Faculdade de Engenharia da Universidade do Porto

Mestrado Integrado em Engenharia Informática e Computação

Unidade Curricular – Concepção e Análise de Algoritmos

E-Stafetas: Transportes de mercadorias em veículos elétricos (Tema 7)

Daniel Filipe Souto Félix – **up201905189**
Miguel Boaventura Rodrigues – **up201906042**
Tiago Caldas da Silva – **up201906045**

2020/2021

Índice

Índice de figuras

Descrição do problema

Uma empresa de entrega de mercadorias ao domicílio decidiu investir em veículos elétricos para realizar as suas entregas. Deste modo, é necessário gerir diariamente a sua frota de recolha/entrega de produtos de forma fazer uma utilização mais eficiente possível da mesma.

Como os veículos são elétricos, têm autonomia limitada, pelo que pode ser necessário efetuar uma recarga a meio da sua rota. Pela rota existem vários pontos de recarga (incluindo a garagem de onde partem), e considerando que os veículos partem totalmente carregados - com a máxima autonomia, é necessário distribuir as várias encomendas pelos diversos veículos que a empresa dispõe, sempre tendo em atenção que estes veículos possuem uma capacidade máxima.

Assim, podemos dividir o problema em cinco partes distintas:

1. Efetuar apenas uma recolha/entrega de uma encomenda por apenas um veículo, considerando que o veículo possui uma autonomia ilimitada e capacidade ilimitada;

2. Efetuar apenas uma recolha/entrega de uma encomenda por apenas um veículo com uma autonomia limitada e capacidade ilimitada;

3. Efetuar todas as recolhas/entregas de um dia com apenas um veículo com autonomia limitada e capacidade ilimitada;

4. Efetuar todas as recolhas/entregas de um dia com apenas um veículo com autonomia e capacidade limitadas;

5. Efetuar todas as recolhas/entregas diárias com mais do que um veículo (com autonomia e capacidade limitadas).

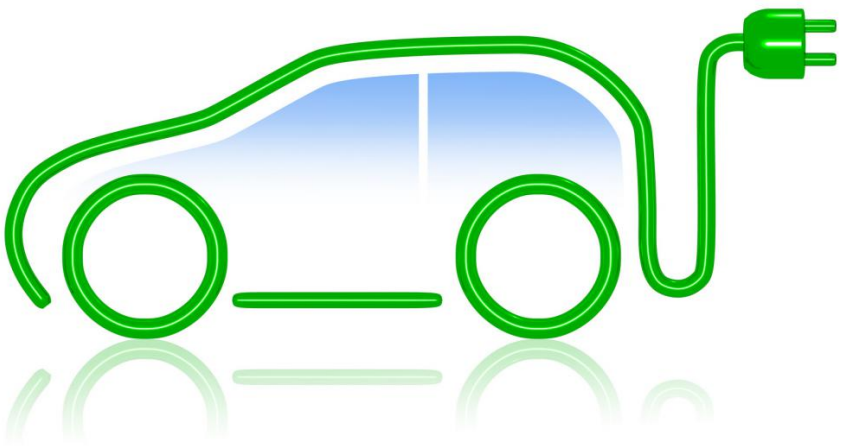


Fig.1 - Imagem ilustrativa do tema

Divisão do problema em subproblemas

1. **Efetuar apenas uma recolha/entrega de uma encomenda por apenas um veículo, considerando que o veículo possui uma autonomia ilimitada e capacidade ilimitada.**

Um veículo, que parte da garagem (com a autonomia máxima), completa a sua tarefa assim que regressar ao ponto de partida depois de recolher e posteriormente entregar uma encomenda.

Assim, dada a complexidade do caminho mais curto de uma tarefa ter de considerar as distâncias de recolha e de uma posterior entrega da encomenda, numa primeira fase, procuramos encontrar qual a rota mais eficiente que um veículo pode tomar.

Deste modo, o caso base do nosso problema passa por tentar encontrar esta solução sem ter em atenção a autonomia e capacidade do veículo. Assim existe apenas a preocupação sobre a procura do caminho mais curto que seja possível de efetuar - considerando todas as adversidades que possam existir na rota - entre os pontos:

Garagem → Local de recolha → Local de entrega → Garagem

Para este cálculo, que será, também, efetuado nas restantes subpartes do problema, é importante considerar a possibilidade de completar a tarefa, por outras palavras, apenas se calcula o caminho mais curto se existem caminhos que liguem os pontos da rota, dois a dois (efetuando-se um estudo prévio à conectividade do grafo).

2. **Efetuar apenas uma recolha/entrega de uma encomenda por apenas um veículo com uma autonomia limitada e capacidade ilimitada**

Uma vez que já descobrimos qual o caminho mais eficiente para uma determinada tarefa, o próximo passo será ter em consideração a autonomia do veículo.

Deste modo, terão de ser acrescentados ao nosso mapa, locais para uma eventual recarga (incluindo a garagem de onde os veículos partem). Assim, o cálculo para o caminho mais curto terá de ter em consideração possíveis desvios para o veículo recarregar.

A autonomia do veículo, dita por quais arestas este pode passar, isto é, a autonomia do veículo é reavaliada de acordo com o peso da aresta pelo qual o veículo acabou de passar. Um veículo fica sem carga assim que a sua autonomia atinge um valor nulo, ficando impedido de recolher ou entregar mais encomendas. Assim, o principal objetivo desta etapa passa por completar uma tarefa evitando que o veículo fique descarregado - utilizando os pontos de recarga sempre que seja possível e necessário. No caso de um veículo ser incapaz de completar uma determinada tarefa devido à sua autonomia, essa encomenda ficará de fora da rota.

3. **Efetuar todas as recolhas/entregas de um dia com apenas um veículo com autonomia limitada e capacidade ilimitada**

Tendo solucionado os subproblemas mais básicos, cabe-nos agora generalizá-los para um dos objetivos da empresa: realizar a recolha e a respetiva entrega de todas as encomendas num determinado dia.

Deste modo, vamos começar por considerar que a empresa apenas dispõe de um veículo com autonomia limitada e com capacidade ilimitada para efetuar todas as suas entregas. Assim, teremos de ter o cuidado de verificar quais os caminhos mais viáveis que o veículo pode tomar, considerando que para entregar uma determinada encomenda, tem sempre de recolhê-la previamente (podendo recolher várias encomendas, sem ter em consideração a sua capacidade, entregando-as posteriormente).

4. Efetuar todas as recolhas/entregas de um dia com apenas um veículo com autonomia e capacidade limitadas

Nesta etapa, e com a aproximação à realidade do problema, passaremos a considerar um novo atributo - a capacidade do veículo de transporte. Aqui é vital o emparelhamento das recolhas e das entregas de forma a nunca exceder a capacidade máxima de um determinado veículo.

5. Efetuar todas as recolhas/entregas diárias com mais do que um veículo (com autonomia e capacidade limitadas)

Finalmente, numa última fase, iremos conciliar vários veículos, dividindo todas as encomendas diárias, pela frota da empresa. Nesta fase, teremos de considerar qual a divisão mais eficiente, dado que esta será a solução final do problema.

Formalização do problema

Dados de entrada

nV – Número de veículos disponíveis que a empresa possui para realizar as encomendas.

A partir de nV são criados veículos (**VY**) que irão possuir:

autM - Valor da autonomia máxima do veículo em unidades de distância (considerando que todos os veículos disponíveis possuem a mesma autonomia).

capM – Valor da capacidade máxima do veículo (considerando que todos os veículos disponíveis possuem a mesma capacidade).

Gm = (V, E) – Grafo dirigido pesado, constituído por:

V – Conjunto de vértices. Estes vértices podem representar:

- R - locais de recolha de encomendas;
- D - locais de entrega de encomendas;
- C - locais de recarga dos veículos (incluindo o ponto inicial/final, a garagem);
- I - possíveis interseções de ruas.

Cada um destes vértices contém:

- (x, y) - As suas coordenadas utilizadas para o cálculo das distâncias;
- Adj $\subseteq$ E – conjunto de arestas adjacente ao vértice.

E – Conjunto de arestas. Estas arestas representam as várias ruas do mapa representado pelo grafo. Cada uma destas arestas contém:

- id – utilizado para diferenciar as arestas
- name – nome da rua que a aresta representa
- w – peso da aresta que representa a distância de um vértice a outro vértice adjacente
- dest $\in$ V – vértice destino da aresta

Este grafo que representa um mapa, possui um vértice que representa o ponto inicial e o ponto final:

G – Garagem. Local que representa o ponto de partida e de chegada dos veículos. Este vértice também é um ponto de recarga.

RD – Conjunto de pares de vértices composto por um local recolha e o respetivo local de entrega, pertencentes ao grafo. Cada par representa uma encomenda. A este conjunto é também adicionado o peso da encomenda, e o seu id - considerando que todas as encomendas têm um id único.

Dados de saída

sV – Conjunto de veículos utilizados. Estes veículos possuem (para além dos atributos já mencionados):

sEV – conjunto de elementos do grafo (arestas $\in E$, e vértices $\in V$, onde é especificado o tipo de vértice **R**, **D**, **C**, **I**, e o id da encomenda se esse for o caso (nulo quando não for)) ordenadas pela ordem de passagem do veículo.

vD – Distância percorrida pelo veículo ao realizar as suas recolhas/entregas:

$$\sum_{Ex \in sEV} w(Ex)$$

sRD – conjunto de encomendas que não foram entregues, por não ser possível recolher/entregar e retornar ao ponto inicial com os recursos (autonomia, capacidade ou número de veículos) que foram dados pelo utilizador, ou devido ao facto do grafo não ser fortemente conexo.

Restrições dos dados

Dados de entrada

O <u>número de veículos</u> disponíveis para utilização é sempre maior ou igual a um:

$$nV \geq 1.$$

<u>Capacidade</u>:

O valor de entrada da capacidade tem de ser maior que zero:

$$capM > 0$$

Deste modo, todos os veículos possuem a mesma capacidade inicial, sendo que esta é igual à capacidade máxima recebida:

$$\forall\, v \in VY,\ cap\ {}^{0}_{v} = capM$$

Por fim, a capacidade de cada veículo nunca pode ser negativa, nem exceder o valor máximo.

$$\forall\, v \in VY, \forall\, t\ ,\ 0 \leq cap\ {}^{t}_{v} \leq capM$$

<u>Autonomia</u>:

O valor de entrada da autonomia tem de ser maior que 0:

$$autM > 0$$

Deste modo, todos os veículos possuem a mesma autonomia inicial, sendo que esta é igual à autonomia máxima recebida:

$$\forall\, v \in VY,\ aut\ {}^{0}_{v} = autM$$

Por fim, a autonomia de cada veículo nunca pode ser negativa, nem exceder o valor máximo.

$$\forall\, v \in VY,\ \forall\, t\ ,\ 0 \leq aut\ {}^{t}_{v} \leq autM$$

<u>Vértices</u>:

Todo o itinerário - para a entrega das encomendas - tem, obrigatoriamente, de ser composto por vértices do mesmo componente fortemente conexo do grafo, ou seja, os veículos têm de ser capazes de entregar qualquer encomenda que recolham. Uma encomenda pode ser entregue por um veículo se este possuir capacidade de a entregar e tiver autonomia suficiente para ir desde o ponto inicial ao ponto de recolha, depois ao ponto de entrega, e retornando, no final, ao ponto de onde partiu (sem descartar a possibilidade de o veículo passar por pontos de recarga a meio do percurso). No caso de uma encomenda não ser alcançável por nenhum veículo, esta não será entregue pela empresa e adicionada a **sRD** - conjunto de encomendas que não foram entregues.

<u>Arestas</u>:

O peso de uma aresta não pode ser negativo, visto que representa uma distância:

$$\forall\, Ex \in E,\ w(Ex) \geq 0$$

Dados de saída

O número de veículos utilizados é sempre menor ou igual ao número de veículos disponibilizado inicialmente:

$$|sV| \leq\ nV$$

Todos os veículos utilizados têm de chegar à garagem com uma capacidade final igual à capacidade máxima, ou seja, não podem voltar à garagem, no final do dia, com encomendas por entregar:

$$\forall\, VYx \in sV,\ cap_f(VYx)\ =\ capM$$

Todas as arestas e vértices que se encontram em **sEV** têm de pertencer ao grafo:

$$\forall\, VYx \in sV,\ \forall\, Ex \in sEV : Ex \in E \cap \forall\, Vx \in sEV : Vx \in V$$

A distância percorrida pelos veículos tem de ser positiva:

$$\forall\, VYx \in sV,\ vD(VYx)\ \geq 0$$

Na sequência ordenada final de cada veículo, o primeiro e o último elemento têm ambos de corresponder ao vértice da garagem:

$$sV[0]\ =\ sV[|sV|\ -\ 1]\ =\ G$$

Funções objetivo

Considerando todas as adversidades já relatadas, a solução mais eficiente para o problema em questão, passa por atingir uma distância total mínima percorrida por todos os veículos aos quais houve, pelo menos, uma encomenda atribuída, procurando-se, também, minimizar o número de veículos utilizados, de forma a conseguir distribuir todas as encomendas para um determinado dia.

Deste modo, as funções objetivos serão, ambas, funções de minimização, pela seguinte ordem de prioridade:

1. $S = min(\sum_{VEx \in sV} vD(VEx))$
2. $z = min(|sV|)$

Contudo, é dada ao utilizador a possibilidade de reverter as prioridades, obtendo uma solução que dá prioridade ao menor número de veículos face à distância total percorrida pelos mesmos:

1. $z = min(|sV|)$
2. $S = min(\sum_{VEx \in sV} vD(VEx))$

Para além disso, aliada a estas funções objetivo, a solução deverá ter em conta a entrega do número máximo de encomendas (para o caso de não ser possível entregar todas as encomendas):

$$min(|sRD|)$$

Algoritmos e técnicas de *design*

Neste ponto vamos explicar os diferentes algoritmos usados, bem como, o seu uso ao longo do nosso programa, também iremos debater algumas das estratégias ou paradigmas que serão usados - seja por implementação própria ou como consequência pelo uso de determinados algoritmos.

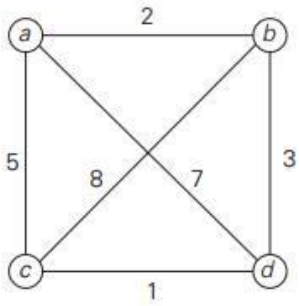
Um aspeto importante a ter em conta é que os algoritmos com os quais vamos trabalhar são sobretudo algoritmos de caminho mais curto e poderão estar sujeitos a pequenas adaptações - como pequenas otimizações ou ligeiras diferenças nos detalhes de implementação - para encaixarem na nossa proposta. Além disso, é importante, desde já, explicar algumas das estratégias, usadas pelos diferentes algoritmos, que serão referidas mais à frente, em particular, a estratégia de força bruta, a estratégia gananciosa e a estratégia de backtracking.

Algoritmos de força bruta

Este é o tipo de algoritmos mais básico e simples. O algoritmo de força bruta é uma abordagem de resolução de um problema de apenas um sentido, baseando-se na iteração de todas as possibilidades disponíveis para resolver um problema.

Um exemplo simples, mas esclarecedor, é o problema de tentar encontrar o código de 4 dígitos de um cofre. Uma abordagem de força bruta baseia-se na tentativa de todas as possibilidades de combinações uma a uma como: 0000, 0001, 0002, até obtermos a combinação correta. Deste modo, no pior caso, iríamos ter de experimentar 10.000 tentativas até encontrar a combinação correta.

Assim, podemos concluir que este tipo de algoritmos são custosos, mas que são muitas vezes utilizados por serem a primeira abordagem que é feita quando se procura a solução de um problema.



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Fig.2 - Exemplo de uso da Força Bruta

Algoritmos Gananciosos e *Backtracking*

Em primeiro lugar, os algoritmos gananciosos, como o próprio nome já deixa antever, procuram obter sempre a melhor solução num dado momento de execução. Apesar de serem definidos de uma forma simples, estes algoritmos nem sempre oferecem a melhor solução. No entanto, quando as sucessivas otimizações locais se traduzem numa otimização global, estes algoritmos revelam-se extremamente eficazes e muito competentes.

Agora, numa outra perspetiva, os algoritmos de *backtracking* são considerados uma otimização relativamente aos algoritmos de força-bruta - aqueles onde é necessário verificar a validade de todas as soluções candidatas. Neste tipo de algoritmos, a partir do momento em que uma solução candidata deixa de ser viável é efetuado um *backtrack*, um retrocesso a um ponto anterior dessa solução candidata e nesse novo ponto efetuar-se-á mais um procura por uma nova solução candidata. Os algoritmos deste tipo são extremamente vulgares, por exemplo, na resolução de labirintos. Isso ocorre como uma consequência de possuírem imensas aplicações no quotidiano.

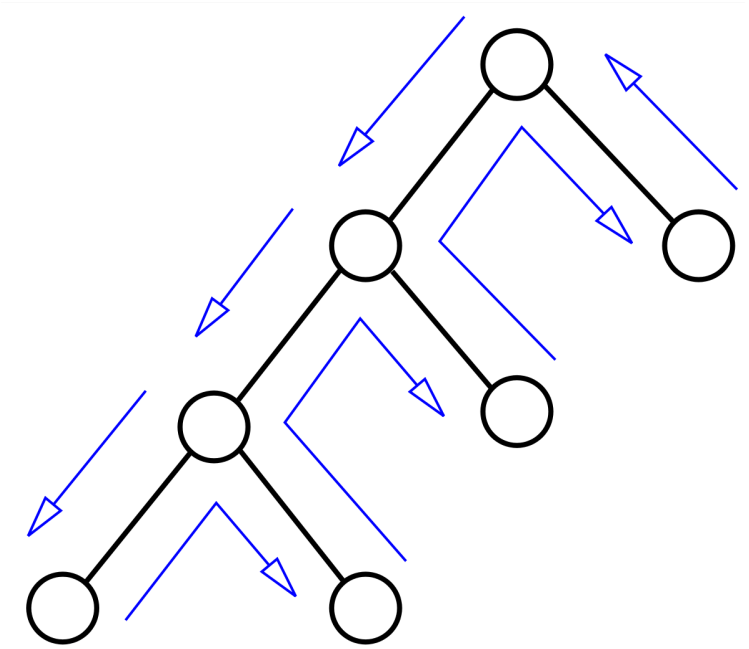


Fig.3 - Técnica de Backtracking

DFS e BFS - Pesquisa em Grafos

Com o alargado número de aplicações que os grafos encontram em diversas situações do quotidiano, surgiu a necessidade de se efetuar pesquisa sobre os seus vértices de uma forma simples, eficaz e ordenada. Desse modo, surgiram duas maneiras de iterar sobre os elementos desta estrutura de dados e que paragem obrigatória para todo aquele que estuda na área de computação - são eles o *DFS* (*Depth First Search*, em português "pesquisa em profundidade") e o *BFS* (*Breadth First Search*, em português "pesquisa em largura").

Primeiro, o *DFS* é provavelmente a maneira mais simples de implementar a pesquisa em grafos. Este método é comumente associado a um "pilha" ou *LIFO* (*Last In, First Out*) , pois é, de uma forma geral, implementado através da recursividade. Aqui, começamos por visitar um dos "filhos" de cada vértice e aí fazemos o mesmo, ou seja, continuamos até chegar a um vértice que não tenha quaisquer "filhos". Depois, vamos retirando os vértices visitados e iteramos sobre os restantes "filhos" do nó acima - ou irmãos - que ainda não tenham sido visitados.

Depois, o *BFS* ou a pesquisa em largura é associado a um outro tipo de estrutura de dados - uma "fila" ou *FIFO* (*First In, First Out*). Aqui, a partir de um vértice inicial adicionamos ao *FIFO* todos os vértices de destino ligados por uma aresta ao vértice em processamento e que ainda não tenham sido visitados/processados. Depois efetuamos uma nova iteração, processando o vértice que esteja no topo desse *FIFO* e removendo durante esse processamento. O processo iterativo terminará assim que o *FIFO* esteja vazio, sinal de que todos os elementos alcançáveis já foram visitados.

Em suma, a pesquisa em largura e a pesquisa em profundidade apresentam diferentes perspectivas para o mesmo problema. Assim, dependerá também do contexto para que se escolha a alternativa mais correta. Por exemplo, quando há necessidade de visitar todos os nós de um grafo não pesado, o *DFS* apresenta-se como uma ótima escolha, visto que a complexidade será de qualquer modo linear para além de que é mais fácil de implementar relativamente ao *BFS*. Por outro lado, o *DFS* não deve ser usado se o objetivo é, a título de exemplo, descobrir se um dado grafo tem ou não ciclos - isto porque com *DFS* é possível detetar ciclos a partir do momento em que um vértice já visitado é inserido no *FIFO*.

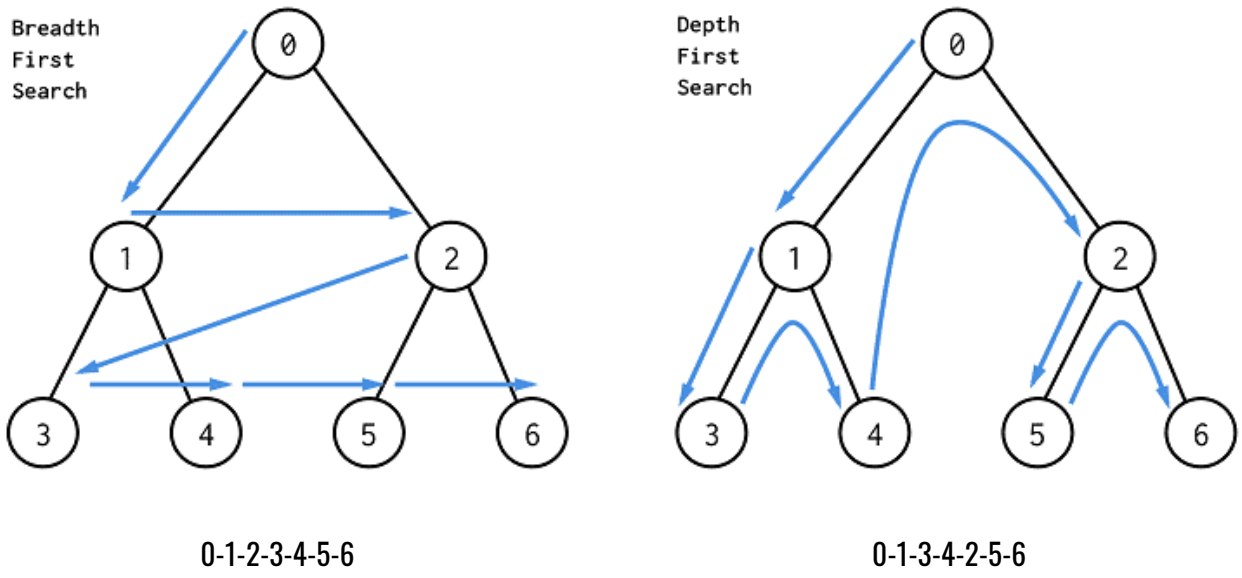


Fig.4 - Diferenças entre os algoritmos BFS e DFS

Conectividade de um Grafo

Apesar da pesquisa num grafo ser algo que merece atenção, o conceito de conectividade também o merece. A conectividade é um conceito muito simples do campo da teoria de grafos, basicamente, refere-se à existência ou não de uma aresta entre dois vértices.

Inicialmente, para determinar se um dado componente de um grafo não dirigido é conexo basta efetuar uma pesquisa em profundidade pelos nós pretendidos, no caso dessa pesquisa visitar todos os vértices desse componente podemos afirmar que esse conjunto de vértices é conexo.

Outros pontos que importa realçar no campo da conectividade são: a biconectividade e pontos de articulação - que aliás possuem uma relação bastante próxima na medida em que um grafo biconexo não pode ter pontos de articulação. Ou seja, um grafo diz-se biconexo quando é possível retirar qualquer um dos vértices sem que o mesmo se torne desconexo. Em contrapartida, os pontos de articulação são os vértices que ao serem retirados de um dado grafo torna-no desconexo. O algoritmo para a determinação deste tipo de pontos é o do cálculo da propriedade $Low(vertex)$ e possui diversas aplicações no quotidiano - como na medição da tolerância a falhas de uma dada rede (água, eletricidade, telecomunicações, etc.).

Não obstante, o nosso foco neste projeto é a manipulação de grafos dirigidos e consequentemente determinar os seus componentes fortemente conexos. Durante as aulas foi apresentado um algoritmo que soluciona esse problema baseado em árvores de expansão e finalizando com uma pesquisa em pré-ordem aos vértices do grafo.

Outro aspeto é o de que, num grafo dirigido, a conectividade pode ter 3 possíveis classificações: não conexo, fracamente conexo e ainda fortemente conexo. Não conexo, tal como o nome já deixa antecipar, estamos perante um grafo onde não é possível alcançar todos os vértices independentemente do vértice inicia, por outras palavras, há pelo menos um vértices que não tem arestas e portanto a lista de adjacência vazia - por exemplo. Num grafo fracamente conexo é possível atingir todos os vértices mas apenas a partir de um conjunto restrito de vértices, ou seja, nem todos os vértices possuem pelo menos uma aresta que tenha como ponto de partida ele próprio. Finalmente, um grafo dirigido fortemente conexo carateriza-se por, a partir de qualquer vértice, ser possível atingir os restantes, noutros termos, para qualquer vértice do grafo há pelo menos uma aresta que parte desse vértice.

Cálculo da conectividade

O algoritmo implementado necessita de, numa primeira fase, utilizar um mecanismo para determinar se uma dada encomenda poderá ser entregue por um dos veículos da frota. Para tal, utilizamos um método simples de pesquisa em largura - explicado mais acima - e onde para cada vértice visitado propriedade *visited* é ativa.

Assim, para todos os vértices onde a propriedade *visited* está ativa, existirá pelo menos um caminho para lá chegar. O uso da pesquisa em largura face à pesquisa em profundidade exige o uso de uma fila para aqueles vértices que ainda não foram processados, mas, em contrapartida, é possível detetar a existência de ciclos em grafos dirigidos - como é o caso da rede viária que utilizamos.

Algoritmos de caminho mais curto

Os algoritmos de caminho mais curto, em poucas palavras, são a essência do nosso programa. Isto deriva pelo facto de lidarmos com uma função objectivo que procura minimizar a distância percorrida pelos veículos de uma frota. No nosso caso utilizamos vários algoritmos, descritos ao longo desta secção, de acordo com o *input* introduzido pelo utilizador. Um exemplo pequeno desse tipo de otimizações, e que serão abordadas mais à frente é o uso do algoritmo de Floyd Warshall para evitar o uso repetido do algoritmo de Djiskstra a cada iteração do nosso algoritmo.

Algoritmo de Dijkstra

Este algoritmo foi concebido por Edsger W. Dijkstra em 1956 e publicado três anos mais tarde. Este algoritmo recebe como dados de entrada um grafo dirigido - sobre o qual serão computados os diferentes caminhos - e um vértice inicial.

Essencialmente , o algoritmo calcula o caminho mais curto a partir do vértice inicial aos restantes vértices. Para atingir isso, o algoritmo de Dijkstra usa uma fila de prioridade mutável, que permite atingir uma complexidade temporal da ordem de $O\ (|E|\ +\ |V|\ log\ |V|)$, o que o torna no algoritmo mais eficiente no campo dos algoritmos de caminho mais curto. Não obstante, é importante ter em conta que para que tal complexidade seja atingida é necessário usar um tipo especial de fila de prioridade - uma *heap de Fibonacci*.

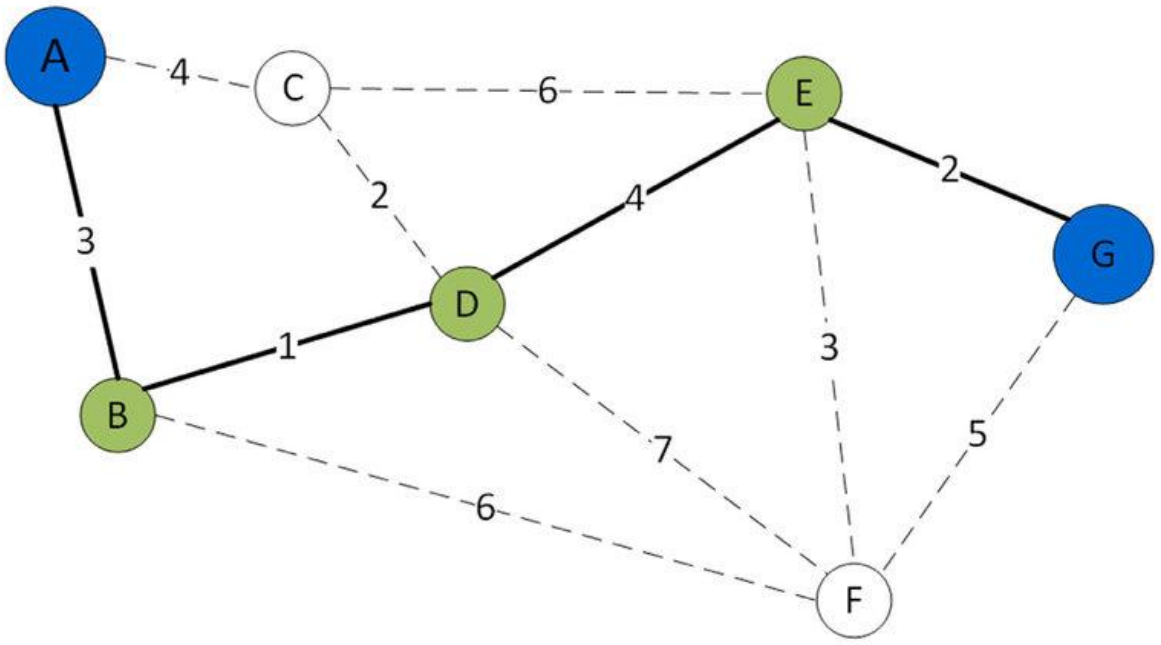


Fig.5 - Resultado da utilização do algoritmo de Dijkstra

Contudo, mesmo sendo um algoritmo extremamente eficiente, existem algumas desvantagens. Um exemplo será o facto de este algoritmo não lidar com ciclos negativos no grafo - o que no nosso caso não será um problema, uma vez que as arestas do grafo terão de ser inteiras e positivas. Também, em relação às arestas de peso positivo note-se na Fig. 5 que para aquele grafo as valores das arestas variam entre um e sete.

Pseudocódigo

$DIJKSTRA\ (graph,\ source)$:
    $Q\ =\ \emptyset$
    $for\ each\ vertex\ v\ in\ graph$:
        $v.dist\ =\ \infty$
        $v.prev\ =\ NIL$
    $source.dist\ =\ 0$
    $Q.insert(source)$

    $while\ Q \neq \emptyset$:
        $u\ =\ Q.extractMin()$
        $for\ each\ vertex\ v\ in\ u.Adj$:
            $if\ u.dist\ +\ weight(u,\ v)\ <\ v.dist$:
                $v.dist\ =\ u.dist\ +\ weight(u,\ v)$
                $v.prev\ =\ u$
            $if\ Q.contains(v)$:
                $Q.decreaseKey(v)$
            $else$:
                $Q.insert(v)$

Nesta versão do algoritmo de Dijkstra existem certos aspetos que saltam à vista. Em primeiro lugar por se tratar de um algoritmo ganancioso, por causa da chamada $Q.extractMin()$. Esta função irá procurar a solução ótima para um subproblema num dado momento da sua execução, por outras palavras, $Q.extractMin()$ retorna a aresta com o menor peso que ainda não foi procurada e a partir daí verificar se ela faz parte de algum caminho mais curto.

Análise do Algoritmo

Finalmente, a análise deste algoritmo está disponível no livro do Cormen - com todo o detalhe e rigor. A correção do algoritmo, ou em inglês *correctness*, trata-se de um prova indutiva onde o caso base é a distância de um dado vértice a ele próprio que será sempre zero. E onde o caso indutivo ocorre com a visita a um vértice desconhecido (ou não visitado) a partir de um vértice para o qual já se sabe a distância mínima desde a origem.

Deste modo, é possível inferir que o caminho mais curto - solução ótima do problema - será a soma das distâncias mais curtas entre os vértices que compõem esse caminho - solução ótima dos subproblemas.

Utilização do Algoritmo

Dado o contexto do nosso problema, o uso deste algoritmo mostrou-se fundamental, não só pela incrível eficiência temporal e espacial que proporciona, mas também pela sua simples implementação mesmo usando uma fila de prioridade mutável, que é abordada no capítulo das estruturas de dados.

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford surgiu em 1955 e foi publicado em 1956 por Lester Ford e, mais tarde, em 1958 por Richard Bellman. Este é um algoritmo, tal como o de Dijkstra, procura soluĭonar o problema de caminho mais curto a partir de um vértice até aos restantes. Aliás os seus dados de entrada são praticamente os mesmos: um grafo dirigido - que contém uma lista de vértices e arestas - e um vértice (que será a origem).

Este algoritmo possui uma complexidade temporal da ordem $O\ (|V|\ *\ |E|)$ - superior ao algoritmo de Dijkstra. Não obstante, é também um algoritmo mais versátil, na medida em que é tolerável a existência de arestas com pesos negativos[1] sendo também capaz de reportar a existência de ciclos negativos no grafo passado como dado de entrada.

Pseudocódigo

$BELLMAN\ -\ FORD\ (graph,\ source)$:
    $for\ each\ vertex\ v\ in\ graph$:
        $v.dist\ =\ \infty$
        $v.prev\ =\ NIL$
    $source.dist\ =\ 0$

    $for\ i\ from\ 1\ to\ |V|\ -\ 1$:
        $for\ each\ edge(u,\ v)\ in\ graph$:
            $if\ u.dist\ +\ weight(u,\ v)\ <\ v.dist$:
                $v.dist\ =\ u.dist\ +\ weight(u,\ v)$
                $v.prev\ =\ u$

    $for\ each\ edge(u,\ v)\ in\ graph$:
        $if\ u.dist\ +\ weight(u,\ v)\ <\ v.dist$:
            $ERROR("Negative\ weight\ cycle")$

Aqui temos uma versão mais abstrata e geral daquilo que este algoritmo faz. Existem alguns aspetos a destacar, sobretudo o uso de programação dinâmica com o uso de listas para guardar os valores referentes às distâncias já calculadas[2]. Por outro lado, a análise deste algoritmo permite também descobrir um pouco mais desses aspetos que devem ser destacados.

[1] Apesar de este tipo de constrangimento ser irrelevante no nosso problema, é importante explicar este algoritmo, visto que será usado como sub-rotina do algoritmo de Johnson - explicado mais adiante.
[2] Mais à frente, no algoritmo de Floyd-Warshall será dada maior atenção à programação dinâmica como consequência das características desse algoritmo.

Análise do Algoritmo

À semelhança do algoritmo anterior - Dijkstra - a análise deste algoritmo pode ser encontrada com o detalhe e rigor característicos no livro do Cormen.

De qualquer modo, e de forma resumida, assenta em dois casos diferentes e trata-se também de uma prova indutiva. Em primeiro lugar o caso base é o cálculo da distância a partir do vértice da origem até ele mesmo e que será sempre 0. Por sua vez, no caso indutivo, chega-se à conclusão de que ao fim de $|V| - 1$iterações sobre todas as arestas os diferentes caminho mais curtos estarão calculados, isto porque uma das características dos caminhos mais curtos é não possuírem ciclos e desse modo o caminho mais curto será no máximo composto por $|V| - 1$ arestas, ou seja, será o caminho mais curto a partir da origem e que chegará a todos os vértices do mesmo componente fortemente conexo desse grafo.

Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é um algoritmo de caminho mais curto e foi formalizado em 1962 por Robert Floyd. O algoritmo tem como dados de entrada simplesmente um grafo - o que já indicia algumas diferenças em relação aos algoritmos anteriores (Dijkstra e Bellman-Ford).

Comparativamente ao algoritmo de Dijkstra, Floyd-Warshall é um algoritmo bem mais poderoso na medida que é capaz de computar os caminhos mais curtos entre quaisquer pares de vértices do grafo de entrada. Todavia, este poder acrescido traduz-se num esforço computacional maior, uma vez que este algoritmo requer uma matriz de adjacência - o que leva a uma complexidade espacial quadrática $\Omega\ (|V|^2)$ - onde serão guardadas as distâncias mínimas e ainda uma complexidade temporal cúbica $\Theta\ (|V|^3)$.

De salientar que este algoritmo, por si só, não foi concebido para guardar os vértices que compõem os caminhos mais curtos, não obstante isso é facilmente ultrapassado com pequenas adaptações, isto é, criar uma nova matriz que contenha informação do vértice anterior para que se possa fazer a reconstrução do caminho desejado.

Pseudocódigo

```
FLOYD − WARSHALL (graph):
    dist = Matrix(|V|, |V|, ∞)
    next = Matrix(|V|, |V|, NIL)

    for each edge(u, v) in graph:
        dist[u][v] = weight(u, v)
        next[u][v] = v
    for each vertex v in graph:
        dist[v][v] = 0
        next[v][v] = v

    for k from 1 to |V|:
        for i from 1 to |V|:
            for j from 1 to |V|:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    next[i][j] = next[i][k]
```

A complexidade deste algoritmo é facilmente compreendida pelo facto de haver três *for's* emparelhados. Outro aspeto muito relevante sobre o algoritmo é que nessas linhas está a ser computada a seguinte fórmula recursiva:

$$ShortestPath(i,\ j\ ,k) = min(ShortestPath(i,\ j,\ k-1),\ ShortestPath(i,\ k,\ k-1) + ShortestPath(k,\ j,\ k -$$

Basicamente, esta fórmula recursiva pode ser descrita como: o caminho mais curto desde o vértice i até ao vértice j passando por k vértices será o mínimo entre o caminho mais curto de i até j passando por $k-1$ vértices ou então a soma dos caminhos mais curtos de i a k e de k a j passando também por $k-1$. Curiosamente, este algoritmo (em particular a fórmula recursiva descrita acima) apresenta uma semelhança

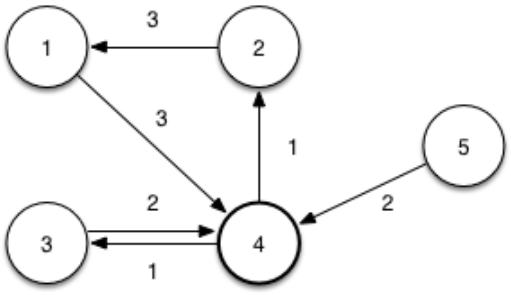
muito próxima ao algoritmo de conversão de *DFA's* (*Deterministic Finite Automaton*) para *RE* (*Regular Expressions*).

De qualquer modo, e para concluir a condição recursiva é ainda necessária uma condição inicial:

$$ShortestPath(i,\ j,\ 0)\ =\ weight(i,\ j)$$

Assim sendo, esta fórmula recursiva pode ser facilmente computada através de programação dinâmica. Neste algoritmo essas características estão bem visíveis, na medida em que os valores necessários para a fórmula recursiva são guardados numa memória (matriz de adjacência com as distâncias) e desse modo não obrigam a que se repita trabalho computacional de forma desnecessária - algo que é realmente custoso.

Após o cálculo dos caminhos mais curtos - distâncias e vértices precedentes, é possível proceder à reconstrução dos caminhos mais curtos através do seguinte algoritmo.



D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	**nil**	**4**	**4**	1	**nil**
2	2	**nil**	**4**	1	**nil**
3	**2**	**4**	**nil**	3	**nil**
4	2	4	4	**nil**	**nil**
5	2	**4**	**4**	5	**nil**

Fig.6 - Matrizes resultantes do algoritmo de Floyd-Warshall

$PATH\ -\ RECONSTRUCTION\ (u,\ v)$:
    $Q\ =\ \emptyset$
    $if\ next[u][v]\ =\ NIL$:
        $return\ Q$
    $Q.add(u)$
    $while\ u \neq v$:
        $u\ =\ next[u][v]$
        $Q.add(u)$
    $return\ Q$

Este algoritmo de reconstrução de caminhos retorna, ao utilizador, uma lista com os vértices ordenados de acordo com a sua visita.

Algoritmo de Johnson

Por fim, o algoritmo de Johnson é, como o apresentado anteriormente - Floyd-Warshall, um algoritmo que calcula os caminhos mais curtos entre quaisquer dois vértices de um grafo dirigido. Apresentado em 1977 por Donald B. Johnson, apresenta alguns aspetos interessantes que nenhum dos algoritmos anteriores aborda - sobretudo a técnica de *reweighting*.

Além de resolver o mesmo problema que o algoritmo de Floyd-Warshall, com uma complexidade temporal da ordem $O\left(|V|^2 \, log \, |V| \; + \; |V| \; * \; |E|\right)$[3] (o que é muito mais eficiente que o $O\left(|V|^3\right)$ de Floyd-Warshall, particularmente em grafos esparsos), este algoritmo aproveita-se de outros dois algoritmos abordados anteriormente - Dijkstra e Bellman-Ford.

Inicialmente o algoritmo irá criar um novo vértice s, que se conectará a todos os outros por arestas com peso nulo. De seguida entra em cena o algoritmo de Bellman Ford. Esta é uma das fases cruciais, pois, é aqui que serão detectados os ciclos de peso negativo (se existirem) e será atribuído a cada vértice um peso que corresponde à distância do caminho mais curto até esse vértice.

No que toca à técnica de *reweighting*, esta permite, sem prejuízo da alteração dos caminhos mais curtos originais, alterar os pesos das arestas para valores positivos - o que permitirá mais à frente a utilização do algoritmo de Dijkstra. Uma vez que no passo anterior foi associado a cada vértice um peso, é agora altura de para cada aresta efetuar o *reweighting*, que seguirá a seguinte fórmula:

$$\widehat{w}(u, \, v) \; = \; w(u, v) \; + \; h(u) \; - \; h(v)$$

Após este cálculo, verifica-se que:

$$\forall \, u, \, v \; \in V : \widehat{w}(u, \, v) \; \geq \; 0$$

Com isto, será possível prosseguir para o último passo do algoritmo, onde será usado, em cada vértice do grafo, o algoritmo de Dijkstra e desse modo descobrir todos os caminhos mais curtos do grafo.

[3] Esta complexidade só é válida se forem usadas as estruturas de dados apropriadas. Neste caso, isto acontece como fruto do uso do algoritmo de Dijkstra onde é abordada esta problemática.

Pseudocódigo

$JOHNSON\ (graph)$:
    $graph.V.add(s)$
    $for\ each\ vertex\ v\ in\ graph$:
        $graph.E.add(s,\ v,\ 0)$

    $BELLMAN\ -\ FORD\ (graph)$
    $for\ each\ vertex\ v\ in\ graph$:
        $v.h\ =\ dist.v$
    $for\ each\ edge(u,\ v)\ in\ graph$:
        $\widehat{w}(u,\ v)\ =\ w(u,v)\ +\ h(u)\ -\ h(v)$

    $D\ =\ Matrix(|V|,\ |V|,\ \infty)$
    $for\ each\ vertex\ u\ in\ graph$:
        $DIJKSTRA\ (graph,\ v)$
        $for\ each\ vertex\ v\ in\ graph$:
            $D[u][v]\ =\ v.dist\ +\ h(v)\ -\ h(u)$

Comparativamente com Floyd-Warshall saltará à vista o facto de este algoritmo guardar as distâncias numa matriz de tamanho $|V|\ *\ |V|$, mas não guardar para uma matriz do mesmo tamanho para que seja possível reconstruir esse caminho.

Por esta razão, apesar de não ser efetivamente utilizado na solução final, é aqui feita uma breve abordagem. Isto é fruto do facto de a implementação do grafo não permitir de uma forma trivial a implementação da reconstrução dos caminhos.

Análise do Algoritmo

Como este algoritmo faz uso de outros que já estão provados como sendo corretos, a preocupação recai sobre o uso da técnica de *reweighting*. Assim sendo, o somatório de todos os pesos (após o *reweighting*) do grafo será dado pela seguinte expressão:

$$\sum_{weights} = w(s,\ p_0)\ +\ h(s)\ -\ h(p_0)\ +\ w(p_0,\ p_1)\ +\ h(p_0)\ -\ h(p_1)\ +\ \dots\ +\ w(p_{n-1},\ p_n)\ +\ h(p_{n-1})\ -\ h(p_n)$$

Simplificando a expressão - removendo os termos que se anulam, chegámos a:

$$\sum_{weights} = (w(s,\ p_0)\ +\ w(p_0,\ p_1)\ +\ \dots\ +\ w(p_{n-1},\ p_n))\ +\ h(s)\ -\ h(p_n)$$

$$\sum_{weights} = \sum_{original\ weights}\ +\ h(s)\ -\ h(p_n)$$

Deste modo, podemos concluir que a primeira parte da expressão trata-se do somatório dos pesos originais do grafo, e que para qualquer aresta do grafo esta estará defasada do seu peso original por um fator de $h(s)\ -\ h(p_n)$, garantindo assim o caminho mais curto.

Considerações Preliminares

O uso dos algoritmos anteriores irá variar ao longo de cada uma das iterações da resolução implementada.

Num primeiro plano, é importante aferir algumas considerações sobre os dados de entrada, em particular, no grafo. Assumindo que o grafo será construído com base em mapas reais, podemos interrogar-nos sobre certos aspetos:

- Será o grafo denso, uma vez que representa uma área mais urbana com um maior número de vértices e/ou arestas?
- Ou será, por outro lado, um grafo pouco denso - provavelmente representando uma zona rural?

Este é um exemplo de perguntas relevantes para que seja possível escolher de uma forma acertada os algoritmos.

Outro facto importante de salientar, é que as complexidades dos algoritmos em questão variam de forma diferentes. Por exemplo, em Floyd-Warshall, grafos densos e com poucas arestas resultarão em tempos de execução menores - comparativamente ao algoritmo de Dijkstra ou Johnson!

Neste caso, considerando um grafo denso, outra questão impõem-se:

- Quais os algoritmos a usar, conforme o número de veículos?

Esta questão coloca alguns problemas, mas começando por um caso mais simples - caso base - de um único automóvel, seria simples a resposta a qual dos algoritmos deveria ser escolhido. Essa resposta seria o Dijkstra pelo facto de que tem a complexidades temporais e espaciais muito apelativas para algoritmos deste género de problemas.

Como foi mencionado acima, este algoritmo é o que apresenta menor complexidade temporal de entre todos os algoritmos de caminho mais curto se usado com as estruturas de dados corretas. Outra vantagem de Dijkstra sobre Floyd-Warshall é a complexidade espacial linear $O(|V|)$ face à complexidade quadrática $O(|V|^2)$.

Contudo, o cenário inverte-se tendo em conta um número de veículos superior ao caso base e ao número de vértices do grafo. Isto porque usando Dijkstra será necessário equacionar vários caminhos mais curtos - uma vez para cada carro ou para todos - o que consequentemente poderá aumentar a complexidade final do programa. Por outro lado, independentemente do número de veículos a circular, usando o algoritmo de Floyd-Warshall basta que este seja executado uma única vez para que se obtenha o caminho mais curto que deve ser seguido por qualquer carro.

Por fim, tendo em consideração todos estes aspetos fica aqui o *link* para uma folha de cálculo[4] onde se averigua qual o melhor algoritmo a usar, tendo em conta um dado número de veículos e um grafo com um número aleatório de vértices e de arestas - para que se obtenha uma certa variação na densidade dos grafos.

[4] https://bit.ly/3dUDvZn - link para a spreadsheet.

Funcionalidades

O nosso programa, como é possível antever pela descrição do problema procurou implementar um programa capaz de lidar com a distribuição de encomendas - *packages* - pelas moradas dos clientes de uma dada empresa numa dada região com uma determinada rede viária.

Ao longo do percurso impõem-se algumas barreiras e obstáculos como o facto de alguns locais não serem acessíveis, a capacidade limitada dos veículos ou até mesmo a autonomia dos veículos da frota. Além disso, e como já mencionado inúmeras vezes anteriormente, demonstrou-se ser conveniente fornecer ao utilizador uma série de configurações que permitam manipular quais os aspetos que pretende minimizar - a distância total percorrida por todos os veículos ou o número total de veículos utilizados.

Uma funcionalidade a destacar é a de que quando um veículo atinge um determinado nível de carga (20%) este irá começar a procurar os locais mais próximos onde possa efetuar a recarga. Para isto foi implementada uma lista com um conjunto de pontos de interesse para que o veículo possa visitar. Estes pontos de interesse fazem uso de uma estrutura de dados especial e que será abordado na secção das estruturas de dados.

Além disso, o programa também permite, após o cálculo das rotas para cada veículo, a visualização dessas mesmas rotas. Para isso é utilizada a *API* fornecida - o *GraphViewer*. Aqui os pontos de interesse são assinalados com cores relevantes para que se possa ter uma melhor noção da sua localização. Os vários pontos de interesse são a garagem, os locais de entrega e de recolha e os postos de reabastecimento para os veículos elétricos.

Por último, o contacto entre o utilizador e o programa é efetuado pelo *interface*. É aqui o local onde o utilizador poderá alterar os diferentes parâmetros de entrada do algoritmo e onde, no final da execução, do algoritmo será mostrado o tempo total de execução. Isto permite, não só ao utilizador mas também aos desenvolvedores mostrar a eficiência do programa e, se porventura algo não estiver totalmente afinado, trabalhar numa possível solução mais rápida e eficiente. Agora, e numa perspetiva mais realista, o facto de um programa deste género não se demonstrar totalmente capaz, pode provocar perdas quer para a empresa, quer para os clientes.

Estruturas de dados

Os desafios apresentados pelo enunciado e a respetiva solução obrigam ao uso de estruturas de dados um tanto quanto sofisticadas - isto para que fosse possível obter a melhor complexidade possível com determinados algoritmos e operações. Eis algumas das mais importantes que foram utilizadas:

Grafos

Segundo a definição, os grafos são uma estrutura de dados composta por vértices e arestas. Apesar de soar a algo muito simples, esta estrutura de dados apresenta uma flexibilidade ímpar na medida em que imensas aspetos do nosso quotidiano podem sempre representados pela forma de um grafo, sobretudo, a rede viária de uma dada localidade, a título de exemplo. Neste caso, os grafos utilizados representam os pontos das estradas por onde os veículos podem passar.

A cada vértice é atribuído um identificador único que permite aceder à sua informação em tempo constante, visto que a sua posição na lista de vértices é mapeada com base nesse mesmo identificador. Ainda relativamente aos vértices existem uma série de características relevantes como o tipo de ponto de interesse que esse vértice possa ser ou não, uma lista de adjacência e ainda outras características que são exigidas pelos próprios algoritmos utilizados, por exemplo o path - um apontador para o vértice anterior no caminho mais curto (ver *Dijkstra*).

Finalmente, no que toca às arestas também temos um identificador único, e dois apontadores para os vértices de origem e destino, respectivamente, ou seja, rede viária, baseada em mapas reais, é representada por um grafo dirigido.

Heaps

As heaps são uma estrutura de dados do tipo árvore e que permitem a remoção do maior ou menor elemento de um *container* em tempo constante - $O(1)$. Esta caraterística chamou particularmente a atenção sobretudo pelo facto de cada veículo ter um conjunto de pontos de interesse a si associados, ou seja, o uso das heaps pela função da *Standard Library* $std::make\_heap()$ permite, a cada iteração, não só verificar qual o veículo mais próximo de um ponto de interesse com também o ponto de interesse mais próximo de cada um dos veículos.

Mutable Priority Queue

A *mutable priority queue* apesar de também ela ser uma heap - Fibonacci heap - merece um lugar destacado na medida em que é vital para que o algoritmo de *Dijkstra*, o mais importante para a execução do programa, apresente uma complexidade temporal da ordem de $O(|E| + |V| \log |V|)$, isto porque neste tipo de heap a operação $decreaseKey$, necessária sempre que um vértice já descoberto tem um caminho novo caminho mais curto, tem complexidade constante.

Stack e Vector

Estes foram os containers da biblioteca *Standard* do C++ utilizados. O uso, relativamente abundante de $std::vector$ocorre fruto da flexibilidade única que esta estrutura oferece - remover e inserir elementos em qualquer posição, etc. Ainda referir brevemente a $std::stack$utilizada para obter a reconstrução dos caminhos pela matriz obtida após o processamento do algoritmo de *Floyd-Warshall*.

Struct Input Parameters

Apesar de não se encaixar totalmente no conceito de estruturas de dados, é importante falar aqui desta estrutura na medida em que esta é responsável por agrupar os dados de entrada listados e descritos mais detalhadamente acima.

Solução

Dadas as novas circunstâncias do problema em questão, a solução varia significativamente da nossa proposta inicial, apresentada no primeiro relatório. Existem 2 casos que são tratados de forma ligeiramente distinta: a empresa apenas tem 1 veículo ao seu dispor e a empresa tem mais do que 1 veículo.

No primeiro caso, inicialmente, o programa utiliza o algoritmo de Dijkstra para calcular as distâncias do veículo (que se encontra na garagem) aos pontos de recolha presentes no grafo (e alcançáveis, como explicado na secção Conetividade dos Grafos). De seguida, estes pontos são colocados na *heap* do veículo, da qual é selecionado o topo, que corresponde ao vértice de recolha mais próximo. Através de uma função própria para o efeito, é obtido o caminho até esse ponto, ditado pelo algoritmo de Dijkstra anteriormente utilizado. Esse caminho é passado ao veículo, que o percorre. Quando chega ao ponto, o veículo recolhe a encomenda, altera o tipo de vértice para este deixar de ser um ponto de interesse e adiciona um novo ponto de interesse à sua *heap*: o vértice de entrega correspondente à encomenda recolhida. Este processo corresponde a uma iteração do algoritmo desenvolvido. De seguida, o passo repete-se, ou seja, o algoritmo de Dijkstra calcula de novo as distâncias, agora referentes ao novo vértice atual do veículo, a *heap* é devidamente atualizada e ordenada e o veículo irá percorrer novo caminho até um ponto de interesse. No final, quando não restarem mais encomendas para recolher ou entregar, o veículo retorna à garagem, através, mais uma vez, do caminho calculado pelo algoritmo de Dijkstra. Durante todo o percurso, o comprimento de cada "rua" (peso da aresta) é adicionado à distância total que o veículo percorre, distância esta que constitui a variável a minimizar. Por último, importante mencionar que, dado que os veículos apresentam autonomia limitada, se esta não for suficiente para avançar de vértice em algum ponto do seu percurso, o veículo irá a um ponto de recarga para que a sua autonomia volte ao valor máximo. A distância (fixa) percorrida para o carregamento é, também, adicionada à distância total que o veículo percorreu.

O segundo caso assemelha-se, em grande parte, ao primeiro, constituindo apenas uma distribuição de tarefas pelos vários veículos disponíveis. É dado adquirido que o algoritmo aloca pelo menos uma recolha/entrega a cada veículo, como forma de maximizar o aproveitamento dos recursos disponíveis. A diferença principal reside na utilização do algoritmo de Floyd-Warshall em detrimento do de Dijkstra. Este apenas necessita de ser executado uma vez, no início do programa, e não a cada iteração, como no caso anterior. Numa fase inicial, após cálculo das distâncias entre vértices, os pontos de recolha são adicionados às *heaps* dos vários veículos. Seguidamente, o veículo que tiver um ponto de interesse mais próximo será o que irá percorrer o caminho (obtido através do algoritmo de Floyd-Warshall, previamente executado) até este. O processo que se segue é análogo ao do primeiro caso e o programa termina quando todas as encomendas estiverem entregues e todos os veículos estiverem de volta à garagem.

Pseudocódigo

```
E − STAFETAS (grafo, nV, autM):
        buildVehicleSet (nV, autM)
        sRD = buildDeliverySet (grafo)
        if nV != 1:
                FLOYD − WARSHALL(grafo)
        source = G

        while not package delivered do:
                if nV = 1:
                        DIJKSTRA(grafo)
                        source = nextPointOfInterest()

                if nV != 1:
                        for each vehicles in VehicleSet:
                                refreshDistances()

                        for each vehicles in VehicleSet:
                                source = assignPackage()

                        if closestPointOfInterest () > NotEnoughAut:
                                charge()
                                source = nextPointOfInterest()
        sV = buildPathsForVehicles ()
        showRoutes()
        return sV, sRD
```

Aqui está uma implementação, de uma forma mais abstrata, do problema em questão. Existem, aqui, vários locais onde todas as estratégias discutidas anteriormente são apresentadas de uma forma condensada.

Análise das complexidades temporais e espaciais

A análise das complexidades temporais e espaciais dos algoritmos de *Floyd-Warshall* e Dijkstra é vastamente conhecida. De qualquer modo, não é possível determinar com exatidão a complexidade temporais e espaciais do código. Todavia fica aqui o link para uma spreadsheet onde é possível consultar os resultados obtidos pelos *benchmarks* à execução do programa[5].

Já no que toca à análise teórica o uso alargado de containers e funções da STL, bem como, as diferentes implementações destas para as diferentes plataformas tornam muito difícil, senão impossível, este tipo de análise com precisão.

[5] https://bit.ly/3dUDvZn - link para a spreadsheet.

Conclusão

Este projeto revelou, para surpresa de todos os elementos, extremamente enriquecedor em vários níveis. Em primeiro lugar, o facto de o problema ser bastante semelhante àqueles pelos quais várias empresas ou outras entidades passam no seu quotidiano levanta mais uma vez a ideia não só da importância, mas também da maneira como certos algoritmos facilitam determinadas ações do nosso quotidiano. Quem nunca usou um *GPS* para descobrir o melhor caminho a tomar até um determinado destino?

Numa fase inicial, foi pedido que aplicássemos diferentes técnicas para conceber um algoritmo com vista à resolução de um problema prévio. Ora isto obrigou-nos a estudar pelo assunto, o que demonstrou ser extremamente positivo, além disso a sua posterior implementação alargou até certo ponto o conhecimento sobre uma das linguagens de programação preferidas daqueles que desenvolvem algoritmos, o C++, sobretudo, pela extrema eficiência de que aufere e ainda da facilidade com que oferece certos mecanismos muito úteis durante o processo de desenvolvimento.

Finalmente, e numa pequena nota relativa aos aspetos positivos e negativos existem vários que se destacam. Positivamente, não só a descoberta de algoritmos que eram desconhecidos da parte dos elementos do grupo aliada à aquisição de certas competências que certamente se tornou muito útil, principalmente, para aqueles que veem na programação uma área de interesse. Por outro lado, numa faceta menos positiva tivemos certas dificuldades em implementar certos procedimentos por não conseguirmos estar totalmente esclarecidos - isto ainda numa fase muito inicial - mas que mais tarde acabou por se resolver!

Contribuição

Dado que o projeto está previsto para ser elaborado em equipa, todos os membros contribuíram de forma equitativa para este, trabalhando mutuamente em grande partes dos desafios que nos foram surgindo na conceção da possível solução.

- Daniel Félix - 33.33%
- Miguel Rodrigues - 33.33%
- Tiago Silva - 33.33%

Bibliografia

BrainKart. 2018. “Exhaustive Search.” BrainKart. https://www.brainkart.com/article/Exhaustive-Search_8013/.

Brčić, David, Marko Valčić, and Serdjo Kos. 2019. “Representation of determined Dijkstra's shortest path.” ResearchGate. https://www.researchgate.net/figure/Representation-of-determined-Dijkstras-shortest-path-Dijkstra-1959_fig10_337784487.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. Third ed. Cambridge, Massachusetts: The MIT Press.

Gupta, Kitty. 2019. “What is the difference between BFS and DFS algorithms.” FreelancingGig. https://www.freelancinggig.com/blog/2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms/.

Maheshwari, M. 2020. “Most important type of Algorithms.” Geeks For Geeks. https://www.geeksforgeeks.org/most-important-type-of-algorithms/.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Algoritmos de Força Bruta. Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/164079/mod_label/intro/01.TCA1.greedy.pdf.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Programação dinâmica. Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/164081/mod_label/intro/04.TCA1.progdinamica.pdf?time=1614253573909.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Funcionamento Correto (Correctness). Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/164082/mod_label/intro/05.TAA1.funcionamentocorreto.pdf?time=1614627000181.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Pesquisa e Ordenação. Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/164083/mod_label/intro/07.grafos2.pdf?time=1615414017142.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Caminho mais curto (Partes I e II). Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/197873/mod_label/intro/09.grafos4.pdf.

Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Ramos, F. Ramos, and G. Leão. 2021. *Técnicas de Concepção de Algoritmos*, Algoritmos de Retrocesso. Porto, Portugal: n.p. https://moodle.up.pt/pluginfile.php/164080/mod_label/intro/03.TCA1.retrocesso.pdf.

Tiwari, Vighnesh. 2019. “Floyd Warshall Dynamic Programming Algorithm.” https://medium.com/@vighneshtiwari16377/floyd-warshall-dynamic-programming-algorithm-e2a899c3e5e6.

Wikipedia. 2021. “Johnson's algorithm.” Johnson's algorithm. https://en.wikipedia.org/wiki/Johnson%27s_algorithm.

Wikipedia. 2021. “Fibonacci heap.” Fibonacci heap. https://en.wikipedia.org/wiki/Fibonacci_heap.

Wikipedia. 2021. “Backtracking.” Backtracking. https://en.wikipedia.org/wiki/Backtracking.

Wikipedia. 2021. “Floyd–Warshall algorithm.” Floyd–Warshall algorithm. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

Wikipedia. 2021. “Dijkstra's algorithm.” Dijkstra's algorithm. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

Wikipedia. 2021. “Bellman–Ford algorithm.” Bellman–Ford algorithm. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.