U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto de Conceção e Análise de Algoritmos (CAL)

MIEIC

EatExpress: Entrega de Comida

Trabalho realizado por:

Turma 4 - Grupo 7 **24/04/2020**

Mariana Oliveira Ramos up201806869@fe.up.pt
Paulo Jorge Salgado Marinho Ribeiro up201806505@fe.up.pt
Rita Sofia Araújo Sá Lopes da Silva up201806527@fe.up.pt

Índice

1ª Parte

2ª Parte

Introdução

No âmbito da unidade curricular Conceção e Análise de Algoritmos (CAL) do Mestrado Integrado em Engenharia Informática e Computação (MIEIC), será desenvolvida uma aplicação de entrega de comida: EatExpress.

Esta aplicação irá processar pontos de partida, recolha e chegada, marcados num mapa de acordo com a localização dos utilizadores, restaurantes bem como dos estafetas.

A aplicação será desenvolvida em C++, com recurso à API GraphViewer para se desenhar o grafo gerado para os diferentes trajetos. Os mapas utilizados serão extraídos da plataforma OpenStreetMaps.

Neste documento será apresentada uma breve descrição do problema bem como a sua formalização e uma proposta de implementação.

Descrição do Tema

A EatExpress é um sistema de entrega de comida entre os restaurantes registados na plataforma e os utilizadores da sua aplicação. Acedendo à aplicação o utilizador pode escolher o restaurante e o prato que deseja. A refeição escolhida é, de seguida, entregue na morada do cliente por um estafeta que utiliza o seu próprio meio de transporte da empresa para lá chegar (a pé, bicicleta, mota ou carro).

Por ser uma rede com vários restaurantes e vários clientes, existem vários percursos que o estafeta pode fazer para entregar a comida ao cliente. Sendo, portanto, necessário avaliar várias hipóteses de forma a otimizar a distância e o tempo de viagem de cada entrega. Coloca-se ainda a hipótese de um cliente ser inacessível a partir de qualquer restaurante, o que requer uma análise de conetividade entre restaurantes e os utilizadores.

Aquando da conclusão do projeto deverá ser possível, para uma dada lista de restaurantes e utilizadores, fazer a distribuição das entregas através dos estafetas de forma otimizada em termos de distância e tempo.

Por forma a ter uma implementação incremental podemos subdividir o problema em variantes com diferentes graus de complexidade:

1. **Apenas existe um estafeta que realiza um pedido de cada vez**

Numa primeira fase considera-se que a plataforma EatExpress tem disponível apenas um estafeta que terá de realizar uma entrega. Nesta situação, o problema reduz-se a encontrar o trajeto mais curto, com início na posição do estafeta quando é realizado o pedido (esta posição é a morada do cliente onde este realizou o último pedido ou a "Casa dos Estafetas"), que passe pelo restaurante indicado, e termine na morada do cliente que efetuou o pedido. Ao estafeta é lhe associado um meio de transporte com uma velocidade média.

2. Apenas um estafeta que pode realizar múltiplos pedidos em simultâneo (carga limitada) (carga ilimitada)

Nesta segunda fase, semelhante ao caso anterior, vamos continuar a considerar apenas um estafeta. No entanto este irá realizar múltiplos pedidos em simultâneo. Isto é, um estafeta pode ser encarregue de um certo número de pedidos que terá de atender numa só deslocação. Assim este irá realizar um trajeto que passe por todos os restaurantes e moradas de clientes dos pedidos selecionados, garantindo sempre que o restaurante do cliente

respetivo é atingido primeiro que este. Nesta fase iremos então acrescentar um novo fator que é a carga máxima do meio de transporte de cada estafeta.

3. Múltiplos estafetas a atender os pedidos (carga ilimitada)

Nesta terceira fase consideramos, ao contrário das fases anteriores, múltiplos estafetas que estão a trabalhar para a aplicação. Assim cada estafeta atende um ou vários pedidos. A escolha do estafeta que atenderá um pedido será realizada em torno de um critério ainda a escolher, mas que poderá ser, por exemplo, o estafeta que estiver mais perto do restaurante requisitado no pedido.

4. Múltiplos estafetas a atender os pedidos (carga limitada)

Nesta quarta fase vamos assumir o problema na íntegra, ou seja, a aplicação tem ao seu dispor vários estafetas cada um com o seu meio de transporte com capacidade limitada. É importante salientar que, ao ter disponível diferentes estafetas, não implica que que todos tenham que estar a circular. Cada estafeta atende um ou vários pedidos diferentes, realizando o mesmo caminho que na primeira ou segunda etapas.

Tal como na terceira iteração o critério de escolha de um estafeta para realizar um pedido ainda irá ser definido. Poderá novamente incidir no que estiver mais perto do restaurante requisitado no pedido, ou por exemplo o que tem mais carga ainda disponível.

Adicionalmente, em qualquer das versões do problema, é necessário antes de procurar definir qualquer percurso, a análise das características subjacentes aos dados. Por exemplo, a conetividade do grafo tem que ter sida em conta como já foi mencionado. Caso um destino de um cliente ou restaurante seja inacessível, esse pedido não deve fazer parte das possíveis rotas dos estafetas, e o utilizador deve ser notificado desta ocorrência. Estas situações serão analisadas num pré-processamento dos dados que será detalhado nos capítulos seguintes.

Formalização do Problema

A solução deste problema passa pela formalização do mesmo em grafos e resolução recorrendo a algoritmos com estas estruturas.

Dados de entrada

De modo a uniformizar a notação procedeu-se às seguintes definições:

G (V, E): grafo dirigido pesado com informação sobre a rede viária em causa. Composto por:

- **V**: conjunto de todos os nós de G, que representam pontos de interesse do mapa, como pontos de turismo, de cuidados de saúde, entre outros, sendo que os que nos interessam são os relacionados com gastronomia, nomeadamente restaurantes. Cada nó apresenta os seguintes atributos:
  - ➔ id - id do nó;
  - ➔ adj - conjunto contido em E, representa o conjunto de arestas (estradas que saem de um determinado nó);
- **E**: conjunto de todas as arestas de G, em que cada aresta representa o caminho entre dois pontos de interesse. Cada aresta apresenta os seguintes atributos:
  - ➔ orig - pertencente a V, representa o nó de partida da aresta;
  - ➔ dest - pertencente a V, representa o nó de chegada de E;
  - ➔ c - custo de percorrer a aresta (quer seja a distância, tempo, combustível, portagens, os algoritmos otimização para a propriedade correspondente);
  - ➔ unic - booleano que indica se uma rua é de sentido único ou não;

R: estrutura de todos os restaurantes inscritos na plataforma, tendo cada um:

- ➔ morada - morada do restaurante, pertencente a V;
- ➔ nome - nome do restaurante;
- ➔ descrição - breve descrição do restaurante e tipos de culinária;
- ➔ menu - conjunto dos pratos disponíveis em cada restaurante;

C: estrutura de todos os clientes registrados na EatExpress, tendo cada um:

- ➔ morada - morada do cliente, pertencente a V, para onde é enviado o pedido que este possa efetuar;
- ➔ nome - nome do cliente;
- ➔ nif - nif do cliente;

L: Estrutura com os diferentes pedidos, tendo cada um:

- ➔ C - cliente que efetuou o pedido;
- ➔ R - restaurante a que foi efetuado o pedido;
- ➔ order - conjunto de pratos pedidos;
- ➔ W - estafeta que vai efetuar o pedido;
- ➔ M - meio de transporte no qual o estafeta vai efetuar o pedido;

W: estrutura de todos os estafetas, tendo cada um:

- ➔ nome - nome do trabalhador na plataforma;
- ➔ nif - nif do trabalhador;
- ➔ pos - posição atual do estafeta;
- ➔ n_pedidos - número de pedidos efetuados por cada estafeta;
- ➔ d_total - número total de quilómetros efetuados por um estafeta;
- ➔ transporte - meio de transporte do estafeta;

M: estrutura de meios de transporte disponíveis na empresa, tendo cada um:

- ➔ nome - nome do veículo;
- ➔ vm - velocidade média do veículo;
- ➔ capacidade - capacidade total do veículo (número de pedidos);

Dados de saída

Os dados de saída, isto é, dados obtidos a partir dos dados de entrada para uso no programa são:

Wf: estrutura de todos os estafetas, 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 estafeta deve efetuar, sendo que parte da localização deste no início do pedido, passa pelo restaurante requisitado, e termina na morada do cliente que efetuou o pedido;
- ➔ **CP**: custo do percurso efetuado pelo estafeta;
- ➔ **T**: tempo que o estafeta leva a efetuar esse percurso no seu meio de transporte;

Funções Objetivo

O objetivo do programa dependerá da escolha do cliente relativa ao seu pedido, e pensamos em duas hipóteses de escolha:

- **Percurso mais barato** - O estafeta efetuará o percurso entre a sua posição inicial e a morada do cliente que efetuou o pedido, passando pelo restaurante requisitado, com menor custo monetário. (menor CP, isto é, min (∑ c)).
- **Percurso menos demorado** - O estafeta efetuará o percurso entre a sua posição inicial e a morada do cliente que efetuou o pedido, passando pelo restaurante requisitado, com menor duração. (menor T, isto é, min (∑ t), em que t depende da velocidade do meio de transporte utilizado, Vm, e da distância da aresta).

Assim a resolução dos vários problemas passará pela minimização da função f=C (nó início,nó fim), sendo C o critério escolhido.
A ter em atenção que selecionando a opção “percurso mais barato” e existindo vários percursos com custos mínimos iguais recorremos ao menos demorado dentro destes. O mesmo se reflete para a opção “percurso menos demorado”. Na possibilidade de existirem vários percursos com tempo mínimo igual optamos pelo mais barato.

Restrições

Nos dados de Entrada

∀v ∈ V, adj(e) ⊆ E, todas as arestas adjacentes a um nó fazem parte do conjunto de arestas do grafo;

∀e ∈ E, orig(e), dest(e) ⊆ V, para todas as arestas, os seus nós de origem e destino pertencem ao conjunto de nós do grafo;

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

∀r ∈ R, morada(r) ∈ V, todos os restaurantes têm a sua morada no conjunto de nós do grafo;

∀r ∈ R, menu(r).size > 0, todos os restaurantes tem pelo menos um prato no menu;

∀c ∈ C, morada(c) ∈ V, todos os clientes têm a sua morada no conjunto de nós do grafo;

∀l ∈ L, C(l) ∈ C, todos os pedidos correspondem a um cliente registrado na plataforma (pertencente ao conjunto de clientes C);

∀l ∈ L, R(l) ∈ R, todos os pedidos são feitos a um restaurante registrado na plataforma (pertencente ao conjunto de restaurantes R);

∀r ∈ R, order(r).size > 0, não é possível efetuar um pedido sem conjunto de pratos;

∀l ∈ L, W(l) ∈ W, todos os pedidos são feitos a um estafeta, trabalhador da plataforma (pertencente ao conjunto de estafetas W);

∀l ∈ L, M(l) ∈ M, todos os pedidos são efetuados num meio de transporte da EatExpress (pertencente ao conjunto de Meios de Transporte M);

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

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

O conjunto de de todos os pontos úteis, isto é, ponto de posição inicial do estafeta, morada do restaurante, 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.

Nos dados de Saída

$P \subseteq V$, 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 “Casa dos Estafetas”), pois todos os percursos partem da posição do estafeta;

Seja PF o último elemento de P, PF = morada(C), pois todos os percursos terminam na morada de um cliente;

$\forall i,j\ (P(i) \in P \wedge P(j) \in P \wedge (ii+1=j) \Rightarrow \exists e \in adj(P(i))\ , dest(e)=P(j)$, 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 estafeta. É constituído pelo prato(s) que foram pedidos, e por dois vértices relativos ao restaurante que o cliente desejou. Numa fase mais avançada, o estafeta terá também associado um meio de transporte com uma capacidade limitada e poderá efetuar vários pedidos em simultâneo.
- Cada estafeta 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 estafetas, que designaremos “Casa dos Estafetas”. Caso contrário, o vértice associado ao estafeta será o vértice correspondente à morada do cliente a quem o estafeta atendeu o seu último pedido.
- Esse vértice da localização do estafeta será o ponto de partida do percurso deste ao atender um pedido, que deverá passar pelo restaurante associado 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 estafeta, 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 da “Casa dos Estafetas”, ponto inicial da localização dos estafetas. Depois disto, faríamos uma DFS aos outros vértices, e terminaríamos a procura caso a “Casa dos Estafetas” fosse alcançada. 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 à “Casa dos Estafetas”, quer de ida, quer de volta, isto é, nos dois sentidos. Isto seria condição suficiente para que qualquer estafeta 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 à “Casa dos Estafetas” e do percurso que parte da “Casa dos Estafetas” para o vértice B. Logo, haveria sempre ligação entre dois quaisquer vértices do grafo.

Uma outra forma seria recorrendo 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 à “Casa dos Estafetas”. 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 na “Casa dos Estafetas”. Posto isto, teriam sido removidos todos os vértices que não tivessem pelo menos uma ligação à “Casa dos Estafetas”, 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 estafeta 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 a “Casa dos Estafetas”. Assim, poderíamos logo eliminar os vértices que não foram alcançados a partir da origem, e que, portanto, nunca seriam alcançáveis pelos estafetas. 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 irá ser, provavelmente, a que será usada na Parte 2 do Projeto, com o acrescento de melhorias que nos poderão surgir durante a sua implementação.

Atendimento de um Pedido

Associado a um pedido está o cliente que o realizou, o estafeta que o atenderá, o restaurante em que será feito o levantamento dos pratos pedidos. Numa fase posterior, em que haverão vários estafetas, a atribuição do estafeta que atenderá o pedido será feita com base num critério ainda a escolher, mas que poderá ser o número de pedidos atendidos (é escolhido o que tiver um menor número), ou a distância por eles percorrida, (é escolhido o que tiver menor distância), por exemplo.

Numa fase mais avançada, a empresa terá também associado um conjunto de meios de transportes, e a cada pedido será também associado um meio de transporte, que será utilizado pelo estafeta, e ao qual está associado uma certa velocidade média. A escolha do meio de transporte a atribuir ao pedido cairá sobre aquele que terá maior velocidade, de entre os disponíveis.

O Atendimento de um Pedido por um estafeta subdivide-se em dois problemas de cálculo do melhor caminho entre dois pontos: o primeiro consiste em calcular o melhor caminho entre o ponto de partida do estafeta e o restaurante especificado no pedido, e o segundo em calcular o melhor caminho entre esse restaurante e o vértice correspondente à morada do cliente que realizou o pedido. Para obter o percurso completo realizado pelo estafeta, basta fazer a junção desses dois caminhos.

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 à “Casa dos Estafetas” 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 estafeta, uma vez que estes partem, num ponto inicial, da “Casa dos Estafetas”, 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, Floyd-Warshall 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 quatro níveis de problemas bem como a sua respetiva interface.

A aplicação EatExpress 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 restaurantes;
- ✓ Adicionar ou remover estafetas;
- ✓ Remover clientes;
- ✓ Resolver as três variantes do problema .

Nesta última opção encontrar-se-á um último menu para escolher entre os vários problemas que descrevemos acima e as diferentes implementações e soluções dos mesmos:

- ✓ Apenas um estafeta a realizar uma única entrega:

Neste modo é pedido ao utilizador que efetue o login, caso já esteja registrado na EatExpress, ou registar-se com uma morada que será adicionada ao grafo caso contrário.

De seguida este pode escolher efetuar um pedido onde terá de escolher um restaurante e um conjunto de pratos. Serão ainda apresentadas duas opções entre as quais o cliente pode escolher: **percurso mais barato ou percurso menos demorado**. Por último ser-lhe-á mostrado o caminho mais curto/barato que parte da posição do único estafeta da plataforma passa pelo restaurante indicado no pedido e termina na morada do cliente.

- ✓ Um estafeta pode efetuar múltiplas entregas em simultâneo (carga limitada) (carga ilimitada):

Neste modo podem ser efetuados múltiplos logins/registos como na primeira etapa, 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, como no primeiro caso, o percurso mais curto/barato que parte da posição do estafeta,

passa pela lista de restaurantes indicados e termina na morada do último cliente atendido, tento sempre em atenção a **carga do veículo**.

- ✓ Múltiplos estafetas a atender os pedidos (carga ilimitada):

Nesta opção o utilizador irá realizar os mesmos passos que no modo anterior, no entanto estarão disponíveis diferentes estafetas com diferentes meios de transporte que os vão realizar. Deste modo será visualizado o percurso dos diferentes estafetas a atenderem os diferentes pedidos/conjunto de pedidos. Baseando-se em combinações de algoritmos utilizados nos casos anteriores, e sem qualquer preocupação com carga, isto é, número máximo de pedidos que podem transportar.

- ✓ Múltiplos estafetas a atender os pedidos (carga limitada):

Nesta última opção o utilizador irá realizar os mesmos passos que no modo anterior, no entanto estarão disponíveis diferentes estafetas com diferentes meios de transporte com **cargas respetivas** que os vão realizar. À semelhança do anterior será visualizado o percurso dos diferentes estafetas a atenderem os diferentes pedidos/conjunto de pedidos.

Conclusão

Em suma, nesta primeira parte do projeto procuramos formalizar um sistema para gerar os percursos mais rápidos, entre os restaurantes e os clientes, para os estafetas da aplicação EatExpress. Optamos por dividir o problema em 4 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. Os tópicos em que cada elemento se dedicou especialmente mais são:

Paulo Ribeiro – Capa, Descrição, Funções Objetivo, Interpretação do Problema, Características do Grafo e Pré-Processamento, Atendimento de um Pedido, Principais Algoritmos (Algoritmo de Pesquisa em Profundidade, Algoritmo de Dijkstra, Algoritmo de Floyd-Warshall).

Mariana Ramos – Descrição e divisão do problema em diferentes iterações, formalização do problema (dados de entrada, dados de saída, restrições), casos de utilização e funcionalidades, conclusão.

Rita Silva – Introdução, Descrição e divisão do problema, Principais Algoritmos (Algoritmo de Tratamento de Dados), edição do relatório, conclusão e bibliografia.

Pensamos ter sido bem-sucedidos na realização desta primeira parte do projeto, tendo cumprido todos os objetivos pedidos e acabando o projeto no prazo indicado. No entanto, ao longo da sua realização tivemos algumas dificuldades, uma vez que a sua escrita sem base empírica por trás, fazendo projeções para o futuro, revelou-se uma tarefa mais complicada do que antecipamos inicialmente (por exemplo, nas funcionalidades, onde estávamos a escrever sobre algo que não existe e ainda não temos garantia que serão possíveis de implementar).

Bibliografia

[1] Slides das aulas teóricas de Análise e Concepção de Algoritmos fornecidos ao longo do semestre

[2] Gustavo Pantuza, 2017. "Busca em Profundidade". Acedido a 10 de abril. https://blog.pantuza.com/artigos/busca-em-profundidade

[3] GeeksforGeeks, 2017. "Floyd Warshall Algorithm | DP-16". Acedido a 11 de abril. https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

[4] UFSC, 2018. "Algoritmo de Dijkstra para cálculo do Caminho de Custo Mínimo". Acedido a 12 de abril. http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html

[5] Wikipedia, 2019. "Problema do caixeiro-viajante". Acedido a 19 de abril. https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante

2ª Parte

Alterações à primeira parte

Depois de ouvir o *feedback* do professor e analisar de novo o nosso trabalho realizamos algumas alterações.

Assim, quanto à divisão do problema em diferentes iterações decidimos na Iteração 2 não considerar ainda a carga do veículo. O estafeta que realiza os múltiplos pedidos tem um veículo com carga ilimitada. Desta forma, o objetivo será apenas percorrer todos os restaurantes / todos os clientes sem consideração da carga.

Quanto à formalização do problema, decidimos remover o menu do Restaurante, o conjunto de pratos do Pedido e o número de pedidos bem como distância total do Estafeta. Para além disso adicionamos ao Estafeta um Meio de Transporte garantindo assim que este utiliza sempre o mesmo que lhe é atribuído. Isto não só nos permite simplificar o problema como também incidir mais na implementação dos algoritmos e comparação de resultados.

Uma outra alteração que temos que salientar é que, como já foi descrito anteriormente, o estafeta parte sempre da posição da morada do cliente ao qual realizou o ultimo pedido. Assim, como criamos um ficheiro do qual recolhemos a informação dos estafetas, optamos por ler as suas posições iniciais fixas. O que se provou útil também na verificação dos algoritmos. Dito isto a “Casa Dos Estafetas” deixou de ser um ponto de referencia no nosso mapa uma vez que não teria qualquer utilidade.

Quanto aos casos de utilização também efetuamos algumas alterações que estão especificadas mais à frente. Entre elas, em vez de ser dada ao utilizador a opção de escolher entre percurso mais barato/percurso menos demorado (no fundo escolher a Função Objetivo do programa), será apresentado o percurso de menor distância.

Estruturas de dados utilizadas

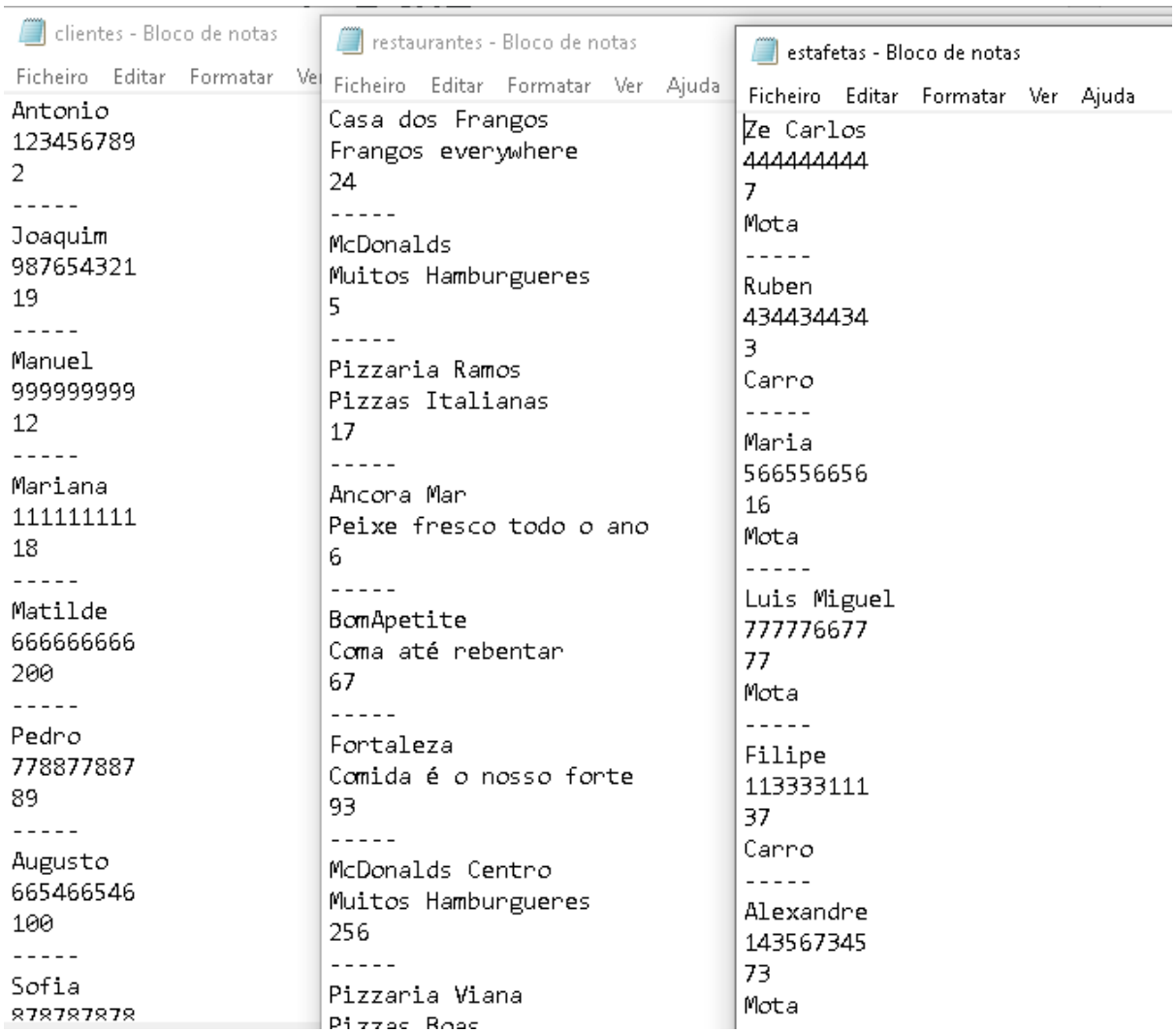
Os dados da empresa EatExpress são lidos a partir de ficheiros, nomeadamente o ficheiro clientes.txt, restaurantes.txt, estafetas.txt e transportes.txt. Em baixo apresenta-se parte destes ficheiros, de forma a ilustrar a sua organização.

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

O ficheiro restaurantes.txt contém os dados dos restaurantes da empresa, nomeadamente os seus nomes, descrições e id do vértice que corresponde às suas localizações.

O ficheiro estafetas.txt contém os dados dos estafetas da empresa, nomeadamente os seus nomes, NIFs, id do vértice que corresponde às suas posições, e o nome do veículo que utiliza.

Por último, o ficheiro transportes.txt contém os dados dos meios de transporte da empresa, nomeadamente o seu nome, capacidade e velocidade.



clientes - Bloco de notas

Ficheiro Editar Formatar Ver

```
Antonio
123456789
2
-----
Joaquim
987654321
19
-----
Manuel
999999999
12
-----
Mariana
111111111
18
-----
Matilde
666666666
200
-----
Pedro
778877887
89
-----
Augusto
665466546
100
-----
Sofia
878787878
```

restaurantes - Bloco de notas

Ficheiro Editar Formatar Ver Ajuda

```
Casa dos Frangos
Frangos everywhere
24
-----
McDonalds
Muitos Hamburgueres
5
-----
Pizzaria Ramos
Pizzas Italianas
17
-----
Ancora Mar
Peixe fresco todo o ano
6
-----
BomApetite
Coma até rebentar
67
-----
Fortaleza
Comida é o nosso forte
93
-----
McDonalds Centro
Muitos Hamburgueres
256
-----
Pizzaria Viana
```

estafetas - Bloco de notas

Ficheiro Editar Formatar Ver Ajuda

```
Ze Carlos
444444444
7
Mota
-----
Ruben
434434434
3
Carro
-----
Maria
566556656
16
Mota
-----
Luis Miguel
777776677
77
Mota
-----
Filipe
113333111
37
Carro
-----
Alexandre
143567345
73
Mota
```

Grafo

A classe “Graph” presente na pasta Graph.h é essencialmente uma adaptação da classe fornecida nas aulas, e contém algoritmos que fomos desenvolvendo durantes 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 estafeta.
- 1, se corresponder à morada de um cliente
- 2, se corresponder à localização de um restaurante
- 3, se corresponder à posição de um estafeta

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.

Pedido

Foi criada uma classe para representar as entregas/encomendas que teriam de ser feitas.
Cada pedido é composto por um Cliente que o efetuou, o Restaurante que este cliente desejou e o Estafeta que irá atender o pedido. A escolha do estafeta que atende o pedido pode ser feita pelo utilizador, ou escolhido pelo programa conforme um critério, isto dependendo de cada Caso de Utilização, pelo que será explicado mais à frente, nos Casos de Utilização.

Cliente

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

Restaurante

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

Estafeta

De modo a implementar as 4 fases descritas anteriormente, optamos por criar uma classe estafeta. Cada estafeta tem um transporte atribuído no qual irá realizar os pedidos. Para além disso contém ainda um nome e *nif* por questões de identificação na visualização dos dados.

Meio de Transporte

Como já foi mencionado, cada estafeta tem lhe atribuído um meio de transporte. Estes estão definidos por um nome, uma capacidade e uma velocidade. A capacidade corresponde ao número máximo de pedidos que este pode ter armazenado, e será abordada no nosso quarto Caso de Utilização, explicado mais à frente.

EatExpress

Por fim criamos uma classe EatExpress que contém toda a informação dos pedidos que estão a ocorrer num determinado momento (vetor “pedidos”) bem como os clientes, restaurantes, estafetas e meios de transporte que pertencem à plataforma.

GraphViewer

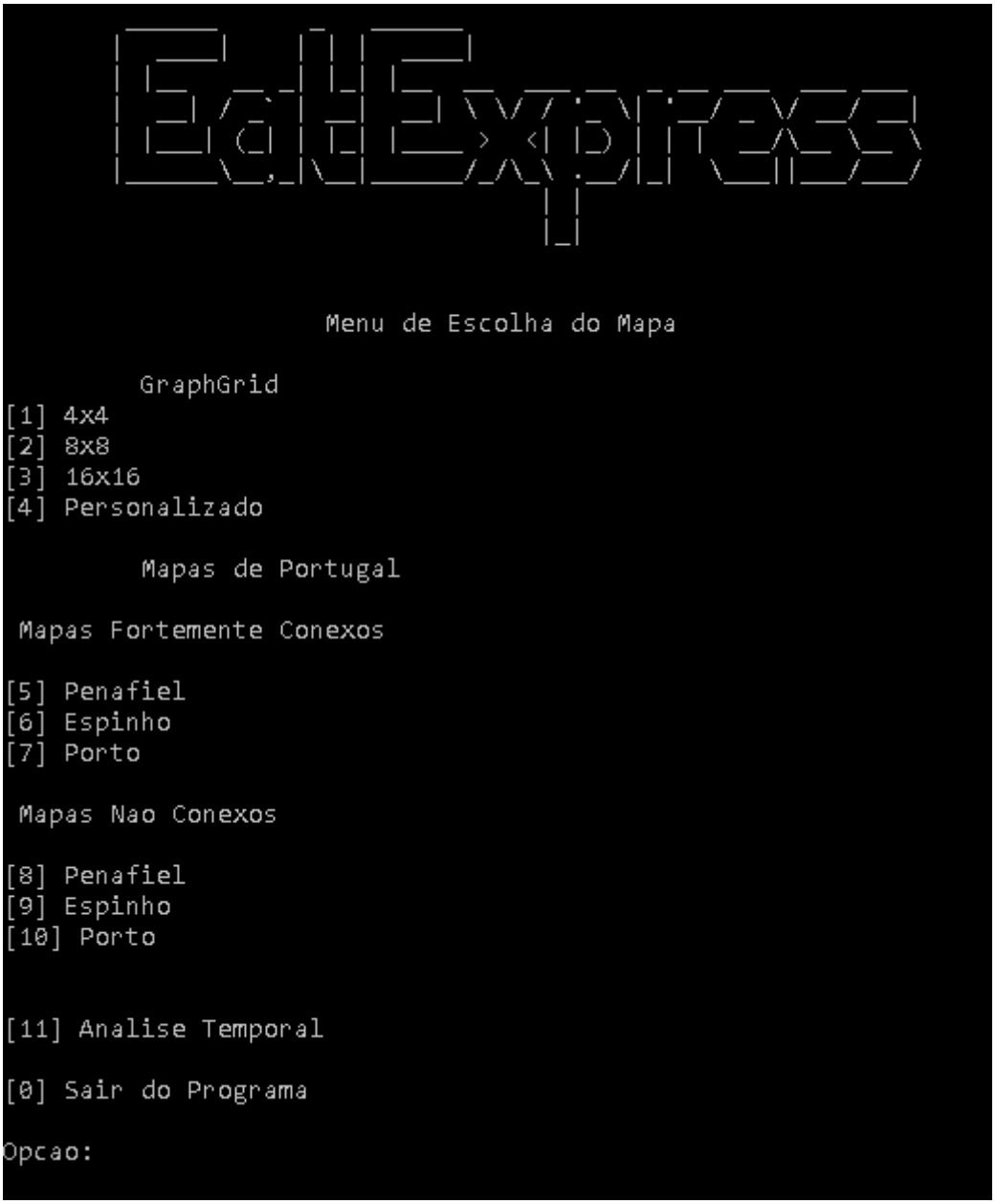
Este módulo, fornecido pelo corpo docente, permitiu-nos uma melhor visualização dos grafos e das soluções obtidas, e uma ilustração dinâmica do caminho que cada estafeta terá de percorrer.

Casos de Utilização

Como já foi mencionado, os casos de utilização implementados sofreram algumas alterações quanto aos previstos na primeira entrega deste trabalho. Assim apresentamos:

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:

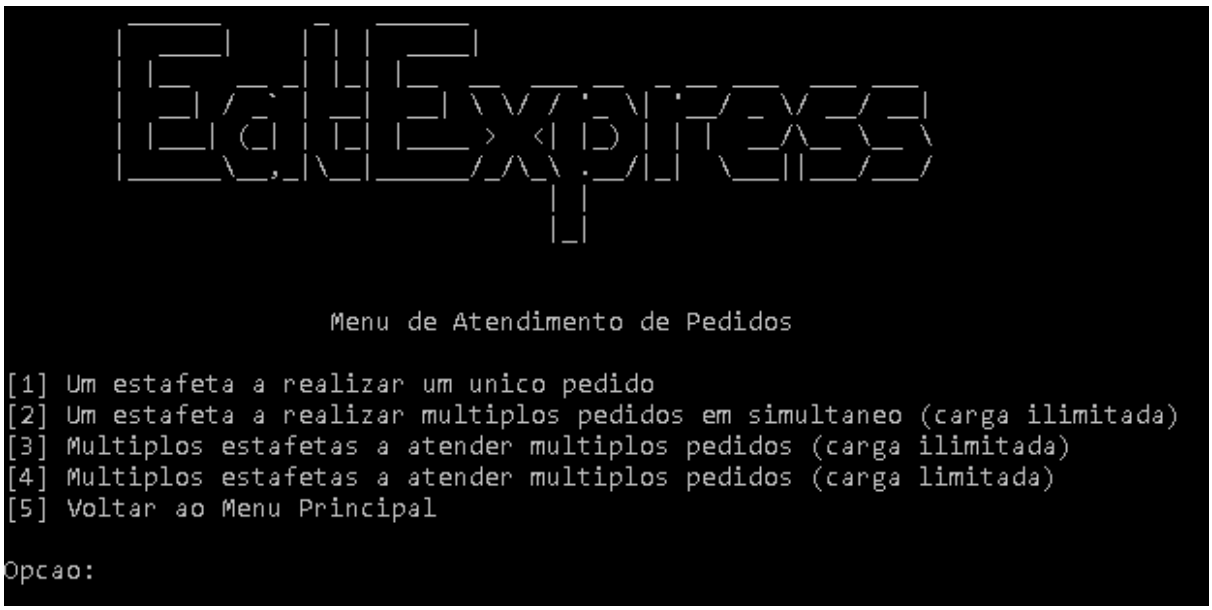
- Avaliar a conetividade do grafo, selecionando apenas uma parte fortemente conexa do mesmo.
- Visualizar o mapa recorrendo ao GraphViewer com os respetivos Clientes (verde), Estafetas (amarelo) e Restaurantes (vermelho) da aplicação.
- Visualizar os dados, ou seja, todos os clientes, restaurantes e estafetas da EatExpress, permitindo um maior controlo dos dados.
- E por último resolver as 4 variantes do problema.

Menu de Escolha do Problema

Quando é selecionada a opção 4 do menu anterior somos redirecionados para um último menu no qual podemos escolher entre as diferentes iterações que implementamos. É importante mencionar que em todas elas tivemos a preocupação de garantir que o estafeta passa primeiro pelo restaurante de um pedido antes de passar pelo cliente desse mesmo pedido, uma vez que é necessário recolher o pedido no restaurante primeiramente.

- Um estafeta a realizar um único pedido – é solicitado ao utilizador que escolha um cliente para efetuar o pedido, o restaurante que deseja, e um estafeta que irá realizar o pedido. De seguida é apresentada num grafo a solução de caminho mais curto para o mesmo.

- Um estafeta a realizar múltiplos pedidos em simultâneo (caga ilimitada) – de forma semelhante à anterior, o utilizador da aplicação pode escolher um conjunto de clientes e restaurantes e ainda um único estafeta que irá realizar todos os pedidos. Assim é mostrado o caminho mais curto no mapa, em que o estafeta entrega todos estes pedidos.
- Múltiplos estafetas a atender múltiplos pedidos (carga ilimitada) – Neste caso mais uma vez é solicitado ao utilizador que escolha um conjunto de clientes e restaurantes, no entanto o estafeta passa a ser escolhido por um critério que consiste em escolher o estafeta que se encontra mais perto do restaurante requisitado em cada pedido.
- Múltiplos estafetas a atender múltiplos pedidos (carga limitada) – Este caso tem uma interface muito semelhante ao anterior, a única diferença é que o estafeta para além de ser escolhido quanto à proximidade do restaurante também está restrito a um segundo fator que é a capacidade do veículo. Este valor restringe o número de pedidos que o estafeta pode acumular no seu veículo, pelo que, por exemplo, se tivesse de atender os pedidos P1(R1 e C1), P2(R2 e C2), P3(R3 e C3) e o seu veículo tivesse capacidade máxima de 2 pedidos, então o percurso R1-R2-R3-C1-C2-C3 não seria opção, mesmo que fosse o mais curto, uma vez que num dado momento desse percurso a capacidade do veículo é excedida (após passar por R3 possuiria 3 pedidos acumulados, mas a carga máxima seria 2). Ora, neste caso, teria de entregar pelo menos um dos pedidos para libertar espaço para outro. Assim é apresentado os caminhos mais curtos dos diferentes estafetas a atender os diferentes pedidos tento em conta a capacidade do veículo.



Conectividade

Este foi um dos primeiros problemas com que nos deparamos aquando da implementação das soluções encontradas e decidimos optar pela primeira abordagem descrita na Parte 1 deste relatório no capítulo "*Características do Grafo e Pré Processamento"* com algumas alterações.

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 Casa dos Estafetas). 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 estafeta e são guardados todos os vértices provenientes da pesquisa num vetor. O estafeta não necessita de regressar ao sitio de onde partiu sendo por isso desnecessário realizar uma avaliação tão detalhada da conetividade como a descrita anteriormente.

Algoritmos implementados e respetiva análise

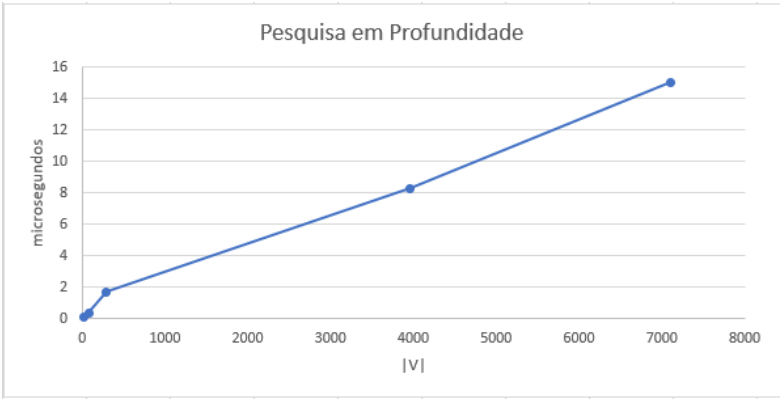
A complexidade teórica dos algoritmos implementados já foi abordada na Parte 1 deste trabalho, pelo que, de forma a não nos tornarmos repetitivos, apenas faremos a análise da complexidade temporal empírica a partir dos resultados obtidos e apresentaremos o pseudocódigo dos algoritmos usados. De notar 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 conetividade 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)
```

Em baixo 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 respetivas 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

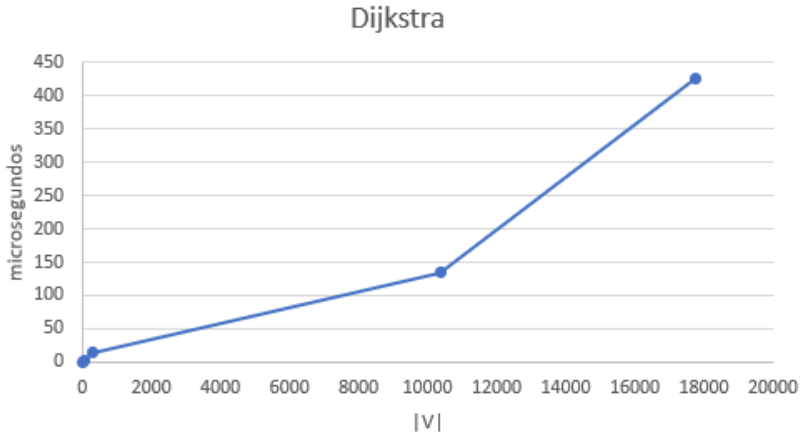
Para descobrirmos o caminho mais próximo entre dois vértices do grafo, decidimos que o melhor e mais simples algoritmo a utilizar seria o Algoritmo de Dijkstra, devido a termos de enfrentar várias vezes esse problema de achar o caminho mais curto entre dois pontos do mapa, pelo que a complexidade temporal 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) ←INF
        path(v) ←NULL
    dist(s) ← 0
    Q ← ∅
    INSERT (Q, (s, 0))                      // Insert s with key 0
    While (Q ≠ ∅) do
        v ←ExtractMin(Q)                    // algoritmo ganancioso
        for each w ∈ Adj(v) do
            if (dist(w) > dist(v) + weight (v, w) then
                dist(w) ←dist(v) + weight (v, w)
                path(w) ← 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 estafeta vai passar, recorremos a este algoritmo para encontrar o caminho mais curto entre cada par desses pontos. Deste modo, possuiremos no final um vetor com o percurso completo do estafeta.

Algoritmo do Vizinho Mais Próximo Adaptado

Este algoritmo está contido no algoritmo da segunda iteração (um estafeta a atender vários pedidos), e será usado por esta, e pela terceira e quarta iterações. 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 se encontra mais próximo do estafeta. No entanto, esta abordagem não seria suficiente, pois temos de garantir que o estafeta passa pelo restaurante X antes de entregar o pedido desse restaurante ao Cliente X. Para a quarta iteração, que considera cargas, foi necessário modificar ligeiramente este algoritmo, pelo que por motivos de simplicidade, apresentaremos em pseudo-código a versão da quarta iteração, em vez da que não considera carga (as únicas alterações são as condições que envolvam a capacidade do veículo). Mas antes do pseudo-código, explicamos melhor o nosso pensamento, que nos levou a desenvolver este algoritmo:

O objetivo deste algoritmo é encontrar uma lista ordenada dos pontos por onde o estafeta 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 (restaurantes e clientes) de forma a que seja o menor percurso do estafeta, o vetor "**restaurantes**", que conterá a cada momento os restaurantes pelos quais o estafeta pode passar (numa fase inicial contém todos eles, pois não há restrições para os restaurantes), e o vetor "**clientes**", que conterá a cada momento os clientes pelos quais o estafeta pode passar (é inicializado vazio, e de cada vez que é adicionado um restaurante ao "result", adiciona-se ao "clientes" o cliente que requisitou esse restaurante, uma vez que o estafeta pode já passar por ele, pois já requisitou o seu pedido no restaurante).

O algoritmo começa então por adicionar todos os restaurantes dos pedidos ao vetor "restaurantes", e é realizado num ciclo, enquanto houver pontos por adicionar (termina quando o vetor "restaurantes" e o vetor "clientes" estão ambos vazios). No caso de se tratar da quarta iteração, uma variável capacidade_atual_estafeta é inicializada com o valor da capacidade do meio de transporte do estafeta que estamos a tratar.

Começa por verificar se ainda existem restaurantes pelos quais o estafeta tem de passar (se restaurantes não está vazio), e se assim for, armazena a info do restaurante mais próximo numa variável "**restaurante_mais_proximo**".

Na primeira iteração (result está vazio), adiciona o restaurante_mais_proximo ao vetor result, retira-o do vetor restaurantes, e adiciona ao vetor clientes o cliente que requisitou esse restaurante, pois já pode passar por ele. No caso de se tratar da quarta iteração, é necessário também diminuir o valor da variável capacidade_atual_estafeta, uma vez que o estafeta possui agora um pedido armazenado.

Nas restantes iterações, faz uso de uma variável "**menor**", que conterá a info do vértice mais próximo do estafeta, podendo este ser um restaurante ou um cliente. Começa por verificar se ainda há restaurantes pelos quais o estafeta tem de passar, e se esse for o caso, menor é inicializado com a info do restaurante que estiver mais próximo do estafeta. Ora, na quarta iteração, este passo só é também realizado se a capacidade_atual_estafeta for maior do que 0, isto é, se este não tiver excedido a capacidade do seu transporte. Caso contrário, menor é inicializado com o primeiro cliente da lista de clientes, e é feito um ciclo que percorre os clientes e que altera o valor de menor se for encontrado um cliente que esteja a menor distância do estafeta do que o valor anterior de menor.

É neste ponto que menor contém o vértice mais próximo do estafeta, pelo qual pode passar. Assim, se menor corresponder a um restaurante (permaneceu com o valor de restaurante_mais_proximo com que foi inicializado), retiramos esse restaurante do vetor restaurantes, e adicionamos ao vetor clientes o cliente que requisitou esse restaurante. No contexto da quarta iteração, será também necessário diminuir a variável capacidade_atual_estafeta, uma vez que o estafeta possui agora mais um pedido acumulado. Caso menor corresponda a um cliente, basta remover esse cliente do vetor clientes. Por último, adiciona-se o vertex com info igual à info armazenada em menor ao vetor "result". Na quarta iteração, é aqui incrementada a variável capacidade_atual_estafeta, uma vez que, após esta entrega ao cliente, o estafeta possui mais uma vaga no seu veículo.

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 estafeta tem de percorrer. Apresenta-se de seguida o seu pseudo-código:

```
result ←∅
restaurantes ←∅
clientes ←∅
for each (pedido ∈ Pedidos) do
        INSERT (restaurantes, pedido->getRestaurante())
T restaurante_mais_proximo, menor
Int capacidade_atual_estafeta = estafeta->getCapacidadeTransporte();
While (restaurantes ≠ ∅ && clientes ≠ ∅)
        if (restaurantes ≠ ∅)
                restaurante_mais_proximo ←getRestauranteProximo(restaurantes)
        if (result = ∅)
                INSERT (result, restaurante_mais_proximo)
                REMOVE (restaurantes, restaurante_mais_proximo)
                capacidade_atual_estafeta--
                INSERT (clientes, getCliente(restaurante_mais_proximo))
        else
                if (restaurantes ≠ ∅ && capacidade_atual_estafeta > 0)
                        menor = restaurante_mais_proximo
                else
                        menor = clientes [0]
                for each (cliente ∈ clientes) do
                        if (dist(result.back(), cliente) < dist(result.back(), menor)
                                menor = cliente
                if (menor = restaurante_mais_proximo)
                        REMOVE (menor, restaurantes)
                        capacidade_atual_estafeta--
                        INSERT (clientes, getCliente(restaurante_mais_proximo))
                else
                        REMOVE (menor, clientes)
                        capacidade_atual_estafeta++
                INSERT (result, menor)
```

Algoritmo utilizado na iteração 1

A implementação da primeira iteração começa na função Um_Estafeta_Um_Pedido() do ficheiro Algoritmos.h. Inicia-se com a apresentação dos vários clientes e restaurantes, e pede ao utilizador que efetue um único pedido, escolhendo um restaurante e um cliente. No final, pede-se também que escolha o estafeta que vai querer responsabilizar pelo seu pedido. Posto isto, temos como dados o pedido que o utilizador efetuou, e o estafeta que os irá atender.

De seguida, é utilizado o Algoritmo de Dijkstra de modo a encontrar os caminhos mais próximos entre cada dois pontos, ou seja, o caminho mais curto entre o estafeta e o restaurante e o caminho mais curto entre o restaurante e o cliente. Após isto, junta-se os dois vetores, construindo assim um vetor resultante com o percurso completo do estafeta (vetor percurso), que será mostrado a partir da função showPathGV (), que mostrará com recurso ao Graph Viewer o caminho percorrido pelo estafeta. Segue-se o seu pseudocódigo:

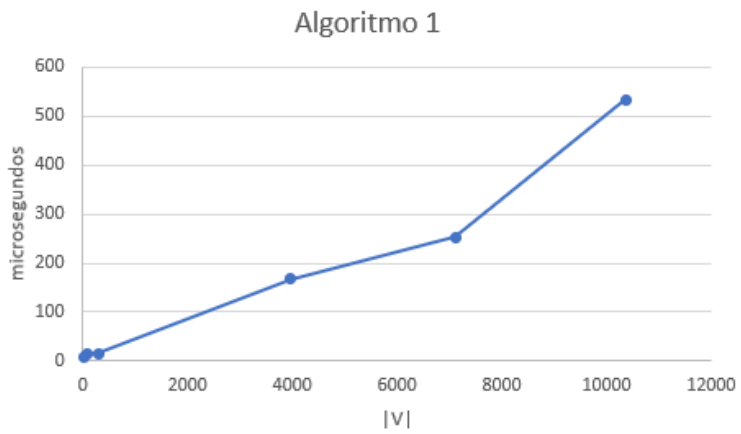
Função Um_Estafeta_Um_Pedido:

```
                                // Conjunto de ciclos e leitura da consola dos inputs do utilizador
getInput ()                     // Neste ponto obtemos o cliente, o restaurante e o estafeta
criarPedido ()                  // Cria-se um único pedido
avaliarConetividade ()          // Verifica se o caminho é possível ou não
vetor percurso = algFase1()     // Algoritmo que trata de um único pedido para um único estafeta
showPathGV ()                   // Mostra o percurso do estafeta no GraphViewer
```

Em que algFase1 é:

```
Dijkstra(estafeta->getPos ())
Vetor estafeta_restaurante = getPath(estafeta->getPos (), restaurante->getMorada())
Dijkstra(restaurante->getMorada ())
Vetor estafeta_restaurante = getPath(restaurante->getMorada (), cliente->getMorada())
restaurante_cliente.erase(restaurante_cliente.begin())                  // Elimina vértice repetido
estafeta_restaurante.insert(estafeta_restaurante.end(),restaurante_cliente.begin(),
restaurante_cliente.end())
return estafeta_restaurante;                    // Vetor que contém todo o percurso do estafeta
```

Segue-se o seu gráfico de análise temporal. Testou-se o algoritmo para 5 grafos: os 3 Grid Graphs, o mapa de Penafiel e o mapa de Espinho.



Algoritmo utilizado na iteração 2

A implementação da segunda iteração começa na função Um_Estafeta_Varios_Pedidos() do ficheiro Algoritmos.h. Inicia-se com a apresentação dos vários clientes e restaurantes, e pede ao utilizador que efetue quantos pedidos quiser, escolhendo para cada pedido um restaurante e um cliente. No final, pede-se também que escolha o estafeta que vai querer responsabilizar pelos seus pedidos. Posto isto, temos como dados os pedidos que o utilizador efetuou, e o estafeta que os irá atender.

De seguida, é chamado o algoritmo da fase 2 (algFase2()), descrito anteriormente (sem utilizar a capacidade dos meios de transporte), criado por nós e baseado na ideia do Nearest Neighbor, que nos devolve o vetor result organizado de forma a respeitar todas as condições (o estafeta tem de passar pelo restaurante X antes de entregar o pedido ao cliente X). 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 estafeta (vetor percurso).

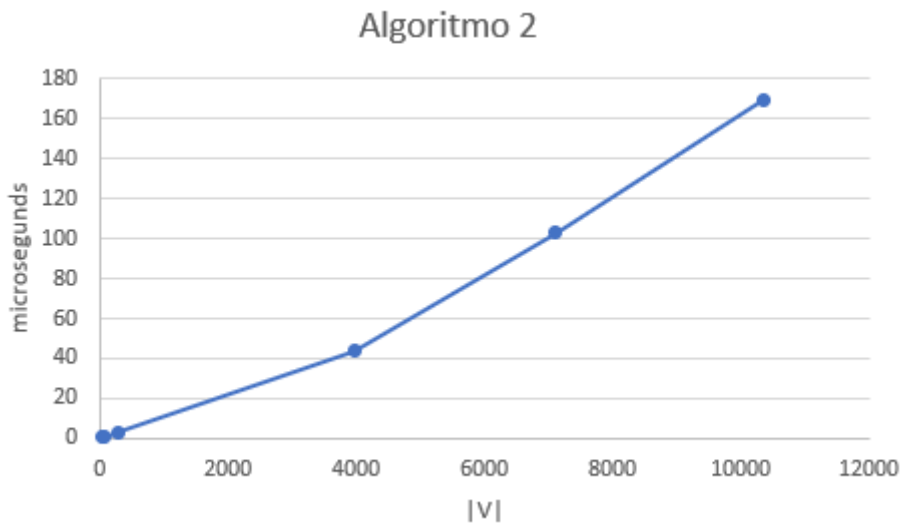
Finalmente, é chamada a função showPathGV() que mostrará com recurso ao Graph Viewer o caminho percorrido pelo estafeta. Segue-se o Pseudocódigo:

Função Um_Estafeta_Varios_Pedidos:

```
                                    // Conjunto de ciclos e leitura da consola dos inputs do utilizador
getInput ()                         // Neste ponto obtemos os vários pedidos e o estafeta
for each pedido ∈ pedidos:
        avaliarConetividade ()  // Verifica se o caminho é possível ou não
vetor percurso = algFase2()  // Algoritmo que trata de um único pedido para um único estafeta
showPathGV ()                           // Mostra o percurso do estafeta no GraphViewer
```

A função algFase2 contém o algoritmo que criamos, descrito em cima e baseado na ideia do Nearest Neighbor Algorithm, sem as partes respetivas à capacidade do veículo, uma vez que nesta fase não é considerada.

Segue-se o seu gráfico de análise temporal. Testou-se o algoritmo para 5 grafos: os 3 Grid Graphs, o mapa de Penafiel e o mapa de Espinho.



Algoritmo utilizado na iteração 3

A implementação da terceira iteração começa na função Varios_Estafetas_Sem_Carga() do ficheiro Algoritmos.h. Inicia-se com a apresentação dos vários clientes e restaurantes, e pede ao utilizador que efetue quantos pedidos quiser, escolhendo para cada pedido um restaurante e um cliente. Posto isto, temos como dados os pedidos que o utilizador efetuou.

Começamos por atribuir a cada pedido o estafeta que estiver mais próximo do restaurante requisitado no pedido. De seguida, percorremos todos os estafetas da empresa. É apresentado no ecrã o número de pedidos que o estafeta foi responsabilizado. Caso esse número seja apenas um, é chamado o algoritmo da fase 1 (algFase1() - um estafeta atende um único pedido), caso seja responsável por mais do que um pedido, é chamado o algoritmo da fase 2 (algFase2()), que corresponde ao algoritmo descrito anteriormente (sem considerar as capacidades), por nós criado e baseado na ideia do Nearest Neighbor, que nos devolve o vetor result organizado de forma a respeitar todas as condições (o estafeta tem de passar pelo restaurante X antes de entregar o pedido ao cliente X). 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 estafeta (vetor percurso).

Cada percurso de cada estafeta 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 estafetas.

Função Varios_Estafetas_Sem_Carga:

```
                                    // Conjunto de ciclos e leitura da consola dos inputs do utilizador
getInput ()                         // Neste ponto obtemos os vários pedidos
for each pedido ∈ pedidos:
        atribuirEstafeta(pedido)    // Atribui o estafeta mais perto do restaurante ao pedido
for each estafeta ∈ estafetas:
        if (num_pedidos = 1)        // Estafeta só é responsável por um pedido
                algFase1(pedido)
        else if (num_pedidos > 1)   // Estafeta é responsável por mais do que um pedido
                algFase2(pedidos)
showMultiplePathsGV ()              // Mostra os percursos de cada estafeta no GraphViewer
```

Algoritmo utilizado na iteração 4

A implementação da quarta iteração começa na função Varios_Estafetas_Com_Carga () do ficheiro Algoritmos.h. Inicia-se com a apresentação dos vários clientes e restaurantes, e pede ao utilizador que efetue quantos pedidos quiser, escolhendo para cada pedido um restaurante e um cliente. Posto isto, temos como dados os pedidos que o utilizador efetuou.

Começamos por atribuir a cada pedido o estafeta que estiver mais próximo do restaurante requisitado no pedido. De seguida, percorremos todos os estafetas da empresa. É apresentado no ecrã o número de pedidos que o estafeta foi responsabilizado. Caso esse número seja apenas um, é chamado o algoritmo da fase 1 (algFase1() - um estafeta atende um único pedido), caso seja responsável por mais do que um pedido, é chamado o algoritmo da fase 4 (algFase4()), que corresponde ao algoritmo descrito anteriormente, por nós criado e baseado na ideia do Nearest Neighbor, que nos devolve o vetor result organizado de forma a respeitar todas as condições (o estafeta tem de passar pelo restaurante X antes de entregar o pedido ao cliente X, e em nenhum momento deve exceder a capacidade máxima do seu veículo). 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 estafeta (vetor percurso).

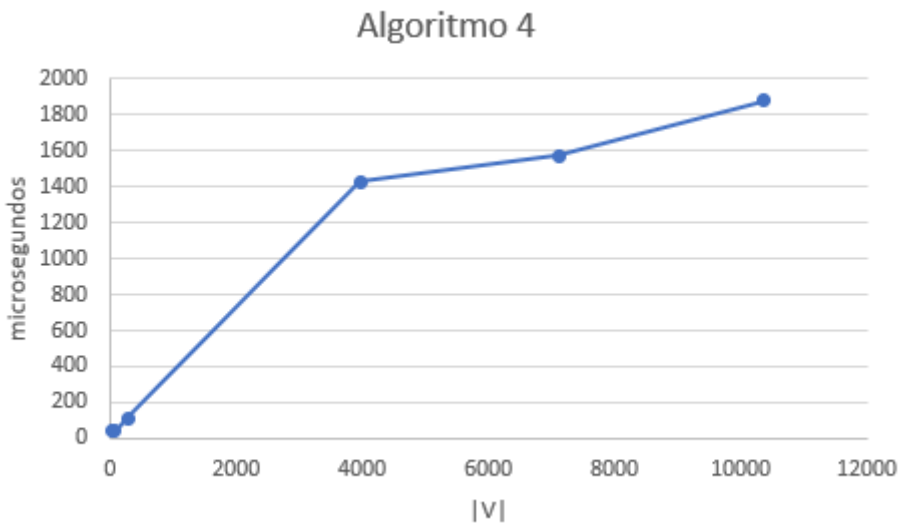
Cada percurso de cada estafeta vai sendo adicionado a um vetor percursos, que será enviado depois a uma função showMultiplePathsGV(), que mostra com recurso ao Graph Viewer todos os caminhos efetuados pelos vários estafetas.

Função Varios_Estafetas_Com_Carga:

```
                                        // Conjunto de ciclos e leitura da consola dos inputs do utilizador
getInput()                              // Neste ponto obtemos os vários pedidos
for each pedido ∈ pedidos:
        atribuirEstafeta(pedido)        // Atribui o estafeta mais perto do restaurante ao pedido
for each estafeta ∈ estafetas:
        if (num_pedidos = 1)            // Estafeta só é responsável por um pedido
                algFase1(pedido)
        else if (num_pedidos > 1)       // Estafeta é responsável por mais do que um pedido
                algFase4(pedidos)
showMultiplePathsGV()                   // Mostra os percursos de cada estafeta no GraphViewer
```

A função algFase4 contém o algoritmo que criamos, descrito em cima e baseado na ideia do Nearest Neighbor Algorithm, desta vez com as partes respetivas à capacidade do veículo incluídas, uma vez que nesta fase são consideradas.

Segue-se o seu gráfico de análise temporal. Testou-se o algoritmo para 5 grafos: os 3 Grid Graphs, o mapa de Penafiel e o mapa de Espinho.



Conclusão

Em suma, nesta segunda parte do projeto procuramos implementar um sistema para gerar os percursos mais rápidos, entre os restaurantes e os clientes, para os estafetas da aplicação EatExpress. Dividimos o problema em 4 iterações com graus de complexidade diferentes, de forma a ter uma implementação incremental. 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 várias formas de resolução dos problemas a tratar. Os tópicos em que cada elemento se dedicou especialmente mais são:

Paulo Ribeiro:

Relatório (2º parte): Algoritmos implementados e respetiva análise (Algoritmo de Dijkstra, Algoritmo do Vizinho mais próximo adaptado, Algoritmos utilizados nas implementações 1,2,3 e 4), Conclusão, Bibliografia

Implementação: Organização do programa em Menus, Criação das leituras de input por parte do utilizador, para cada iteração, Criação do Algoritmo utilizado para organização prévia dos pontos de interesse antes de se obter o percurso do estafeta, Implementação de todas as iterações descritas, Implementação das funções que permitem a apresentação dos percursos e dos grafos no GraphViewer, para além de outras funções que necessitamos aquando da implementação.

Mariana Ramos:

Relatório (2º parte): Alterações à primeira parte, Estruturas de dados utilizadas, Conetividade, Casos de Utilização, Algoritmos implementados e respetiva análise (Algoritmo de Pesquisa em Profundidade, Componentes Fortemente Conexas), gráficos com o tempo de execução dos algoritmos.

Implementação: Implementação de todas as iterações descritas, implementação das funções que permitem atribuir estafetas na iteração 3 e 4, implementação das funções que permitem a apresentação de múltiplos percursos e dos grafos no GraphViewer, para além de outras funções que necessitamos aquando da implementação. Implementação de todas as funções relativas a análise de conetividade.

Rita Silva:

Relatório (2º parte): Alterações à primeira parte, Estruturas de dados utilizadas, escrita dos ficheiros txt.

Implementação: Implementação de algumas funções de leitura de input pela parte do utilizador, implementação dos menus.

Pensamos ter sido bem-sucedidos na realização desta segunda parte do projeto, tendo cumprido todos os objetivos que tínhamos, adaptando alguns, como já esperávamos, pois a prática levou-nos a pensar em abordagens que fizessem mais sentido, e acabando o projeto no prazo indicado.

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.