. Primeiro, põe o vértice como visitado. De seguida, para todos os vértices vizinhos não visitados, chama-se recursivamente. Este algoritmo tem uma complexidade temporal de O(|V| + |E|).

O seguinte algoritmo, representado pela função **dfsRemoveUnvisited,** começa por inicializar todos os vértices (complexidade de O(|V|)). De seguida, realiza uma pesquisa em profundidade no vértice que recebe como argumento(O(|V| + |E|). Finalmente, apaga todos os vértices não visitados e todas as arestas que tinham como vértice de destino um vértice não visitado(O(|V| + |V|*|E|)). Logo, a complexidade deste algoritmo será de O(|V| + |E| + |V|*|E|).

8.1.3 Análise empírica da complexidade

8.2. 1ª Fase

8.3.1 Pseudo-código

```
farm -> Objeto FarmFresh2You que contem a quinta (idFarm) e a garagem (idGarage)
graph -> Grafo
filename -> nome do ficheiro de onde lê os cabazes
resultFilename -> nome do ficheiro para onde exporta o resultado
baskets -> vetor de baskets com a localização da entrega (idDelivery)

aStarAlgorithm(farm, graph, filename, resultFilename):

    baskets = readBasketsFromFile(filename);

    aStarAlgorithmGraph(graph, idFarm, idDelivery);

    exportResultsToFile(graph, resultFilename, idFarm, idDelivery);

    aStarAlgorithmGraph(graph, idDelivery, idGarage);

    exportResultsToFile(graph, resultFilename, idDelivery, idGarage);
```

```
id1 -> id do vertice de origem
id2 -> id do vertice de chegada
vertice1 -> vertice de id1
vertice2 -> vertice de id2

aStarAlgorithmGraph(id1, id2):

    InitializingRoutine();

    Queue Q;

    INSERT-QUEUE(Q, vertice1)

    while (Queue not empty):
        vertice <- EXTRACT-MIN(Qu);
        setVerticeVisited(vertice);

        if (vertice equals vertice2)
            END;

        for(neighborVertice of Vertice):

            newGValue = calculateGValue(neighborVertice);

            if(newGValue < gValue(neighborVertice))

                updateValues(neighborVertice)

                if(neighborVertice not visited)
                    INSERT-QUEUE(Q, neighborVertice);
                else
                    DECREASE-KEY(Q, neighborVertice);
```

Figura 8.3 e 8.4: Algoritmo desenvolvido pelo grupo e algoritmo A* implementado

8.3.2 Análise teórica da complexidade

Analisando a primeira função, verifica-se que começa com uma chamada à função **readBasketsFromFile.** Esta função tem complexidade O(n), em que n é o número de cabazes. Como neste caso só existe um cabaz, a função realizar-se-á em tempo constante.

A segunda função chamada, a **aStarAlgorithmGraph** é uma simples implementação do algoritmo A*. Este algoritmo começa por inicializar todos os vértices com a função **InitializingRoutine,** que tem uma complexidade O(|V|). De seguida, cria uma queue e insere o vértice de origem. Na implementação feita, é usada uma Fibonacci Heap. Depois, enquanto a queue não estiver vazia, extrai o mínimo desta (na Fibonacci Heap corresponde a uma operação com complexidade O(log n)) e atualiza os vértices vizinhos, adicionando-os à queue se lá não estiverem (O(1)), ou atualizando a chave (O(1)). Se ao extrair da queue um vértice, esse for o destino, o algoritmo acaba. No pior caso, este algoritmo tem complexidade O(|V| + |E|*log n), uma vez que o pior dos casos será quando percorre todas as arestas, realizando o

ciclo while |E| vezes, e, a cada iteração deste ciclo, realiza a operação Extract-Queue (O(log n)). A soma com o |V| vem do facto de se realizar sempre a rotina inicial.

A terceira função chamada é a **exportResultsToFile.** Esta função percorre os vértices visitados, exportando os seus índices para um ficheiro, logo, terá a mesma complexidade[4] do **aStarAlgorithmGraph,** ou seja, O(|E|).

Finalmente, somando as componentes todas, o algoritmo tem complexidade igual a O(|V| + |E|*log n), no pior dos casos.

8.3.3 Análise empírica

Como podemos verificar pelo gráfico, o aumento do tamanho do grafo (e por consequente, das arestas) implica o aumento do tempo de execução.

[4] Esta função utiliza a função reverse do vector, mas como esta tem complexidade linear ao número de elementos, logo não se considera esta função no cálculo da complexidade

8.3. 2ª Fase

8.3.1. Pseudocódigo

```
farm -> Objeto FarmFresh2You que contém a quinta (idFarm) e a garagem (idGarage)
graph -> Grafo
filename -> Nome do ficheiro de onde lê os cabazes
resultsFilename -> Nome do ficheiro para onde exporta os resultados
baskets -> Vetor de baskets com a localização da entrega (idDelivery)
route -> Vetor com o percurso final (apenas contém os pontos relativos à farm, cabazes ou garage)
cabazDest -> Destino de um cabaz
dist -> Destância de um subpercurso

heldKapAlgorithm(farm, graph, filename, resultsFilename):
    start = idFarm;
    end = idGarage;

    baskets = readBasketsFromFile(filename);

    heldKarpCore(start, end, baskets, graph, resultFilename);
```

```
heldKarpCore(startVertex, endVertex, baskets, graph, resultFilename):
     Para cada cabaz:
          Para cada cabaz restante em baskets:
               startVertex = Ultimo elemento de route;
               aStarAlgorithmGraph(startVertex, cabaz);
               dist = calculatePathSize(startVertex, cabaz);
               Se este sub-percurso for o mais pequeno:
                    Adicionar cabazDest ao vetor route;
                    Eliminar este basket do vetor baskets;
                    exportResultsToFile(resultFilename, startVertex, cabazDest);
     startVertex = Ultimo elemento de route;
     aStarAlgorithmGraph(startVertex, endVertex);
     dist = calculatePathSize(startVertex, endVertex);
     Adicionar garagem ao vetor route;
     exportResultsToFile(resultFilename, startVertex, endVertex);
```

Figura 8.5 e 8.6: Algoritmo desenvolvido pelo grupo e algoritmo de Held-Karp implementado

8.3.2. Análise teórica da complexidade

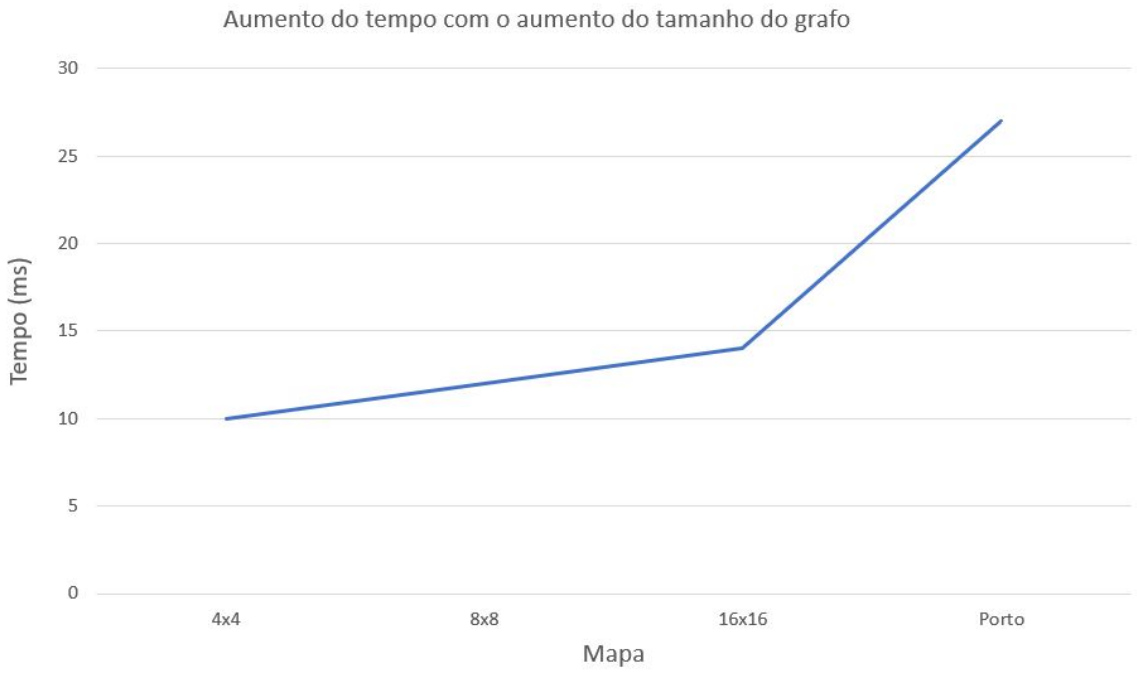
Analisando a primeira função, verifica-se que começa com uma chamada à função **readBasketsFromFile.** Tal como indicado anteriormente, esta função tem uma complexidade O(n) que será dependente do número de cabazes.

De seguida, é realizada a chamada à função **aStarAlgorithm** que tem complexidade O(|V| + |E| * log n) e uma chamada à função **exportResultsToFile** com complexidade O(|E|), explicadas anteriormente. Como podemos observar, esta chamada será repetida n * (n + 1) / 2 vezes. Logo a complexidade temporal do ciclo será O((n * (n + 1) / 2) * |V| + |E| * log n).

Depois, é realizada uma última chamada à função **aStarAlgorithm** e uma chamada à função **exportResultsToFile**.

Concluindo, somando as componentes todas, o algoritmo tem complexidade igual a O((n * (n + 1) / 2) * |V| + |E| * log n), no pior dos casos.

8.3.3. Análise empírica



Ao observar o gráfico acima apresentado conseguimos verificar que o tempo de execução do algoritmo implementado aumenta com o número de nós. É menor num grafo 4x4 e maior num grafo representante do Porto com muitos mais nós.

8.4. 3ª Fase

8.4.1. Pseudo-código

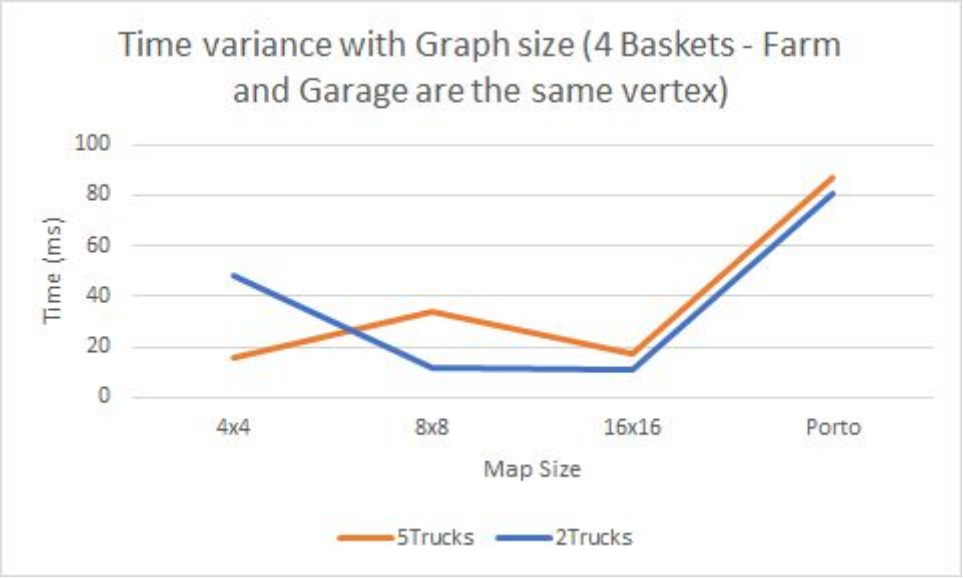
```
farm -> Objeto FarmFresh2You que contem a quinta (idFarm) e a garagem (idGarage)
graph -> Grafo
filenameB -> nome do ficheiro de onde são lidos os cabazes
filenameT -> nome do ficheiro de onde são lidos os camiões existentes na frota da empresa
resultFilename -> nome do ficheiro para onde exporta o resultado

thirdPhaseAlgorithm(farm, graph, filenameB, filenameT, resultFilename)
    baskets = readBasketsFromFile(filenameB);
    trucks = readTrucksFromFile(filenameT);
    currentCapacity = 0;

    for (truck of trucks):
        maxCap = truck.getCap();
        while (baskets != empty & currentCapacity < maxCap) {
            if( maxCap > (currentCapacity + basket.getVolume())){
                currentCapacity += basket.getVolume();
                truck.addBasket(basket);
                baskets.erase(basket);
            }
        }
        heldKarp(farm, garage, truck.BasketsToDeliver, graph, resultFilename);
        currentCapacity = 0;
    }
}
```

8.4.2. Análise teórica da Complexidade

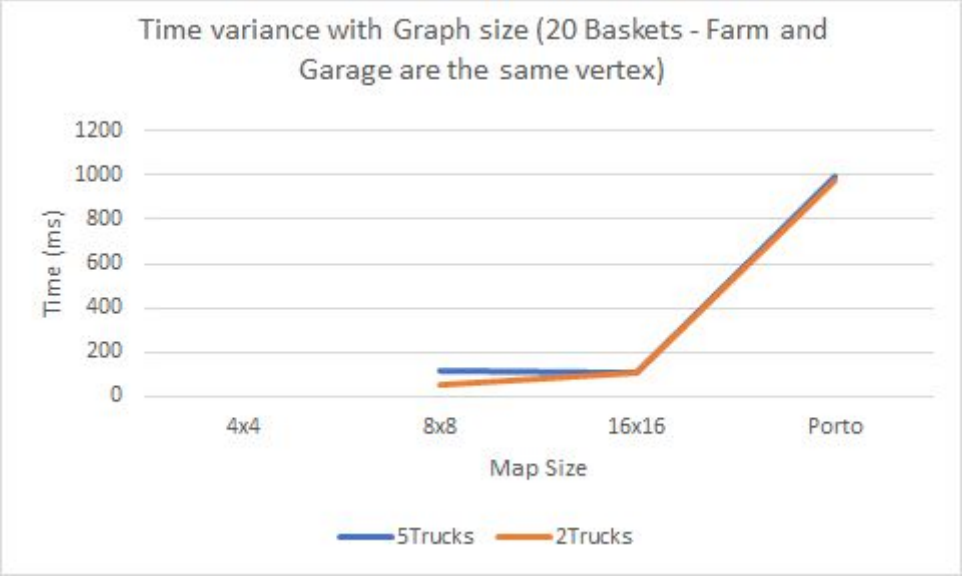
A terceira fase implementa o algoritmo Held-Karp desenvolvido na 2ª fase, cuja complexidade temporal é exponencial e igual a $O(2^{|V|} * |V|)$ a fim de buscar o percurso mais curto para cada camião que acabou de ser preenchido com baskets. Como em cada iteração do vetor de camiões, é adicionada uma certa quantidade de baskets ao vetor-atributo basketsToDeliver, adicionamos à complexidade total O(|T|*|B|), cujos valores de |T| e |B| ficam a depender tanto da quantidade de cabazes que caberá em um determinado camião, quanto do total de camiões utilizados pela empresa. Sendo o pior caso a utilização de todos os camiões para realizar as entregas e, portanto, totalizando $O(2^{|V|} * |V| + |T| * |B|)$.

8.4.3. Análise empírica

É possível perceber no gráfico abaixo que apenas quando a dimensão do grafo aumenta consideravelmente é que percebemos um crescimento grande no tempo de execução. Pode-se perceber também que quando temos uma quantidade pequena de baskets a serem entregues, ter 2 ou 5 trucks muda o comportamento quando o grafo muda pouco de dimensão.



Porém, quando aumentamos consideravelmente o número de cabazes, a diferença entre o tempo de execução não depende quase nada da quantidade de camiões. Apesar disso, o tempo triplica ao início e sobe bastante ao utilizarmos o grafo da cidade do Porto.



9. Conclusão

A criação de um sistema de gestão de encomendas para a empresa fictícia FarmFresh2U foi dividida no presente projeto em 3 fases, a fim de fasear o processo de busca por soluções computacionais e eficazes tanto em termos temporais quanto espaciais.

Outra estratégia utilizada foi identificar os problemas presentes nas fases que podem ser resolvidos com algoritmia. Os principais foram:

- Shortest-Path Problem (SP), que se caracteriza pela procura do caminho mais curto entre dois pontos;
- Travelling Salesman Problem (TSP), que também busca o caminho mais curto entre dois pontos, porém com várias paragens no plano de viagem;
- Vehicle Routing Problem (VRP), que além de solucionar o anterior, leva em consideração o fato de haver mais de um camião para realizar as encomendas;
- Conectividade, onde foi analisada a questões relacionadas ao pré-processamento dos grafos e remoção daqueles que não foram considerados como parte da área fortemente conexa do grafo.

Na segunda parte deste relatório, foi apresentada a implementação dos algoritmos propostos.

De acordo com a análise feita nos capítulos anteriores, a solução para o SP foi feita com o algoritmo A*, devido à sua elevada eficácia temporal. Floyd Warshall e Bellman-Ford, apesar de aceitarem arestas com pesos negativos em seu processamento, mostram-se ser mais lentos e de complexidade de tempo elevada. Mesma coisa acontece ao utilizar Dijkstra/Dijkstra-Bidireccional, portanto, decidiu-se portanto utilizar A* para encontrar o caminho mais curto entre dois pontos na primeira fase.

Já o TSP foi tratado com o algoritmo Held-Karp, que apesar de não ter a menor complexidade temporal, encontra sempre uma solução ótima.

Quanto ao VRP, utilizamos um algoritmo criado pelos autores deste artigo, visando encontrar caminhos mínimos para todos os camiões e ao mesmo tempo otimizando o preenchimento dos mesmos sem exceder as capacidades.

No último problema, para analisar a conectividade do grafo e ainda percorrer o mesmo a fim de realizar o pré-processamento de dados, foi realizada uma Depth-First Search aplicada no início do programa logo após a importação do grafo.

Durante o presente trabalho, a equipa pôde aplicar conhecimentos adquiridos na disciplina Concepção de Algoritmos mas também a partir de pesquisas próprias, tais como recursão, dynamic programming, greedy algorithms, divide and conquer strategies e até mesmo bruteforce.

Os membros da equipa aplicaram um esforço semelhante no decorrer do projecto cujas partes foram principalmente elaboradas segundo a seguinte divisão:

- **Deborah Lago:**

Formatação do relatório, Capa, Lista de Símbolos e Definições, Introdução, Função Objetivo, Conclusão, Pesquisa dos algoritmos Bellman-Ford/Dijkstra/Dijkstra-Bidirecional, implementação de um algoritmo para resolução da 3ª fase e respectivas análises empírica e de complexidade, implementação da Depth-First Search, função readTrucksFromFile.

- **João Paulo Silva da Rocha:**

Identificação do Problema, Formalização do Problema, Pesquisa dos algoritmos Floyd-Warshall/Held-Karp/Utilizando Árvore de Expansão Mínima, Implementação de um algoritmo para a resolução da 2ª fase e as respetivas análises empírica e de complexidade, implementação da função calculatePathSize.

- **Ricardo Amaral Nunes:**

Pesquisa dos algoritmos A*/Kosaraju/Tarjan e análise da 1ª fase, implementação do Algoritmo A*, 1ª fase, estrutura inicial do código, estrutura da função de medição de tempo, análise empírica e de complexidade da 1ª fase, análise empírica e de complexidade da DFS, estrutura dos menus interativos, função readBasketsFromFile.

10. Referências

Artigos:
T. H. Cormen, Algorithms Unlocked, MIT Press, Cambridge, 2013
O. Dustin, Comparative Analysis of the Algorithms for Pathfinding in GPS Systems, The Fourteenth International Conference on Networks, Poland, 2015

Youtube:
Vídeos do canal MIT OpenCourseWare - MIT 6.006 Introduction to Algorithms, Fall 2011
https://www.youtube.com/watch?v=-JjA4BLQyqE

GeeksforGeeks:
https://www.geeksforgeeks.org/strongly-connected-components/
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/?ref=lbp

Wikipedia:
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
https://en.wikipedia.org/wiki/Multi-fragment_algorithm
https://en.wikipedia.org/wiki/Vehicle_routing_problem
https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

Outros sites:
https://www.ime.usp.br/~pf/algoritmos_para_grafos/
https://brilliant.org/wiki/bellman-ford-algorithm/
https://brilliant.org/wiki/floyd-warshall-algorithm/
https://www.rand.org/pubs/papers/P510.html