Relatório CAL - 1ª Parte Trabalho Prático
11/4/2019

Tema 8 — SecurityVan: Entrega de Valores

Grupo D - Turma 4

Eduardo Ribeiro —> up201705421@fe.up.pt

Eduardo Macedo —> up201703658@fe.up.pt

Diogo Machado —> up201706832@fe.up.pt

Nota: a 2ª parte do relatório pode ser encontrada no final.

Descrição do Tema

O trabalho está relacionado com o conceito de veículos especializados em transporte e entrega de valores, que transportam grandes quantias de dinheiro e outros objetos valiosos, de um ponto para outro.

O trabalho consiste em implementar um sistema que permite a identificação de rotas ótimas para tais transportes, de modo a ser utilizado por uma empresa que se especializa em transporte de valores. Inicialmente, deve ser considerado que a empresa apenas dispõe de um veículo, a fazer todos os trajetos/entregas. Numa fase mais posterior, considerar-se-à que a empresa tem vários veículos e que os seus trabalhos/trajetos poderão ser especializados por tipo de cliente (bancos, museus, etc).

Estes transportes implicam a recolha prévia dos valores em certos pontos, para depois os entregar.

Deveremos, então, inicialmente, definir a sequência ordenada dos pontos de interesse pelos quais o(s) veículo(s) terão de passar, considerando que os trajetos deste(s) começarão e acabarão sempre na central. Há que ter em conta que os destinos têm de aparecer depois que as origens correspondentes, na sequência ordenada.

Um exemplo de um trajeto que um veículo pode fazer é:

central —> origem1 —> origem2 —> destino2 —> origem3—> destino3 —> destino1 —> central

Depois de se saber a sequência ordenada dos pontos de interesse, é necessário saber qual o caminho mais rápido de um ponto até outro, dois a dois, definindo assim a melhor rota para o veículo.

A entrega só poderá ser efetuada se existirem caminhos que liguem a central dos camiões, as origens, os destinos, e de volta à central, dois a dois. Ou seja, para a entrega se poder concretizar, é necessário que os pontos façam parte do mesmo componente fortemente conexo do grafo, pelo que é de extrema importância a avaliação da conectividade do mesmo.

Por último, será também necessário considerar que obras nas vias públicas poderão tornar certas zonas do mapa inacessíveis, podendo ser impossível chegar a certos clientes; devem ser identificados os pontos de recolha/entrega com acessibilidade reduzida.

Identificação e Formalização do Problema

Dados de Entrada

- R —> tabela com valores das distâncias/tempos mínimos entre cada par ordenado de vértices, e possivelmente o caminho que leva a essas distâncias/tempos (ver perspetiva de solução);

- Ti —> sequência de trajetos/entregas que será necessário fazer, sendo i o i-ésimo elemento. Cada entrega será caracterizada por:

    - type —> tipo de cliente (museu, banco, loja, …) (na 1ª parte do problema, isto não vai ser considerado, uma vez que há apenas um veículo para todo o tipo de transportes);

    - origem —> vértice em que será necessário ir recolher os bens;

    - destino —> vértice em que será necessário entregar os bens, depois de estarem dentro do veículo;

    - ID —> um ID único e específico, para cada trajeto/entrega.

- Ci —> veículo número i, que está disponível para fazer um percurso (na 1ª parte do problema, apenas vai ser considerado um único veículo). Cada um tem:

    - type —> tipo de entrega/trajeto (museu, banco, loja, …) em que o veículo é especializado (na 1ª parte, o único veículo irá fazer todos os trajetos, pelo que não terá especialidade).

- G = (V, E) —> grafo dirigido(*) pesado, representando o mapa da cidade. Este grafo irá ser constituído por vértices (V) e arestas (E).

Cada vértice (representam pontos de importância no mapa: possíveis clientes, interseções entre vias de trânsito, etc.) é caracterizado por:

- gpsCoords —> coordenadas GPS do ponto no mapa (opcional; ver perspetiva de solução);

- type —> o vértice representa um museu, banco, loja, ou nenhum dos anteriores;

- Adj $\subseteq$ E —> arestas que saem desse vértice.

Cada aresta (representam ruas, pontes e outras vias) é caracterizada por:

- w —> peso da aresta, que representa o tempo/distância do percurso entre os dois vértices que esta liga;

- ID —> identificador único de uma aresta;

- dest $\in$ V —> vértice de destino da aresta;

- S $\in$ V —> vértice inicial, que representa a central de onde os veículos saem.

(*) —> é considerado dirigido, pois algumas ruas são de sentido único.

Dados de Saída

- Gf = (Vf,Ef) grafo dirigido pesado, tendo Vf e Ef os mesmos atributos que V e E.
- Cf —> sequência ordenada de todos os veículos usados, cada um com:
  - type —> tipo de serviço/trajeto que esse veículo fez (igual ao desse veículo no início do algoritmo) (na 1ª parte do problema, não vai ser considerado, porque será um veículo a fazer todos os trajetos).
  - D —> vetor ordenado com os IDs das arestas pelo qual o veículo terá de passar. Pode haver repetidos.
  - Tf —> vetor com os IDs dos trajetos/entregas que esse veículo fez.

Restrições

Restrições nos dados de entrada (Pré condições):

- $\forall$ i $\in$ [1; T.size()] :
  - type(T[i]) = “BANK” v “MUSEUM” v “SHOP” —> o tipo de transporte/cliente terá de ser: agência bancária, ou museu, ou loja.
  - origem(T[i]) $\in$ V —> a origem terá de ser um dos vértices do grafo.
  - destino(T[i]) $\in$ V —> o destino terá de ser um dos vértices do grafo.
  - type(origem(T[i])) = type(destino(T[i])) = type(T[i]) —> será considerado apenas trajetos de museu para museu, banco para banco, etc. O tipo do vértice de destino terá de ser igual ao tipo do trajeto, bem como o tipo da origem. Esta restrição também assegura que a origem e o destino serão sempre ou um banco ou um museu ou uma loja, e não vértices normais.
  - origem(T[i]) ≠ destino(T[i]) —> os vértices de destino e origem terão de ser diferentes.
- $\forall$ V, type(V) = “BANK” v “MUSEUM” v “SHOP” v “NONE” —> para além de termos vértices a representar os possíveis clientes, também temos vértices “normais”.
- $\forall$ C $\in$ Ci, type(C) = “BANK” v “MUSEUM” v “SHOP”, exceto na 1ª parte do problema, em que apenas é considerado um veículo para tudo.
- type(S) = “NONE” —> a central dos veículos será um vértice do tipo “NONE”.

- $\forall$ E, w(E) > 0 —> o peso das arestas terá de ser positivo uma vez que representam distâncias/tempos.

- $\forall$ E, E deve ser utilizável pelo(s) veículo(s). As arestas que não cumprirem isto serão removidas por um pré-processamento, e não serão incluídas no grafo G inicial.

- Seja L o conjunto de vértices constituído por S (central), todos as origens, e todos os destinos. É necessário que todos os elementos de L estejam no mesmo componente fortemente conexo do grafo (CFC).

  Deste modo, a partir de qualquer elemento do conjunto, é possível aceder qualquer outro, no grafo dirigido. Dado que todos os percursos começam e acabam no mesmo ponto (central), então todos os vértices do percurso devem pertencer ao mesmo CFC. Cada percurso então pode ser considerado como um CFC, sendo que como terão pelo menos um vértice em comum (central), irão originar um grande CFC, composto pela central, todas as origens, e todos os destinos (conjunto L).

Restrições nos dados de saída (Pós condições):

- Cf.size() <= Ci.size() —> obviamente, não será possível utilizar mais veículos do que os disponíveis no início.

- $\forall$ C $\in$ Cf:

  - $\forall$ i $\in$ [1; Tf.size()], type(C) = type (Tf[i]) —> um veículo de um dado tipo só pode fazer trajetos/entregas desse tipo (não aplicável à 1ª parte do projeto).

  - Seja di o primeiro elemento de D. É preciso que di $\in$ Adj(S), uma vez que todos os percursos começam a sair da central dos veículos.

  - Seja df o último elemento de D. É necessário que dest(df) = S, uma vez que todos os percursos devem acabar por retornar à central dos veículos.

  - $\forall$ i $\in$ [1; Tf.size()], $\exists$ d1, d2 $\in$ D tal que (dest(d1) = origem(Tf[i])) ^ (dest(d2) = destino(Tf[i])) ^ (d1 < d2) —> se um trajeto foi incluído no vetor de trajetos desse veículo, significa que este concluiu o dado trajeto, ou seja, passou pelo menos uma vez na origem, e depois no destino do trajeto em causa. Por causa disto, o vetor de arestas pelo qual o veículo terá de passar terá de possuir pelo menos duas arestas, cujos destinos sejam a origem e o destino do trajeto, respetivamente, sendo que esta última terá de vir depois da primeira, no vetor ordenado (isto não elimina a possibilidade de se passar no destino antes da origem).

$$\min \sum_{c \in Cf} \sum_{d \in D} w(d)$$

—> os caminhos percorridos pelos veículos serão os que permitirem fazer as entregas na menor distância/tempo total possível.

Função objetivo

A solução ótima do problema reside em, como já sabemos, encontrar as rotas ótimas para a entrega dos valores por parte dos veículos. Ou seja, queremos que eles terminem as suas tarefas/trajetos na menor distância/tempo total possível. Assim, a solução ótima irá passar pela minimização da soma das distâncias/tempos por todos os veículos utilizados:

$$\min \sum_{c \in Cf} \sum_{d \in D} w(d)$$

Perspetiva de solução

Para resolver o problema proposto, formulamos um plano com cinco etapas, sendo algumas delas “pré-etapas” auxiliares:

0) Cálculo dos valores da tabela auxiliar, que será utilizada para acelerar o algoritmo em si (só necessário da primeira vez que o algoritmo é executado);

1) Remoção das arestas indesejáveis para a execução do algoritmo:
- 1ª iteração) remoção das que representam vias inutilizáveis pelos veículos (estas serão indicadas no grafo com peso = 0);
- 2ª iteração) no caso de haver mais que uma aresta com a mesma origem e destino, remoção das que não têm peso mínimo (caso improvável, mas possível);

2) Verificar se os pontos de interesse (central, origens e destinos) pertencem todos a um comum Componente Fortemente Conexo;

3) Ordenação dos pontos de interesse (origens/destinos) a ser incluídos no(s) trajeto(s):
- Numa etapa inicial, o algoritmo será apenas para um veículo, sendo que a ordenação será entre todas as origens e destinos em questão;
- Posteriormente, passarão a poder ser utilizados vários veículos, sendo que será feita uma divisão das entregas por diferentes trajetos, cada um associado a um dos veículos (é tido em conta, também, a especialização dos veículos por tipos de cliente).

4) Sabendo já a ordem do(s) trajeto(s), calcular as arestas a serem percorridas, de modo a garantir a rota ótima para cada veículo, com o tempo mínimo.

Os **passos 0) e 1)** apenas teriam de ser executados quando se é inserido um novo grafo. O passo 2) ocorreria numa fase de pré-processamento, sendo que o algoritmo em si seria composto pelos passos 3) e 4).

Para o **segundo passo**, utilizaremos o algoritmo apresentado nas aulas teóricas para determinação dos CFCs de um grafo:

1. Pesquisa em profundidade no grafo G origina uma floresta de expansão, numerando os vértices em pós-ordem;
2. Inverter todas as arestas de G;
3. Segunda pesquisa em profundidade no grafo invertido, começando sempre pelo vértice não visitado de numeração mais alta;
4. Cada árvore obtida é um CFC;

5. Percorrer cada CFC até encontrar um que inclua todos os vértices do percurso (se não for encontrado nenhum, então não existe trajeto possível, logo o algoritmo não pode ser executado).

O **terceiro passo** poderia ser resolvido de várias formas, mas para o fazer em tempo útil, e sem utilizar uma quantidade desproporcional de recursos, foram concebidas duas possíveis abordagens, que determinam o tipo de dados que estarão contidos na tabela auxiliar:

Abordagem 1 – O peso de cada aresta representaria o tempo de viagem entre os dois vértices ligados por esta. Para além disso, cada vértice teria as suas coordenadas GPS, de modo a possibilitar o cálculo da distância diagonal entre dois dados pontos do mapa. Na tabela auxiliar seriam armazenadas as distâncias diagonais entre todos os pares de vértices. A ordenação das origens e destinos de cada trajeto iria ser feita da seguinte maneira:

- Começa-se por ter apenas a Central como ponto de partida e de chegada (percurso Central —> Central). A comparação das distâncias entre os pontos é feita da seguinte maneira:
    - Se houver um caminho direto entre os pontos em questão (os pontos são adjacentes), utiliza-se o valor da aresta que os liga como dado de comparação; caso contrário, acede-se à tabela auxiliar, e utiliza-se a distância diagonal entre os dois vértices;

- Escolhe-se um par origem-destino arbitrário. Para cada posição possível da sequência (entre os pontos de partida e de chegada, que devem manter-se), calcula-se o acréscimo de distância total do percurso, caso a origem seja aí inserida:
    - Exemplo: para inserir a Origem 2 (O2 daqui em diante) entre a Central (C) e O1 – calcula-se

      (dist(C, O2) + dist(O2, O1) – dist(C, O1) **)**

    - Caso particular: para a inserção do primeiro par origem-destino, não é necessário este cálculo, pois há apenas uma posição possível.
- A origem é adicionada na posição para a qual o acréscimo de distância/tempo total é menor;
- Para o destino correspondente, é feito o mesmo processo, mas apenas para as posições a seguir à sua origem.
- Também seria possível fazer da forma inversa (inserir primeiro o destino, e inserir a origem da mesma maneira, numa posição anterior ao destino), sendo que uma hipótese seria calcular ambas as possibilidades, que seguem uma heurística diferente, e eleger a que melhor cumpre a função objetivo.

Abordagem 2 – Para a abordagem 2 o processo é semelhante ao da abordagem 1. No entanto, neste caso, não seria utilizado o conceito de coordenadas GPS para os pontos do mapa. Em vez disso, será armazenado na tabela o tempo de viagem mínimo entre cada par ordenado de dois vértices do grafo (obtido através das arestas, que representam tempos), bem como, para cada par ordenado de vértices, a última aresta a ser percorrida no caminho entre eles, de modo a saber que arestas é que fazem parte do caminho que origina esse tempo mínimo. Há duas maneiras de fazer esses cálculos:

- Se o grafo for esparso (**$|E| = O(V)$**), será preferível utilizar o algoritmo de Dijkstra para cada um dos vértices do grafo (**$O(|V| * |E| * \log(V))$**)
- Se o grafo for mais denso (**$|E| = O(V^2)$**), será mais adequado usar o algoritmo de Floyd-Warshall (**$O(V^3)$**)

No cálculo das distâncias entre os pontos, utiliza-se sempre os valores armazenados na tabela, dado que temos o valor calculado para qualquer par de vértices. Apesar de exigir mais na etapa de preparação, esta abordagem trabalharia sempre com valores certos, e não com aproximações, como é o caso das distâncias diagonais pelas coordenadas GPS da abordagem 1. Para além disso, as arestas a serem percorridas para cada veículo poderão ser simplesmente obtidas através da tabela, sendo que não será necessário repetir esses cálculos.

Na segunda etapa do terceiro passo, será necessário efetuar a divisão das entregas em percursos, possivelmente para mais que um veículo, de modo a melhor cumprir a função objetivo. Aqui, cabe-nos decidir o que é mais importante:

- minimizar o tempo total de viagem (e assim, os custos de operação para a empresa);
- efetuar todas as entregas no menor tempo possível.

Embora este segundo ponto seja importante, além de não ser tão interessante resolver o problema desta maneira (afinal, a solução ótima seria sempre enviar um carro a cada origem, e depois ao destino correspondente), a solução resultante não faria sentido no contexto do problema, já que desperdiçar recursos não faz, de todo, parte de uma estratégia de negócio ideal. Assim, apresenta-se o problema de minimizar a distância/ tempo total percorrida pelos carros.

Para cada entrega:

- verifica-se qual o tipo de entrega (banco, museu, loja, …);
- identifica-se os veículos correspondentes a esse tipo (são os veículos candidatos a fazer essa entrega);
- para cada veículo candidato, tenta-se inserir a origem na sua sequência de trajetos, pelo método apresentado na abordagem 1 ou 2; o acréscimo da distância total para esse veículo será **deltaOrigem**; insere-se também o destino dessa entrega na sequência, cujo acréscimo será **deltaDestino**;

• a entrega será atribuída ao veículo cuja soma **deltaOrigem + deltaDestino** assuma o valor mínimo; naturalmente, a origem e o destino serão inseridos nas posições que resultem neste acréscimo mínimo, segundo as abordagens 1 e 2.

Este método permite atribuir uma dada entrega a um veículo que já tenha um percurso definido, ou a um veículo que ainda não tem percurso, e irá sair da central (irá escolher o que compensar mais).

Esta solução garantirá um balanço positivo na grande maioria dos casos, poupando, simultaneamente, muito tempo de processamento para o cálculo de novos percursos (em comparação, por exemplo, a uma resolução do tipo *brute-force*).

No **quarto passo** é necessário, para a sequência do trajeto de cada veículo, calcular o caminho mais curto entre cada par de vértices consecutivos. Tal pode ser feito de várias maneiras:

- aplicação do algoritmo de Dijkstra, com começo no primeiro vértice do par, parando o algoritmo quando for processado o segundo vértice do par;
- aplicação do Dijkstra bidirecional entre os dois vértices do par (pode haver ganhos temporais, mas devido à necessidade de computar o grafo invertido seria ocupada mais memória, e o tempo de pré-processamento poderia invalidar a vantagem temporal);
- entre outros…

Independentemente do algoritmo aplicado, iremos ter as arestas que o veículo deverá percorrer para seguir a rota ótima. Este conjunto de arestas irá ser retornado como dado de saída do algoritmo.

NOTA: Caso seja adoptada a abordagem 2 no terceiro passo do algoritmo, este quarto passo não será necessário, uma vez que as arestas a percorrer já estão calculadas, e presentes na tabela auxiliar.

Casos de utilização

- Importação de um grafo a partir de um ficheiro de texto;
- Visualização do grafo e da informação que este representa (vias de trânsito e pontos de interesse numa cidade) no GraphViewer;
- Importação dos veículos disponíveis, bem como o tipo de transporte que cada um efetua, a partir de um ficheiro de texto;
- O utilizador tem a possibilidade de escolher os pontos de interesse para a execução do algoritmo (origens e destinos, verifica a validade dos pontos escolhidos);
- O utilizador tem a possibilidade de exigir, ou não, a especialização dos veículos para o seu trajeto (escolha entre vários veículos para tudo ou vários veículos especializados);
- Depois de calculadas as rotas ótimas para cada veículo, mostrar as arestas escolhidas e os vértices por onde passam no GraphViewer (para cada veículo, uma cor diferente, por exemplo).

Conclusão

Desde cedo percebemos que existem diversas maneiras de tentar resolver este tipo de problemas. Após uma cuidada análise do enunciado do tema e da matéria das aulas teóricas, bem como aconselhamento por parte do professor das aulas práticas e dos monitores, achamos que conseguimos estruturar um bom plano, que leva a uma boa solução ao problema da identificação das rotas ótimas.

A nosso ver, a parte mais exigente e difícil do trabalho foi a determinação de uma estratégia que levasse à ordenação das origens e destinos por que cada veículo tem de passar. Como não há nenhum algoritmo que faça isto diretamente, definimos nós próprios uma estratégia de ordenação dos pontos de interesse.

Também achamos desafiador conceber a parte do algoritmo que decide que trajetos é que devem ser feitos por que veículos.

De resto, não foram encontradas mais nenhumas dificuldades de grande calibre. Somos da opinião que com este trabalho, dominamos agora a matéria que incide na manipulação de grafos com algoritmos, pelo que nos encontramos bem preparados para a implementação desta ideia na 2ª parte do trabalho.

Diogo Machado - up201706832@fe.up.pt

Tarefas:
- descrição do tema;
- identificação e formalização do problema (dados de entrada e saída, restrições);
- perspetiva de solução (principalmente passos 3 e 4);

Esforço Dedicado: 1 / 3

Eduardo Ribeiro - up2017054212@fe.up.pt

Tarefas:
- descrição do tema;
- identificação e formalização do problema (dados de entrada e saída, restrições);
- perspetiva de solução (principalmente passos 0, 1, 2 e 4);
- casos de utilização;

Esforço Dedicado: 1 / 3

Eduardo Macedo - up201703658@fe.up.pt

Tarefas:
- descrição do tema;
- identificação e formalização do problema (restrições, função objetivo);
- perspetiva de solução (principalmente passos 0, 1 e 4);
- casos de utilização;

Esforço Dedicado: 1 / 3

Bibliografia Utilizada: slides das aulas teóricas.

Relatório CAL - 2ª Parte Trabalho Prático
11/4/2019

Diferenças entre o planeamento da 1ª parte e a respetiva implementação final

Como seria de esperar, à medida que foi implementada a nossa visão do projeto, percebemos que existiam certos aspetos relativos ao nosso planeamento que teriam de ser modificados, por diversas razões.

Primeiramente, foi necessária a adaptação da nossa visão e implementação, devido aos grafos que nos foram disponibilizados pelos monitores: estes contém arestas bidirecionais, e não unidirecionais, como era esperado.

Seguidamente, acrescentamos mais alguns algoritmos e funcionalidades ao nosso programa, que achamos pertinentes no contexto do problema.

Em suma, o trabalho disponibiliza as seguintes opções:

- Leitura e carregamento de um grafo a partir de um ficheiro de texto;
- Leitura e carregamento de listagens de veículos e entregas, bem como a identificação do vértice do grafo que representa a central (devem ser especificados pelos utilizadores);
- Cálculo de uma tabela contendo a distância e o último vértice no caminho entre todos os pares de vértices acessíveis desde a central (feito com o algoritmo de Dijkstra ou com Floyd-Warshall; a escolha é do utilizador);
- Apresentação no GraphViewer do grafo total da região;

- Apresentação no GraphViewer do grafo constituído pelos vértices acessíveis a partir da central dos veículos (através de uma pesquisa em profundidade começando na central);

- Aplicação de um algoritmo de deteção de pontos de articulação, e identificação no GraphViewer dos mesmos (contribuí para a avaliação da conectividade do grafo);

- Cálculo das rotas ótimas para os veículos, tendo em conta as entregas que devem ser feitas e o tipo de veículos e entregas que estão definidos;

- Apresentação no GraphViewer dos resultados, sendo dada a opção de display dos resultados para todos os veículos, ou apenas para um grupo definido pelo utilizador.

Relativamente às heurísticas definidas na 1ª parte para calcular as rotas ótimas para os veículos, estas foram mantidas na implementação.

Discussão sobre estruturas de dados utilizadas

Tabela

A principal estrutura de dados utilizada no nosso projeto, gerada num pré-processamento de uma parte do grafo total dado, é a tabela referida acima, que contém a distância e o último vértice (path) no caminho entre todos os pares de vértices. A informação da tabela é calculada numa fase inicial, quando o utilizador indicar o vértice onde a central dos veículos se situa. Optamos pela utilização desta tabela pois, embora a fase de pré-processamento seja um bocado mais demorada, as distâncias e os paths nunca mais precisam de ser calculados (desde que não se mude a central, o que consideramos um acontecimento pouco comum). Para o cálculo das rotas ótimas para as entregas futuras que venham a ser processadas, basta aceder, em tempo constante, aos conteúdos da tabela.

A estrutura de dados utilizada para a tabela é:

```
unordered_map<Key, pair<double, Vertex<Node>*> >
```

Struct da chave:

```
struct Key {

    Vertex<Node>* first;
    Vertex<Node>* second;

    Key(Vertex<Node>* first, Vertex<Node>* second) {
        this->first = first;
        this->second = second;
    }

    bool operator==(const Key &other) const {
        return ((first == other.first)
                && (second == other.second));
    }

};
```

Hash function:

```
namespace std {

    template<>
    struct hash<Vertex<Node> > {

        size_t operator()(Vertex<Node>*& n) const {

            return ((hash<double>()(n->getInfo().getX())
                    ^ (hash<double>()(n->getInfo().getY()) << 1)) >> 1)
                    ^ (hash<int>()(n->getInfo().getID()) << 1);

        }

    };

    template<>
    struct hash<Key> {

        size_t operator()(const Key &k) const {

            return ((hash<Vertex<Node>*>()(k.first))
                    ^ (hash<Vertex<Node>*>()(k.second)) << 1);

        }

    };

}
```

Ou seja, para cada par de vértices, haverá um par que indica a distância entre os dois, representada por um double, e o path (vértice anterior no caminho), representado pelo endereço do vértice.

Inicialmente, no nosso programa, a tabela era gerada para todos os vértices do grafo total. Embora isso possa trazer algumas vantagens, sendo um pré-processamento mais abrangente, chegamos a conclusão que era demasiado demorado e pouco eficiente, pelo que optamos então por gerar a tabela para apenas os vértices acessíveis a partir da central.

Outra estratégia que tentamos implementar foi a não repetição de entradas na tabela (ex: se o par 1 - 2 já está na tabela, não meter o par 2 - 1). Para determinar a distância de um par não presente na tabela, apenas era necessário pesquisar a distância para o par invertido (como os grafos são bidirecionais, a distância entre 1 - 2 é igual a 2 - 1). Porém, para calcular o path, era necessário uma função mais complexa e recursiva, o que levava a perdas significativas de performance ao calcular o trajeto dos veículos. Por isso, apesar de fazer com que a tabela ocupe mais memória, optamos por guardar todos os pares na mesma.

Por fim, aproveitamos para referir também que, numa fase inicial, a estrutura utilizada para a tabela era um map, e não um unordered_map. Acabamos por trocar para o último, uma vez que as operações de pesquisa e inserção se fazem em tempo constante (em média), contrariamente ao que acontece no map (tempo logarítmico).

Graph, Vertex e Edge

As classes fornecidas nas aulas que representam grafos, vértices e arestas também foram utilizadas neste trabalho. A maior parte do funcionamento das mesmas não mudou, mas foram acrescentados alguns métodos que achamos conveniente ter.

Node

Uma vez que queríamos guardar várias informações em cada vértice do grafo, definimos uma classe Node, que contém um ID específico do vértice, as coordenadas X e Y do mesmo, o tipo de vértice (uma enumeração definida por nós) e as coordenadas X e Y de display (para o GraphViewer). O grafo utilizado é, então, um Graph<Node>.

```
enum TYPE { BANK,
            FIN_ADVICE,
            ATM,
            TAX_ADVISOR,
            AUDIT,
            MONEY_MOV,
            OTHER,
            CENTRAL };

class Node {

private:
    /**
     * ID do node.
     */
    int id;

    /**
     * Coordenada X do node.
     */
    double xCoord;

    /**
     * Coordenada Y do node.
     */
    double yCoord;

    /**
     * Tipo do node (BANK, MUSEUM, SHOP, etc)
     */
    TYPE type;

    /**
     * Coordenada X a ser utilizada para fazer display do nó no GraphViewer.
     */
    int displayX = 0;

    /**
     * Coordenada Y a ser utilizada para fazer display do nó no GraphViewer.
     */
    int displayY = 0;
```

Delivery

Para representar as entregas que têm de ser feitas, definimos uma classe Delivery, com alguns atributos e métodos de acesso aos mesmos. É mantido um vetor de deliveries, de modo a saber quais as entregas que têm de ser feitas nesse momento.

```
class Delivery {

private:

    /**
     * ID da entrega.
     */
    int id;

    /**
     * Tipo da entrega.
     */
    TYPE type;

    /**
     * Vertice origem da entrega.
     */
    Vertex<Node>* origem;

    /**
     * Vertice destino da entrega.
     */
    Vertex<Node>* destino;
```

Vehicle

Para representar os veículos que estão disponíveis num determinado momento, foi criada a classe Vehicle, que contém o ID específico do veículo e o tipo do mesmo.

Contém também um vetor de vértices, que representa os vértices pelos quais o veículo terá de passar para cumprir o seu trajeto (ex: se a central for 4, e o veículo tiver de

fazer uma entrega de 2 a 3, o vetor será {4, 2, 3, 4}). Para além disso, armazena um vetor com todas as entregas que este irá fazer no seu percurso.

Para além das típicas funções get e set, tem uma função que indica qual a melhor posição para inserir a origem e o destino de uma entrega no vetor de vértices dele, bem como o acréscimo de distância que a entrega iria implicar no percurso do veículo.

```
class Vehicle {

private:

    /**
     * ID do veiculo.
     */
    int id;


    /**
     * Tipo do veiculo.
     */
    TYPE type;


    /**
     * Vetor que ira conter os vertices pelos quais o veiculo tera de passar para cumprir o seu percurso.
     */
    vector<Vertex<Node>* > vehiclePath;


    /**
     * Vetor que contem as entregas atribuidas ao veiculo.
     */
    vector<Delivery> deliveries;
```

Discussão sobre a conectividade do grafo

A conectividade do grafo é avaliada de varias maneiras. Primeiramente, quando é fornecido pelo utilizador o vértice que representa a central dos veículos, é feita uma **pesquisa em profundidade** a partir desse vértice, o que resulta na identificação dos vértices do grafo que são acessíveis a partir da central. Este subconjunto de vértices representa o conjunto de vértices que “interessam” para as entregas e as rotas ótimas; se um determinado nó não é acessível a partir da central, que é o ponto de início e de fim de todos os percursos, então não é possível fazer nenhuma entrega que passe por lá, sendo esse nó inútil.

Fig: Cálculo dos nós acessíveis a partir da central

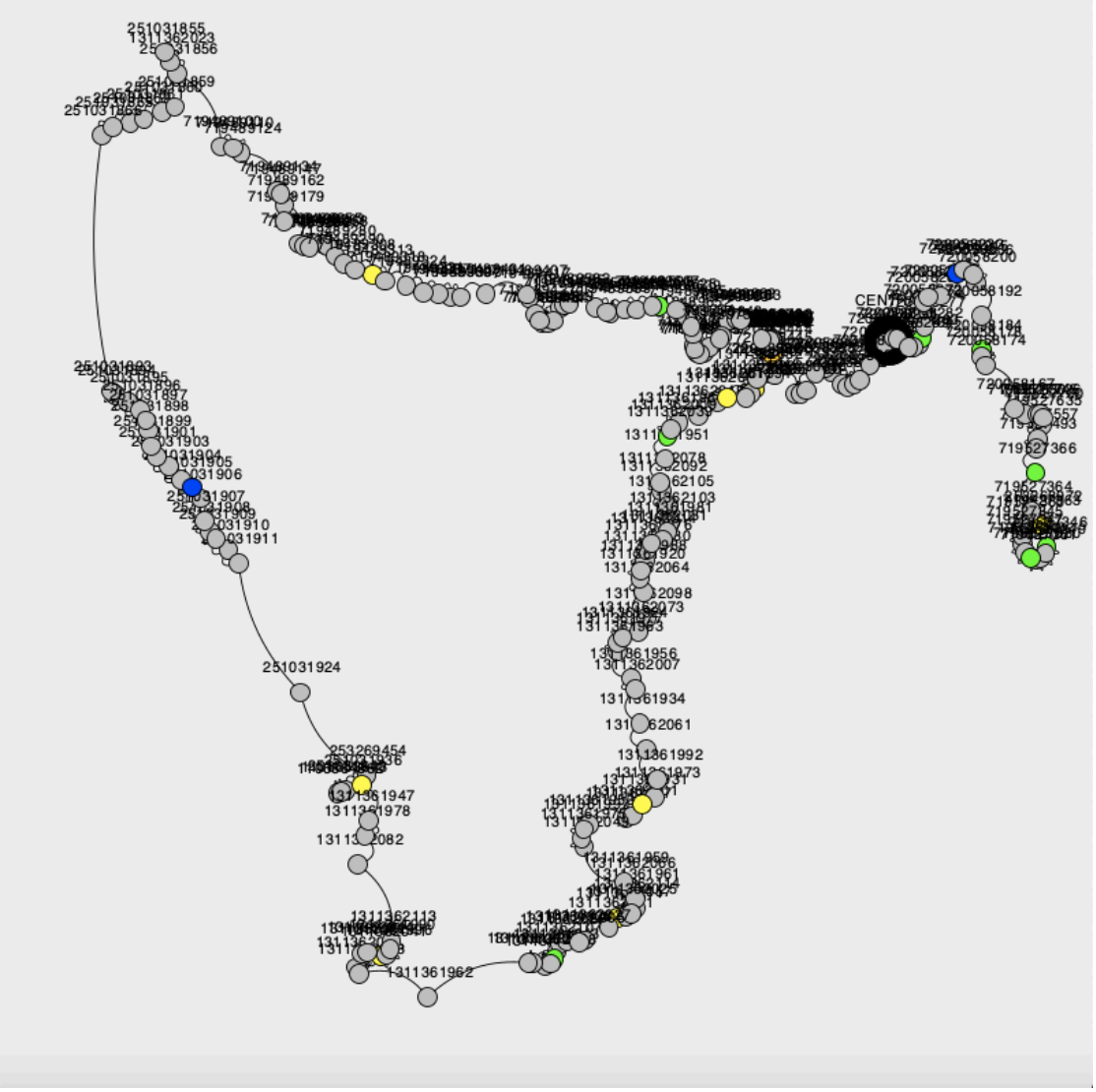
```
vector<Vertex<Node>* > calculateAccessNodesDisplayCoords(Graph<Node> graph, Vertex<Node>* central, int& width, int& height) {

    vector<Vertex<Node>* > accessNodes;
    double maxX = 0, maxY = 0, minX = INF, minY = INF;
    height = 800;


    for (auto v : graph.getVertexSet())
        v->setVisited(false);

    // faz uma visita em profundidade a partir da central; uma vez que o grafo e bidirecional,
    // isto faz com que o vetor retorne com todos os vertices acessiveis a partir da central.
    graph.dfsVisit(central, accessNodes);
```

Assim, depois de o utilizador definir as entregas que devem ser feitas, é verificado se os vértices de origem e destino de cada entrega pertencem a esse grupo de vértices acessíveis a partir da origem: se tal não for o caso, então a entrega não pode ser feita, e é retirada do vetor de entregas a fazer.

```
void pathExists(vector<Vertex<Node>* > accessNodes, vector<Delivery>& deliveries) {

    // como todos os percursos comecam e acabam na central, mesmo que as entregas
    // acabem por ser feitas por varios veiculos, nao tendo um que passar em todos
    // os pontos, e necessario na mesma que haja um caminho entre todos os pontos.

    // central nao pode ser origem nem destino!

    bool impDelivery = false;

    // verificacao dos pontos
    for(int i = 0; i < deliveries.size(); i++) {

        Delivery d = deliveries.at(i);

        if((find(accessNodes.begin(), accessNodes.end(), d.getOrigem()) == accessNodes.end()) ||
           (find(accessNodes.begin(), accessNodes.end(), d.getDestino()) == accessNodes.end())) {

            impDelivery = true;
            cout << "Delivery " << d.getID();
            cout << " (" << d.getOrigem()->getInfo().getID();
            cout << " -> " << d.getDestino()->getInfo().getID() << ") ";
            cout << "cannot be done." << endl;

            // apaga do vetor das entregas
            deliveries.erase(deliveries.begin() + i);
            i--;

        }
    }

    if(impDelivery)
        cout << "The impossible deliveries were removed." << endl;
    else
        cout << "Every delivery is valid!" << endl;
}
```

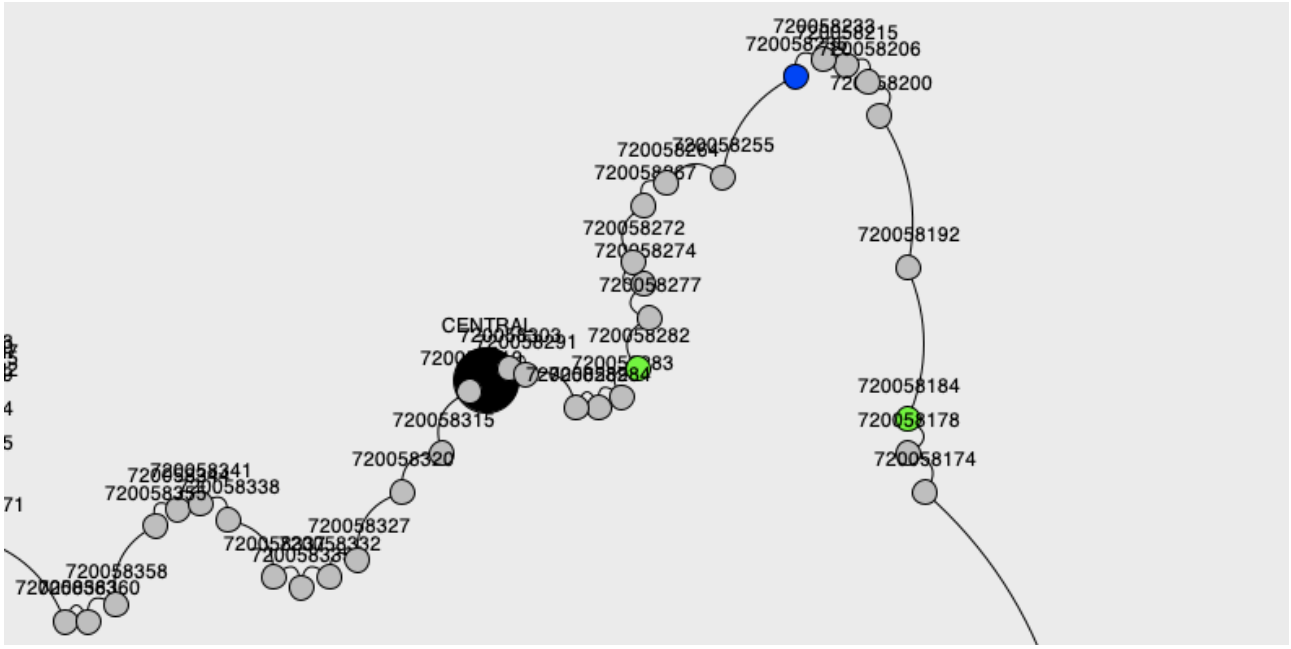
Como uma opção auxiliar do programa, implementamos uma opção de apresentação do grafo que contém os vértices acessíveis a partir da central, bem como os IDs de cada um, de modo ao utilizador poder definir entregas que utilizam pontos válidos.

Fig: Display, no GraphViewer, de todos os nós acessíveis a partir da central (utilizado o grafo de Coimbra).



NOTA: cada cor de vértice representa um tipo diferente: o vértice preto e grande representa a central, os cinzentos são os nós “normais”, os azuis são os pontos onde há bancos, etc.

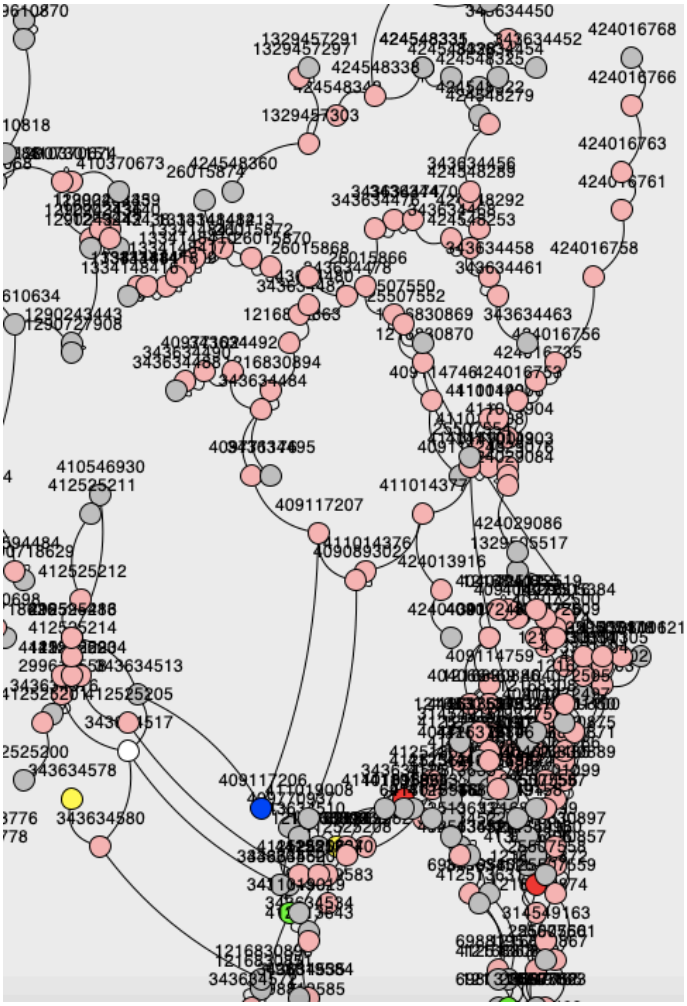
Fig: Imagem anterior, com zoom numa parte do grafo próxima da central.



Finalmente, e tal como o enunciado referia, obras nas vias públicas podem fazer com que certas zonas se tornem inacessíveis, inviabilizando o acesso ao destino de alguns clientes; era dito que se deveriam identificar pontos de recolha e entrega com pouca acessibilidade.

Para isto, e tendo em conta que os grafos são bidirecionais, decidimos implementar um **algoritmo de deteção de pontos de articulação**, que, se fossem removidos (o equivalente à ocorrência de obras neles, ficando inacessíveis), tornariam o grafo desconexo, impedindo o percurso a outros nós. Este algoritmo é aplicado ao sub-grafo constituído pelos vértices acessíveis a partir da central. O resultado do algoritmo é apresentado no GraphViewer, apresentando os pontos de articulação com a cor rosa.

Fig: Parte do gráfico do Porto, com os pontos de articulação a rosa.



Discussão sobre os principais casos de uso implementados

Nesta secção vão ser descritos os casos de utilização e funcionalidades que o programa apresenta.

Na imagem abaixo podemos observar o “menu principal” do programa.

```
------------------------------------------------
                CHOOSE AN OPTION
------------------------------------------------

0: Load new graph
1: Read new vehicle file and generate new table
2: Display graph
3: Display accesible graph from the central
4: Display articulated points
5: Read delivery file and calculate vehicle paths
6: Exit program
Option: 1
```

Como se pode observar, são dadas várias opções ao utilizador, como por exemplo carregar um novo grafo de um ficheiro de texto, ler vários ficheiros ficheiros que contém informação relevante, calculas as rotas ótimas para as entregas e apresentar todos os resultados no GraphViewer.

Carregamento de um novo grafo

Como já foi dito, é dada a opção ao utilizador para carregar um de vários grafos. Para além dos grafos geográficos fornecidos pelos monitores, é também disponibilizado um grafo de “Teste”, significativamente mais pequeno, feito por nós, para testar a execução do programa.

```
---------------------------------------------
                Choose Map
---------------------------------------------

Please select the desired map:
0 -> Aveiro
1 -> Braga
2 -> Coimbra
3 -> Ermesinde
4 -> Fafe
5 -> Gondomar
6 -> Lisboa
7 -> Maia
8 -> Porto
9 -> Portugal
10 -> Viseu
11 -> Teste

Value: 8

Building graph...
Done!

Removing useless edges...
Done!
```

Ler ficheiro dos veículos e central, e geração da tabela

De modo a definir os veículos que estarão disponíveis para fazer entregas, o utilizador deverá definir um ficheiro de veículos, que deverá conter as seguintes informações:

ID do vértice que será a central —>
Nº de veículos listados —>

ID do veículo; tipo de trajetos que faz —>
ID do veículo; tipo de trajetos que faz —>
… —>

```
111443920
7
1 BANK
3 ATM
4 FIN_ADVICE
5 ATM
6 MONEY_MOV
7 TAX_ADVISOR
8 TAX_ADVISOR
```

```
-------------------------------------------
          Central and Vehicle File
-------------------------------------------

Please insert the central and vehicles file name:
VehicleFile.txt
```

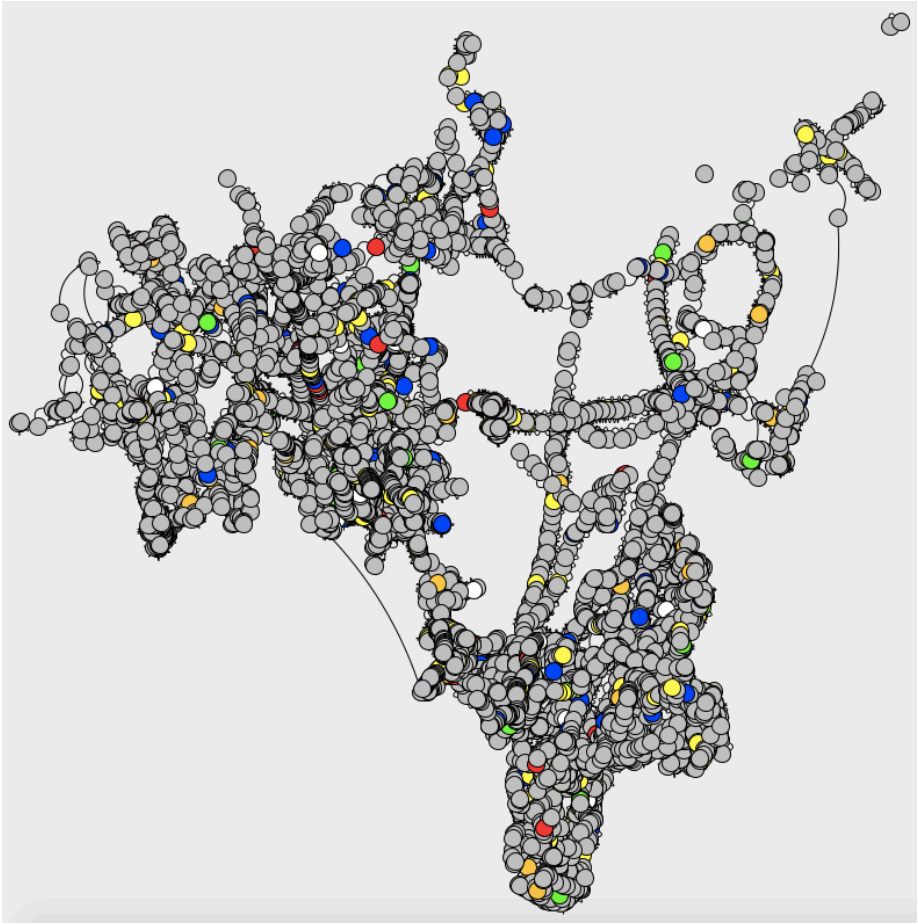
Depois de o ficheiro estar definido e selecionado, o programa, através de uma pesquisa em profundidade, calcula todos os nós acessíveis a partir da central.

Para a geração da tabela, é perguntado ao utilizador que algoritmo é que este prefere que seja utilizado no cálculo da informação que seria armazenada nela.

```
-------------------------------------------
          Central and Vehicle File
-------------------------------------------

Please insert the central and vehicles file name:
VehicleFile.txt
Calculating accessible nodes...
Done!

What algorithm should be used to fill the table?
NOTE: If the graph has number of edges roughly equal or lower than the number of nodes,
Dijkstra is recommended. Otherwise, Floyd-Warshall might be the better option.
0 -> Dijkstra
1 -> Floyd-Warshall
Option: 0

Building table...
Done!
```

Desenho/Exibição do grafo total no GraphViewer

É dado como opção a exibição do grafo total no GraphViewer; as dimensões dos nós e da janela são modelados de modo a que o grafo todo caiba na janela.



Desenho/Exibição do grafo com nós acessíveis no GraphViewer

Antes de gerar a tabela, são identificados os nós do grafo que são acessíveis a partir da central, através de, como já foi referido, uma pesquisa em profundidade. É dada ao utilizador do programa a possibilidade de visualizar o sub-grafo constituído por esses nós.

Estes estão identificados com os seus IDs para que, se o utilizador ainda não o tenha feito, definir nós de origem e destino válidos para as entregas.

Uma imagem que mostre o resultado da escolha desta opção pode ser encontrada mais atrás no relatório.

Desenho/Exibição dos pontos de articulação do grafo com nós acessíveis no GraphViewer

Como já foi referido anteriormente, se o grafo com os nós acessíveis estiver a ser exibido, poderá selecionar-se a opção de exibição dos pontos de articulação do mesmo, de modo a ser possível, de uma maneira mais fácil, a identificação dos nós que possivelmente terão pouca acessibilidade; tem como objetivo avaliar a conectividade do grafo.

Uma imagem que mostre o resultado da escolha desta opção pode ser encontrada mais atrás no relatório.

Ler ficheiro das entregas, aplicação de algoritmos para o cálculo de rotas ótimas, e apresentação dos resultados

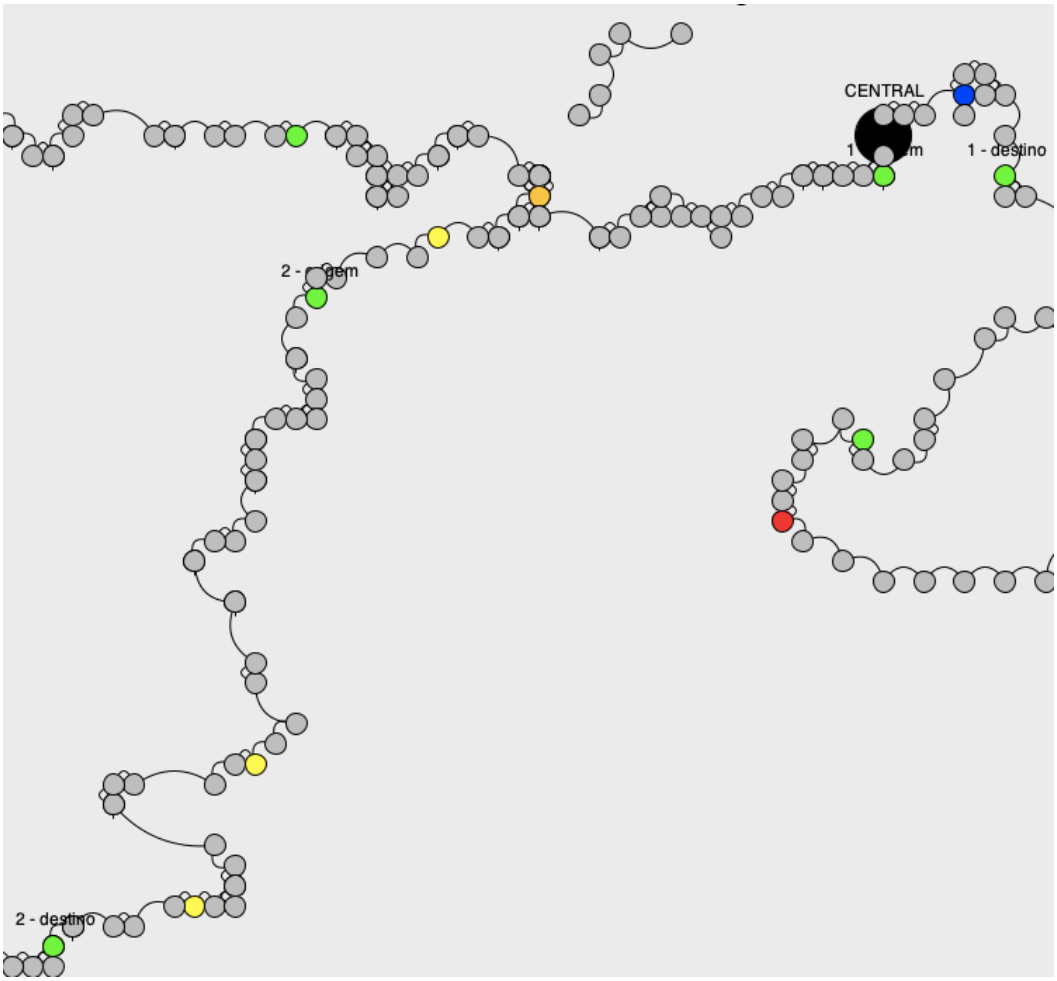
Antes de escolher esta opção, o utilizador deverá definir um ficheiro de texto contendo informação sobre as entregas que se pretendem que sejam efetuadas:

Nº de entregas a ler —>
ID da entrega; ID do vértice de origem; ID do vértice de destino —>
ID da entrega; ID do vértice de origem; ID do vértice de destino —>
… —>

```
11
1 998240321 998240237
2 392953854 998240226
3 390064496 506348132
4 391277672 998240440
5 624265125 26019988
6 26019976 26019990
7 402746347 392953852
8 391276720 390064918
9 998240278 998240375
10 391276859 391276858
11 998240268 998240555
```

Depois de ler a informação contida no ficheiro, e se a janela do GraphViewer contendo o grafo total estiver aberta, os nós de origem e destino de cada entrega são assinalados no mesmo:

Fig: identificação das origens e destinos das entregas lidas do ficheiro.



Seguidamente, é verificada a validade das entregas lidas do ficheiro: apenas podem ser feitas as entregas cujas origens e destinos pertençam ao vetor de vértices que são acessíveis a partir da central, que foi calculado previamente. As entregas que não cumprirem este requisito são retiradas do vetor de entregas, apenas sendo feitas as entregas válidas.

```
-----------------------------------------------
               Deliveries File
-----------------------------------------------

Please insert the deliveries file name:
DeliveryCoimbra.txt


The delivery nodes can be seen in the main graph.
Verifying the graph's conectivity and if all deliveries are possible...
Every delivery is valid!
Done!
```

A seguir, ele atribuí cada entrega a um veículo, de modo a gerar rotas ótimas para cada um deles. Os algoritmos e as heurísticas utilizadas nesta parte serão explicadas mais a frente no relatório.

Assim que este processamento termina, os seus resultados são exibidos no terminal para o utilizador tomar conhecimento de quais entregas é que cada veículo irá fazer, e quais as entregas que não poderão ser feitas (p. ex., não há veículos que façam esse tipo de entregas):

```
Assigning deliveries to vehicles...
Done!

For vehicle of id 1, the path is:
720058272 720058272
And the deliveries are:
-


For vehicle of id 3, the path is:
720058272 1311362039 1311361943 720058282 720058184 720058272
And the deliveries are:
1 2

For vehicle of id 4, the path is:
720058272 720058272
And the deliveries are:
-


For vehicle of id 5, the path is:
720058272 720058272
And the deliveries are:
-


For vehicle of id 6, the path is:
720058272 720058272
And the deliveries are:
-
```

```
For vehicle of id 7, the path is:
720058272 720058272
And the deliveries are:
-


For vehicle of id 8, the path is:
720058272 720058272
And the deliveries are:
-


And the deliveries that couldn't be made were:
-


To display the path of the vehicles, please press ENTER.
```

Finalmente, para o utilizador tomar conhecimento do caminho exato de cada veículo, é dada a opção de display, no GraphViewer, de um grafo, que representa os caminhos percorridos pelos veículos, de modo a fazerem as entregas que lhes foram atribuídas. O utilizador tem a liberdade de escolher que caminhos de que veículos é que pretende observar no grafo gerado:

```
To display the path of the vehicles, please press ENTER.

-----------------------------------------------
              CHOOSE AN OPTION
-----------------------------------------------

0: View all deliveries
1: View deliveries for a type of transport
2: View custom group of deliveries
3: Exit
Option:
```

Se escolher a opção 1:

```
Option: 1

-----------------------------------------------
              CHOOSE AN OPTION
-----------------------------------------------

0: Bank
1: Financial Advice
2: ATM
3: Tax Advisor
4: Audit
5: Money Movement
Option: 0

Vehicles selected: 1
Calculating and displaying the paths for the vehicles...
Done!
```

Se escolher a opção 2:

```
Option: 2

-----------------------------------------------
              CHOOSE YOUR OPTIONS
-----------------------------------------------

0: Done!
1: Vehicle 1
3: Vehicle 3
4: Vehicle 4
5: Vehicle 5
6: Vehicle 6
7: Vehicle 7
8: Vehicle 8
Option:
```

Fig: Grafo completo de um exemplo de um trajeto de um veículo.

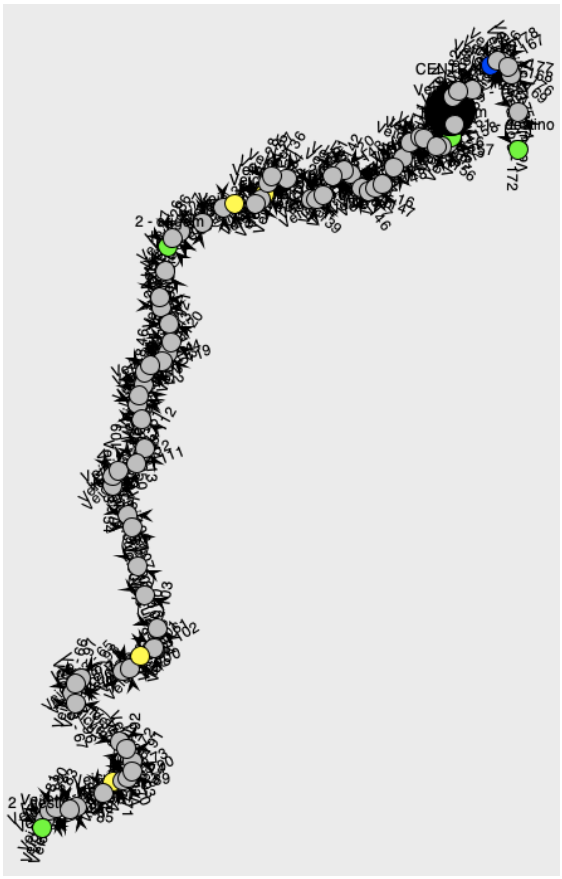
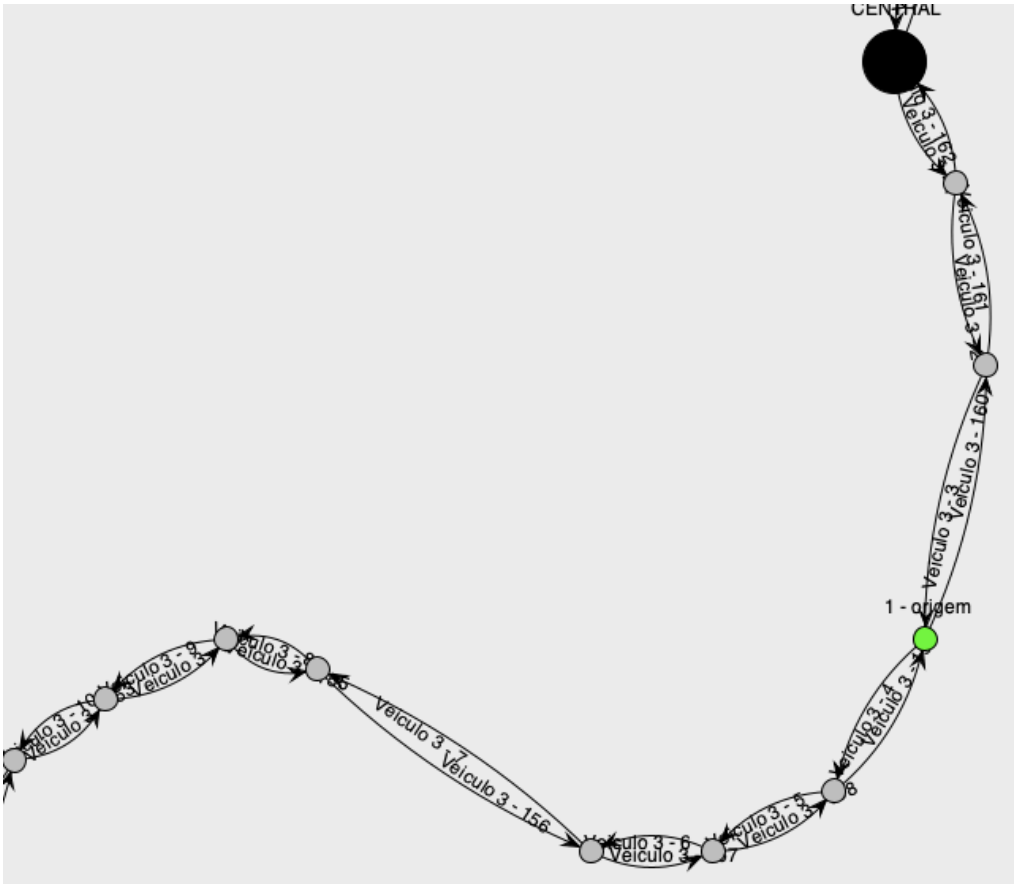


Fig: Close up do grafo apresentado acima. Cada origem e destino está devidamente assinalada, bem como a central. É indicado, para cada aresta, o veículo que a percorre, e o número da mesma na sequência de arestas que constitui o caminho do veículo.



Apresentação dos algoritmos implementados

Pesquisa em Profundidade (DFS Visit)

O primeiro algoritmo que é utilizado no nosso programa é uma pesquisa em profundidade a partir da central dos veículos, de modo a tomar conhecimento de todos os vértices acessíveis a partir da mesma. É um passo muito importante no decorrer do programa uma vez que a maioria do resto das operações e algoritmos aplicados incide apenas neste sub-conjunto de vértices.

Fig: Pseudocódigo para a pesquisa em profundidade.

```
DFSVisit(v, R):                  // - v is a vertex (1st one is the central)
                                 // - R is the return vector, and
    visited(v) ⟵ true             // it is passed by reference
    insert(R, v)
    for each e ∈ adj(v) do
        w ⟵ dest(e)
        if not visited(w) do
            DFSVisit(w, R)
```

Cálculo da Tabela: Dijkstra para todos os vértices

Devido ao tema deste trabalho estar ligado com ruas, estradas e redes viárias, optamos por realizar um pré-processamento dos vértices que são de interesse (ou seja, os acessíveis a partir da central) e guardar os resultados numa tabela.

O primeiro algoritmo que implementamos para o pré-processamento da tabela foi o algoritmo de Dijkstra, uma vez que os grafos que nos foram disponibilizados (bem como a grande maioria dos grafos que representam redes viárias) são esparsos, ou seja, o número de arestas é aproximadamente igual ao número de vértices do grafo.

A ideia aqui é, depois de identificar os vértices acessíveis a partir da central, executar o algoritmo de Dijkstra começando em cada um deles, de modo a saber, para cada par de vértices, a distância entre eles e qual o último vértice a passar no caminho mais curto.

Fig: Pseudocódigo para o algoritmo de Dijkstra utilizado para preenchimento da tabela de pré-processamento. O algoritmo é aplicado uma vez para cada nó acessível.

```
DijkstraInsertIntoTable(T, s):                          // - G=(V,E), s ∈ V
                                                        // - T is a table where
    for each v ∈ V do                                   // the values of distance
        dist(v) ⟵ INF                                    // and path are stored
        path(v) ⟵ NULL                                   // - s is the starting point
    dist(s) ⟵ 0                                         // for the algorithm
    Q ⟵ ∅
    insert(Q, s)

    while Q ≠ ∅ do
        v ⟵ extractMin(Q)

        if v ≠ s do
            insert( T, ( (s, v), (dist(v), path(v)) ) )     // inserts the values of distance
                                                            // and path into the table, using
        for each e ∈ adj(v) do                              // the vertices as a key
            oldDist ⟵ dist(dest(e))

            if dist(v) + weight(e) < dist(w) do
                dist(w) ⟵ dist(v) + weight(e)
                path(w) ⟵ v

                if oldDist == INF
                    insert(Q, dest(e))
```

Cálculo da tabela: Floyd-Warshall

Numa fase mais posterior do projeto, consideramos a possibilidade de acrescentar um outro algoritmo para fazer o preenchimento da tabela. Acabamos por escolher e implementar o algoritmo de Floyd-Warshall, devido à sua eficiência para grafos densos, mas também devido à simplicidade do seu código.

Como já foi referido anteriormente, o utilizador tem a liberdade de escolher quer o Dijkstra quer o Floyd-Warshall para preencher a tabela de pré-processamento, tendo em conta o grafo que este carregou para o programa.

Fig: Pseudocódigo para o algoritmo de Floyd-Warshall utilizado para preenchimento da tabela de pré-processamento.

```
FloydWarshallInsertIntoTable(A, T):     // - A is a vector containing the nodes
                                        // accessible from the central
    for each iVert ∈ A do               // - T is the table where the values
        for each jVert ∈ A do           // will be stored (see Dijkstra)

            int value
            if iVert == jVert do
                value ⟵ 0
            else do
                value ⟵ INF

            insert(T, ( (iVert, jVert), (value, NULL) ) )

        for each e ∈ adj(iVert) do
            update T[iVert, dest(e)] to (weight(e), iVert)

    for each kVert ∈ A do
        for each jVert ∈ A do

            if kVert ≠ jVert do
                for each iVert ∈ A do
                    if dist( T[iVert, kVert] ) == INF
                    OR dist( T[kVert, jVert] ) == INF do
                        continue

                    val ⟵ dist( T[iVert, kVert] ) + dist( T[kVert, jVert] )

                    if val < dist( T[iVert, jVert] ) do
                        update T[iVert, jVert] to ( val, path( T[kVert, jVert] ) )
```

Algoritmo de Deteção de Pontos de Articulação

De modo a avaliar de uma melhor maneira a conectividade do grafo, foi implementado um algoritmo de deteção de pontos de articulação, de modo a ser possibilitada uma mais fácil identificação dos pontos de entrega e recolha com pouca acessibilidade.

A primeira aplicação tem como vértice inicial o vértice que representa a central dos veículos; este algoritmo apenas irá afetar os nós acessíveis a partir da central. A nossa implementação utiliza uma técnica semelhante a uma pesquisa em profundidade, com o objetivo de identificar os pontos de articulação.

Fig: Pseudocódigo para o algoritmo implementado para a deteção de pontos de articulação.

```
calcArticulationPoints(v, R):   // - v is a vertex (1st one is the central)
                                // - R is the return vector, and
    visited(v) ⟵ true           // it is passed by reference
    low(v) ⟵ counter
    num(v) ⟵ counter             // before the start, counter is initialized
    counter(v) ⟵ counter + 1     // with 1

    for each e ∈ adj(v) do

        w ⟵ dest(e)

        if not visited(w) then
            parent(w) ⟵ v
            calcArticulationPoints(w, r)
            low(v) ⟵ min( low(v), low(w) )

            if (low(w) >= num(v))
                insert(r, v)

        else if parent(v) ≠ w
            low(v) ⟵ min( low(v), num(w) )
```

Algoritmos e Heurísticas utilizadas para cálculo das rotas ótimas para os veículos

O maior desafio e objetivo principal do nosso tema é o cálculo de rotas ótimas para os vários veículos da agência, de modo a completar uma série de entregas definidas pelo utilizador.

Para isto (como foi explicado na 1ª entrega), optamos pela divisão do problema em partes:

1) Definição dos pontos de interesse a que cada veículo teria de passar, atribuindo-lhes entregas tendo em conta o acréscimo de distância que seria provocado no seu percurso. No fim desta parte, para cada veículo que tem entregas deverá ter um percurso do género:

central —> origem1 —> origem2 —> destino2 —> origem3—> destino3 —> destino1 —> central

2) Depois de definido o percurso para todos os veículos, proceder à determinação do caminho exato que eles vão ter de percorrer, ou seja, calcular e identificar os vértices exatos pelos quais um determinado veículo terá de passar, de modo a este chegar ao ponto B a partir do ponto A, na menor distância possível. Isto é feito para a sequência toda de vértices, dois a dois.

Relativamente à **primeira parte do processo**, inicialmente temos um vetor contendo todas as entregas que têm de ser feitas. Para cada entrega, verifica-se qual é o melhor veículo para a fazer, e quais as melhores posições para inserir a origem e destino da entrega no percurso que esse veículo já tem.

Para saber quanto é o acréscimo de distância que uma determinada entrega provocaria no caminho de um veículo, faz-se da seguinte maneira:

- Primeiro, verifica-se se o veículo já tem entregas. Se não tiver, o seu percurso será Central —> Central, logo só há uma posição válida para a entrega: o percurso do veículo ficaria Central —> Origem —> Destino —> Central. É calculada a distância que o trajeto implicaria (acedendo à tabela), que será o acréscimo que essa entrega provocará no percurso do veículo (cuja distância total é 0, pois ele ainda não tinha entregas para fazer). As posições da origem e do destino na sequência é fixa, uma vez que a origem tem de vir antes do destino.

- No caso de o veículo já ter entregas, faz-se o seguinte:

  - Percorre-se todas as posições da sequência de pontos de interesse do veículo. Para cada posição, calcula-se o acréscimo de distância que seria provocado por passar na origem do trajeto entre os pontos anterior e posterior a essa posição. Chamemos a esse acréscimo **curDeltaOrigin**.

  - Para todas as posições a seguir à posição da origem (uma vez que temos de passar na origem antes de passar no destino), faz-se a mesma coisa para o destino, calculando o acréscimo de distância que ocorreria se passássemos lá, para cada posição. Chamemos a esse acréscimo **curDeltaDestination**.

  - Seja **curDelta** a soma **curDeltaOrigin + curDeltaDestination**, que se refere a uma posição específica para a origem, e outra para a central. Se **curDelta** é menor que o acréscimo mínimo ate ao momento, **minDelta**, então é porque foi descoberta uma maneira nova de ordenar o caminho do veículo de modo a que a sua distância seja mínima: o valor desta última variável é atualizada, bem como os índices de inserção na sequência para a origem e o destino.

Assim, para cada entrega, sabe-se o acréscimo de distância mínimo que ela provoca em cada veículo, bem como as posições que a origem e o destino da mesma devem ter na sequência de pontos de interesse, para conseguir esse acréscimo mínimo.

Sabendo isto, para cada entrega, calcula-se o acréscimo mínimo para todos os veículos cujo tipo é igual ao tipo da entrega: a entrega é atribuída, obviamente, ao veículo cujo acréscimo é menor.

Este processo é feito para todas as entregas, atribuindo cada uma ao veículo que compensar mais.

Em relação à **segunda parte do processo**, para gerar a sequência completa de vértices pelo qual cada veículo terá de passar, faz-se o seguinte:

- Para cada par consecutivo de vértices da sequência do veículo, calcula-se (acedendo à tabela), os vértices exatos pelos quais o veículo terá de passar para este ir, na menor distância possível, de um ponto a outro.

- "Juntando" todas as arestas resultantes deste processo, descobre-se então a rota ótima para o veículo!

Figs: Pseudocódigo para a heurística implementada para a primeira parte da determinação das rotas ótimas para cada veículo.

```
TestInsertDelivery(v, d, T, pOrig, pDest):          // - v is the vehicle for which the distance will be calculated
                                                    // - d is the delivery to be tested
    orig ⟵ orig(d)                                   // - T is the table with the values  used to compute the output
    dest ⟵ dest(d)                                   // - pOrig and pDest are the indexes of the path where the
    if empty( deliveries(v) ) do                    // delivery could be inserted to best fulfill the goal of the
        pOrig ⟵ 1                                    // program (passed by reference)
        pDest ⟵ 2
        central ⟵ path(v)[0]
        deltaOrig ⟵ getDist(central, orig, T) + getDist(orig, central, T)
        deltaDest ⟵ getDist(orig, dest, T) + getDist(dest, central, T) - getDist(orig, central, T)

        // calculates the "delta" of the total path distance by inserting the delivery into the
        // vehicle's path, in search of a minimum value (this is the particular case in which
        // the vehicle has no other deliveries, making it so there is only one possibility for
        // how to insert the origin and destination into its path)

        return deltaOrig + deltaDest


    minDistance ⟵ INF

    for each i from 1 to size( path(v) ) do
        prev ⟵ path(v)[i-1]
        next ⟵ path(v)[i]
        curDeltaOrig ⟵ getDist(prev, orig, T) + getDist(orig, next, T) - getDist(prev, next, T)

        for each j from i to size( path(v) ) do
            prev ⟵ path(v)[j-1]
            next ⟵ path(v)[j]
            if i == j do
                curDeltaDest ⟵ getDist(orig, dest, T) + getDist(dest, next, T) - getDist(orig, next, T)
            else do
                curDeltaDest ⟵ getDist(prev, dest, T) + getDist(dest, next, T) - getDist(prev, next, T)

            if curDeltaOrig + curDeltaDest < minDistance do
                minDistance = curDeltaOrig + curDeltaDest
                pOrig = i
                pDest = j + 1

    return minDistance
```

```
AssignDeliveryToVehicle(V, d, T):       // - V is a vector, containing all the
                                        // vehicles (passed by reference)
    bestPosOrig ⟵ -1                    // - d is the delivery to be assigned
    bestPosDest ⟵ -1                    // - T is the table with the values
    bestDelta ⟵ INF                     // used to perform this task
    bestVehicle ⟵ NULL

    for each v ∈ V do

        if type(v) == type(d) do
            int pOrig, pDest
            curDelta ⟵ TestInsertDelivery(v, d, T, pOrig, pDest)

            if bestDelta - curDelta > RESIDUAL do    // RESIDUAL is a macro that was defined
                bestPosOrig ⟵ pOrig                  // in order to avoid residual errors when
                bestPosDest ⟵ pDest                  // comparing bestDelta and curDelta
                bestDelta ⟵ curDelta
                bestVehicle ⟵ v


    if bestVehicle == NULL do
        return false                    // no suitable vehicle was found

    insert( path(bestVehicle), (orig(d), bestPosOrig) )
    insert( path(bestVehicle), (dest(d), bestPosDest) )
    insert( deliveries(bestVehicle), d )

    for each i from 1 to size(V)        // makes it so that, if the result
        if V[i] == bestVehicle          // in terms of total distance would
            swap( V[i], V[i+1] )        // be the same between an already used
                                        // vehicle and an empty one, priority
                                        // is given to new, emptier vehicles,
                                        // for the sake of time saving
    return true
```

```
AssignDeliveries(V, D, T):              // - V is a vector, containing all the
                                        // vehicles (passed by reference)
                                        // - D is the vector of all deliveries
                                        // - T is the table with the values

    for each d ∈ D do
        if AssignDeliveryToVehicle(V, d, T)   // if it returns true, it means the delivery
            remove d from D                   // was assigned to a vehicle successfully;
                                              // we can remove the delivery from the vector
```

Fig: Pseudocódigo para a heurística implementada para a segunda parte da determinação das rotas ótimas para cada veículo.

```
displayVehiclePaths(G, V, T, width, height)::              // - V is a vector, containing all the
                                                           // vehicles (passed by reference)
    GV ⟵ initGraphViewer()                                 // - G is the graph
    idAresta ⟵ 1                                           // - T is the table with the values
    numArestasAnterior ⟵ 1                                 // - width is the GraphViewer window width
                                                           // - height is the GraphViewer window height


    for each v ∈ V do
        if hasDeliveries(v)
            path ⟵ getPath(v)

            for each i from path.size - 1 to 1

                v ⟵ path.at(i)          // gets consecutive vertexes from the path
                s ⟵ path.at(i - 1)

                while s != v
                    t ⟵ getPath(s, v, T)   // t is the last vertex in the shortest path from s to v

                    displayNode(GV, t)       // display in GraphViewer
                    displayNode(GV, v)
                    displayEdgeBetween(GV, t, s)
                    v = t
                    idAresta++


            // from here on now, its just displaying the results in GraphViewer

            for each i from 1 to idAresta - numArestasAnterior
                setEdgeLabel(GV, idAresta - i, 'Vehicle' + getID(v) + ' - ' + i)

            numArestasAnterior ⟵ idAresta

            for each d ∈ getDeliveries(d)
                setVertexLabel(GV, getOrigin(d), getID(getOrigin(d)) + ' - origin')
                setVertexLabel(GV, getDest(d), getID(getDest(d)) + ' - dest')


    rearrange(GV)
```

Análise da complexidade temporal dos algoritmos utilizados

Complexidade Teórica dos Algoritmos

Nota: |V| e |E| referem-se não ao número de nós e arestas do grafo total, mas ao número de nós e arestas do sub-grafo constituído pelos vértices acessíveis a partir da central de veículos.

Pesquisa em profundidade (DFS Visit)

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

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

Dijkstra

Uma vez que a nossa implementação utiliza uma fila de prioridades mutáveis, as operações de extração e inserção de elementos nessa fila irão impactar a complexidade deste algoritmo.

Sendo que o número total de extrações e inserções é |V|, e sendo que cada operação será executada em tempo logarítmico no tamanho da fila, que assume um valor máximo de |V|, então este aspeto do algoritmo terá uma complexidade de O(|V| * log|V|).

Também devido ao uso da fila de prioridades mutáveis, será necessária a execução de um procedimento para atualizar as posições dos elementos da fila, com base nos dados processados (uma vez que o vértice de distância mínima não é necessariamente o mais antigo, como aconteceria no caso de um grafo não pesado).

Esta operação será feita no máximo |E| vezes, sendo que neste caso seria executada para o destino de cada uma das arestas. Cada operação será executada em tempo

logarítmico no tamanho da fila, que assume um valor máximo de |V|, pelo que este aspeto do algoritmo terá uma complexidade de O(|E| * log|V|).

Dado que o algoritmo calcula os dados a partir de um vértice para todos os outros do grafo, cada vértice será processado uma vez, sendo que para cada um deles, as suas arestas serão também processadas apenas uma vez. Esta vertente do algoritmo terá uma complexidade de O(|V| + |E|).

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

No entanto, o nosso caso de uso do algoritmo tem algumas diferenças em relação ao algoritmo “standard”:

1) O algoritmo não é aplicado ao grafo completo, mas apenas ao sub-grafo que é composto pelos vértices que são acessíveis a partir da Central. Isto traduz-se num número bastante reduzido de vértices e arestas totais para processar.

2) O algoritmo é executado para cada um dos vértices do sub-grafo referido, funcionando do mesmo modo que o algoritmo de Floyd-Warshall, pelo que a complexidade previamente calculada deve ser multiplicada por um fator igual ao número de vértices do sub-grafo a ser utilizado.

Complexidade para a construção da tabela: O(((|V| + |E|) * log|V|) * |V|)

Floyd-Warshall

Dado que o algoritmo consiste em um ciclo, durante o qual percorre todos os vértices do grafo, sendo que para cada um executa um segundo ciclo, que faz o mesmo, chamando para cada um dos seus vértices um terceiro ciclo, que efetua o processamento propriamente dito para os vértices respetivos do primeiro, segundo e terceiro ciclos, então terá uma complexidade de O(|V| * |V| * |V|).

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

Pontos de articulação

O algoritmo tem início num vértice, a partir do qual será efetuada uma pesquisa em profundidade, na qual os vértices serão numerados no seu processamento. Apesar de o processamento ser diferente, o modo de percorrer o grafo neste algoritmo é semelhante ao da pesquisa em profundidade, tendo este algoritmo a mesma complexidade, pelas razões supramencionadas.

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

Algoritmos e Heurísticas utilizadas para cálculo das rotas ótimas para os veículos

Relativamente à **primeira parte do processo**, que foi explicada na secção anterior do relatório, podemos dividi-la em 3 partes:

1) Cálculo do acréscimo e das posições de inserção da origem e do destino, para um determinado veículo e entrega;
2) Determinação de qual é o veículo que produz o menor acréscimo, para uma dada entrega;

3) Atribuição de todas as entregas aos veículos.

Para as explicações dadas a partir daqui, não esquecer que como se utiliza unordered_map para a tabela, as operações de pesquisa e acesso são feitas em tempo constante.

Foquemo-nos primeiramente no **tópico 1)**, mais concretamente na função testInsertDelivery(), da classe Vehicle. No melhor dos casos, o veículo em questão não tem nenhuma entrega, logo a origem e o destino apenas têm uma posição válida na sequência. A complexidade temporal no melhor caso é, então, constante.

Se o veículo já tem entregas, então é porque o seu vetor de pontos de interesse por onde o veículo tem de passar já não contém só a central, mas as origens e destinos

dessas entregas. O algoritmo percorre uma vez esse vetor, para calcular a melhor posição para a origem, e para cada uma dessas vezes percorre as posições seguintes do vetor, para calcular a melhor posição para o destino. Se considerarmos |P| como o tamanho do vetor dos pontos de interesse, a complexidade é:

Complexidade: O(1), no melhor caso;
O(|P| * |P|), no caso médio / pior caso.

Em relação ao **tópico 2)**, observemos a função assignDeliveryToVehicle(). Esta calcula qual o melhor veículo para uma dada entrega, chamando a função anteriormente descrita para todos os veículos disponíveis, e comparando os valores gerados. Seja |C| o número de carros disponíveis:

Complexidade: O(|C| * |P| * |P|)

Finalmente, relativamente ao **tópico 3)** (função assignDeliveries()), todas as entregas são atribuídas a um veículo (quando houver pelo menos um veículo apto para as fazer). Tal é feito, executando a função anteriormente descrita para todas as entregas. Seja |D| o número total de entregas:

Complexidade: O(|D| * |C| * |P| * |P|)

Relativamente à **segunda parte do processo**, gera-se a sequência completa de vértices pelo qual cada veículo terá de passar.

Para fazer isto, faz-se: para cada veículo (se este tiver entregas associadas), percorre-se o seu vetor de pontos de interesse e executa-se um ciclo para encontrar o caminho mais curto entre dois pontos consecutivos do vetor (relembramos que os acessos à tabela são feitos em tempo constante).

Para esse ciclo interior, o processo é o seguinte: tendo os vértices S e V, do vetor do ponto de interesses do veículo, para saber o caminho mais curto de S até V, acede-se à tabela para saber o último vértice no caminho entre eles os dois: chamemos a esse vértice T, por exemplo. No fim da iteração, o vértices V é substituído por T, e na próxima iteração, acede-se à tabela para saber o último vértice no caminho entre S e V, atualiza-se V de novo… isto, até S coincidir com V, o que indica que já calculamos o caminho na sua totalidade.

No melhor dos casos, se S e V forem adjacentes, apenas uma iteração do ciclo irá ser executada, pelo que a complexidade irá ser constante.

No pior dos casos, S e V estão em pontas opostas do grafo, sendo que para calcular o caminho entre eles, é necessário processar o grafo inteiro, pelo que a complexidade seria linear relativamente ao número de vértices do grafo (|V|).

No caso médio, pode considerar-se que é preciso percorrer |V| / 2 vértices.

Assim, podemos concluir que, para esta segunda parte do processo:

Complexidade: O(|C| * |P|), no melhor caso;
O(|C| * |P| * |V|), no pior caso e no caso médio.

Complexidade Empírica dos Algoritmos

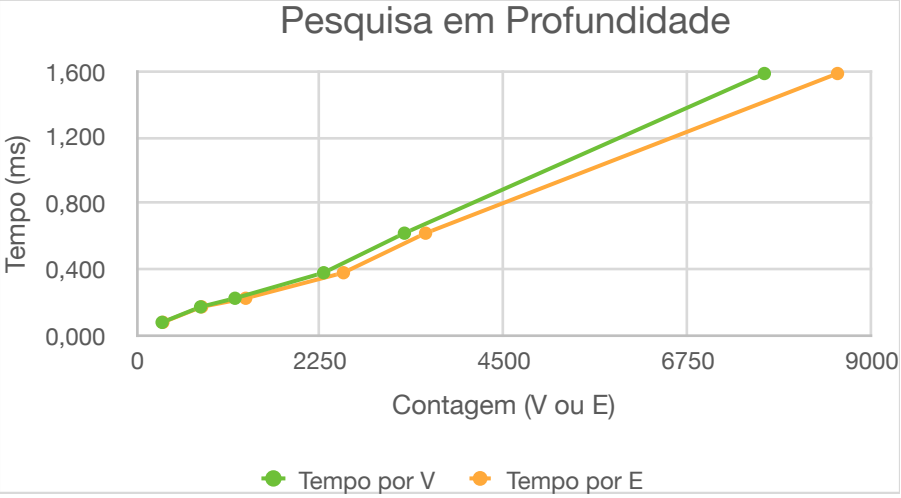
O número de vértices e arestas apresentados para cada um dos casos não se referem ao grafo total da cidade, mas sim ao sub-grafo dos vértices acessíveis a partir da central, o que obviamente depende grandemente do vértice onde a central for definida.

Para além disso, o número de arestas que é apresentado em cada um dos casos interpreta as arestas como sendo bidirecionais; o que acontece na realidade, no nosso programa, é que cada aresta “bidirecional” é representada como um par de arestas direcionadas em sentidos opostos.

NOTA: os tempos medidos podem não ser totalmente exatos, dado que a sua extração foi efetuada numa máquina virtual, devido a conflitos entre o sistema operativo Windows e a biblioteca chrono de C++.

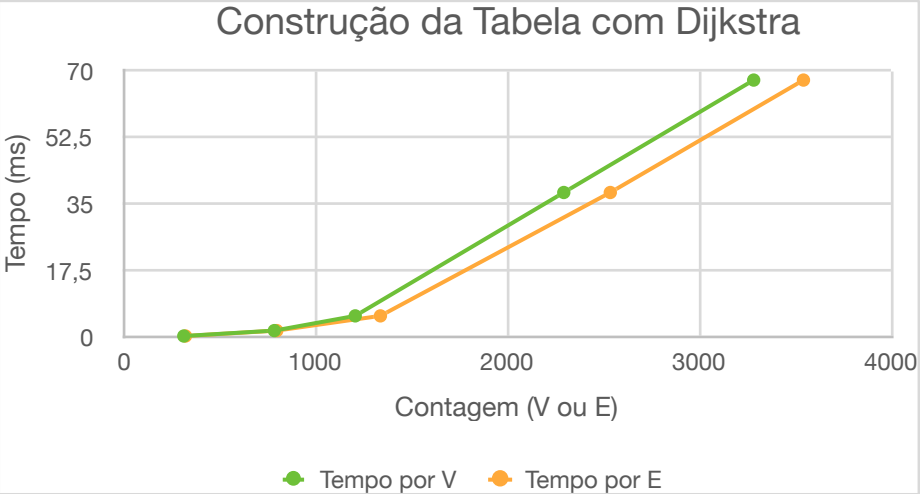
Pesquisa em profundidade (DFS Visit)

V	E	Tempo (ms)	Cidade
317	326	0,086	Aveiro
788	801	0,179	Viseu
1209	1340	0,231	Fafe
2294	2536	0,385	Ermesinde
3282	3541	0,622	Gondomar
7687	8582	1,582	Porto



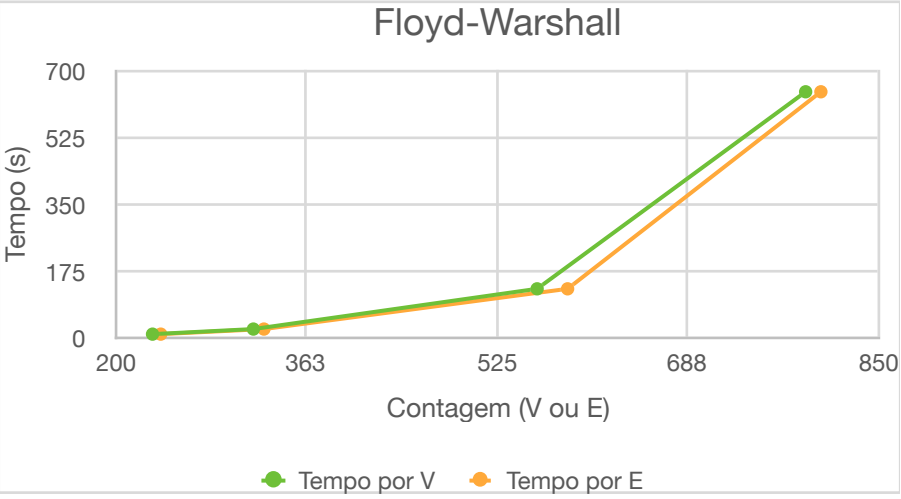
Dijkstra

V	E	Tempo (ms)	Cidade
317	326	146	Aveiro
788	801	1565,9	Viseu
1209	1340	5439,1	Fafe
2294	2536	37882,3	Ermesinde
3282	3541	67441,6	Gondomar



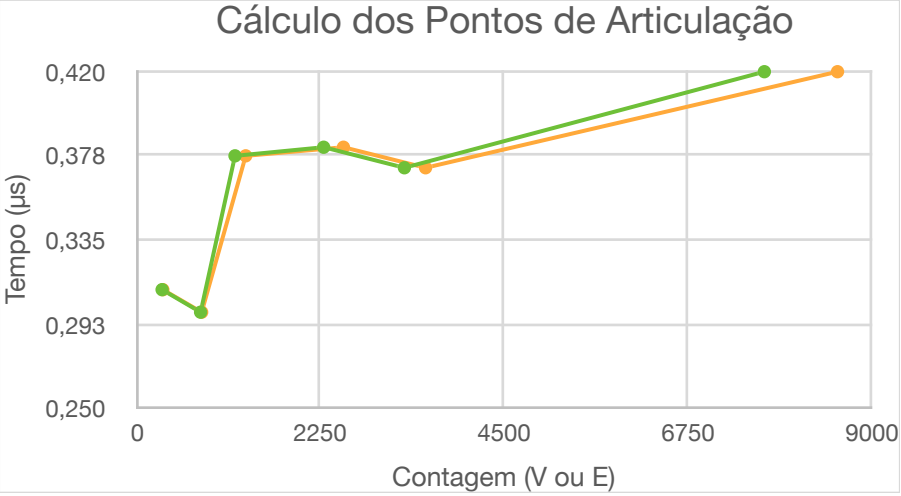
Floyd-Warshall

V	E	Tempo (s)	Cidade
231	238	7,951	Viseu
317	326	21,294	Aveiro
559	585	127,103	Coimbra
788	801	645,235	Viseu



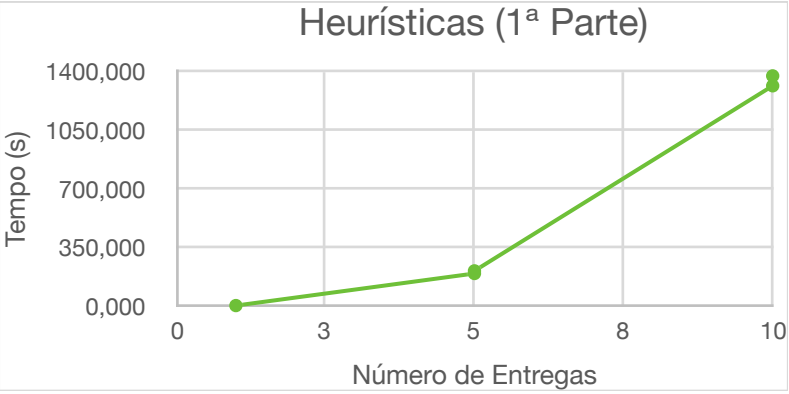
Pontos de articulação

V	E	Tempo (µs)	Cidade
317	326	0,310	Aveiro
788	801	0,298	Viseu
1209	1340	0,377	Fafe
2294	2536	0,381	Ermesinde
3282	3541	0,371	Gondomar
7687	8582	0,419	Porto

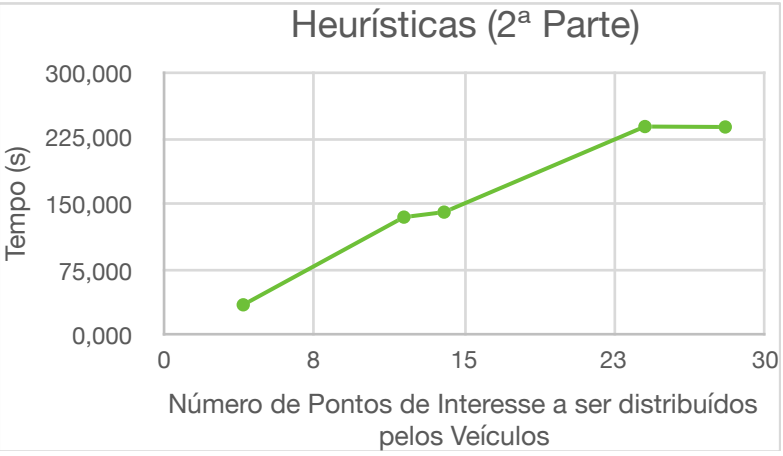


Algoritmos e Heurísticas utilizadas para cálculo das rotas ótimas para os veículos

Heurísticas (1ª parte)	Num Veículos	Num Entregas	Tempo (s)
Cidade:	1	1	4,461
Fafe	1	5	195,856
Num Vértices Acessíveis:	2	5	211,581
1209	2	10	1315,910
	4	10	1375,070



Heurísticas (2ª parte)	Num Veículos	Média Pts Interesse p/ Veículo	Pts Interesse	Tempo (s)
Cidade:	1	4	4	34,087
Fafe	1	12	12	134,566
Num Vértices Acessíveis:	2	7	14	140,299
1209	2	12	24	238,376
	4	7	28	237,865



Conclusão geral do trabalho

Apesar de nos depararmos com algumas dificuldades ao longo do projeto, somos da opinião que conseguimos arranjar uma boa solução para o problema das rotas ótimas que nos foi proposto. Para além da aplicação de certos algoritmos dados nas aulas, como o Dijkstra, F.W, deteção de pontos de articulação, etc., tivemos que pensar e criar heurísticas que nos conduzissem à solução ótima, sem gastar muito tempo e memória.

Também foi testada a nossa capacidade de adaptação a certos fatores, uma vez que (como seria de esperar) nem tudo o que planeamos na 1ª entrega para a implementação acabou por ser feito, ou porque outro algoritmo compensava mais, ou porque essa ideia foi posta de lado, etc.

Em suma, somos da opinião que os objetivos pretendidos com este projeto de grupo foram atingidos, quer a nível individual quer a nível coletivo.

Cotações

Diogo Machado (up201706832):

- Estruturação dos menus do programa, e interface com o utilizador;
- Adaptação e implementação das estratégias e heurísticas definidas para o cálculo de rotas ótimas dos veículos;
- Implementação do algoritmo de Floyd-Warshall para o preenchimento da tabela;
- Configuração do display dos resultados calculados no GraphViewer;
- Redação do relatório;
- Etc.

Cotação: 30%

Eduardo Ribeiro (up201705421):

- Estruturação dos menus do programa, e interface com o utilizador;
- Adaptação e implementação das estratégias definidas para o cálculo de rotas ótimas dos veículos;
- Implementação do algoritmo de cálculo dos pontos de articulação;
- Implementação do algoritmo de Dijkstra para o preenchimento da tabela;
- Implementação da identificação dos pontos do grafo acessíveis a partir da central;
- Redação do relatório;
- Etc.

Cotação: 40%

Eduardo Macedo (up201703658):

- Definição da estrutura e leitura dos ficheiros de texto utilizados para a importação de dados para a aplicação;
- Configuração do display dos resultados calculados no GraphViewer;
- Deteção e teste das funcionalidades do programa;
- Etc.

Cotação: 30%

Notas e observações

O programa, para além de fazer a leitura de ficheiros dados pelos monitores relativos ao grafo, lê também ficheiros em que estão especificados os veículos e as entregas; estes têm de ser definidos pelo utilizador antes de correr o programa ou antes de selecionar a opção de leitura dos mesmos.

Nos ficheiros enviados na submissão, apenas estão presentes dados relativos a Aveiro. Para testar em outros grafos, é necessário fazer novos ficheiros relativos à nova cidade/país.

Juntamente com os grafos disponibilizados, encontra-se uma pasta com um grafo de Teste, significativamente mais pequeno e simples, que foi utilizado à medida que o trabalho foi desenvolvido, para testar o correto funcionamento das funcionalidades do mesmo.

Os ficheiros de informação relativa à latitude e longitude não foram utilizados no nosso programa, uma vez que não achamos necessário.

Como bibliografia utilizada apenas foram utilizados os slides das aulas teóricas.