Projeto de Conceção e Análise de Algoritmos

BosHBus: transporte de trabalhadores

(Tema 5 parte 1)

Luís Miranda up201306340@fe.up.pt
Nuno Oliveira up201806525@fe.up.pt
Romeu Flores up201806287@fe.up.pt

Índice

1. Descrição do tema

Neste trabalho, pretende-se implementar um sistema que permita a gestão do transportes pela empresa BosHBus. Inicialmente, os veículos estão numa garagem e, percorrendo um caminho com destino às instalações da BosH Global, recolhe os trabalhadores em pontos de entrada e saída previamente identificados. No final do horário de trabalho, os veículos fazem o caminho inverso, das instalações da BosH até à garagem, deixando aos trabalhadores no mesmo local da entrada.

Inicialmente é apenas usado um veículo mas na medida de otimizar o caminho percorrido pode-se usar mais. Para além disso, BosHBus poderá ser contratada por outras empresas o que se traduz num destino intermédio diferente.

A título ilustrativo, um trajeto possível seria:
Origem→ P.Recolha1→ P.Recolha2→ Destino→ P.Recolha2→ P.Recolha1 → Origem

Devemos então, primeiramente, tratar o grafo que representa o mapa onde vamos trabalhar. Com isto, devemos remover todas as arestas que correspondem a áreas inacessíveis, marcar todos os ponto de recolha, garagem e instalações da empresa. Só depois passaríamos ao processamento dos pontos de recolha (ordenando-os e atribuí-los ao respectivo veículo) e cálculo efetivo do percurso. Finalmente, tendo em conta que as arestas do grafo serão bidirecionais o caminho de volta para a garagem será exatamente o mesmo, só que na ordem inversa, passando novamente por todos os pontos de recolha.

2. Formalização do problema

2.1. Dados de entrada

G(V,E) → Grafo não dirigido pesado (considera-se que todas as ruas têm 2 sentidos) que representa o mapa da região para o(s) autocarro(s) circular(em). Cada vértice será caracterizado por:

- ID - Nome do vértice (único)
- Type - Tipo do vértice
- GPS - Coordenadas GPS
- Ai - Arestas que saem do vértice - Cada uma terá o seu próprio identificador único, vértice de destino e um atributo que indica o seu peso, ou seja, distância diagonal dos vértices que ela liga.

TD → Tabela de distâncias que guardará o peso de cada aresta

Ti → Os transportes a serem utilizados

Pi → Os percursos a fazer que consistem em:

- PRi - Conjunto de vértices que serão pontos de recolha
- E - Destino intermédio (considerando que a garagem será o destino final, as instalações da empresa serão o destino intermédio)
- ID - ID único do percurso

PP → coordenadas GPS do vértice que representa a garagem

OFFi → arestas inacessíveis devido a obras públicas

2.2. Dados de Saída

Gf(V,E) → O mesmo grafo não dirigido pesado dos dados de entrada
Cf → Todos os veículos, cada um com os seguintes atributos:

- ID - ID único do veículo
- Tr - Vetor ordenado com as arestas pelas quais o veículo terá de transitar

2.3. Restrições

- Os tipos dos vértices só podem ser “PARAGEM”, “PRECOLHA”, ”INTERSECAO” ou “POINT”;
- Tanto a garagem como o destino intermédio terão de ser vértices do grafo mas ambos distintos entre si;
- Os “PRECOLHA” são obrigatoriamente “PARAGEM” mas o inverso é necessariamente verdade. Para além disso os vértices que dizem respeito a pontos de recolha não podem coincidir com os da garagem e destino intermédio;
- Ambos a garagem e destino intermédio serão vértices do tipo “POINT”;
- Por cada destino intermédio diferente haverá no mínimo um veículo diferente;
- Um veículo só pode ser utilizado num percurso;
- O peso de cada aresta tem que ser positivo tirando no caso de ser inacessível que será representada por -1;
- É imperativo que os vértices que representam a garagem, destino(s) intermédio(s) e pontos de recolha façam parte de um comum componente fortemente conexo do grafo o que significa

que dado um qualquer par de vértices (v,w) existe um trajeto de v para w e vice-versa. Isto é importante dado que no fim do dia o autocarro terá de fazer o trajeto inverso para deixar os trabalhadores no respetivo ponto de recolha e voltar à garagem;

- Obviamente, a origem da primeira aresta do vetor ordenado de arestas de cada veículo, tal como o destino da última tem de ser a garagem. Da mesma maneira pelo meio do vetor, duas arestas seguidas tem de ter as instalações da empresa como destino e origem, respetivamente. Assim, cada trajeto de um veículo tem no mínimo 4 arestas (2 em cada sentido porque tem que passar no mínimo por 1 ponto de recolha);

2.4. Função Objetivo

O problema reside em encontrar em encontrar um trajeto ótimo que passe em pontos de recolha pré-determinados. Para tal pode-se usar vários veículos, se necessário.

$$\min \sum_{c \in Cf} \left(\sum_{d \in Tr} w(Tr) \right)$$

3. Perspetiva de solução

A solução para o problema proposto pode ser dividida em várias etapas distintas:

1) A obtenção de um grafo, no qual todos os vértices são ou paragens ou interseções na via. Para isto utilizaremos os mapas fornecidos, bem como a visualização dos respectivos grafos no *GraphViewer*.

2) Fazer um pré-processamento do grafo obtido, removendo as arestas indesejadas para a execução do algoritmo, nomeadamente as que representam vias públicas inutilizáveis devido a obras nas mesmas. Com esta finalidade, estas arestas serão indicadas no grafo com um peso igual a -1. Para além disso serão marcados todos os pontos de recolha porque nem todas as paragens correspondem necessariamente pontos de recolha no âmbito deste projeto.

3) Verificar se os vértices que correspondem à garagem, instalações da empresa e pontos de recolha pertencem todos a um componente fortemente conexo comum a partir do método dado na aula teórica sobre conectividade em grafos (pesquisa em profundidade, inversão das arestas seguida de uma nova pesquisa em profundidade começando pelo vértice mais alto não visitado). Se tal não se verificar, o que ainda que improvável é possível, significa que o trajeto não será possível.

4) Ordenação dos vários pontos de recolha, entre a origem e o destino intermédio (e vice-versa), a serem incluídos no trajeto. O peso de cada aresta corresponderá à distância entre ambos os vértices, e cada

vértice terá a respectiva coordenada necessária para calcular a mesma distância. Para esta ordenação, será seguido o seguinte processo:

- Inicialmente existe apenas a origem e o destino intermédio;
- Escolhe-se um ponto arbitrário, e para cada posição possível da sequência de pontos já ordenados, calcula-se a nova distância total do percurso. Para tal, soma-se a distância entre o novo ponto e os seus pontos adjacentes e subtrai-se a distância entre ambos os pontos previamente adjacentes (Exemplo: para inserir o ponto C entre A e B: dist(A,C)+dist(C,B)-dist(A,B));
- Introduz-se este novo ponto no lugar com menor acréscimo de distância.

5) Aplicação de um algoritmo com objetivo de perceber se o uso de mais veículos será beneficial para o trabalho. Se tal acontecer, a cada veículo serão associados os pontos de recolha que lhe competem, visto que cada ponto apenas pode ser percorrido por um veículo. Numa fase inicial do projeto este ponto não vai ser tido em conta visto que apenas se vai utilizar um veículo.

6) Cálculo do caminho mais curto entre cada par consecutivo de vértices, para obtenção do percurso óptimo. Este passo será conseguido através da aplicação do algoritmo de Dijkstra, um algoritmo ganancioso bastante usado em problemas de busca de caminhos ótimos, como o proposto. Este algoritmo permite calcular qual o percurso com menor distância entre dois pontos distintos, começando neste caso na origem (garagem), e percorrendo os vários pontos de recolha até ao destino intermédio (empresa). O percurso de volta da empresa à garagem será depois o inverso, visto ser também ele o ótimo.

4. Casos de utilização

- Leitura e carregamento de um determinado grafo (a partir de um ficheiro de texto fornecido) e visualização do mesmo com a ferramenta *GraphViewer*;
- Leitura e carregamento dos veículos disponíveis (não implica que serão utilizados) e da garagem de onde eles partem(comum a todos);
- Leitura e carregamento dos pontos de recolha no mapa para cada trabalho (apenas pode escolher entre os pontos que são considerados paragens). Isto é feito pelo utilizador;
- Cálculo do número de veículos necessários para otimizar determinado trabalho;
- Cálculo do(s) percurso(s) ótimo(s) para realizar determinado trabalho;
- Visualização no *GraphViewer* os trajetos de cada veículo após terem sido feitos todos os cálculos;
- Visualização no *GraphViewer* os pontos de recolha com acessibilidade reduzida (tal pode acontecer dado ao grafo em si ou a obras na via pública);

5. Conclusão

Sendo este um problema de otimização de percurso há vários procedimentos e algoritmos diferentes que poderíamos adotar para o resolve. No entanto, optamos pelo algoritmo de *Dijkstra* por acharmos ser o mais vantajoso no que diz respeito ao nosso tema.

Considerámos também que após calcular o caminho ótimo da garagem às instalações da empresa, o de retorno seria o mesmo dado que as arestas do grafo são bidirecionais.

Para já achamos que a nossa maior dificuldade será identificar quando seria benéfico usar mais que um veículo e atribuir os respetivos pontos ao mesmo.

De resto, somos da opinião que chegamos a uma boa solução para o problema e estamos preparados para a fase seguinte.

Esforço Individual:

Nuno Oliveira - up201806525

- Quantitativo: 50%

Luís Miranda - up201306340

- Quantitativo: 50%

Romeu Flores - up201806287

- Quantitativo: 0%

Quanto à divisão de tarefas, o Luís Miranda dedicou-se principalmente à perspectiva de solução e o Nuno Oliveira à formalização do problema e aos casos de utilização, mas ajudando-se mutuamente em todo o processo.

6. Bibliografia

Como referência relativamente à estrutura do relatório usamos os um dos disponibilizados no moodle e no que diz respeito ao resto do trabalho restringimo-nos aos slides das aulas teóricas.

2ª Parte

7. Diferenças/mudanças em relação à 1ª parte

Infelizmente, não nos foi possível fazer tudo que nos tínhamos proposto fazer na primeira parte do relatório por vários motivos. Não servindo para nos desculpar, um dos membros do nosso grupo simplesmente deixou de comunicar connosco e acabou por não fazer nada, algo que nos deixou confusos a princípio e acabamos por gerir mal o tempo devido a isso. Tal facto não se sucedeu apenas nesta cadeira e as entregas praticamente todas na mesma semana só aumentaram aos problemas. Algumas funcionalidades não implementadas que gostaríamos de ter feito sâo:

- Uso de vários veículos com o propósito de otimizar um mesmo serviço
- Utilizar os veículos e serviços como um elemento da empresa. Da maneira que foi feito, o nosso programa apenas apresenta a solução para um qualquer serviço com um veículo. Não tem em conta um número de veículos que podem ou não ser utilizados e os serviços que já está ou não a realizar.

Para além disto, como é de esperar, à medida que se ia desenvolvendo o projeto, a nossa visão do mesmo mudou o que levou a algumas outras alterações relativamente à estrutura do mesmo.

- A tabela com a distância de cada node a todos os outros foi inicialmente implementada mas como demorava mesmo

muito tempo a construir e não trazia vantagens suficientes para compensar essa perda de tempo decidimos não a usar e remover do projeto por completo.

- Os tipos possiveis de cada node mudaram podendo ser um de “NONE”, “PRECOLHA”, “FACTORY” e “GARAGEM”. Inicialmente todos são “NONE”, só mais tarde é que se muda consoante o necessário.
- As edges em obras bem como as que têm peso menor ou igual a zero ao contrário do que foi dito antes não são marcadas com -1 mas sim removidas do grafo por completo.

8. Principais casos de uso implementados

- Leitura e carregamento de um grafo corresponde a uma cidade à escolha do utilizador a partir de um ficheiro de texto
- Complementando a leitura anterior é lido também de outro ficheiro associado à cidade escolhida o id do node da garagem e os edges que dizem respeito às ruas onde existem obras (se houver) que podem ser alterados ao gosto do utilizador (não como funcionalidade do programa, mas o conteúdo do ficheiro pode ser alterado em runtime e se for carregado novamente, terá informação atualizada)
- Apresentação no GraphViewer do mapa total carregado inicialmente, não tendo em conta a garagem, mas já sem as edges inacessíveis

- Apresentação do GraphViewer do grafo acessível apenas a partir da garagem. Este novo grafo é determinado através duma pesquisa em profundidade
- Output do ID de todas as nodes acessíveis a partir da garagem e de instruções para criar um serviço. Esta opção serve mais como um “help” para ajudar o utilizador.
- Leitura e carregamento de um serviço a partir de um ficheiro com o nome à escolha do utilizador. Este ficheiro tem ser colocado no local indicado e ter uma estrutura específica explicada mais tarde. Após leitura é calculado o caminho mais curto que realiza o serviço utilizando um de dois algoritmos à escolha do utilizador e tal será apresentado no GraphViewer.

9. Estruturas de dados usadas

- **Graph / Vertex / Edge**

Para estas classes foi utilizado maioritariamente o código para as mesmas fornecido nas aulas, embora tenham sido acrescentados alguns métodos que achamos necessário para facilitar o resto do trabalho.

```
/**************************** Graph  ****************************/

<T>
class Graph {
    vector<Vertex<T> *> vertexSet;    // vertex set
    double ** W = nullptr; // distance
    int **P = nullptr; // path
    int findVertexIdx(const T &in) const;
```

```
/************************ Vertex  *************************/

 <T>
class Vertex {
    T info;                       // content of the vertex
    vector<Edge<T> > adj;         // outgoing edges

    double dist = 0;              // dist
    Vertex<T> *path = NULL;       // path
    int queueIndex = 0;           // required by MutablePriorityQueue

    bool visited = false;         // auxiliary field
    bool processing = false;      // auxiliary field

    void addEdge(Vertex<T> *dest, double w);
    void addEdge(Vertex<T> *dest, double w, bool display);
```

```
/********************** Edge  ****************************/

<T>
class Edge {
    Vertex<T> * dest;      // destination vertex
    double weight;         // edge weight
    bool displayGV;     //needed because we add the same edge twice and should only display 1
```

- **Node**

Definimos uma classe “Node” com o objetivo de guardar várias informações relativas a cada vértice (um ID, coordenadas X e Y, o tipo

de vértice - Ponto de Recolha, Garagem ou Empresa - e coordenadas X e Y de display, para permitir o seu display no GraphViewer).

```
enum Type{
    PRECOLHA,
    GARAGEM,
    FACTORY,
    NONE
};

class Node{
private:
    int id; // ID do vértice
    double xCoord; // coordenada X
    double yCoord; // coordenada Y
    Type type; // tipo de vértice
    int graphViewerX; // coordenada X para o GraphViewer
    int graphViewerY; // cordenada Y para o GraphViewer
```

- **Vehicle**

Criamos a class “Vehicle” com o objetivo de representar os diversos veículos utilizados pela empresa de transporte para realizar os serviços requisitados pelas diversas empresas. Cada veículo contém informação sobre o seu ID e a lista de “Edges” (ordenadas) que terá de percorrer para completar o trajeto necessário para cumprir o serviço que lhe foi atribuído.

```
class Vehicle{

private:
    int id; // ID do veiculo
    vector<Edge<Node>> PRordenados; // vetor com as edges a percorrer, ordenadas
```

- **Service**

Os serviços a prestar são representados pela classe “Service”, que contém como atributos um ID, o vértice da garagem, o vértice do destino (localização da empresa), um vetor com os diferentes vértices dos “pontos de recolha”, onde o veículo terá de passar obrigatoriamente, e o veículo que prestará esse mesmo serviço.

```
class Service{

private:
    int id; // ID do veículo
    Vertex<Node>* garagem; // vértice da garagem
    Vertex<Node>* destino; // vértice da empresa
    vector<Vertex<Node>*> pontosRecolha;    //vetor dos pontos de recolha
    Vehicle vehicle;    // veículo atribuido;
```

10. Discussão sobre os casos de utilização

Após compilar com sucesso e correr o programa, este será o menu inicial que o utilizador verá:

```
HELLO, WHAT DO YOU WANT TO DO?

[0] Load a graph and and generate CFC after reading its info .txt file (DO THIS BEFORE ANYTHING ELSE!!!)
[1] Display full graph
[2] Display accesible nodes from the garage
[3] Explain how to generate a service and list accesible nodes from the garage (in case you need help)
[4] Load a service and display optimal path solution
[5] Exit program
```

O utilizador é avisado que tem de carregar um grafo antes de fazer outra coisa qualquer, mas mesmo que tente não irá conseguir.

```
[0] Load a graph and and generate CFC after reading its info .txt file (DO THIS BEFORE ANYTHING ELSE!!!)
[1] Display full graph
[2] Display accesible nodes from the garage
[3] Explain how to generate a service and list accesible nodes from the garage (in case you need help)
[4] Load a service and display optimal path solution
[5] Exit program
3


You must first load a graph!

[0] Load a graph and and generate CFC after reading its info .txt file (DO THIS BEFORE ANYTHING ELSE!!!)
[1] Display full graph
[2] Display accesible nodes from the garage
[3] Explain how to generate a service and list accesible nodes from the garage (in case you need help)
[4] Load a service and display optimal path solution
[5] Exit program
```

Opção 0 - Carregar um grafo

Se o utilizador escolher a opção, ser-lhe-á apresentado outro “menu” para escolher de que cidade pretende carregar o grafo.

```
Please select the map which you want to load:
[0] Aveiro
[1] Braga
[2] Coimbra
[3] Ermesinde
[4] Fafe
[5] Gondomar
[6] Lisboa (not recommended, may take a while due to huge size)
[7] Maia
[8] Porto
[9] Viseu
[10] Cancel...
0

Reading graph file...
Done!

Generating CFC...
Done!
```

Em cidades como aveiro e fafe com 3000 nodes e edges ou menos carregar o grafo é praticamente instantâneo mas outras como Braga com 20000 de cada já demoram um pouco mais dependendo da máquina (nos nossos computadores braga, por exemplo, demora cerca de 15 segundos). Inicialmente, nós fazíamos duas pesquisas lineares a cada edge para encontrar os vértices no grafo aos quais associar a mesma. Mais tarde, porque achamos que a leitura estava demasiado lenta decidimos transformar a pesquisa linear numa pesquisa binária pelo vetor (explicado melhor na parte dos algoritmos implementados). Obviamente, para tal poder ser feito ordenamos o vetor dos vertex pelo ID da node depois de acabar de os ler por completo usando o sort da standart library e uma função sortById como comparador.

```
bool sortById(const Vertex<Node>* a,const Vertex<Node>* d){
    return a->getInfo().getId()<d->getInfo().getId();
}
```

Agora já quase todas as cidades são processadas em tempo relativamente aceitáveis. Mesmo assim, Lisboa continua a ser relativamente lento dado ao facto de ter cerca de 75000 nodes e 91000 edges pelo que não é aconselhado que se escolha, ainda que eventualmente funcione. Este nosso problema podia ter sido resolvido se na classe graph usássemos uma hashtable de vértices em vez de um vetor, mas inicialmente não tínhamos a noção do quão "pesado" seria esta leitura e quando nos apercebemos preferimos implementar outras funcionalidades do que refazer grande parte do código.

Relativamente a tags, tendo o nosso tema em conta, decidimos que não seria necessário usar visto que todos os pontos do mapa podem ser pontos de recolha (quem os escolhe é o utilizador que cria o serviço). Assim, o tipo de cada ponto é inicialmente "NONE" só sendo alterado aquando da leitura do ficheiro da garagem ou de um serviço.

Após criar o grafo completo lêmos a id do node da garagem e os nodes inutilizáveis a partir de um ficheiro com um nome constante que está no diretório "files" dentro de um outro diretório com o nome da cidade. A título de exemplo, se o utilizador escolher carregar o grafo de Aveiro, o programa lê o os nodes no ficheiro "nodes_x_y_aveiro.txt", as edges no "edges_aveiro.txt" sendo que ambos se encontram no diretório "mapas/aveiro/" e a garagem e nodes em obras no ficheiro "aveiro_info.txt" que está no diretório "files/aveiro/". O conteúdo deste ficheiro será:

```
1131574752                  ----> Id do node onde fica a garagem
------                      ----> Separador
(392953699, 1237291488)     ----> A partir de aqui só podem aparecer edges e todas que aparecerem
(1103236590, 457547432)           serão removidadas do grafo
```

Este ficheiro pode ser alterado à vontade com algumas condições:

- O ficheiro tem sempre que ter na primeira linha um ID de um node que exista na cidade;
- O nome do ficheiro não pode ser alterado;
- As edges em obras, se existirem, têm também de existir na cidade;
- Não é obrigatório ter edges no ficheiro, mas se tiver o separador é obrigatório. Se não lá estiver, a node que estaria no seu lugar vai ser ignorada;

Nota sobre o ficheiro: já existe um default para todas as cidades com uma garagem escolhida aleatoriamente e o único que indica edges em obras é o de aveiro.

Após encontrada a garagem procedemos a remover do grafo as edges indicadas no ficheiro. O próximo passo é uma pesquisa em profundidade a partir da garagem para obter o grafo conexo (explicado mais ao pormenor no ponto 11. relativo à conetividade) e voltar ao menu inicial.

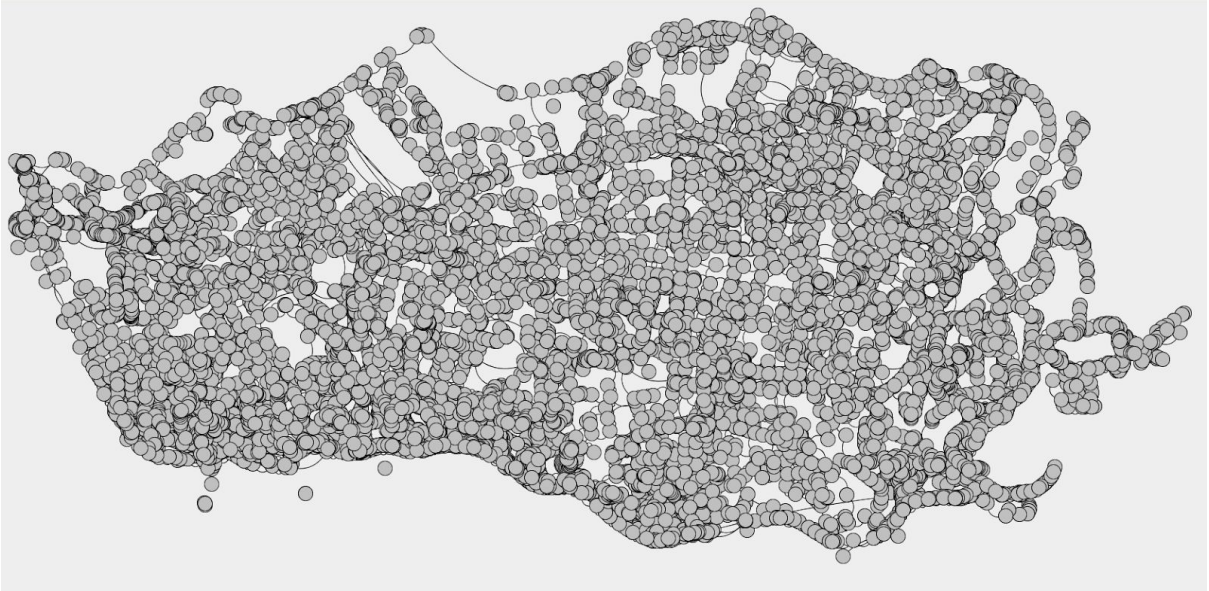
Opção 1 - Visualizar grafo completo

Depois de ler um grafo o utilizador pode visualizá-lo por completo se escolher esta opção. As dimensões da janela são calculadas com uma razão ou em termos mais populares, “regra de três simples”:

```
w = (int) ((xMax-xMin)*h/(yMax-yMin));
```

Onde h é uma constante constante de 750, e os restantes valores são calculados a partir de todos os nodes do grafo.

Como exemplo, o grafo completo do porto:

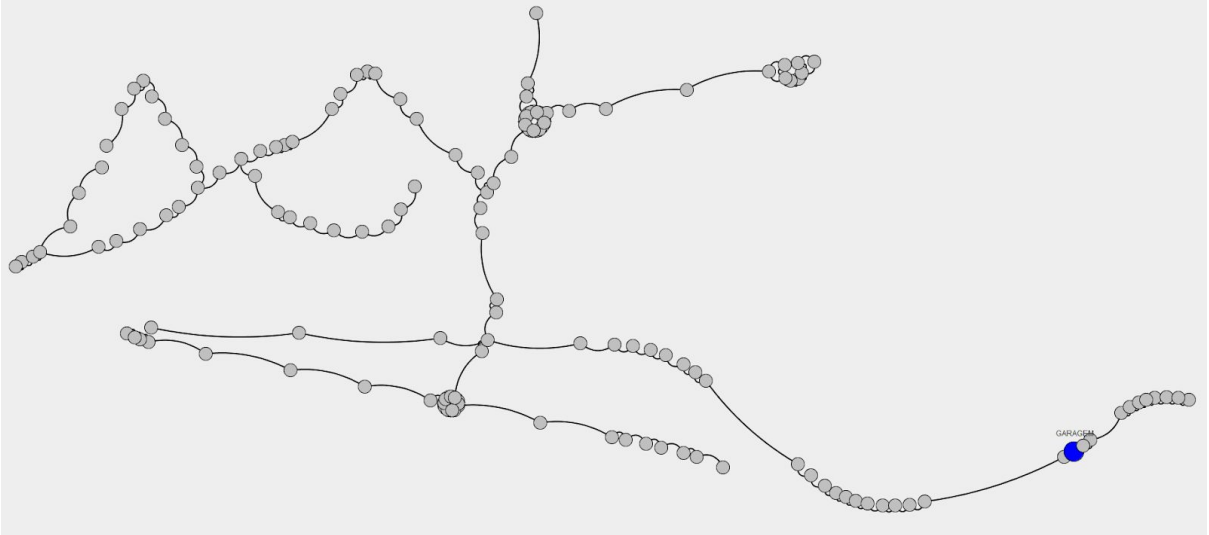


Como dá para ver ainda tem muitos pontos inacessíveis, mas já não tem as arestas em obras, se forem indicadas no ficheiro respectivo. É importante referir que como estamos a tratar as edges sendo bidirecionais, cada edge é escrita nos dois nodes correspondentes, no entanto, nesta opção do graphviewer apenas mostramos uma para facilitar a leitura.

Opção 2 - Visualizar grafo conexo

Esta opção é muito semelhante à primeira na medida em que quando se quer fazer isto, já temos a informação do grafo conexo, não é preciso calculá-lo. Outra semelhança à opção anterior é o facto de apenas mostrar uma edge “undirected” de um node a outro, quando fisicamente são guardadas duas para o efeito de serem bidirecionais. Este grafo é à partida mais percetível, mais limpo e a garagem encontra-se marcada a azul, com um tamanho superior ao resto dos pontos.

Como exemplo, o grafo conexo de aveiro:



Opção 3 - Ajuda para criar um serviço

Nesta opção programa irá imprimir o id de todas as nodes acessíveis a partir da garagem (exclusive) em ordem crescente, dado que o vetor é ordenador à prior para o efeito de usar a pesquisa binária.

```
(410351373);    (410351377);    (410351379);    (410351380);    (410351381);    (410351382);    (410351535);    (4103515
37);    (410351538);    (410351542);    (410351543);    (410351544);    (410351545);    (410351546);    (410351547);
        (410351548);    (410351549);    (410351550);    (428671585);    (428671586);    (428671587);    (428671588);
        (428671589);    (428671590);    (428671591);    (428671592);    (428671594);    (464755680);    (464755686);
        (464755691);    (464755697);    (464755701);    (464755705);    (464755736);    (464755740);    (464755741);
        (464755743);    (464755752);    (464755754);    (464755757);    (464755760);    (464755799);    (464755803);
        (464755810);    (464755812);    (464755815);    (464755818);    (464755821);    (464755824);    (464755827);
        (464755835);    (464755839);    (464755842);    (464755844);    (464755847);    (464755849);    (464755878);
        (464755879);    (464755880);    (464755881);    (464755882);    (464755883);    (464755884);    (464755885);
        (464755886);    (464755888);    (464755890);    (464755892);    (464755895);    (464755897);    (1030926683);
        (1030926685);   (1030926687);   (1030926704);   (1030926705);   (1030926707);   (1030926708);   (1030926710);
        (1030926711);   (1030926712);   (1103236582);   (1103236587);   (1103236593);   (1103236595);   (1103236606);
        (1103236646);   (1131574669);   (1131574671);   (1131574704);   (1131574707);   (1131574731);   (1131574733);
        (1131574736);   (1131574742);   (1131574745);   (1131574760);   (1131574761);   (1131574786);   (1131574788);
        (1131574803);   (1131574806);   (1131574810);   (1131574821);   (1131574825);   (1131574829);   (1131574831);
        (1131574847);   (1131574849);   (1131574863);   (1131574864);   (1131574872);   (1131574874);   (1131574875);
        (1131574883);   (1131574886);   (1131574887);   (1131574893);   (1131574895);   (1131574897);   (1167473294);
        (1167473301);   (1167473302);   (1167473306);   (1167473312);   (1167473317);   (1167473322);   (1167473324);
        (1167473326);   (1167473333);   (1167473334);   (1167473335);   (1167473342);   (1167473343);   (1167473344);
        (1167473348);   (1167473350);   (1167473351);   (1167473354);   (1167473357);   (1167473363);   (1167473364);
        (1167473366);   (1167473367);   (1167473368);   (1167473369);   (1167473371);   (1167473372);   (1167473378);
        (1167473381);   (1167473383);   (1167473385);
```

Para além disso, esta opção dá output duma breve explicação sobre como fazer um serviço para ajudar e facilitar a vida ao utilizador:

```
About the service criation:
1 - Create a text file with whatever name you whish (i.e. banana.txt)
2 - The first line of the file should be the integer of the ID of your factory's node and nothing else
3 - The next should also be an integer indicating how many pickup nodes are about to be read (that you will list) and no
thing else
4 - All the lines following these should only cointain one integer corresponding to the ID of a pickup node
5 - Put that file inside the corresponding city folder in the 'files' directory

Example of the content of a service file:

15202115
2
561235
5165518

------------------
```

Opção 4 - Carregar um serviço e calcular solução

Na quarta opção o programa vai primeiro pedir ao utilizador que insira nome do ficheiro que diz respeito ao serviço. Este ficheiro deve seguir a seguinte estrutura:

```
1131574707              ------> ID do node corresponde ao destino/instalações da fábrica
3                       ------> Número de pontos de recolha
1131574874              ------> A partir daqui deverá ter o ID dos nodes correspondentes a pontos de recolha
428671587
428671585
```

Toda a informação no ficheiro deverá ser preenchida pelo utilizador (apesar de já haver um serviço de teste para gondomar e aveiro que pode servir como exemplo) seguindo algumas regras:

- O id do destino tem que existir no grafo conexo da cidade e tem que ser diferente da garagem.
- Na segunda linha tem que ter um número que indica a quantidade de pontos de recolha que vamos ler a seguir e estes todos têm também de ser todos diferentes da garagem.
- No mínimo um dos pontos de recolha tem de pertencer ao grafo conexo da cidade para que o serviço seja considerado “válido”. Se

algum(ns) dos pontos não pertencer(em), o programa vai calcular o caminho ignorando esse(s) ponto(s) e no fim avisa quais pontos foram de facto ignorados.

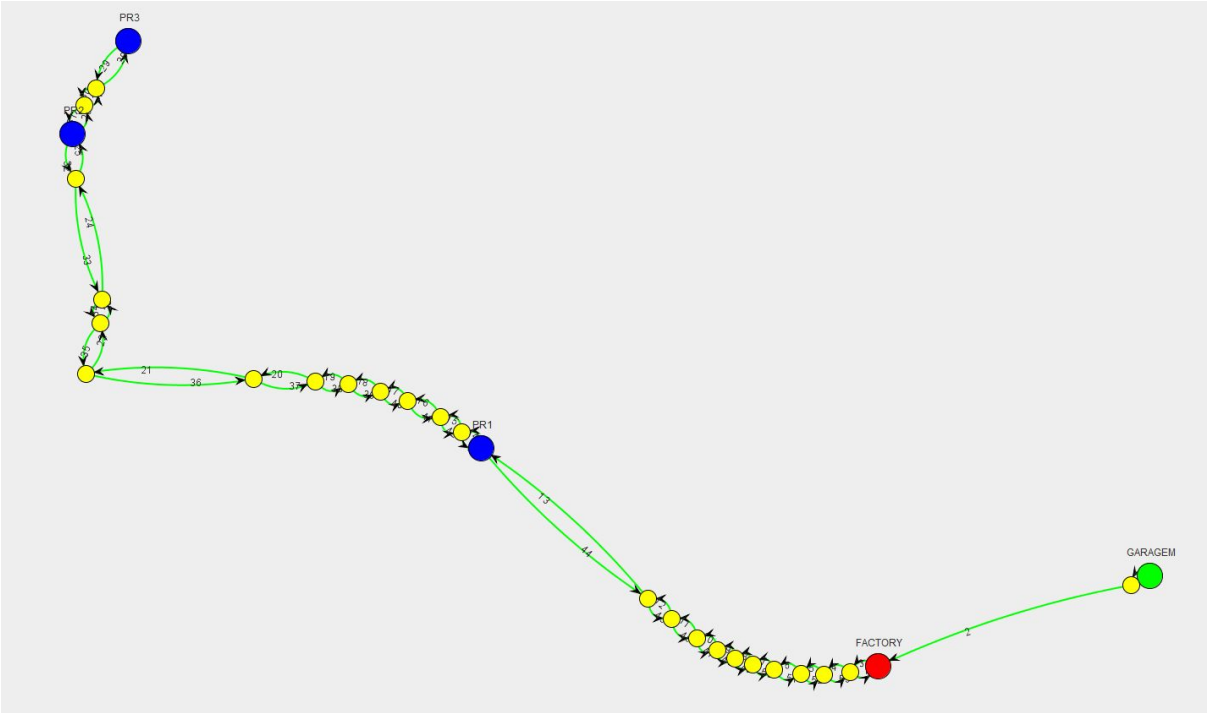
Depois de ler o serviço com sucesso para um objeto do tipo “Service” o programa pede ao utilizador para escolher um dos algoritmos disponíveis para calcular o caminho. Escolhido o algoritmo, é calculado o caminho e atribui-se o mesmo a um veículo. Este veículo irá por sua vez ser atribuído ao serviço que acabou de ser processado e o próximo passo é dar display da solução no graphviewer. O que o utilizador visualiza enquanto isto tudo acontece é o seguinte:

```
[0] Load a graph and and generate CFC after reading its info .txt file (DO THIS BEFORE ANYTHING ELSE!!!)
[1] Display full graph
[2] Display accesible nodes from the garage
[3] Explain how to generate a service and list accesible nodes from the garage (in case you need help)
[4] Load a service and display optimal path solution
[5] Exit program
4


Reading service...
Insert the target service file name (no need for the directory and sufix but MUST be .txt):
teste
Done!
Calculating path...
What algorithm should be used?
0 -> Dijkstra's Shortest Path
1 -> Bellman-Ford algorithm
Option:0

 Working, this may take a while depending on CFC size.
Done!
Displaying service!
```

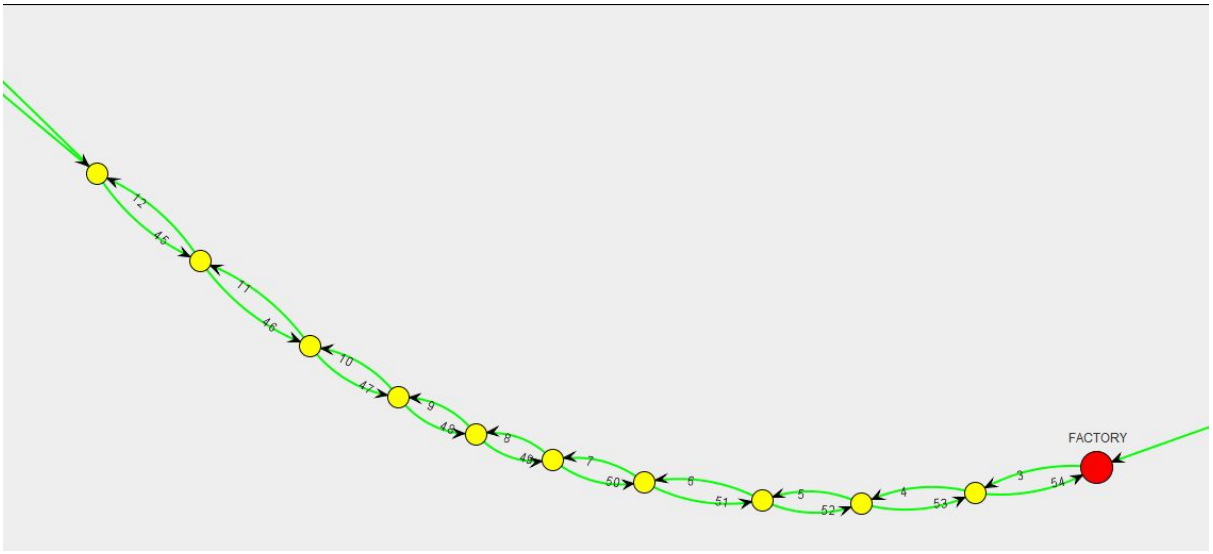
Como exemplo do display, segue-se um pequeno serviço para o grafo conexo de aveiro mostrado anteriormente.



Nesta função nós decidimos apenas mostrar os nodes e edges que efetivamente fazem parte do caminho bem como atribuir cores e tamanhos diferentes consoante o tipo:

- Verde: garagem
- Azul: ponto de recolha
- Vermelho: destino intermédio / instalações da empresa
- Amarelo: node normal pelo qual é preciso de passar
- Todos os pontos amarelos têm tamanho reduzido em relação aos outros

De notar também, que as edges são desenhadas como “directed” e têm como label o número da ordem em que é suposto ser usada. Desta maneira, com grafos mais densos pode ficar um pouco confuso, mas se for feito zoom continua a ser bastante percetível.



11. Discussão sobre a conetividade

A conectividade do grafo é calculada quando é lido o ID do node que corresponde à garagem numa determinada cidade. Depois de a encontrar é utilizada a seguinte função para obter tudo o que é acessível a partir da garagem.

```
vector<Vertex<Node>*> cleanEdgesNVertex(Graph<Node> graph, Vertex<Node>* garage){
    vector<Vertex<Node>*> visitedVertex;
    //--------------------CLEAN USELESS EDGES--------------------
    for(auto v : graph.getVertexSet()) {
        for (int i = 0; i < v->getAdj().size(); i++) {
            if (v->getAdj().at(i).getWeight() <= 0) {
                v->removeEdge(i);
                i--;
            }
        }
        v->setVisited( v: false);
    }
    //--------------------GET CFC----------------------------
    graph.DepthFirstSearch(garage,  &: visitedVertex);

    return visitedVertex;
}
```

Primeiramente é realizada uma pesquisa linear no grafo para remover todas as edges com um peso menor ou igual a 0 (são inúteis)

ao mesmo tempo que garante que nenhum dos pontos está marcado como visited (pode acontecer devido a várias leituras consecutivas).

De seguida começando na garagem, é feita uma pesquisa em profundidade pelo grafo.

```
<T>
void Graph<T>::DepthFirstSearch(Vertex<T> *v, vector<Vertex<T>* > & accessible) const {
    v->visited = true;
    accessible.push_back(v);
    for (auto & e : v->adj) {
        auto w = e.dest;
        if ( ! w->visited)
            DepthFirstSearch(w, & accessible);
    }
}
```

Nesta pesquisa a cada node novo visitado, o endereço deste será adicionado a um vetor (visitedVertex) que no fim vai conter todos os nodes acessíveis a partir da garagem.

Assim, ao ler um serviço para uma cidade se este conter algum node que não esteja nesse vetor, significa que não será possível realizá-lo por completo. O algoritmo continuará a processar o resto dos pontos e irá calcular um caminho ótimo não tendo em conta pontos inacessíveis indicando, antes de terminar, que nodes não foram usados no cálculo do caminho .

12. Algoritmos Implementados

Pesquisa binária por um vetor ordenado de Vertex

O primeiro algoritmo utilizado é o de pesquisa binária que, a partir de um vetor ordenado pesquisa o elemento do "meio" e afere se o elemento que estamos à procura é de facto esse. Se não for, altera o limite esquerdo ou direito e pesquisa novamente. Isto repete-se até encontrar o elemento ou até o limite esquerdo ser maior que o direito(neste caso retorna nullptr).

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m - 1
        else:
            return m
    return unsuccessful
```

Pesquisa em profundidade (Depth First Search)

O segundo algoritmo utilizado é o de pesquisa em profundidade que, partindo do vértice da garagem, reconhece todos os vértices acessíveis a partir deste. A aplicação deste algoritmo permite reduzir bastante o número de vértices com que o programa e os seus outros algoritmos têm de trabalhar após esta operação.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)
```

Algoritmo de Dijkstra

Como os grafos com que trabalhamos são maioritariamente esparsos, ou seja tem aproximadamente o mesmo número de vértices e arestas, o algoritmo de dijkstra mostra-se eficiente para o objetivo pretendido. Após identificar e isolar todos os vértices que são acessíveis a partir da central, a aplicação do algoritmo de dijkstra a cada um deles permite obter a distância entre os mesmos, e qual o caminho a percorrer para obter o valor de distância mais curto possível entre a garagem e a central.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
```

Algoritmo de Bellman-Ford

Este algoritmo é bastante semelhante ao de Dijkstra, no entanto permite trabalhar com grafos em que existe arestas com peso negativo. Se existir um caminho em que a soma dos pesos seja negativo, desde a origem, este será sempre o caminho mais “barato”, pelo que este algoritmo permite detetar e excluir estes casos.

```
function bellmanFord(G, S)
        for each vertex V in G
                        distance[V] <- infinite
                        previous[V] <- NULL
        distance[S] <- 0
        for each vertex V in G
                for each edge (U,V) in G
                        tempDistance <- distance[U] + edge_weight(U, V)
                        if tempDistance < distance[V]
                           distance[V] <- tempDistance
                           previous[V] <- U

        for each edge (U,V) in G
                If distance[U] + edge_weight(U, V) < distance[V}
                        Error: Negative Cycle Exists
```

Algoritmo de Floyd-Warshall

Este algoritmo é mais eficiente para grafos densos do que os anteriores. Ao ser executado, ao contrário dos primeiros algoritmos abordados, este avalia a distância entre todos os pares de vértices do grafo, em vez de apenas entre os dois pontos desejados, guardando a informação relativa ao caminho mais curto entre cada par.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

13. Análise de Complexidade dos Algoritmos Implementados

Para as seguintes descrições, será utilizado |V| e |E| para representar o número total de vértices e arestas, respectivamente, que cada algoritmo utiliza no seu processo, sendo que, tirando no algoritmo de pesquisa em profundidade, onde são verificados todos os vértices do grafo, todos os outros atuarão apenas no subgrupo de vértices alcançáveis desde a central (obtidos com este primeiro algoritmo).

Análise teórica dos Algoritmos:

Pesquisa binária por um vetor ordenado de Vertex

Cada elemento do vetor sujeito à pesquisa binária é percorrido apenas uma vez, fazendo log(n) comparações entre os mesmos, pelo que a sua complexidade será O(log(|V|)).

Complexidade: O(log|V|)

Pesquisa em profundidade (Depth First Search)

Cada vértice do grafo é processado apenas uma vez, e para cada um dos vértices, todas as suas arestas são processadas apenas uma vez.

Complexidade: O(|V| + |E|)

Algoritmo de Dijkstra

Como o algoritmo opera desde um dos vértices para todos os outros, todos os vértices serão processados uma vez, pelo que para cada um deles as suas arestas serão igualmente processadas uma vez, o que resulta numa complexidade de O(|V| + |E|).

Ao utilizar uma fila de propriedades mutáveis, as operações de inserção e extração também impactarão a complexidade. A fila terá |V| operações, e executa cada uma delas num tempo logarítmico no tamanho da fila (que terá um valor máximo de |V|), o que resulta numa operação de complexidade O(|V| * log|V|). Além disso, esta fila necessitará também de atualizar todas as posições dos elementos já inseridos, tendo isto um valor máximo de |E|, também num tempo logarítmico no tamanho da fila, o que se traduz em O(|E| * log|V|).

O(|V| + |E| + |V| * log|V| + |E| * log|V|) = O((|V| + |E|) * log|V|)

Complexidade: O((|V| + |E|) * log|V|)

Algoritmo de Bellman-Ford

Embora seja mais simples que o algoritmo de Dijkstra, este algoritmo percorre, para cada vértice do conexo, todas as arestas possíveis, o que lhe atribui uma complexidade de O(|V| * |E|), a qual é bastante superior à do algoritmo anterior.

Complexidade: O(|V| * |E|)

Algoritmo de Floyd-Warshall

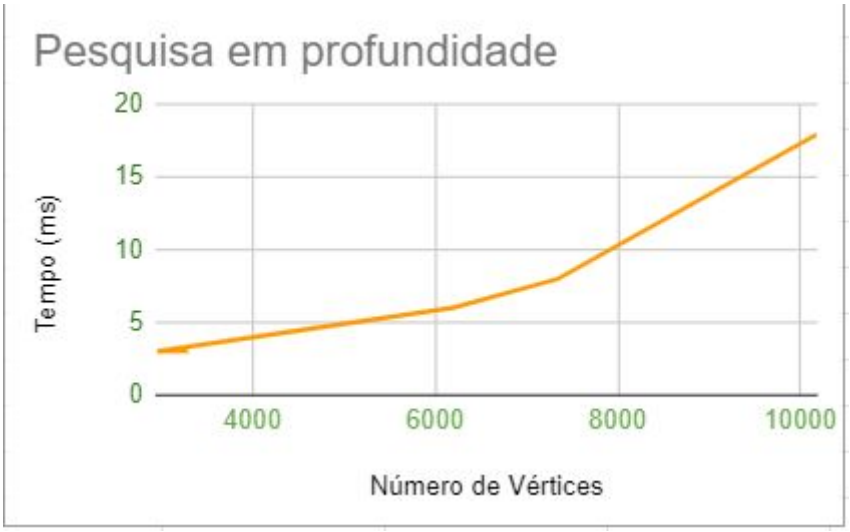
Este algoritmo consiste num ciclo que percorre todos os vértices do grafo, sendo que para cada um volta a repetir a mesma operação. Após estes dois ciclos, voltará a percorrer todos os vértices para fazer o processamento do caminho mais curto, o que se traduz numa complexidade de O(|V| * |V| * |V|).

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

Análise temporal empírica:

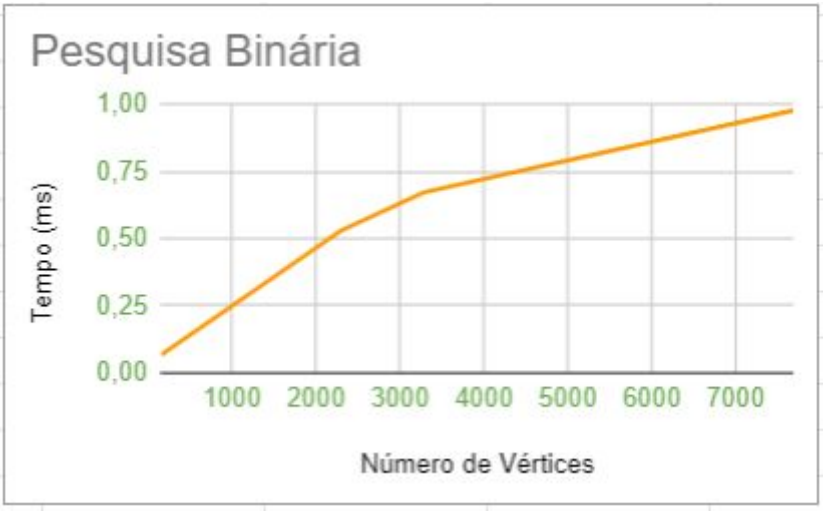
Pesquisa em profundidade

Cidade	V	E	Tempo (ms)
Aveiro	3294	3135	2,991
Ermesinde	2949	3134	2,983
Viseu	6193	6175	5,984
Gondomar	7340	7504	7,972
Porto	10176	10765	17,952



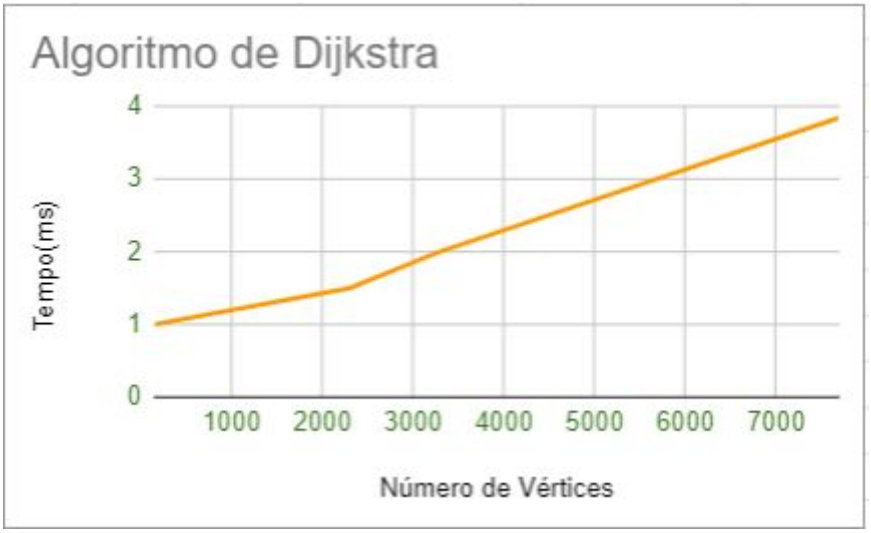
Pesquisa Binária por um vetor ordenado

Cidade	V	E	Tempo (ms)
Aveiro	152	158	0,066
Ermesinde	2294	2355	0,53
Gondomar	3282	3299	0,673
Porto	7687	7840	0,979



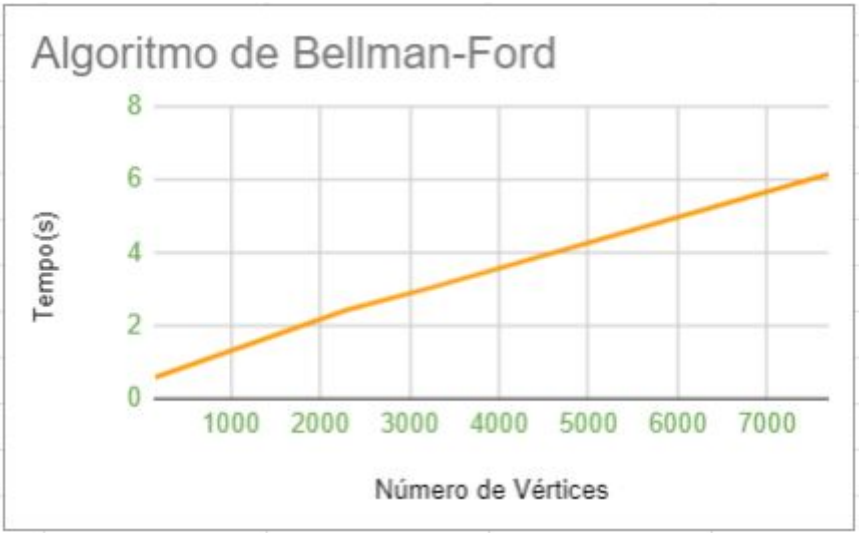
Algoritmo de Dijkstra

Cidade	V	E	Tempo (ms)
Aveiro	152	158	0.997
Ermesinde	2294	2355	1.495
Gondomar	3282	3299	1.994
Porto	7687	7840	3.847



Algoritmo de Bellman-Ford

Cidade	V	E	Tempo (s)
Aveiro	152	158	0.565
Ermesinde	2294	2355	2.421
Gondomar	3282	3299	3.061
Porto	7687	7840	6.158



14. Conclusão e algumas considerações

Embora, como já dito antes, não tenhamos conseguido fazer tudo a que nos tínhamos proposto na primeira parte, somos da opinião que o que de facto fizemos, fizemos bem. Inicialmente no nosso código havia a opção de o utilizador escolher o algoritmo que prefere para encontrar o caminho para o serviço, Dijkstra ou Bellman-Ford. Mais tarde apercebemo-nos que este último não fazia grande sentido dado que as edges nulas ou negativas teriam de ser removidas no contexto do projeto. Ainda assim, deixamos o algoritmo como opção visto que encontra a solução ótima como deve e embora a performance seja ligeiramente pior é na mesma decente.

Considerando isto e tendo praticamente só um e meio algoritmo (sendo o de Bellman-Ford algo redundante neste contexto) para encontrar o caminho a funcionar, decidimos tentar adicionar o de Floyd-Warshall mesmo sabendo que iria ser pior em termos de performance devido ao rácio vertex/edges. Infelizmente não o conseguimos implementar por completo devido a um par de bugs que não conseguimos corrigir a tempo pelo que este se encontra comentado no código-fonte. Em retrospectiva, esta decisão não foi das melhores visto que nos consumiu mais tempo do que devia, tempo que podia ter sido usado para implementar outras funcionalidades e até acabamos por não conseguir acabar o dito algoritmo.

Finalmente, apesar de todos os problemas e contratempos que enfrentamos ao realizar este projeto somos da opinião que, a nível individual conseguimos atingir os objetivos do mesmo.

15. Esforço dedicado

Esforço Individual:

Nuno Oliveira - up201806525

- Quantitativo: 50%

Luís Miranda - up201306340

- Quantitativo: 50%

Romeu Flores - up201806287

- Quantitativo: 0%

À semelhança da primeira parte, o Luís Miranda e o Nuno Oliveira foram-se ajudando mutuamente ao longo do projeto pelo que se pode dizer que ambos estiveram envolvidos em tudo.