FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Projeto de Conceção e Análise de Algoritmos (CAL)

**MIEIC**

## e-Stafetas: Transportes de Mercadorias


Turma 4 - tema 7:

Pedro Miguel Novais Vale - up201806083

João Pedro Ramos de Sousa - up201806496

# Índice

# Descrição Sucinta do Problema

Uma empresa de entrega de mercadorias ao domicílio optou por investir em veículos elétricos para realizar as suas entregas. A frota de veículos fica estacionada no parque da empresa, que está equipado com pontos de recarga.

A empresa tem de gerir, a cada dia, pedidos de recolha de produtos nas lojas, para depois entregar na morada dos clientes. O itinerário a ser definido para os veículos deve ter em consideração a autonomia do veículo.

# Iterações

## Iteração 1 : apenas 1 veículo na frota autonomia ilimitada e carga ilimitada

Numa primeira fase, considera-se que a frota da empresa é apenas composta por um veículo elétrico, que tem autonomia e a carga ilimitada. Assim, o objetivo é apenas de determinar qual o caminho mais curto que começa na empresa, que passa por todos os pontos de interesse e que, por fim, regresse ao ponto de partida, parque da empresa.

É importante notar que uma viagem entre dois pontos de interesse só pode ser efetuada se existirem caminhos que liguem esses pontos, em ambos os sentidos. Portanto, todos os pontos de interesse devem fazer parte do mesmo componente conexo do grafo. Esta necessidade advém do facto que, da empresa, o veículo deve conseguir alcançar todos os pontos de interesse e, depois, deve poder regressar à empresa. Assim, torna-se fundamental efetuar uma análise da conectividade do grafo.

Certas vias de comunicação podem não poder ser usadas pelo piquete, devido à existência de obras nas vias públicas, pelo que será necessário desprezar certas arestas durante o processamento do grafo.

## Iteração 2 : vários veículos na frota autonomia limitada e carga ilimitada

Numa segunda fase, considera-se que a frota da empresa é composta por vários veículos elétricos, que têm autonomia limitada e a carga ilimitada. Assim, o objetivo é determinar qual o caminho mais eficiente que começa na empresa, que passa por todos os pontos de interesse até que a bateria necessite de recarregar, combinando as atividades de todos os veículos, de modo a permitir o máximo de clientes atendidos e o minimizar o número de veículos utilizados.

## Iteração 3 : vários veículos na frota autonomia limitada e carga limitada

Numa terceira fase, para além das restrições apresentadas na Iteração 2, adicionamos a restrição da carga do veículo que faz com que o veículo não possa exceder uma certa carga, assim tem que entregar algumas das suas encomendas antes de poder receber mais.

# Formalização do problema

## Dados de entrada

**G**i - grafo, composto por:

- **V** - vértices (pontos da cidade) com:
    - R - loja que irá ser feita a recolha/morada do cliente a entregar o pedido
    - $R_{max}$ - carga da recolha a ser feita
    - Adj $\subseteq$ E - conjunto de arestas que partem do vértice
- **E** - arestas (conexão entre vértices) com:
    - w - valor da aresta (representando a distância entre dois vértices)
    - ID - identificador único da aresta
    - dest $\in$ Vi - vértice de destino

**P**i - conjunto de pedidos feito pelos clientes:

- **IDpedido** - id associado ao pedido;
- **IDloja** - id da loja que foi feita a compra;
- **IDcliente** - id do utilizador que fez o pedido;
- **carga** - tamanho da encomenda

**CL**i - conjunto de clientes:

- **IDcliente** - id associado ao cliente
- **nome** - nome do cliente
- **GPScoords** - morada do cliente

**L**i - conjunto de lojas:

- **IDloja** - id associado à loja
- nome - nome da loja
- GPScoords - morada da loja

**CR**i- conjunto de carros:

- **IDcarro** - id associado ao carro
- **autonomia** - bateria do carro
- **carga total** - carga que o carro possui

S ∈ Vi - vértice inicial (parque da empresa)

T ⊆ Vi - vértices intermédios (localização das lojas)

M ⊆ Vi - vértices finais (localização das casas)

## Dados de saída

**Gf = (Vf, Ef)** - grafo dirigido pesado, sendo Vf e Ef os mesmos atributos que V e E.

**CR**f: estrutura de todos os carros, semelhante à estrutura inicial, tendo cada um:

- ➔ **P**: sequencia ordenada de nós (pertencentes a V) que constituem o percurso que define o caminho que o carro deve efetuar, sendo que parte da localização deste no início do pedido, passa pela loja requisitada, pela morada do cliente que efetuou o pedido e acaba no parque da empresa;
- ➔ **CP**: custo do percurso efetuado pelo carro;

# Funções Objetivo

A solução ótima do problema passa por minimizar o número de carros usados e a distância total percorrida por estes, de forma a que todos os pedidos sejam entregues, minimizando o número de carros utilizados

# Restrições

## Dados de entrada

$\forall v \in V$, adj(e) $\subseteq$ E, todas as arestas adjacentes a um nó fazem parte do conjunto de arestas do grafo;

$\forall e \in E$, orig(e), dest(e) $\subseteq$ V, para todas as arestas, os seus nós de origem e destino pertencem ao conjunto de nós do grafo;

$\forall e \in E$, c(e) > 0, não há ruas com custo 0 ou negativo, pois todas têm um certo comprimento ou tempo para atravessar;

$\forall l \in L$, morada(r) $\in$ V, todos as lojas têm a sua morada no conjunto de nós do grafo;

$\forall cl \in CL$, morada(c) $\in$ V, todos os clientes têm a sua morada no conjunto de nós do grafo;

$\forall p \in P$, CL(l) $\in$ CL, todos os pedidos correspondem a um cliente registrado na plataforma (pertencente ao conjunto de clientes C);

$\forall p \in P$, L(l) $\in$ L, todos os pedidos são feitos a uma loja registrado na plataforma (pertencente ao conjunto de lojas J);

$\forall r \in L$, carga > 0, não é possível efetuar um pedido com carga vazia;

$\forall l \in CL$, W(l) $\in$ W, todos os pedidos são feitos a um carro, trabalhador da plataforma (pertencente ao conjunto de carros W);

$\forall l \in P$, CR(l) $\in$ M, todos os pedidos são efetuados num meio de transporte da e-Stafetas (pertencente ao conjunto de Meios de Transporte M);

∀w ∈ W, pos(w) ∈ V, as posições atuais dos diferentes carros correspondem a um nó do grafo.

∀m ∈ CR, vm(m) > 0, não existem veículos com velocidades médias negativas;

O conjunto de de todos os pontos úteis, isto é, ponto de posição inicial do carro, morada da loja, morada do cliente, fazem todos parte de um mesmo componente fortemente conexo do grafo. Ou seja ∀v1, v2 ∈ Vn sendo P (v1, v2) a sequência ordenada de vértices do percurso que liga v1 a v2 então P (v1, v2) != {} e P(v2, v1)!={}. Existe sempre um caminho que ligue quaisquer dois pontos úteis um ao outro.

## Dados de saída

P ⊆ V, todos os pontos de P têm que ser vértices do grafo;

Seja P0 o primeiro elemento de P, P0 = pos(W) inicial (morada do cliente que efetuou o último pedido, ou no caso de ainda não ter efetuado nenhum pedido, ponto de interesse denominado parque da empresa), pois todos os percursos partem da posição do carro;

Seja PF o último elemento de P, PF = PI(parque_da_empresa), pois todos os percursos terminam no parque da empresa;

∀i,j (P(i) ∈ P∧P(j) ∈ P ∧(ii+1=j)=> ∃e ∈adj(P(i)) ,dest(e)=P(j) , isto é, para quaisquer dois vértices de P consecutivos, são adjacentes (têm ligação entre eles);

CP > 0;

T > 0;

# Perspetiva de Solução

## Técnicas de Conceção

### Interpretação do Problema

Após uma análise do problema, solidificamos a nossa visão deste, destacando-se os tópicos:

- Um pedido é realizado por apenas um cliente e atendido por um único carro da frota. É constituído pelo tamanho da carga, e por dois vértices relativos a loja que o cliente desejou. Numa fase mais avançada, o carro terá também associado um meio de transporte com uma capacidade limitada e poderá efetuar vários pedidos em simultâneo.

- Cada carro terá associado um vértice correspondente à sua localização atual. Caso não tenha atendido ainda nenhum pedido, este vértice será o vértice relativo a um ponto inicial da localização dos carros, que designaremos parque da empresa. Caso contrário, o vértice associado ao carro será o vértice correspondente à morada do cliente a quem o carro atendeu o seu último pedido.

- Esse vértice da localização do carro será o ponto de partida do percurso deste ao atender um pedido, que deverá passar pela loja associada ao pedido, e terminar no vértice correspondente à morada do cliente que efetuou o pedido.

# Características do Grafo e Pré-processamento

O grafo que a nossa aplicação irá abordar será um grafo dirigido, tendo em conta que é uma representação de uma rede viária, pelo que as estradas poderão ter apenas um sentido. Será também um grafo pesado, uma vez que o peso das arestas não será unitário, mas variará conforme a distância ou o custo destas, conforme a escolha do cliente no pedido.

O pré-processamento dos dados terá como objetivo encontrar sempre um percurso para o carro, de modo a que todos os pedidos tenham sucesso. Para isto, teremos de tornar o grafo fortemente conexo, de modo a garantir que haverá sempre um caminho que ligue um qualquer ponto A a um qualquer ponto B e um caminho que ligue esse ponto B ao ponto A.

Uma forma de o conseguirmos seria realizar uma **Depth-First-Search** partindo do vértice inicial, e assim poderíamos remover quaisquer vértices não visitados nessa DFS, que correspondem a pontos de interesse que nunca seriam alcançáveis a partir do parque da empresa , ponto inicial da localização dos carros. Depois disto, faríamos uma DFS aos outros vértices, e terminaríamos a procura caso o parque da empresa fosse alcançado. Caso não fosse alcançada, podíamos remover o vértice, sendo que este não teria retorno para o ponto inicial. Desta forma, seriam removidos todos os vértices que não teriam pelo menos uma ligação ao parque da empresa, quer de ida, quer de volta, isto é, nos dois sentidos. Isto seria condição suficiente para que qualquer carro tenha liberdade máxima de movimento e para que qualquer pedido tivesse sucesso, ou seja, fosse sempre encontrado um percurso para este, uma vez que, no pior dos casos, esse percurso fosse a conjunção do percurso que leva o vértice A ao parque da empresa e do percurso que parte do parque da empresa para o vértice B. Logo, haveria sempre ligação entre dois quaisquer vértices do grafo.

Uma outra forma seria recorrer ao Algoritmo de **Floyd-Warshall**, que pode ser relevante para este pré-processamento do mapa de estradas. A descrição deste algoritmo será mais aprofundada na temática dos Principais Algoritmos mas, resumidamente, tem como objetivo encontrar as menores distâncias entre todos os pares de vértices num grafo dirigido pesado. Estas distâncias estariam armazenadas numa matriz de adjacências W, em que W[v1][v2] conteria o valor da menor distância percorrida desde o vértice v1 até ao v2, e W[v2][v1] conteria o valor da menor distância percorrida desde o vértice v2 até ao v1 (sentidos diferentes). Tendo isto em conta, após correr este algoritmo, poderiam ser removidos todos os vértices v1 tal que W[v0][v1] = INF (valor muito elevado em cada célula

da matriz no momento anterior ao algoritmo, exceto nas células da matriz em que v1=v2, nas quais seria atribuído o valor 0), uma vez que estes correspondem aos vértices que nunca seriam alcançáveis partindo do vértice correspondente ao parque da empresa. Da mesma forma, poderiam ser eliminados aqueles vértices em que W[v1][v0] = INF, isto é, que não teriam qualquer percurso que partisse destes e terminasse no parque da empresa. Posto isto, teriam sido removidos todos os vértices que não tivessem pelo menos uma ligação o parque da empresa, quer de ida, quer de volta, isto é, nos dois sentidos, pelo que podemos afirmar que o grafo é agora fortemente convexo. Como explicado em cima, isto seria condição suficiente para que qualquer carro tenha liberdade máxima de movimento e para que qualquer pedido tivesse sucesso, ou seja, fosse sempre encontrado um percurso para este. Este algoritmo é, no entanto, demorado, tendo em conta a sua complexidade temporal, explicada na temática dos Principais Algoritmos.

Para não tornar este processo tão demorado, poderíamos aplicar primeiro o algoritmo de DFS, descrito anteriormente, tendo como vértice origem, o parque da empresa. Assim, poderíamos logo eliminar os vértices que não foram alcançados a partir da origem, e que, portanto, nunca seriam alcançáveis pelos carros. Desta forma, reduzimos o número de vértices que o algoritmo de Floyd-Warshall tem de tratar, e não seria necessária a primeira verificação (W[v0][v1] = INF), pois estes vértices já foram removidos na DFS.

Esta abordagem será, provavelmente, a que será usada na Parte 2 do Projeto, com o acrescento de melhorias que poderão surgir durante a sua implementação.

# Principais Algoritmos

## Algoritmo de Tratamento dos Dados

A preparação dos dados é realizada facilmente assumindo que já se tem numa estrutura C as moradas dos clientes que irão efetuar os diferentes pedidos e numa estrutura V os vértices do grafo, basta percorrer uma vez os clientes e adicioná-los ao respetivo vértice, atualizando a sua informação.

## Algoritmo de Pesquisa em Profundidade

Um dos primeiros algoritmos que o programa executará será o Algoritmo de Pesquisa em Profundidade (**Depth-First Search**), isto se este for a opção escolhida para pré processamento de modo a garantir que há sempre caminhos possíveis entre dois vértices, nos dois sentidos.

Resumidamente, este algoritmo procura percorrer todos os nós filhos do nó origem o mais profundo possível para só depois retroceder, pelo que a nossa pesquisa em profundidade segue uma política de visitar sempre os nós mais profundos primeiro. Abordaremos uma versão recursiva deste, uma vez que as arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tenha arestas a sair dele. O nó origem pode ser qualquer nó do grafo, o que implica que esse nó será o nó inicial da busca.

No nosso caso, usaremos o nó correspondente ao parque da empresa como origem, ou seja, a busca partirá deste nó. Há que ter em atenção que este algoritmo pode entrar em loop e nunca terminar a sua execução se for encontrado um ciclo. Para controlar isto, cada vértice tem como atributo um boleano, que indica se já foi ou não visitado durante a pesquisa, que começa a false em todos os vértices (exceto o nó origem), e se torna true à medida que exploramos cada nó. Assim, se tivermos alcançado um nó com esse boleano a true, saberemos que o grafo tem, pelo menos, um ciclo. No final do algoritmo, os vértices que não tiverem sido visitados pela pesquisa serão os que nunca poderão ser alcançados a partir do vértice origem. Neste contexto, estes vértices devem ser eliminados, tendo em conta que não poderão ser alcançados por nenhum carro, uma vez que estes partem, num ponto inicial, da parque da empresa, ponto utilizado como origem da pesquisa.

Como o algoritmo precisa, no pior caso, de percorrer todos os vértices e todas as arestas, a complexidade temporal é O (|V| + |A|). Porém, se escolhermos utilizar a DFS para tornar o grafo fortemente conexo, como descrito no Pré-Processamento, este algoritmo será

realizado para todos os vértices do grafo, tendo como origem cada um deles, pelo que a complexidade temporal passará a ser O (|V| * (|V| + |A|)), no pior caso.

## Algoritmo de Dijkstra

Uma das hipóteses que estamos a considerar implica recorrer ao **Algoritmo de Dijkstra** para obter o percurso que se adequa ao pedido, uma vez que encontra o caminho mais curto de um vértice para outro num grafo dirigido em tempo computacional - O([arestas + vértices] * log (vértices)).

Ora, o **Algoritmo de Dijkstra** é assim uma boa escolha, em que a distância será obtida pela soma dos pesos das arestas. Trata-se de um algoritmo ganancioso, uma vez que procura maximizar o ganho imediato (neste caso, minimizar o custo ou a distância) em cada passo.

Este algoritmo não é opção para grafos com pesos negativos, uma vez que não garante a exatidão da solução nessas situações, mas tendo em conta que este não é o caso, o algoritmo é perfeitamente aplicável. É uma boa opção se o grafo for esparso (|E|~|V|), como é o caso das redes viárias.

Claramente, teremos de adaptar o Algoritmo de forma a que funcione das maneiras descritas anteriormente, isto é, respeitando todas as restrições e tendo como resultado a Função Objetivo, escolhida pelo cliente.

## Algoritmo de Floyd-Warshall

Caso seja escolhida a segunda abordagem explicada no Pré-Processamento, já teremos armazenados os dados da matriz relativos aos caminhos mais curtos entre todos os pares de vértices, pelo que estes poderiam ser aproveitados para resolver os problemas que o programa aborda.

Este algoritmo poderia ser substituído por uma execução repetida do algoritmo de Dijkstra, que teria complexidade temporal O(|V| * (|V| + |E|) * log|V|), no entanto, FloydWarshall seria uma melhor opção, tendo em conta que é melhor que o de Dijkstra se o grafo for denso ((|E|~|V|^2), e que, mesmo em grafos menos densos, pode ser melhor devido à sua simplicidade de código.

Baseia-se numa matriz de adjacências W, em que W[v1][v2] conteria o valor da menor distância percorrida desde o vértice v1 até ao v2, e W[v2][v1] conteria o valor da

menor distância percorrida desde o vértice v2 até ao v1 (sentidos diferentes). Inicialmente, cada célula da matriz é iniciada com um valor muito elevado, INF, à exceção das células em que o número da linha é igual ao da coluna, em que essa distância toma o valor 0. À medida que se percorre os vértices, estas células vão sendo atualizadas com a menor distância entre os vértices v1 e v2, pelo que no final da execução do algoritmo teremos completa a matriz com as distâncias mais curtas entre todos os pares de vértices. Serão também armazenados numa outra matriz, também bidimensional, os predecessores de cada célula (correspondente a uma aresta), de modo a poder construir, no final do algoritmo, o caminho mais curto entre dois quaisquer vértices.

A invariante do ciclo principal consiste, portanto, no facto de em cada iteração k desse ciclo, W[i][j] ter a distância mínima desde o vértice i a j, usando apenas vértices intermédios do conjunto {0, …, k}. A fórmula de recorrência do algoritmo pode ser traduzida da seguinte forma:

W[i][j] (iteração k) = mínimo(W[i][j] (iteração k-1), W[i][k] (iteração k-1) + W[k][j] (iteração k-1)) Este algoritmo tem complexidade temporal O(V^3). Já a sua complexidade espacial é O(|V|^2), uma vez que tanto a matriz que vai armazenar os valores das distâncias mais curtas como a matriz que vai armazenar os predecessores são de duas dimensões.

# Casos de utilização e funcionalidades

Temos como objetivo implementar soluções para os três níveis de problemas bem como a sua respetiva interface.

A aplicação e-Stafetas terá inicialmente um menu que permita escolher entre selecionar um mapa, operar sobre o mapa selecionado ou sair. De seguida, num outro menu, seriam apresentadas as seguintes opções:

- Visualizar o mapa recorrendo ao GraphViewer;
- Adicionar ou remover lojas;
- Adicionar ou remover carros da frota;
- Remover clientes;

Vários carros podem efetuar múltiplas entregas em simultâneo (carga limitada) (autonomia limitada):

Neste modo podem ser efetuados múltiplos logins/registos, sendo acrescentados ao grafo múltiplas novas moradas de clientes. A quando de cada um desses logins os clientes serão apresentados com a opção de efetuar os respetivos pedidos. No fim, quando não existirem mais pedidos a serem efetuados, seleciona-se a opção que permitirá ver os vários percursos efetuados pelos carros, que partem do parque da empresa, passa pela lista de lojas indicadas, na morada do último cliente atendido e acabam no parque da empresa, tendo sempre em atenção a carga e autonomia do veículo.

# Conclusão

Nesta primeira parte do projeto procuramos formalizar um sistema para gerar os percursos mais rápidos, entre as lojas e os clientes, para os e-Stafetas. Optamos por dividir o problema em 3 iterações com graus de complexidade diferentes, de forma a ter uma implementação incremental. Para cada um desses subproblemas foi procurada uma solução que ainda irá ser colocada em prática e aprofundada na segunda parte do trabalho. Para além disso, apresentamos os casos de utilização e algumas funcionalidades que iremos implementar.

A proposta de trabalho continha um intuito educativo, que requeria da nossa parte que compreendêssemos e usássemos, não só novas estruturas como grafos, mas também diferentes algoritmos de pesquisa nos mesmo abordados nas aulas.

Relativamente ao trabalho desempenhado por cada um dos elementos do grupo, procuramos entreajudar-nos e discutir cada tópico do relatório, na procura da melhor abordagem do tema, sendo que fizemos todos os tópicos juntos, no que resulta a 50% de participação de cada um.

Pensamos ter sido bem-sucedidos na realização desta primeira parte do projeto. No entanto, ao longo da sua realização tivemos algumas dificuldades ao fazer projeções para o futuro, revelando-se uma tarefa mais complicada do que antecipamos inicialmente por não sabermos ainda como irá funcionar o programa a nível prático.

2ª Parte

# Alterações à primeira parte

Verificamos que a carga não era necessária para o nosso projeto, então decidimos retira-la do carro e dos pedidos.

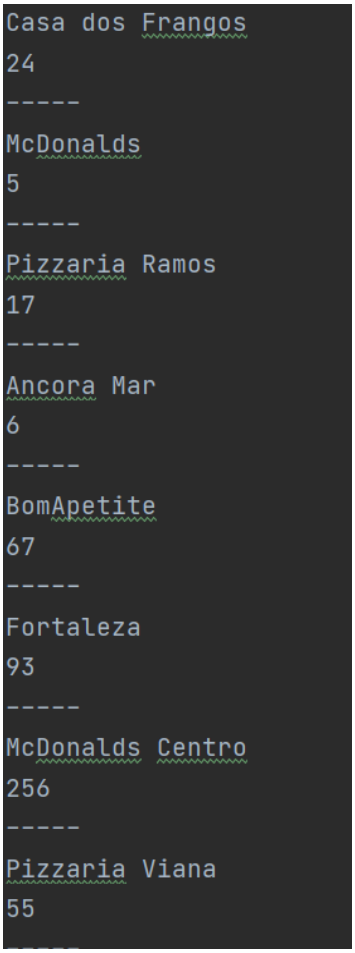
# Estruturas de dados utilizadas

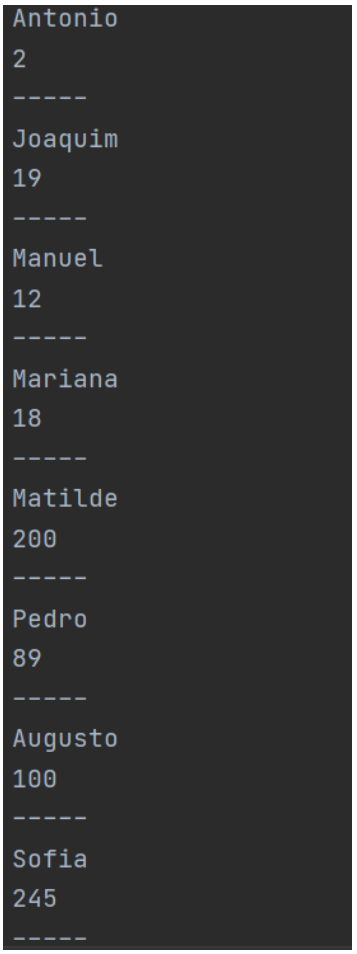
Os dados da empresa E-Estafetas são lidos a partir de ficheiros, nomeadamente o ficheiro clients.txt, stores.txt e cars.txt . Em baixo apresenta-se parte destes ficheiros, de forma a ilustrar a sua organização.

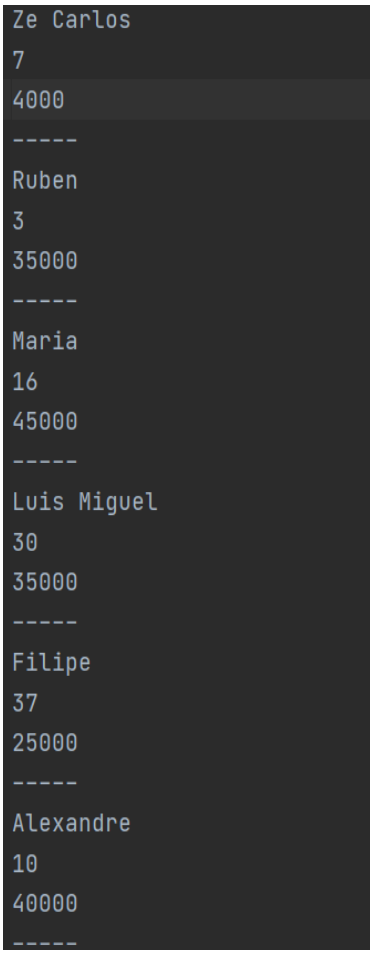
O ficheiro clients.txt contém os dados dos clientes da empresa, nomeadamente os seus nomes e id do vértice que corresponde às suas moradas.

O ficheiro stores.txt contém os dados das lojas da empresa, nomeadamente os seus nomes e id do vértice que corresponde às suas localizações.

O ficheiro cars.txt contém os dados dos carros da empresa, nomeadamente os seus nomes e id do vértice que corresponde às suas posições.



**Grafo**

A classe “Graph” presente na pasta Graph.h é essencialmente uma adaptação da classe fornecida nas aulas, e contém algoritmos que fomos desenvolvendo durante as aulas práticas, para além de outros métodos que sentimos necessidade de criar durante a implementação. O programa possuirá uma instância desta classe, que conterá todos os vértices e arestas do grafo escolhido pelo utilizador no primeiro Menu.

**Vertex**

A classe “Vertex” presente também na pasta “Graph.h” corresponde a um vértice do grafo, que no nosso contexto simboliza pontos do mapa. Adicionamos um atributo inteiro “type”, que especifica a importância do Vertex para o nosso contexto, podendo ser dos tipos:

- 0, se não tiver interesse para o contexto, ou seja, se for apenas um ponto de deslocação do carro.

- 1, se corresponder à morada de um cliente

- 2, se corresponder à localização de uma loja

- 3, se corresponder à posição de um carro

**Edge**

A classe “Edge” também presente no ficheiro “Graph.h” trata-se da ligação entre dois vértices, pelo que corresponde no nosso contexto ao caminho entre dois pontos do mapa.

**Request**

Foi criada uma classe para representar as entregas/encomendas que teriam de ser feitas.

Cada pedido é composto por um Client que o efetuou, a Store que este cliente desejou e o Car que irá atender o pedido. A escolha do carro que atende o pedido é feita pelo programa conforme um critério.

**Client**

Esta classe representa um cliente existente na empresa. É constituída por um nome e uma morada que corresponde à informação de um dos vértices do grafo.

**Store**

Semelhante à classe anterior esta classe representa uma loja da plataforma. É constituída por um nome e morada que, mais uma vez, corresponde à informação de um dos vértices do grafo.

**Car**

Semelhante à classe anterior esta classe representa um carro da plataforma. É constituída por um nome e morada que, mais uma vez, corresponde à informação de um dos vértices do grafo.

**Company**

Por fim criamos uma classe Company que contém toda a informação dos pedidos que estão a ocorrer num determinado momento (vetor “requests”) bem como os clientes, lojas e carros que pertencem à plataforma.

# Casos de Utilização

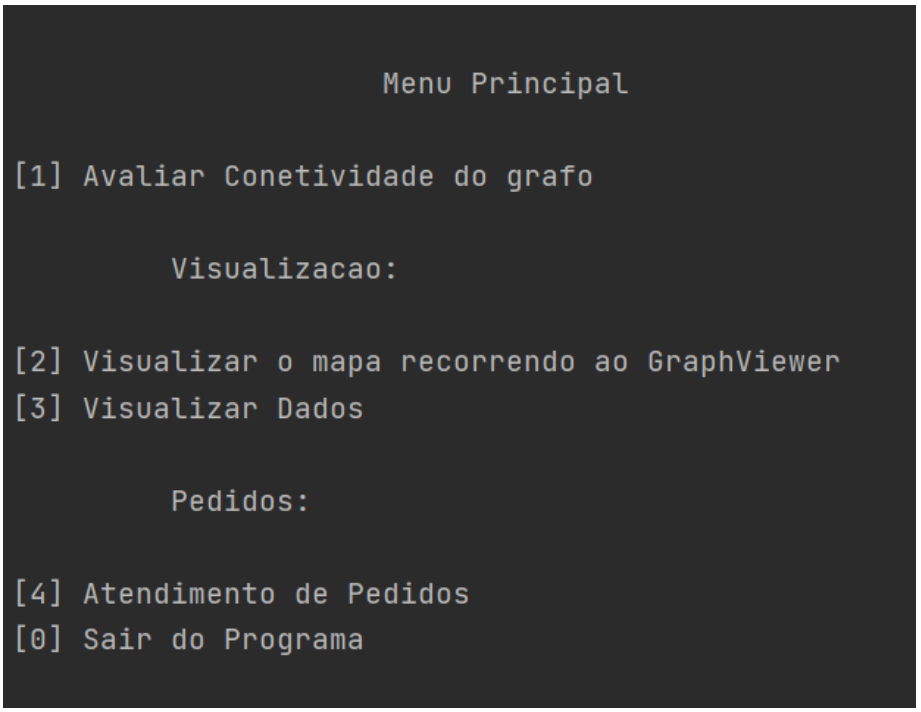
### Menu de Escolha do mapa

Neste menu é dada a possibilidade ao utilizador de escolher um mapa no qual vão ser realizados todos os algoritmos e soluções implementadas.


### Menu Principal

Neste menu apresentamos as seguintes possibilidades:

1. Avaliar a Conectividade do Mapa.
2. Visualizar o mapa recorrendo ao GraphViewer com os respetivos Clientes (verde), Carros(laranja) e Lojas(vermelho) da aplicação.
3. Visualizar os dados, ou seja, todos os clientes, lojas e carros da Company, permitindo um maior controlo dos dados.
4. E por último efetuar pedidos, que permite visualizar no GraphViewer o mapa e os carros a efetuar os pedidos.




**Menu de Escolha dos Pedidos**

Nesta parte aparecem todos os clientes e lojas disponíveis, onde as vamos selecionar para efetuar pedidos.

# Conectividade

De modo a facilitar a fase inicial sem remover totalmente a praticidade do problema, optamos por adicionar uma *flag* no início do programa (*bidiretional_edges*) que nos permite carregar e correr os algoritmos em modo grafo dirigido ou com arestas bidirecionais.

Assim sendo, o teste de conetividade implementado na opção “Avaliar Conetividade” foi não só uma simples verificação de que o ponto final e o ponto inicial se encontram na mesma componente fortemente conexa, através de um método de Pesquisa em Profundidade com começo num vértice definido como ponto de referência (antes denominado de Parque da Empresa). Mas também o cálculo da Componente Fortemente Conexa (CFC) que contém este ponto.

Devido à possibilidade de o grafo ser ou não dirigido, o cálculo da CFC pode ser feito de duas formas, uma mais eficiente que a outra: caso base é o primeiro algoritmo descrito na primeira parte deste trabalho (Pesquisa em Profundidade -> inverter Grafo -> Pesquisa em Profundidade). Por outro lado, como no caso as arestas sejam bidirecionais é como se o grafo já estivesse invertido, este algoritmo pode ser reduzido a uma simples Pesquisa em Profundidade.

Optamos ainda por não remover os vértices que não se encontram na mesma CFC que o ponto de referência inicial. A cada algoritmo é realizada uma nova Pesquisa em Profundidade que parte do vértice da posição atual do carro e são guardados todos os vértices provenientes da pesquisa num vetor.

# Algoritmos implementados e respetiva análise

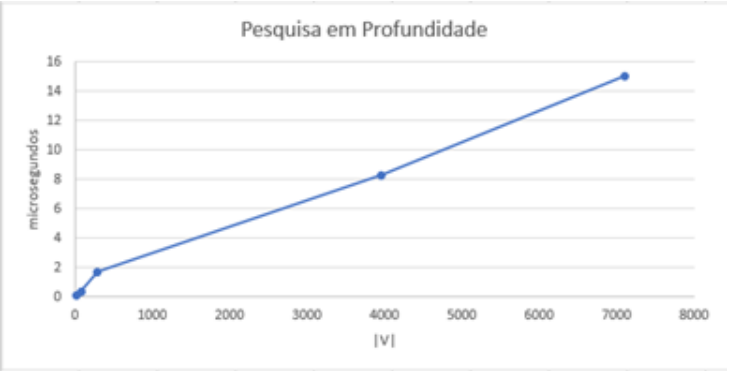
A complexidade teórica dos algoritmos implementados já foi abordada na Parte 1 deste trabalho, pelo que, de forma a não nos tornarmos repetitivos, apenas faremos a análise da complexidade temporal empírica a partir dos resultados obtidos e apresentaremos o pseudocódigo dos algoritmos usados. De notar que os tempos não podem ser comparados em valor absoluto entre algoritmos uma vez que diferentes algoritmos são corridos sob diferentes condições. Os tempos serão medidos em milissegundos podendo mudar a variável independente.

**Pesquisa em Profundidade**

Este algoritmo não sofreu qualquer alteração em relação ao planeado, por isso achamos apenas relevante apresentar uma possível implementação através do seu pseudo código e os resultados obtidos dos testes efetuados. É utilizado na análise da conectividade do grafo.

```
DFS(G, v_initial):
        Vector<Vertex> result
        For each v pertencente a V
                Visited(v) <- false
        DFS_VISIT (v_initial, result)
        Return result;
DFS_VISIT(v_initial, vector<Vertex> result):
        Visited(v)<- true
        Result<-v
        for each w pertencente a Adj(v)
                if not visited(w)
                        DFS_VISIT(w,result)
```

Abaixo encontra-se o gráfico com os valores obtidos a partir de 5 mapas-grafos disponibilizados pelos professores (não foi corrido nos mapas originais de Portugal devido à sua dimensão). Os grafos não são, contudo, aleatórios e há um salto grande no número de vértices dos Grid para o de Penafiel e do de Penafiel para Espinho, fazendo com que os resultados não sejam categóricos.




**Componentes Fortemente Conexas**

Quando se trata de um grafo não dirigido, o algoritmo para obter a componente fortemente conexa de um grafo passa apenas por realizar uma Pesquisa em Profundidade (DFS) pelo que a sua análise seria redundante.

Pelo contrário, em grafos dirigidos, a análise da componente fortemente conexa é mais complexa e leva à necessidade de duas pesquisas em profundidade. Uma que será realizada no início, de seguida fazemos uma inversão do grafo (uma vez que não guardamos antecedentes em cada vértice) e uma pesquisa no fim. Obtemos então o seguinte pseudocódigo para um grafo G = (V, E) e vértice origem v pertencente a V:

**Analisar_Conetividade(G,v):**

vector<Vertex> solution1 = **DFS**(G,v);

graphInverted= invert(G);

vector<Vertex> solution2 = **DFS** (graphInverted,V)

vector<Vertex solution=intersect (solution1, solution2);

return solution;

Quanto à complexidade temporal do algoritmo de pesquisa em profundidade sabemos que é |V|, pelo que, ao ser feito duas vezes, equivale a 2|V|. A inversão do grafo é efetuada simplesmente percorrendo todos os vértices (|V|) e as suas respectivas arestas (|E|) invertendo as direções das mesmas. Assim podemos concluir que a complexidade deste algoritmo é |V|+|E|. Por último é efetuada uma interseção dos dois grafos que consiste em percorrer DFS do grafo normal e comparar com todos com todos os vértices da DFS do grafo invertido. Esta operação pode variar muito a sua complexidade uma vez que depende das soluções encontradas por cada pesquisa de profundidade. E poder-se-ia ter encontrado uma forma mais eficiente de a fazer minimizar estes custos.

**Algoritmo de Dijkstra**

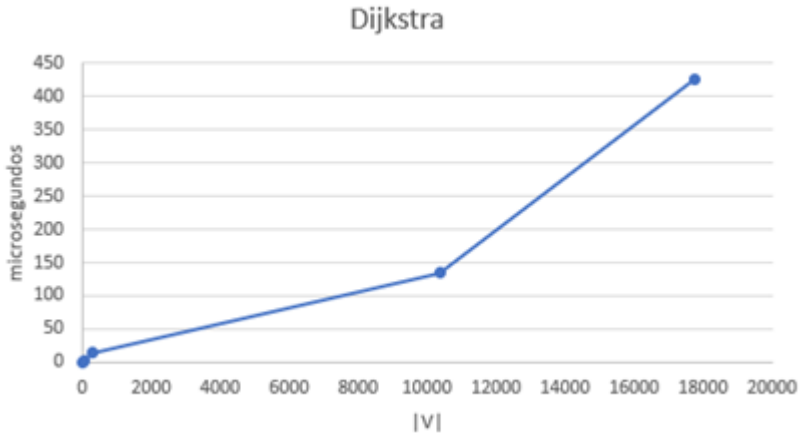
Para descobrirmos o caminho mais próximo entre dois vértices do grafo, decidimos que o melhor e mais simples algoritmo a utilizar seria o Algoritmo de Dijkstra, devido a termos de enfrentar várias vezes esse problema de achar o caminho mais curto entre dois pontos do mapa, pelo que a complexidade temporal relativamente baixa desta algoritmo nos tenha induzido a usá-lo. Uma descrição mais profunda deste foi realizada na primeira parte do relatório, pelo que resta apresentar o seu pseudocódigo, para um grafo G = (V, E) e vértice origem s pertencente a V:

```
for each v ∈ V do:                          // Tempo de execução: O ((|V| + |E|) * log(|v|)
    dist(v) ßINF
    path(v) ßNULL
    dist(s) ß 0
    Q ß ⌀
    INSERT (Q, (s, 0))                      // Insert s with key 0
    While (Q ≠ ⌀) do
        v ßExtractMin(Q)                    // algoritmo ganancioso
        for each w ∈ Adj(v) do
            if (dist(w) > dist(v) + weight (v, w) then
```

```
dist(w) ßdist(v) + weight (v, w)
path(w) ß v
if (w ∉ Q) then                    // old dist(w) was INF
    INSERT (Q, (w, dist(w)))
else
    DECREASE-KEY (Q, (w, dist(w)))
```

Análise do Algoritmo:

A análise deste algoritmo foi já detalhada na primeira parte deste relatório, pelo que apenas realçamos a sua complexidade temporal: O ([arestas + vértices] * log (vértices)). O facto de possuir baixa complexidade temporal levou-nos a escolher este algoritmo, para calcularmos o menor percurso entre dois vértices. No gráfico a baixo apresentamos o gráfico com os valores obtidos a partir 5 grafos-mapas.




Assim, após aplicarmos o algoritmo que criamos, do qual resulta num vetor dos pontos de interesse pelos quais o carro vai passar, recorremos a este algoritmo para encontrar o caminho mais curto entre cada par desses pontos. Deste modo, possuímos no final um vetor com o percurso completo do carro.

**Algoritmo do Vizinho Mais Próximo Adaptado**

Este algoritmo está contido na função de encontrar os caminhos feitos pelo carro para atender aos pedidos. Foi desenvolvido por nós, e segue a lógica do Algoritmo do Vizinho Mais Próximo (Nearest Neighbor Algorithm), uma vez que procuramos sempre o vertex que tem a menor soma das seguintes distâncias: distância a que se encontra do ponto atual e distância que esse esse ponto tem do parque da empresa, devido ao facto de o carro ter sempre que voltar ao parque para recarregar. No entanto, esta abordagem não seria suficiente, pois temos de garantir que o carro passa pela loja X antes de entregar o pedido desta loja ao Cliente X e o carro tem um limite de distância que pode percorrer.

O objetivo deste algoritmo é encontrar uma lista ordenada dos pontos por onde o carro tem de passar, de modo a que seja o percurso mais curto.

Recorremos a três vetores: o "**result**", um vetor de Vertex<T>*, que conterá os pontos dos pedidos (lojas e clientes) de forma a que seja o menor percurso do carro, o vetor "**stores**", que conterá a cada momento as lojas pelos quais o carro pode passar (numa fase inicial contém todos eles, pois não há restrições para as lojas), e o vetor "**clients**", que conterá a cada momento os clientes pelos quais o carro pode passar (é inicializado vazio, e de cada vez que é adicionado uma loja ao "**result**", adiciona-se ao "clientes" o cliente que requisitou essa loja, uma vez que o carro pode já passar por ele, pois já requisitou o seu pedido na loja).

O algoritmo começa então por adicionar todas as lojas dos pedidos ao vetor "**stores**", e é realizado num ciclo, enquanto houver pontos por adicionar (termina quando o vetor "**stores**" e o vetor "**clients**" estão ambos vazios). Uma variável "**baterry**" é inicializada com o valor da autonomia do carro que estamos a tratar.

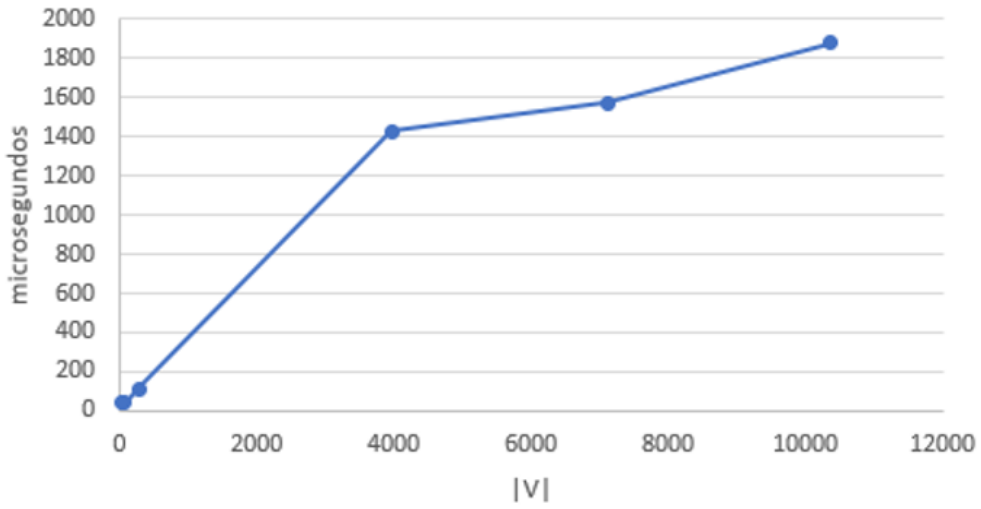
Depois, verificamos se "**result**" está vazio. Se estiver, procuramos a loja mais próxima e removemo-la do vetor "**stores**". Assim verificamos se o percurso: Parque->Loja(a analisar)->Cliente(do pedido)->Parque é possível, ou seja, se a distância desse percurso é menor que "**baterry**", a autonomia do carro. Se for esse o caso, adicionamos o vertex associado à loja ao "**result**". Se não estiver vazio, iremos encontrar qual é o ponto seguinte a procurar, através do vertex que tem a menor soma da distância a que se encontra do ponto atual e distância que esse esse ponto tem do parque da empresa. Este ponto será entre uma loja do vetor "**stores**" ou um cliente do vetor "**clients**".

Se este ponto for uma loja, removêmo-la de “**stores**” e adicionamos o cliente associado à loja a um vetor auxiliar “**auxclient**”, junto com todos os clientes que faltam atender. Ordenamos este vetor pelo critério da distância a que se encontra do ponto atual e distância que esse esse ponto tem do parque da empresaDepois e criamos um vetor auxiliar de percurso “**auxordering**” com os ponto presentes em “**result**”, a loja a analisar e todos estes clientes ordenados. Verificamos se a distância deste percurso é menor que a autonomia do carro e se for adicionamos a loja a “**result**” e o cliente associado a ao vetor “**clients**”.

Se o ponto for um cliente, verificamos se o percurso em “**result**” mais o cliente a analisar é menor que a autonomia do carro. Se for adicionamos “**result**”.

Posto isto, após o final do algoritmo, quando o ciclo while terminar, teremos o vetor ordenado de forma a respeitar todas as condições impostas, e de forma ser o percurso mais curto que o carro tem de percorrer.

Posteriormente, é utilizado o Algoritmo de Dijkstra de modo a encontrar os caminhos mais próximos entre cada dois pontos desse vetor, construindo assim um vetor resultante com o percurso completo do carro(vetor percurso), que é retornado.



**Algoritmo final utilizado**

Inicia-se com a apresentação dos vários clientes e lojas, e pede ao utilizador que efetue quantos pedidos quiser, escolhendo para cada pedido uma loja e um cliente. Posto isto, temos como dados os pedidos que o utilizador efetuou.

Verificamos se é possível realizar o pedido, ou seja, se existem caminhos possíveis para tal e se isso se verificar, adicionámo-lo aos "**requests"** da Company.

É apresentado todos os pedidos possíveis e ao percorrer o vetor de carros da Company, chamamos o algoritmo descrito anteriormente, que retorna o percurso completo realizado pelo carro.

Cada percurso de cada carro vai sendo adicionado a um vetor percursos, que será enviado depois a uma função showMultiplePathsGV(), que mostra com recurso ao Graph Viewer todos os caminhos efetuados pelos vários carros.

# Conclusão

Em suma, nesta segunda parte do projeto procuramos implementar um sistema para gerar os percursos mais rápidos, entre as lojas e os clientes, para as lojas da aplicação E-estafetas. Para além disso, conseguimos implementar os casos de utilização e funcionalidades descritas na primeira parte.

Acreditamos que compreendemos não só novas estruturas como grafos, mas também diferentes algoritmos abordados nas aulas.

Relativamente ao trabalho desempenhado por cada um dos elementos do grupo, procuramos entreajudar-nos e discutir cada tópico do relatório, na procura da melhor abordagem do tema, sendo que fizemos a maioria dos tópicos juntos, no que resulta a 50% de participação de cada um.

# Bibliografia

Para esta segunda etapa (implementação do código e continuação deste relatório), não foi necessário consultar outras fontes para além das especificadas na bibliografia da primeira parte deste relatório.

Destacam-se, no entanto, os slides apresentados durante as aulas teóricas de CAL, que consultamos repetidamente, para a implementação dos algoritmos desenvolvidos nas aulas práticas e para o conhecimento de lógicas de outros algoritmos que pudéssemos usar, nomeadamente a lógica do Nearest Neighbor Algorithm, que abordamos durante a criação do nosso algoritmo para determinação do vetor ordenado de pontos de interesse, já descrito em cima.