Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

Smart Delivery

Tema 4: Empresas de Distribuição de Mercadorias - Parte 1

Professor Regente: Rosaldo José Fernandes Rossetti

Professor das aulas Teórico-Práticas: Francisco Xavier Richardson Rebello de Andrade

Grupo B - Turma 6:

Gaspar Santos Pinheiro | up201704700@fe.up.pt

Manuel Monge dos Santos Pereira Coutinho | up201704211@fe.up.pt

Tito Alexandre Trindade Griné | up201706732@fe.up.pt

Índice - 1º Parte

Índice - 1º Parte

Descrição do Tema

Uma empresa de distribuição de mercadorias necessita, diariamente, de recolher itens de um **depósito** (fábrica ou armazém) e levá-los aos respetivos **destinos** (habitações ou armazéns) através de uma frota de camiões. No final das entregas, os camiões ter-se-ão de deslocar de novo para o depósito ou para uma garagem. Cada item do depósito terá associado o destino, o volume/peso correspondente e outras informações relevantes.

Torna-se imperativo encontrar o melhor trajeto para cada camião por forma a minimizar o percurso realizado, tornando a entrega o mais célere possível e reduzindo os custos de transporte o mais possível, tomando em atenção fatores como a capacidade do camião (volume e/ou peso máximo que suporta).

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

1. **Um camião de capacidade ilimitada entrega uma mercadoria**

Nesta abordagem, considera-se que a empresa tem disponível apenas um camião que terá de realizar uma entrega. Nesta situação, o problema reduz-se a encontrar o trajeto mais curto, com início no depósito, que passe pelo ponto de entrega, e termine na garagem, que poderá coincidir com o depósito.

2. **Um camião de capacidade ilimitada entrega todas as mercadorias**

Semelhante ao caso anterior a empresa continua a deter apenas um camião, capaz de transportar qualquer quantidade de itens, mas agora existem várias mercadorias a serem entregues. Neste caso, o problema passa por encontrar o trajeto mais curto com início no depósito que passe por todos os pontos de entrega e termine na garagem (mais uma vez, pode coincidir com o depósito).

3. **Frota de camiões de capacidade limitada entregam todas as mercadorias**

Por último assume-se o problema na íntegra, ou seja, a empresa tem ao seu dispor vários camiões, cada um com capacidade respetiva e limitada. É importante salientar que, ao ter disponível vários camiões, não implica que os tenha de usar todos, sendo do interesse da empresa precisamente o contrário, entenda-se, o objetivo será minimizar o percurso tomado por cada camião e utilizar o mínimo número de camiões. Em geral, o objetivo é diminuir os custos incorridos nas entregas. Isto tem de ser complementado com as condições já enunciadas nas implementações anteriores.

Poder-se-ia ter criado ainda uma variante antes desta onde apenas um camião limitado teria de fazer diversas viagens para satisfazer todas as entregas, mas consideramos que a diferença para esta última fase não era grande ao ponto de justificar a existência de mais uma etapa. A única

diferença no cálculo dos caminhos será que, no problema final, a capacidade do veículo pode variar enquanto na etapa extra seria constante, ou seja, o que os distingue é a existência de um valor constante ou de uma variável.

Adicionalmente, em qualquer das versões do problema, é necessário, antes de procurar definir qualquer percurso, a análise das características subjacentes aos dados, por exemplo, a conectividade do grafo tem de ser tida em conta. Caso um destino de uma mercadoria seja inacessível, essa mercadoria não deve fazer parte das possíveis escolhas de inventário dos camiões, e o utilizador deve ser notificado desta ocorrência. O inverso também não é desejável, isto é, se não existe um caminho de retorno ao vértice inicial num dos destinos. Estas situações serão analisadas num pré-processamento dos dados.

Formalização do Problema

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

1. Dados de Entrada

M - estrutura com as diferentes mercadorias, tendo cada uma:

- dest → pertencente a V, local de entrega da mercadoria;
- id → número identificativo da encomenda;
- vol → volume (peso ou espaço) de uma dada encomenda;

C - estrutura com os diferentes camiões que a empresa detém, cada um com o seguinte atributo:

- capacity → capacidade total do camião (no caso de ser ilimitado, o valor deve ser igual a ∞);

G(V, E) - grafo dirigido pesado composto por:

- **V** - os vértices que representam os vários pontos no mapa com os seguintes atributos:
  - initial → booleano que permite inferir se o vértice é um depósito e, por isso, o vértice inicial;
  - final → booleano que permite inferir o ponto final do trajeto, podendo este ser o mesmo que o inicial (ser um depósito);
  - vol → volume (peso ou espaço) acumulado de uma dada encomenda (0 se não existe nenhuma mercadoria para esse vértice);
  - adj → contido em E, representa o conjunto de arestas (estradas que saem de determinado vértice);
  - enc → contido em M, representa o conjunto de encomendas a entregar neste vértice;
- **E** - as arestas que representam as várias estradas (podendo ter apenas um ou ambos os sentidos), sendo constituídas por:
  - orig → pertencente a V, representa o ponto de partida da aresta;
  - dest → pertencente a V, representa o ponto de chegada de E;
  - w → custo de percorrer a aresta/estrada (quer seja a distância total, tempo, combustível, portagens, os algoritmos otimizarão para a propriedade correspondente);

2. Dados de Saída

Cf - estrutura com todos os camiões usados, semelhante à inicial em que cada um deve ter:

capacity → capacidade do camião (não relevante, mas é um dado que faz parte);

P → sequência ordenada de vértices a visitar pertencentes a V, cujo primeiro tem a flag initial verdadeiro e o último a final;

Mf → subconjunto de M que representa as mercadorias a entregar;

3. Restrições

Nos Dados de Entrada:

- $\forall m \in M, vol(m) > 0$, pois não há encomendas sem volume (ou com ele negativo);
- Assim sendo também $\forall c \in C, capacity(c) > 0$; $\forall v \in V, vol(v) \geq 0$(pode não ter encomendas);
- $\forall e \in E, w(e) > 0$(não há ruas com custo 0 ou negativo, pois todas têm um certo comprimento ou tempo para atravessar);
- $\forall v \in V\ adj(v) \subseteq E$, todas arestas adjacentes fazem parte do conjunto de arestas do grafo;
- $\forall v \in V\ enc(v) \subseteq M$, todas as encomendas que se encontram na fila para entregar num vértice têm de pertencer ao conjunto de mercadoria inicial M;
- $\cup_{v \in V} \quad enc(v) = M$, todas as encomendas são para algum ponto do mapa e, por isso, a união das entregas de todos os pontos tem de ser igual ao conjunto inicial destas;
- $\cap_{v \in V} \quad enc(v) = \{\}$, ou seja, não há uma encomenda para dois pontos diferentes do mapa;
- $\forall v \in V \sum_{m \in enc(v)} \quad vol(m) = vol(v)$, para garantir coerência nos dados, a soma das encomendas a serem entregues num vértice, tem de ser igual ao seu valor acumulado guardado;
- $\forall m \in M\ \forall c \in C, vol(m) < capacity(c)$, ou seja, não há nenhuma entrega com volume maior que a capacidade de um camião;
- $\forall v \in V, vol(v) = \sum_{m \in M \wedge dest(m) = v} \quad vol(m)$, ou seja, volume em cada vértice é igual à soma do volume das encomendas para esse mesmo vértice;
- Seja o conjunto $U = \{v \in V | \ initial(v) = true \vee final(v) = true \vee vol(v) > 0\}$, o conjunto de todos os pontos úteis (pontos onde se terá de deixar uma encomenda, depósito e garagem), é necessário que todos os seus elementos façam parte de um mesmo componente fortemente conexo do grafo. Quer isto dizer que $\forall u1, u2 \in U$, sendo $T(u1, u2)$a sequência ordenada de vértices do percurso que liga $u1$a $u2$ então $T(u1, u2) \neq \{\ \}$e $T(u2, u1) \neq \{\ \}$, isto é, existe sempre um caminho que ligue quaisquer dois pontos úteis um ao outro (justificação para uma restrição mais forte do que aparenta ser necessária

no subcapítulo sobre as Técnicas de Concepção);

- $\exists!\ v\ \in V$tal que initial(v) = true, pois tem de existir ponto inicial (depósito);
- $\exists!\ v\ \in V$tal que final(v) = true, pois tem de existir uma garagem (nem que seja o próprio depósito). Poderia haver mais do que um, mas segundo o enunciado apenas nos vamos focar nesta situação;
- $\forall m\ \in\ M, dest(m)\ \in V$ e da mesma forma: $\forall e\ \in\ E, orig(e)\ \in V\ \wedge dest(e)\ \in V$;
- $\forall e\ \in\ E, e\ \in adj(orig(e))$, ou seja, qualquer aresta pertence ao conjunto de adjacentes do seu vértice original;
- $\forall e\ \in\ E, e$ é utilizável pelo camião, isto é, se o grafo for carregado do OpenStreetMaps ou bases semelhantes, ruas que um camião seja incapaz de utilizar (ruelas por exemplo) não devem ser adicionadas;
- Do ponto inicial deve ser possível chegar ao final para que o percurso possa ser completado, logo se $v1, v2\ \in V, (initial(v1)\ =\ true\ \wedge\ final(v2)\ =\ true \wedge v1 \neq v2) \Rightarrow adj(v1)\ \neq\ \{\ \}$ (numa situação real o contrário também teria de ser verdade, pois os camiões terão de voltar eventualmente ao depósito, mas este caminho não é o nosso objeto de estudo);

Nos Dados de Saída:

- $\forall c\ \in\ Cf, capacity(c)\ \geq \sum_{m\in\ Mf(c)}\ \ vol(m)$, um camião não pode levar um volume maior que a sua capacidade;
- $|Cf|\ \leq\ |C|$, não se pode usar mais camiões do que aqueles disponíveis inicialmente;
- Se $|Cf|\ <\ |C| \Rightarrow \forall v \in V, vol(v)\ =\ 0$, se não foi necessário utilizar todos os camiões é porque todas as entregas foram feitas, uma vez que não há entregas de maior volume que um camião (assumindo que as entregas inacessíveis já foram descartadas);
- $\forall c\ \in\ Cf\ \forall m\ \in\ Mf(c), \sum_{m\in Mf(c)}\ \ vol(m)\ = \sum_{v\in V}\ \ vol(v)\ *\ u$, sendo u = 0 se quando o camião passou em v o seu volume era 0 (quer porque foi retirado por outro camião ou pelo próprio quer porque não havia inicialmente), u = 1 caso contrário. Se u = 1, $v\ \in P$;
- $P \subseteq V$, todos os pontos de P têm de ser vértices do gráfico;
- Seja p0 o primeiro elemento de P, $initial(p0)\ =\ true$, pois todos os camiões partem do depósito;
- Seja pf o último elemento de P, $final(pf)\ =\ true$, pois todos os camiões terminam na garagem (que pode coincidir com o depósito);
- $\forall i, j, (p(i) \in P \wedge p(j) \in P \wedge i\ +1\ =\ j)\ \Rightarrow \exists\ e\ \in adj(p(i)), dest(e)\ =\ p(j)$, ou seja, quaisquer dois vértices consecutivos na lista de P são adjacentes (têm ligação entre eles);

4. Funções Objetivo

A resolução dos vários problemas passa pela minimização de essencialmente duas funções:

1) $f = |Cf|$, minimizar o número de camiões usados;
2) $g = \sum_{c \in Cf} \quad \sum_{p \in P(c)} \quad w(e)$, sendo e a aresta que liga dois pontos consecutivos do conjunto P e, por isso, $e \in adj(p)$, minimizar a soma do custo dos caminhos;

Quando apenas temos um camião, a função 1) deixa de fazer sentido sendo a minimização do peso total das arestas a nossa prioridade, nos casos em que existe mais do que um camião, a função 1) passa a ser a prioridade, pois assumimos que o custo de mais um camião na estrada é superior ao custo de um caminho ligeiramente maior.

Perspetiva de Solução

Técnicas de Conceção

Preparação dos Dados

Para uma mais fácil resolução do problema, transformar-se-á o processo de distribuição num processo de levantamento, ou seja, cada ponto terá um volume que deseja ver levantado, assim o valor atribuído a cada vértice corresponde às encomendas que lhe têm de ser entregues e evita-se um falso problema da mochila em que se tentaria maximizar a quantidade de encomendas com que carregar o camião inicialmente em vez de minimizar o custo da viagem também. Consequentemente os vértices a percorrer serão aqueles cujo seu volume associado seja maior que 0 (sinal de que há mercadoria a ser distribuída).

Resolve-se ainda a questão de encomendas para um mesmo sítio, pois já estarão todas acumuladas e concentradas neste ponto evitando que vários camiões necessitem de passar pelos mesmo pontos. Este pressuposto heurístico tem também as suas desvantagens: caso o nó se situe num sítio de muita passagem (perto do depósito por exemplo), entregas a esta casa poderiam ser feitas com espaços sobresselente nos diferentes camiões dado que a passagem é quase obrigatória, em vez de "gastar" um camião quase exclusivamente para esta tarefa, contudo vamos admitir que a empresa está inserida num grafo denso em que a probabilidade de isto acontecer é reduzida e, assim, compensa quase sempre a entrega dos itens na sua totalidade a uma casa (até do ponto de vista do cliente).

Pré-processamento - Conectividade do Grafo

Para a nossa análise da conectividade admitimos que o grafo dado é dirigido, uma vez que, ao ser uma representação dum sistema de estradas, as mesmas podem ter apenas um sentido e, consequentemente, para uma aresta que liga um vértice A a outro B, pode não ser válido fazer ambos os percursos, de A para B e de B para A.

Como mencionado nas restrições iniciais, podemos afirmar que todos os pontos "úteis" do grafo (depósito, pontos de entrega e garagem) terão de pertencer à mesma componente fortemente conexa. Uma análise inicial poder-nos-ia levar a pensar que apenas um caminho unidirecional com início no depósito e final na garagem, passando ou não (caso em que seriam todos descartados) pelos pontos de entrega. Contudo, facilmente nos apercebemos que, num contexto real, os camiões terão de regressar eventualmente ao depósito (quer seja a garagem um ponto diferente deste ou o mesmo), logo, apesar de o estudo dele não ser do nosso interesse, terá de existir também um caminho de regresso, fazendo com que a zona onde se encontram todos este pontos tenha de ser uma componente fortemente conexa.

Os algoritmos que verificam esta propriedade em grafos tomam um vértice aleatório como ponto de partida para este teste. No nosso caso, devido à preponderância do depósito - sendo daqui que partem todas as encomendas e um dos únicos sítios que a sua existência é requerida - utilizá-lo-emos como ponto de partida para o estudo da conectividade.

A informação obtida pela análise da conectividade tem de fornecer a seguinte informação:

- Quais os vértices, caso existam, que, partindo do vértice inicial (depósito), não são alcançáveis, ou seja, não existe nenhum percurso que ligue o vértice inicial a esses vértices, direta ou indiretamente;
- Se existem vértices que, mesmo alcançáveis a partir do vértice inicial, o contrário já não aconteça, não se tratando, por isso, de vértices pertencentes à mesma componente fortemente conexa;

Uma vez recolhidos, estes vértices serão removidos do grafo uma vez que não obedecem às restrições dos dados de entrada. O output deste pré-processamento deve ser um grafo fortemente conexo que inclua o vértice inicial.

Existem várias alternativas para realizar este processamento, das quais duas nos parecem mais pertinentes:

1. Uma forma possível seria realizar uma **Depth-First-Search** a todos os vértices do grafo e verificar se em todos a DFS passava pela totalidade dos vértices. Apesar de simples de implementar, uma abordagem deste gênero implicaria realizar muitos cálculos desnecessários.

   Uma simples melhoria a esta hipótese, possível dado o caso presente, seria primeiro fazer uma DFS partindo do vértice inicial. Uma vez concluída, qualquer vértice não visitado na DFS era removido logo do grafo. Assim, evita-se perder recursos a realizar DFS's em vértices que nunca seriam alcançáveis partindo do depósito e, portanto, irrelevantes para o nosso problema. De seguida, basta fazer uma DFS aos restantes vértices, terminando a procura caso o vértice inicial seja encontrado. Caso chegue ao fim da pesquisa e não seja encontrado, esse vértice também pode ser eliminado uma vez que não existe um percurso de retorno para o vértice inicial. Neste caso, podia ser vantajoso, quando se analisa as arestas de um vértice, verificar primeiro se alguma liga ao vértice inicial, e apenas em caso contrário, começar a processar cada um dos vértices das arestas.

   Uma outra estratégia podia ser, após serem eliminados os vértices não atingidos a partir do vértice primeiro, fazer de novo uma DFS partindo do primeiro nó, mas invertendo previamente as direções das arestas, como é explicado em "The Algorithm Design Manual" (*5.10.2 Strongly Connected Graphs*, *pag. 181*). Assim, os vértices alcançados pela segunda DFS

equivalem aos vértices nos quais é possível regressar ao vértice inicial. Na realização da segunda parte do trabalho será necessário testar qual das duas estratégias se torna mais proveitosa para o tipo de dados com que iremos lidar.

2. Aplicando o algoritmo de **Floyd-Warshall** ao grafo, o mesmo iria descobrir o caminho mais curto entre todos os pares de vértices. O resultado estaria numa matriz de adjacências A, na qual, se assume que se terá inicializado as distâncias com ∞.

   Analisando esta matriz, sendo o vértice inicial v_i , percorre-se a coluna de índice do vértice v_i, pelo que todos os vértices v_x ,tal que, $A[v_i][v_x] = \infty$ são removidos uma vez que, se mantiveram a distância inicial, indica que não existe um percurso de v_i até ao mesmo. De seguida, o inverso é feito, isto é, todos os vértices que verifiquem a condição $A[v_x][v_i] = \infty$ são também removidos.

   Dependendo do caso, este algoritmo pode ser bastante moroso. Imaginemos, por exemplo, um grafo no qual temos um subgrafo extenso e fortemente conexo, mas que não se encontra ligado ao subgrafo onde o vértice inicial se encontra. Todas as distâncias dos vértices desse subgrafo seriam calculadas, sendo no final, todos esses vértices eliminados.

   Uma forma de, possivelmente, melhorar esta implementação seria realizar primeiro uma DFS partindo do vértice inicial e eliminando todos os vértices que não tivessem sido alcançados. Só depois se realizava o algoritmo de Floyd-Warshall ao grafo, agora atualizado. Seguindo esta implementação a primeira verificação não seria necessária ($A[v_i][v_x] = \infty$), ou seja, todos os vértices já são alcançáveis partindo do inicial, sendo apenas verificado o contrário. Esta otimização teria de ser testada na segunda parte de modo a concluir se de facto, melhora significativamente a *performance*.

Cálculo de distâncias

Depois de nos certificarmos de que todos os pontos são alcançáveis e descartar as encomendas impossíveis de entregar, ou impossíveis de regressar uma vez entregues, passamos ao cálculo das distâncias entre pares de pontos do grafo. Relembramos o facto de, após o pré-processamento, o grafo obtido é fortemente conexo, ou seja, a partir de qualquer vértice é possível chegar aos outros.

Caso se utilize o algoritmo de Floyd-Warshall para o teste da conectividade (2.) , temos a vantagem que todas as distâncias já estarão calculadas, pelo que, só é necessário que o algoritmo guarde numa matriz P[*i*][*j*] os vértices predecessores no caminho mais curto de *i* a *j*, de forma a ser possível criar uma função *getPath()* que retorne a sequência de vértices que fazem parte do caminho mais curto entre dois vértices.

Caso não se opte por utilizar o algoritmo de Floyd-Warshall, o cálculo das distâncias entre os vértices dependerá da solução pretendida, bem como do algoritmo utilizado.

Começando pela primeira solução - encontrar o melhor caminho para realizar uma entrega com um camião - está lhe subjacente a determinação das distâncias mínimas entre os vértices pelo que, neste caso, fazer este cálculo acaba por ser redundante à própria natureza da variante.

Para as restantes versões e respetivos algoritmos, é indispensável ter disponível as distâncias entre quaisquer dois pontos do grafo. Nestes casos, pode-se optar por utilizar um dos algoritmos usados na resolução da variante inicial. Quando é chamada a função para obter a distância entre dois vértices, verifica-se primeiro se essa distância já não foi calculada (recurso a uma estrutura que guarde os valores já calculados), caso não o tenha sido, é obtida recorrendo ao método escolhido, guardada na estrutura e finalmente devolvida.

Entrega de uma encomenda por um camião

Este problema pode ser interpretado como um problema de cálculo do melhor (mais curto) caminho entre dois pontos que passe também por um certo ponto escolhido, considerando o depósito como origem, a garagem (que poderá ser localizada no próprio depósito) como destino e o local de entrega da encomenda como o ponto escolhido.

Se o teste da conectividade for feito com recurso ao algoritmo de Floyd-Warshall, este problema fica resolvido utilizando a função *getPath()*, referida anteriormente, desde o depósito até ao local de entrega, e deste este até à garagem.

Por outro lado, caso a condição anterior não se concretize, é possível usar uma pesquisa bidirecional, aplicando um algoritmo de procura do caminho mais curto a partir do depósito e uma outra aplicação do algoritmo a partir da garagem no grafo invertido.

Quando cada uma das pesquisas encontrar o vértice correspondente ao local de entrega, estas terminam retornando o caminho encontrado até ao local de entrega. A junção dos dois caminhos dará o resultado pretendido.

Como algoritmo de pesquisa tentar-se-á usar tanto o A* como o Dijkstra e comparar os resultados obtidos por cada.

Estes algoritmos são no fundo bastante idênticos, distinguindo-se o primeiro pela função heurística a que recorre para aumentar a sua eficácia e ajudar a que a pesquisa evolua sempre na direção desejada com vista a diminuir a distância entre o vértice inicial e o final (que no nosso caso será o ponto de entrega). Assim sendo, podemos estimar esta distância de várias formas que compararemos entre si:

1. Recorrendo à distância euclidiana que calcula o comprimento do segmento de reta que une os dois pontos em causa;
2. Recorrendo à distância de Manhattan que calcula a soma do valor absoluto das diferenças das coordenadas dos dois pontos. Esta forma de determinar distâncias foi pensada para a

organização moderna das cidades muitas vezes feita em quadrículas (daí o nome Manhattan, inspirado nas ruas desta cidade), logo é de esperar que funcione melhor para este tipo de planeamento urbanístico;

3. Recorrendo ao quadrado da distância euclidiana. Esta forma não temos garantias que funcione, mas iremos ver se evitando a raiz quadrada nos cálculos e fazendo o quadrado dos pesos quando necessários, produz ou não melhores resultados devido à diminuição do overhead causado pela operação original.

Entrega de todas as encomendas por um camião

Este problema representa um caso particular do *Travelling Salesman Problem*, na medida em que o ponto inicial pode ser diferente do final. No entanto, esta pequena diferença não irá afetar a solução, dado que para este problema só é importante a ordem das entregas das encomendas e, para simplificação do problema, não vamos ter em consideração o local da garagem aquando da ordenação das entregas, uma vez que este é obrigatoriamente o último local para onde o camião se tem de dirigir.

A solução mais direta para este problema seria tentar todas as permutações possíveis, isto é, testar todas as rotas possíveis que partem do depósito, passam por todos os locais de entrega e retornam à garagem e, de seguida, verificar qual das rotas era a mais eficaz. Ou seja, estaríamos a usar **Brute-Force-Search**. O problema desta solução é, evidentemente, a sua impraticabilidade dada a **complexidade temporal** de **O(|V|!)**. Desta forma, torna-se necessária a procura de heurísticas que procurem obter uma solução aproximada.

A heurística do vizinho mais próximo (**Nearest Neighbour**) é de fácil implementação e apresenta uma melhoria significativa face à solução mencionada anteriormente. No entanto, este algoritmo nem sempre leva uma solução ótima dada a sua natureza gananciosa, sendo que, em média, leva uma rota 25% maior do que a mais eficaz possível.

Existem algumas maneiras de melhorar esta aproximação com a família de algoritmos apelidada de pesquisa local que procuram otimizar estes problemas, entres os quais constam o **2-opt**, **3-opt** e **Lin-Kernighan.** Estes são progressivamente mais complexos de implementar, pelo que não se garante a presença de todos numa futura fase do trabalho (uma breve explicação de cada um destes encontra-se no subcapítulo seguinte sob o título de Algoritmos de Pesquisa Local).

Entrega de encomendas em camiões limitados

De todos, este é o problema mais complicado tratando-se de uma variante do problema de roteamento de veículos em que estes são capacitados. Tratando-se de outro problema NP-difícil (resolúvel em tempo não polinomial), os algoritmos vulgarmente utilizados para o tratar, ultrapassam bastante o âmbito desta cadeira. Entre estes constam algoritmos genéticos, pesquisas tabus (semelhantes aos métodos de pesquisa local, mas não ficam presos num mínimo local, pois concebem a possibilidade de piorar ligeiramente a solução corrente numa iteração para a melhorar substancialmente na próxima), entre outros.

Uma hipótese mais simples de solução passaria por enviar um camião segundo o algoritmo anterior que quando se encontrasse cheio seguiria para a garagem e assim sucessivamente (esta aparente temporalidade, não se traduz para o caso real pois os caminhos são pré concebidos e, por isso, os camiões podem partir ao mesmo tempo). Este algoritmo é longe de perfeito, mas é uma boa primeira maneira de garantir que todas as encomendas são entregues (se existir camiões suficientes) sem que a capacidade dos mesmo seja ultrapassada.

No final de encontrarmos uma solução poderão ser aplicados algoritmos semelhantes aos referidos para a variante anterior para tentar otimizar a solução encontrada.

Principais Algoritmos

0. Algoritmo de Tratamento dos Dados

A preparação dos dados é realizada facilmente assumindo que já se tem numa estrutura M as encomendas que necessitam de ser entregues e numa estrutura V os vértices do grafo, basta percorrer uma vez as encomendas e adicioná-las ao respectivo vértice, atualizando o seu peso. Optou-se por adicionar um atributo com o peso total, pois apesar de ocupar mais espaço, acreditamos que, o tempo que se poupará em *run-time* a fazer esta soma, o justifique.

```
for(Mercadoria m : M){
    Vertex v = V.find(m.houseId);

    v.addEncomenda(m);
    v.addWeight(m.peso);
}
```

1. Algoritmo de Pesquisa em Profundidade

O algoritmo de pesquisa em profundidade aproveita a recursividade para determinar todos os vértices alcançáveis a partir de um inicial. De modo a evitar recursão infinita, caso o grafo tenha ciclos, cada vértice tem um atributo "*visited*" que indica se já foi ou não percorrido.

A função começa por ser chamada e analisa o primeiro vértice, marcando-o como *visited*. De seguida, são percorridas as arestas, a função é de novo chamada para cada vértice da aresta que ainda não tenha sido visitado. No final, os vértices marcados como visited são alcançáveis através do primeiro vértice, os restantes não.

Este algoritmo é utilizado em várias aplicações, tais como:

- Para grafos não pesados, produz a árvore mínima de expansão;
- Detetar ciclos no grafo, caso seja encontrado um vértice já marcado como *visited*, o grafo tem pelo menos um ciclo;
- Encontrar um caminho entre dois vértices (não necessariamente o mínimo), basta começar num dos vértices mantendo numa pilha os vértices percorridos até ser encontrado o segundo vértice;

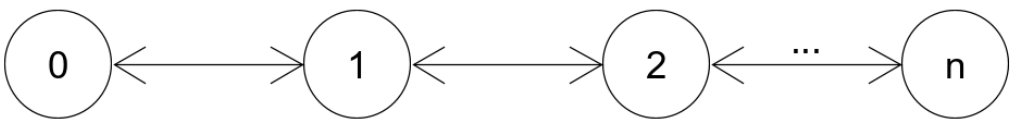
No nosso caso, sendo o objetivo avaliar a conectividade do grafo tendo em atenção o vértice inicial, uma possível implementação é demonstrada no seguinte pseudocódigo:

```
CLEAN();

v_init = V.inital;

DFS(v_init);

for(v : V){
    if(v.visited == false)
        V.remove(v);
}

for(v : V){
    CLEAN();

    if(v.initial == false)
        DFS_init(v);
        if(v_init.visited == false)
            V.remove(v);
}
```

```
CLEAN():
for(Vertex v : V){
   v.visited = false;
}

DFS(v):
v.visited = true;
for (w : v.adj){
   if(w.visited == false)
       DFS(w);
}

DFS_init(v):
v.visited = true;
for (w : v.adj){
    if(w.init == true)
        w.visited = true;
    else if(w.visited == false)
        DFS_init(w);
}
```

Este algoritmo apresenta complexidade temporal **O(|V| x (|V| + |E|))** . Cada limpeza (CLEAN) tem complexidade **O(|V|)**. O algoritmo clássico DFS tem complexidade temporal **O(|V| + |E|)** dado que cada vértice só é visitado uma vez ou nunca, sendo os vértices adjacentes ao visitado os próximos a serem processados. Isto acontece ainda na primeira pesquisa em profundidade ao vértice inicial. De seguida têm-se um ciclo que percorre os restantes vértices e realiza uma pesquisa em profundidade que pode terminar antes de concluída se o vértice inicial for processado. Contudo, no pior caso todos os vértices serão percorridos sendo o inicial encontrado (ou não) no final, pelo que, a complexidade temporal ainda se assume ser **O(|V| + |E|)**, em que, |V| e |E| podem ser menores caso tenham sido eliminados vértices com a primeira DFS. Apesar de ser possível com cada DFS reduzir o número de vértices do grafo (especialmente com a primeira pesquisa), no caso que que o grafo seja fortemente conexo, isto é, cada vértice é alcançável a partir de qualquer outro vértice, o pré-processamento não seria necessário (não provoca alterações) e consequentemente a complexidade temporal é máxima. Obtemos então uma complexidade temporal do algoritmo final de **O(|V| + (|V| + |E|) + |V| x (|V| + (|V| + |E|)))**, equivalente a ter **O(|V| x (|V| + |E|))**.

Quanto à complexidade espacial, a natureza recursiva do algoritmo implica que, no pior caso, a função é chamada |V| vezes (ver figura), obtendo-se uma complexidade **O(|V|)**.



2. Floyd–Warshall Algorithm

O algoritmo de Floyd-Warshall aplica-se a grafos pesados dirigidos, calculando a distância mínima entre quaisquer pares de vértices do grafo. Só é possível aplicá-lo a grafos nos quais não existam ciclos com arestas de peso negativo.

A resolução do problema pode ser feita também utilizando uma execução repetida do algoritmo de Dijkstra, contudo, este último só é aconselhável caso o grafo seja esparso (sendo o algoritmo de Floyd-Warshall inequivocamente melhor em grafos densos). A densidade de um grafo simples dirigido é calculada através da seguinte fórmula $D = \frac{|E|}{|V|(|V|-1)}$, variando entre 0 (mínimo) e 1 (máximo). Ainda em casos onde o grafo seja esparso, o algoritmo de Floyd-Warshall pode ser mais eficiente dada a sua simples implementação em código.

O algoritmo toma proveito de uma matriz de adjacência D[*i*][*j*], que irá conter as distâncias mínimas entre quaisquer dois vértices *i* e *j*, que é inicializada segundo os pesos ∞ se *i* ≠ *j* ou 0 se *i* = *j*. São então encadeados 3 ciclos que percorrem a totalidade dos vértices usando 3 variáveis *i, j, k*, tal que, em cada iteração, o valor da matriz em D[*i*][*j*] é atualizado caso D[*i*][*j*] > D[*i*][*k*] + D[*k*][*j*], ou seja, caso o percurso mais eficiente de momento entre *i* e *j* tenha um peso maior ao percurso de *i* a *k* seguido do percurso de *k* a *j*, este último passa a ser o percurso mais eficiente. Por outras palavras, se um percurso que passe pelo vértice intermédio *k* é menor que o melhor até agora encontrado, esse percurso passa a ser o ótimo (mais curto) até ser, possivelmente, substituído por outro mais curto.

Este algoritmo pode ser utilizado para encontrar ciclos negativos presentes num grafo. Simplificando, a cada iteração é analisada a diagonal da matriz de adjacências, verificando se algum dos seus valores é negativo. Isto porque, num grafo em que não haja ciclos com arestas negativas, o caminho mais curto entre um vértice e ele próprio é sempre 0 (como é inicializada a diagonal da matriz). Isto não será o caso para todos os vértices naqueles que pertencerem a um ciclo negativo, pois ao percorrer esse ciclo, o caminho mais curto tornar-se-ia menor que 0. Desta forma, ao verificar a diagonal da matriz é possível verificar a sua ocorrência.

Para o nosso problema, o cálculo das distâncias interessa-nos primeiramente para encontrar vértices que sejam inatingíveis a partir do vértice inicial ou que o mesmo seja inalcançável uma vez nesses vértices e, consequentemente, eliminá-los do conjunto de vértices.

Uma possível implementação do algoritmo, específico ao nosso caso e sem a otimização referida (ver Pré-Processamento), encontra-se demonstrada no seguinte pseudocódigo:

```
FLOYD_WARSHALL():
double D[V.size()][V.size()];

for(int i = 0; i < V.size(); i++){
    for(int j = 0; j < V.size(); j++){
        if(i == j)
            D[i][j] = 0;
        else
            D[i][j] = ∞
    }
}

for(Vertex vertex : V) {
    for(Edge edge : vertex.adj) {
        D[vertex.index][edge->vertex.index] = edge.weight;
    }
}

for(int k = 1; k < V.size(); k++){
    for(int i = 0; i < V.size(); i++){
        for(int j = 0; j < V.size(); j++){
            if(D[i][j] > D[i][k] + D[k][j])
                D[i][j] = D[i][k] + D[k][j]
        }
    }
}
```

```
VERIFICATION():
int init = V->inital.index;
vector<Vertex> unreachable;

for(int k = 0; k < V.size(); k++){
    if(D[init][k] = ∞ || D[k][init] = ∞)
        unreachable.add(V[k]);
}

for(Vertex vertex : unreachable)
    V.remove(vertex);
```

O algoritmo de Floyd-Warshall tem complexidade temporal **O(|V|³)**, visto que, de modo a calcular o caminho mais curto entre todos os pares de vértices, **O(|V|²),** em ambos os sentidos, **O(2|V|²)**, compara-se com todos os possíveis vértices intermédios, **O(|V|)**, o que resulta numa complexidade de **O(|V| x 2|V|²)** equivalente a **O(|V|³)**.

Em termos de complexidade temporal, dado que é necessário guardar os valores numa matriz bidimensional, **|V| x |V|**, e considerando que não existe recursividade, a resultante complexidade é de **O(|V|²)**.

Importante de referir que, a razão pela qual se torna mais eficiente usar este algoritmo em grafos densos ao invés de uma execução repetida do algoritmo de Dijkstra (complexidade temporal **O(|E| x log(|V|))**) é porque, enquanto que a complexidade temporal do algoritmo de Floyd-Warshall se mantém **O(|V|³)**, a execução repetida de Dijkstra pode resultar numa complexidade temporal no pior caso de **O(|V|³ x log(|V|))**.

3. Algoritmo de Dijkstra

O algoritmo de Dijkstra pretende encontrar o caminho de menor peso desde um dado vértice até todos os outros vértices, num grafo dirigido.

Este algoritmo vai constantemente acumulando as distâncias desde o ponto inicial até ao vértice que está ser processado, e usa estes valores para poder fazer a análise do caminho mais curto. Desta forma, pode também ser utilizado apenas para calcular a distância mínima entre dois vértices do grafo.

O algoritmo Dijkstra, a ser usado, pode ter uma modificação relativamente ao estudado nas aulas, dado que para o problema em questão não há a necessidade de calcular o caminho mais curto para todos os outros vértices, mas apenas para o local de entrega da encomenda (e desta para a garagem), assim sendo, quando finalmente se extrair da fila de prioridade o vértice correspondente ao ponto onde se encontra a encomenda, o algoritmo poderá terminar.

Em seguida, é apresentado o seu pseudocódigo:

```
// G = (V, E), s = vértice inicial, d = local de entrega
Dijstra(G, s, d):
    for(Vertex v : V){
        v.distance = ∞;
        v.path = null;
    }

    s.distance = 0;
    priorityQueue q = {}; // Minimum priority queue

    q.insert(s);

    while(!q.isEmpty){
        Vertex v = q.extractMin; // Greedy

        if(v == d)
            return G.getPath(s, d);

        for(Edge w : v.adj){
            if(w->vertex.distance > v.distance + w.weight){
                w->vertex.distance = v.distance + w.weight;
                w->vertex.path = v;

                if(q.find(w->vertex)) // vertex already in queue
                    q.decrease_key;
                else
                    q.insert(w->vertex);
            }
        }
    }
```

Este algoritmo, dependendo da estrutura onde se guarde os vértices a analisar, poderá apresentar **complexidade temporal** de **O((|V|+|E|) *|V|)** caso se guarde num vetor ou *array* que exija procurar o elemento mais pequeno sempre que se quer extrair um novo vértice. Terá complexidade de **O((|V|+|E|) * log|V|)** caso usemos uma fila de prioridade já que o tempo de inserção de um elemento numa destas é logarítmico, **O(log N)**), podendo ser otimizado até **O(|V| * log|V| + |E|)**, caso se use *Fibonacci Heaps*. Há ainda uma fase de preparação do grafo que necessita de percorrer todos os seus vértices, **O(|V|)**, e no final a função getPath() também terá de, no limite, percorrer todos os vértices

(caso bastante improvável em que o caminho mais curto do depósito ao ponto de entrega passa por todos os vértices), complexidade **O(|V|).**

Espacialmente a complexidade será de **O(|V|)** devido à necessidade do uso de uma estrutura auxiliar, que poderá armazenar até à totalidade dos vértices do grafo.

4. Algoritmo de Dijkstra Bidirecional

Para a implementação deste algoritmo são necessárias algumas operações adicionais, como a inversão do grafo e uma consequente adaptação do algoritmo de Dijkstra para operar sobre o gráfico invertido e para ser capaz de obter o caminho calculado na ordem correta.

Como a ideia da pesquisa bidirecional é ter duas pesquisas a correr em "simultâneo", um partindo da origem e outro partindo do destino, tentaremos usar *multithreading* para alcançar esse objetivo.

Sendo que cada pesquisa termina quando encontrar o local de entrega da encomenda, não há necessidade de termos um ciclo que termina quando as duas pesquisas encontram o mesmo vértice, como é habitual nas pesquisas bidirecionais.

O algoritmo a ser usado poderá apresentar, portanto, o seguinte pseudocódigo:

```
// G = (V, E), src =vertice de origem, dest = vertice de destino
// delivery = local de entrega de encomendas

BidirectionalDijkstra(G, src, dest, delivery):

    //Initialize reversed edges
    initReversedGraph

    //Use multi-threading to alternate between normal and reversed Dijkstra
    dijsktraResult = Dijkstra(G, src, delivery);
    dijsktraReverseResult = DijsktraReverse(G, dest, delivery);

    return append(dijsktraResult, dijsktraReverseResult);
```

Para obter o grafo revertido, pode-se percorrer todos os vértices do grafo e, para cada um destes, guardar um vetor de arestas que neles entram. Desta forma, a adaptação de Dijkstra para o gráfico revertido, terá, apenas, de percorrer este vetor em vez daquele que guarda as arestas que saem de cada vértice. Esta operação terá uma **complexidade temporal** de **O(|V|+|E|)** e espacial de **O(|E|)**, já que terá de guardar novamente todas as arestas, mas na ordem inversa.

O resto do algoritmo é simplesmente Dijkstra aplicado duas vezes e, portanto, a sua **complexidade temporal** será a mesma que o original, **O((|V|+|E|) * log|V|)**. A inversão do grafo original requer a criação de uma lista de arestas invertidas para cada vértice, portanto a complexidade **espacial** deste algoritmo será igual à de Dijkstra mais o número de arestas do grafo, ou seja, **O(|V|+|E|)**. Assim, acaba por ter as mesmas complexidades que o algoritmo de Dijkstra original.

5. Algoritmo A* (A estrela)

Este algoritmo é em tudo semelhante ao Dijkstra (não o fosse ele apenas um melhoramento heurístico deste último). Sendo assim, também o A* calcula a menor distância entre dois pontos através de um algoritmo ganancioso que avalia primeiro os vértices que considera mais promissores.

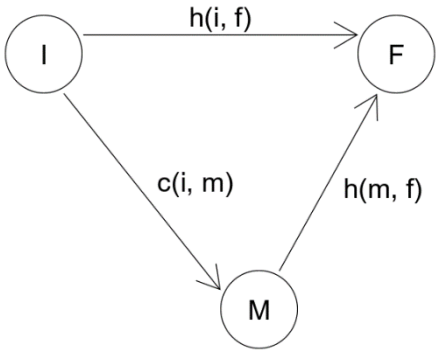
A função que permite avaliar a tomada de decisão pode ser escrita como $f(v) = g(v) + h(v)$, com $g(v)$ igual ao custo desde o vértice inicial até ao ponto v, e $h(v)$ a função heurística a implementar, que estima a distância deste ponto ao final. É também de salientar que ambas as funções devem vir nas mesmas unidades para que a função f faça sentido. Conseguimos ainda perceber facilmente que o algoritmo Dijkstra não passa de uma situação particular deste em que a função *h* para qualquer ponto tem o valor 0.

As diferenças prendem-se, portanto, em como está ordenada a fila de prioridade que, devido à introdução de uma função heurística a ser contabilizada para o seu custo/peso, apresentará uma composição diferente do que teria caso se usasse o algoritmo precedente.

Conseguimos notar a introdução deste novo elemento no seguinte pseudocódigo:

```
Heuristic(a, b):
    return (sqrt((a.x - b.x)^2 + (a.y - b.y)^2)); // Using Euclidian distance
    |
    return (abs(a.x - b.x) + abs(a.y - b.y));     // Using Manhattan distance
    |
    return ((a.x - b.x)^2 + (a.y - b.y)^2)        // Using squared Euclidian distance


// G = (V, E), src = vértice inicial, dest = local de entrega
A_Star(G, src, dest):
    for(Vertex v : V){
        v.distance = ∞;
        v.path = null;
    }

    src.distance = 0;
    priorityQueue q = {}; // Minimum priority queue

    q.insert(s, 0);       // The priority is given by the second argument

    while(!q.isEmpty){
        Vertex v = q.extractMin; // Greedy

        if(v == dest)
            return G.getPath(src, dest);

        for(Edge w : v.adj){
            heuristic_distance = v.distance - Heuristic(v, dest) + w.weight + Heuristic(w->vertex, dest);

            if(w->vertex.distance > heuristic_distance){
                w->vertex.distance = heuristic_distance;
                w->vertex.path = v;

                if(q.find(w->vertex)) // vertex already in queue
                    q.decrease_key;
                else
                    q.insert(w->vertex, heuristic_distance);
            }
        }
    }
```

Como podemos observar também aqui podemos utilizar a mesma condição de paragem, contudo para que seja encontrado uma solução ótima é necessário que a função seja consistente, ou seja, para qualquer ponto inicial **i**, médio **m** e final **f**, $h(i,f) \leq c(i,m) + h(m,f)$, sendo *h* a função heurística a implementar (euclidiana no caso da figura) e c o custo da aresta. Podemos observar tal fenómeno facilmente através da desigualdade triangular (no limite m, estaria sobre o segmento de reta [i f] e, portanto, a distância seria igual):



As complexidades deste algoritmo serão exatamente as mesmas que no de Dijkstra normal, sendo também preciso recorrer a este algoritmo duas vezes caso se o queira implementar na sua forma mais simples.

6. Algoritmo A* Bidirecional

Como já tinha acontecido para o algoritmo de Dijkstra, também este se poderá correr a sua versão bidirecional, partindo um lado do depósito e outro da garagem com referência ao local da encomenda.

Assim sendo, todos os passos do capítulo 2.3 podem ser aqui aplicados mudando apenas a função chamada:

```
// G = (V, E), src =vertice de origem, dest = vertice de destino
// delivery = local de entrega de encomendas

BidirectionalA_Star(G, src, dest, delivery):

    //Initialize reversed edges
    initReversedGraph

    //Use multi-threading to alternate between normal and reversed A*
    A_StarResult = A_Star(G, src, delivery);
    A_StarReverseResult = A_StarReverse(G, dest, delivery);

    return append(A_StarResult, A_StarReverseResult);
```

Assim sendo, também as complexidades temporal e espacial são idênticas.

7. Algoritmo do Vizinho Mais Próximo

A heurística do vizinho mais próximo (nearest neighbor) é um algoritmo ganancioso que escolhe como seguinte encomenda a ser entregue aquela que estiver mais próxima do local atual e que ainda não foi entregue.

Para o nosso problema, em que o camião tem sempre de começar no depósito e terminar na garagem, a principal função deste algoritmo será ordenar os locais de entrega, sendo que o camião fará a entrega das encomendas por essa mesma ordem.

Este algoritmo apresenta o seguinte pseudocódigo:

```
// DPs = delivery places
NearestNeighbor(G, s, DPs):
    vector result;

    for(Vertex v : DPs){
        v.distance = getDistFrom(s, v);
    }

    priorityQueue q = DPs; // Minimum priority queue by distance

    while(!q.isEmpty){
        Vertex v = q.extractMin;
        result.add(v);

        for(Vertex w : q)
            w.distance = getDistFrom(v, w);
    }

    return result;
```

Tal como referido no capítulo do cálculo das distâncias, a função *getDistFrom()*, referida no pseudocódigo acima, pode ir buscar os valores desejados à matriz de distâncias calculadas pelo algoritmo de Floyd-Warshall, caso este seja utilizado no teste da conectividade. Desta forma, esta operação terá uma **complexidade temporal constante** (**O(1)**). Por outro lado, caso o algoritmo de Floyd-Warshall não seja utilizado, teremos de utilizar um algoritmo de pesquisa do caminho mais curto, como Dijkstra ou A*, para obter as distâncias pretendidas. Para qualquer um destes algoritmos a **complexidade temporal** é **O((|V|+|E|) * log|V|).**

Sendo que esta operação é feita para cada local de entrega em cada iteração do ciclo *while*, a **complexidade temporal** do algoritmo apresentado acima será, para o primeiro caso, **$O(N^2)$**, em que N é o número de entregas a serem realizadas pelo camião. Para o segundo caso apresentado, a **complexidade temporal** será de **$O((|V|+|E|) * \log|V| + N^2)$**.

Para aplicar o algoritmo é apenas necessário uma *minimum-priority queue* para armazenar os locais de entrega que faltam processar e ao mesmo tempo colocar aquele que tem menor distância no topo. Desta forma, a **complexidade espacial** deste algoritmo é **O(N)**.

8. Algoritmos de Pesquisa Local

Neste capítulo apenas se mencionará brevemente como funcionam os algoritmos **2-opt,** o seu sucessor **3-opt** e explicar resumidamente a sua generalização **Lin-Kernighan**.

O primeiro e mais simples destes algoritmos é o 2-opt que, como todos os outros, parte de uma solução (podendo esta ser aleatória) e tenta melhorá-la. Numa primeira fase, começa por tentar descruzar pares de caminhos e, após isto, continua a trocar duas partes da solução encontrada com vista a otimizá-la. A cada iteração compara-se o custo da nova permutação com o da anterior e guarda-se, evidentemente, a que tiver menor.

No nosso caso, podemos começar este algoritmo com uma solução vinda do algoritmo do vizinho mais próximo explicado anteriormente.

```
repeat until no improvement is made {
start again:
    best_distance = calculateTotalDistance(existing_route);

    for(i = 1; i < number of nodes eligible to be swapped - 1; i++){
        for(k = i + 1; k < number of nodes eligible to be swapped; k++){
            new_route = 2optSwap(existing_route, i, k);
            new_distance = calculateTotalDistance(new_route);

            if(new_distance < best_distance){
                existing_route = new_route;
                best_distance = new_distance;
                goto start_again;
            }
        }
    }
}

2optSwap(route, i, k){
    1. take route[0] to rout[i-1] and add them in order to new_route
    2. take route[i] to route[k] and add them in reverse order to new_route
    3. take route[k+1] to end and add them in order to new_route

    return new_route
}
```

Também de notar que apesar deste algoritmo correr até que se convirja para um mínimo local, o número de iterações pode ser limitado criando outra condição de saída do ciclo.

Cada iteração tem ainda **complexidade temporal $O(n^2)$**, e **espacial** constante, **O(1)**, pois não necessita de nenhuma estrutura auxiliar.

A ideia do **3-opt** é a mesma, mas em vez de trocar 2 arestas, em cada repetição do ciclo são eliminadas e recombinadas 3, testando todas as 7 possibilidades de recombinação destas. Assim sendo a complexidade temporal deste algoritmo passa a ser **$O(n^3)$** tornando-o mais lento que o 2-opt, mas podendo encontrar melhores soluções.

Finalmente, a otimização heurística de **Lin-Kernighan** generaliza os outros dois casos e aplica o mesmo tipo de algoritmo, mas com δ (assumindo que $\delta = 2$ no 2-opt e $\delta = 3$ no 3-opt) adaptativo, ou seja, em cada passo decidirá quantas arestas necessitam de ser trocadas para encontrar o caminho mais curto.

Devido à sua maior complexidade, e ao facto de já não se inserir inteiramente no âmbito desta cadeira, a implementação deste é pouco provável, não sendo, por isso, menos digno de menção, uma vez que se enquadra e representa o culminar dos algoritmos apresentados neste capítulo.

Casos de Utilização

Pretende-se que venham a ser implementadas soluções para todos os 3 níveis de problemas bem como a sua respetiva interface.

Assim, é de prever que o programa tenha um primeiro menu que permita escolher entre selecionar um mapa, operar sobre o mapa carregado ou sair.

As operações possíveis no menu seguinte passariam por visualizá-lo recorrendo ao **GraphViewer**, adicionar e remover depósito e garagem, testar a sua conectividade, verificação de ligação entre eles e outra opção para resolver as três variantes de problema.

Dentro desta última opção, encontrar-se-á um menu para escolher entre os vários problemas e as diferentes implementações dos mesmos:

1. **Um camião a realizar uma entrega**: neste modo é requerido ao utilizador que selecione um local de entrega e será lhe demonstrado o caminho mais curto que parte do depósito, passa pelo local de entrega e termina na garagem. Como algoritmos a usar, o utilizador pode escolher entre **Dijkstra** e **A***, **unidirecional** e **bidirecional**.
2. **Um camião a realizar um conjunto de entregas**: neste modo o utilizador seleciona uma data de entregas e é lhe apresentado um caminho que comece no depósito, realize todas as entregas e acabe na garagem. Haverá pelo menos duas opções de algoritmos a escolher: a opção com a heurística do **vizinho mais próximo** e pelo menos mais uma com uma **pesquisa local** (provavelmente a 2-opt) em que o utilizador poderá dizer qual o número máximo de iterações que deseja ver feitas ou a partir de que limite é que considera que a otimização já é desprezável (*threshold*);
3. **Frota de camiões com capacidade limitada**: após inserir vários camiões, com as respetivas capacidades, o utilizador assinala ainda as várias mercadorias e o peso/volume correspondentes, sendo depois calculadas as diferentes rotas que cada camião deverá fazer, baseado em combinações dos algoritmos utilizados nos casos anteriores.

Conclusão

A realização deste relatório foi mais complicada do que aquela que primeiramente se esperava. A escrita de muitos dos capítulos sem base empírica por trás, fazendo projeções para o futuro realizou-se uma tarefa mais difícil do que pensamos que seria inicialmente (exemplo mais gritante talvez seja o dos casos de utilização onde estamos a escrever sobre algo que não existe nem sabemos se os conseguiremos implementar todos), pois não tínhamos forma de ver se uma certa resolução resulta ou se será eficaz.

A organização e estrutura também não foi fácil, uma vez que alguns dos tópicos quase que se intercalavam e a sua distinção não nos era óbvia (identificação das técnicas de conceção e os principais algoritmos a serem implementados por exemplo).

A outra grande dificuldade foi a falta de hábito no tratamento de problemas de índole mais prática que tivessem de recorrer a processos heurísticos muito pouco abordados nas aulas. Por causa disto, inicialmente não sabíamos bem por onde começar a pesquisar e quando começamos a encontrar o que queríamos, deparámo-nos com soluções muito mais complexas do que aquelas que pensamos ser o objetivo deste trabalho e da cadeira (as resoluções para a última variante do nosso problema utilizam muitas vezes algoritmos genéticos e outras meta heurísticas).

Ainda de salientar que o facto de o trabalho ter sido realizado maioritariamente nas férias e ter sido feriado na nossa última aula prática, a discussão com o professor das práticas sobre estes algoritmos ficou dificultada.

Apesar de tudo, achamos que conseguimos conciliar espantosamente bem o esforço e trabalho de cada um sendo a divisão do trabalho realizado um pouco complicada, pois acabamos por participar na conceção de quase todas as partes e às retificações que iam sendo acrescentadas, graças à ferramenta de edição online de texto da google (Google Docs).

Referências bibliográficas

- Slides das aulas teóricas de Análise e Concepção de Algoritmos fornecidos ao longo do semestre do ano letivo de 2018/2019;
- Skiena, Steven S. *The Algorithm Design Manual*. 2nd ed., Springer, 2008.
- Vehicle Routing, http://people.brunel.ac.uk/~mastjjb/jeb/or/vrp.html
- Depth First Search or DFS for a Graph, https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
- Applications of Depth First Search, https://www.geeksforgeeks.org/applications-of-depth-first-search/
- Dijkstra's Algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Bi-Directional Dijkstra's Algorithm, https://github.com/rast-7/Bi-Directional-Dijkstra
- Floyd-Warshall Algorithm, https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm
- Floyd-Warshall Algorithm | DP-16, https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/
- Comparison of Dijkstra's and Floyd-Warshall Algorithms, https://www.geeksforgeeks.org/comparison-dijkstras-floyd-warshall-algorithms/
- A* Search Algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm
- Introduction to the A* Algorithm, https://www.redblobgames.com/pathfinding/a-star/introduction.html
- Implementation of A*, https://www.redblobgames.com/pathfinding/a-star/implementation.html
- Efficient Point-to-Point Shortest Path Algorithms, http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf
- A* Pathfinding for Beginners, https://web.archive.org/web/20170604064300/http://www.policyalmanac.org/games/aStarTutorial.htm
- Introduction to A*, http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
- Heuristics, http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- Euclidean vs Chebyshev vs Manhattan distance, https://lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/
- Admissible Heuristics, https://en.wikipedia.org/wiki/Admissible_heuristic

- An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, http://akira.ruc.dk/~keld/research/LKH/LKH-2.0/DOC/LKH_REPORT.pdf
- Implementing the Lin-Kernighan heuristic for the TSP, https://www.zib.de/borndoerfer/Homepage/Documents/WS11/WS11-CIP-TU-12.pdf
- Problema do Caixeiro Viajante, https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante
- Travelling Salesman Problem, https://en.wikipedia.org/wiki/Travelling_salesman_problem
- Problema do Roteamento de Veículos, https://pt.wikipedia.org/wiki/Problema_de_roteamento_de_ve%C3%ADculos
- Vehicle Routing Problem, https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Vehicle Routing Problem, https://www.youtube.com/watch?v=A1wsIFDKqBk
- Nearest Neighbor Algorithm, https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

Smart Delivery

Tema 4: Empresas de Distribuição de Mercadorias - Parte 2

Professor Regente: Rosaldo José Fernandes Rossetti

Professor das aulas Teórico-Práticas: Francisco Xavier Richardson Rebello de Andrade

Grupo B - Turma 6:

Gaspar Santos Pinheiro | up201704700@fe.up.pt

Manuel Monge dos Santos Pereira Coutinho | up201704211@fe.up.pt

Tito Alexandre Trindade Griné | up201706732@fe.up.pt

Estruturas de Dados Usadas

Grafo

A classe grafo presente na pasta Graph no ficheiro *Graph.h* é essencialmente uma adaptação da classe fornecida nas aulas: acrescentaram-se 4 novos parâmetros para definir os limites (cima, baixo, esquerda e direita) deste, compatibilizando-o, assim, com o GraphViewer; adicionou-se um método para duplicar, outro para inverter o grafo e alguns *getters* e *setters* para permitir a separação dos algoritmos desta classe; por último foi também refeito o destrutor, uma vez que o fornecido não removia/destruía realmente as matrizes criadas, mas antes repunha-as com o valor origem, não libertando o espaço alocado e provocando vários erros de bad_alloc.

Aresta

Esta classe encontra-se na mesma pasta que a anterior sob o nome *Edge.h* e consiste apenas numa separação em *headers* do header *Graph.h* fornecido nas aulas práticas.

Vértice

Semelhante ao Grafo, esta classe é uma adaptação da *Vertex* que se encontrava no ficheiro disponibilizado no decorrer da disciplina mudado para um novo *header* - *Vertex.h*. Assim sendo, não só se adicionou valores de X e Y a cada vértice, como também duas formas de calcular a distância entre dois vértices: a distância Euclidiana e a distância de Manhattan (explicadas no relatório anterior).

Entrega

Foi criada uma classe para representar as entregas/encomendas que teriam de ser feitas (*Delivery*). Cada é composta por um ID, volume correspondente e destino representado por uma classe explicada mais à frente (*MapInfo*).

Camião

Para ajudar à resolução da 3ª versão do problema, fizémos uma classe que pudesse representar um camião e correspondente tarefa (caminho que terá de fazer). Um *Truck*, além do seu ID identificativo e capacidade, tem também um vetor com as Encomendas que terá de entregar e outro com o caminho que terá de percorrer (vector de *Vertex*).

MapInfo

Esta classe é aquela que faz de argumento do Grafo genérico (o grafo e seus algoritmos foram deixados template, para fácil reutilização e adaptação a outro problema). Esta classe tem apenas o ID lido dos ficheiros disponibilizados e duas flags a indicar se se trata do ponto final e/ou do inicial.

Aplicação

A classe *Application* serve como base a todo o programa. Não só guarda o grafo do mapa escolhido na sua integridade, como também o grafo reduzido representativo da componente fortemente conexa

calculada após a escolha dos pontos final e inicial sob o qual serão corridos a maior parte dos algoritmos.

Nesta classe constam ainda um vetor com todos os depósitos carregados das tags, dois *arrays* de vectores dos vários tipos de loja - um para o grafo inteiro e outro para o grafo pequeno - que são acessados por meio de um enum que especifica o tipo de loja, um vector com a totalidade das encomendas e ainda um outro com os camiões. Esta classe pode também ser interpretada como a "central".

GraphViewer

Este módulo, fornecido pelos docentes da cadeira, permite-nos uma melhor visualização dos grafos e das soluções obtidas.

Conectividade

Este foi um dos primeiros problemas com que nos deparamos aquando da implementação das soluções implementadas.

Os mapas fornecidos pelo corpo docente não se encontravam nas melhores condições para que a posterior aplicação dos algoritmos resultasse em soluções interessantes: havia falta de muitas ruas, com rotundas desconexas, levando a que a maior parte das zonas fortemente conexas fossem de tamanhos extremamente reduzidos (em 7 vezes com pontos aleatórios, apenas num conseguiu-se encontrar uma zona que não fosse unitária - tinha 3 vértices. Este teste também foi feito com base nas tags sugeridas tendo resultados semelhantes).

Comunicado este problema, foi nos sugerido que todas as arestas que importássemos dos ficheiros deveriam ser consideradas como sendo bidirecionais. Apesar de não fazer inteiramente sentido num contexto real (as rotundas ficam com dois sentidos, por exemplo), decidimos adotar esta estratégia para nos permitir resultados mais interessantes.

Contudo, como esta simplificação não só facilita imenso os algoritmos posteriores, mas também remove um bocado da praticidade do problema, optamos por adicionar uma flag no início do programa que nos permite carregar e correr os algoritmos em modo grafo dirigido ou com arestas bidirecionais.

Assim sendo, os testes de conectividade implementados foram não só uma simples verificação de se o ponto final e o inicial se encontram na mesma CFC (componente fortemente conexa), através de um método de procura em profundidade com começo no vértice definido como depósito, mas também o cálculo da CFC que contém este ponto.

Devido à possibilidade do grafo ser ou não dirigido, o cálculo da CFC pode ser feito de duas formas uma mais eficiente que a outra: caso base é o algoritmo descrito na primeira parte deste trabalho (DFS -> inverter Grafo -> DFS), mas como, caso as arestas sejam bidirecionais, é como se o grafo já estivesse invertido, este algoritmo pode ser reduzido a uma simples DFS (ou BFS).

Casos de Utilização

Os casos de utilização implementados acabaram por ser mais ou menos os previstos na primeira parte deste trabalho:

Menu Inicial

Neste primeiro menu é dada a possibilidade de escolher um mapa e mostrá-lo ou operar sobre ele caso este já se encontre carregado. É possível escolher qualquer um dos 11 mapas disponibilizados pelo monitor, contudo mapas como o de Portugal poderão demorar algum tempo a carregar devido à sua dimensão. O mapa é visto noutra janela com recurso ao GraphViewer e a terceira opção encaminha-nos para um novo menu.

```
============================================================

              Welcome to the Smart Delivery

============================================================

OPTIONS:

  1 - Choose Map
  2 - Show Map
  3 - Map Operations
  0 - Exit

Option ?
```

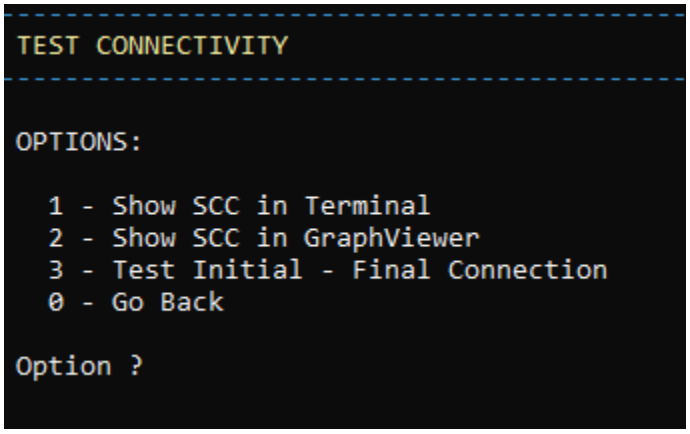
Menu Mapa

Aqui o utilizador pode escolher entre voltar para trás e escolher outro mapa, ir para um menu que permite definir, remover, mostrar no GraphViewer ou informações mais detalhadas no terminal sobre um ponto inicial ou final, testar a conectividade do grafo e ainda prosseguir para a resolução dos problemas

```
---------------------------
MAP OPERATIONS
---------------------------

OPTIONS:

  1 - Initial Point
  2 - Final Point
  3 - Test Conectivity
  4 - Solve Problems
  0 - Go Back

Option ?
```

Menu Conectividade

Após definir pelo menos um ponto inicial, o utilizador pode escolher testar a conectividade em função deste ponto deparando-se assim com 3 possibilidades: mostrar a componente fortemente conexa no terminal, mostrá-la com recurso ao GraphViewer e fazer um simples teste de verificação se existe caminho entre o ponto definido para início e aquele que se escolheu para garagem, utilizando uma pesquisa em profundidade.



Menu Problemas

Este menu serve apenas como ponto de passagem para a escolha de uma das 3 variantes do problema ou de visualização do novo grafo reduzido (do menu anterior para este, ocorre uma seleção dos vértices atingíveis).

Menu Problema 1

Caso a primeira opção seja escolhida (*One Truck - One Delivery*), o utilizador é levado para um menu de seleção do tipo de entrega que pretende fazer. Após esta escolha são nos apresentados 3 algoritmos que, aquando da sua conclusão, apresentarão a solução com recurso ao GraphViewer. O ponto de entrega terá a cor azul, amarelos, ligeiramente mais pequenos, serão os vértices que compõem o percurso, a verde, grande, o inicial e a vermelho o final. As arestas apresentadas são aquelas que fazem parte da solução, indicando a numeração nelas a ordem a percorrer. Os vértices cinzentos mais pequenos são aqueles desprezáveis para a solução.

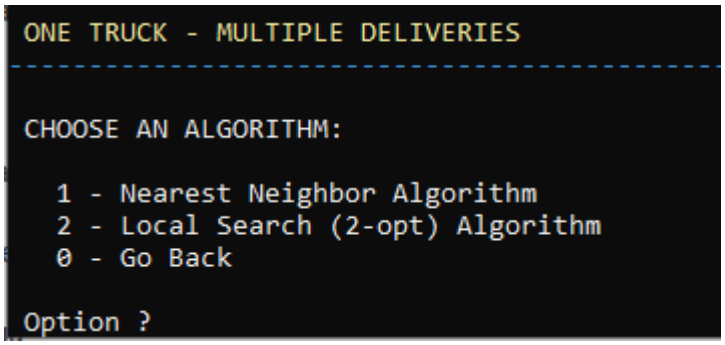
Os possíveis algoritmos para a resolução desta variante são o Dijkstra, o A* com a heurística da distância euclidiana e o A* com a heurística da distância de Manhattan. Todos estes são corridos na sua forma bidirecional em threads separados: um thread a começar na origem e outro no final. Caso o grafo seja dirigido é, ainda necessário recorrer à inversão do mesmo, caso contrário apenas o tem de se duplicar.

Menu Problema 2

Semelhante ao que acontecia no anterior, o utilizador é levado para um menu onde escolhe as entregas que pretende fazer com a diferença que neste existe a possibilidade de escolher mais do que uma entrega, acabando a seleção quando se carrega no 0.

Depois disto, encontramos um menu que nos permite escolher qual heurística/otimização pretendemos usar (esta versão já se trata de um problema NP-Difícil como explicado na primeira parte). Podemos escolher a heurística do vizinho mais próximo quer com recurso ao cálculo da tabela do algoritmo de Floyd-Warshall quer usando a distância euclidiana, ou o algoritmo de procura local 2-opt.

O diagrama de cores é semelhante ao anterior.



Menu Problema 3

Finalmente, temos a opção da resolução do problema na sua completude. Aqui o utilizador não só é inquirido acerca dos locais de entrega, como também do volume da sua encomenda (nesta última versão os camiões apresentam uma capacidade máxima). Os veículos e suas capacidades são lidos de um ficheiro texto chamado *trucks.txt* e só após isto é que é feito o cálculo de um caminho.

Após a obtenção das encomendas e do seu volume, o utilizador, tal como para o problema 2, é questionado sobre se pretende usar Floyd-Warshall ou A* com distância euclidiana para o cálculo dos melhores trajetos para os diferentes camiões.

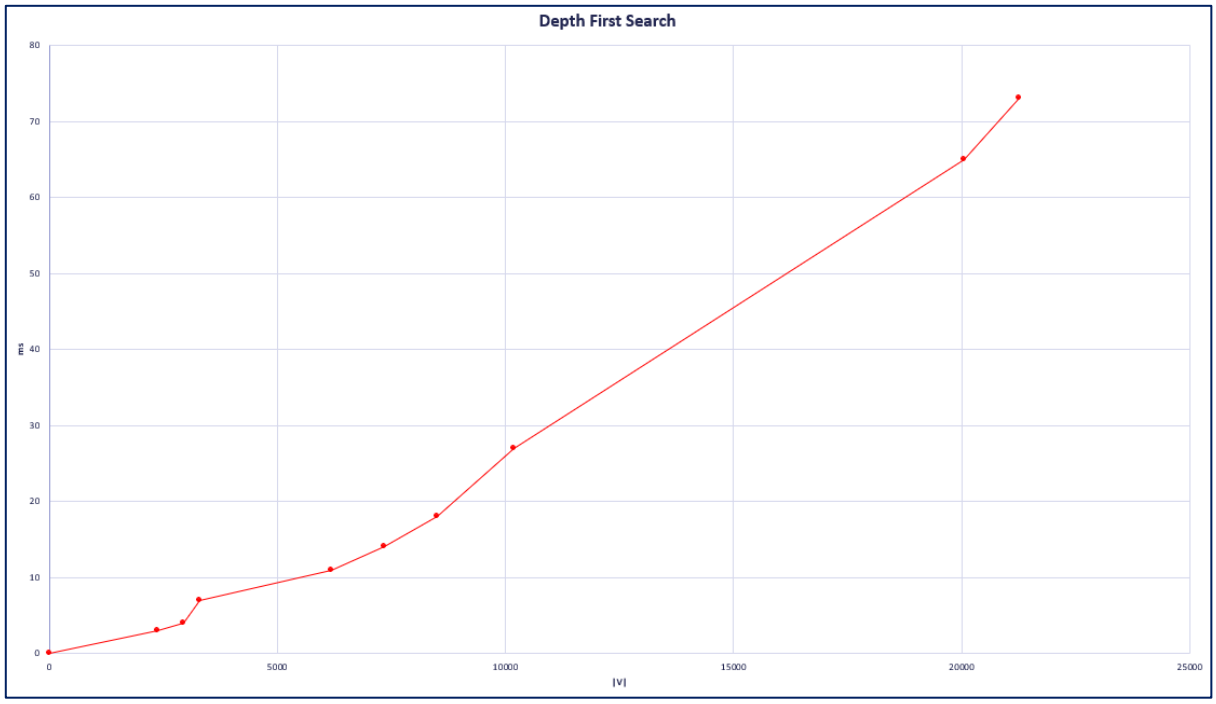
Algoritmos

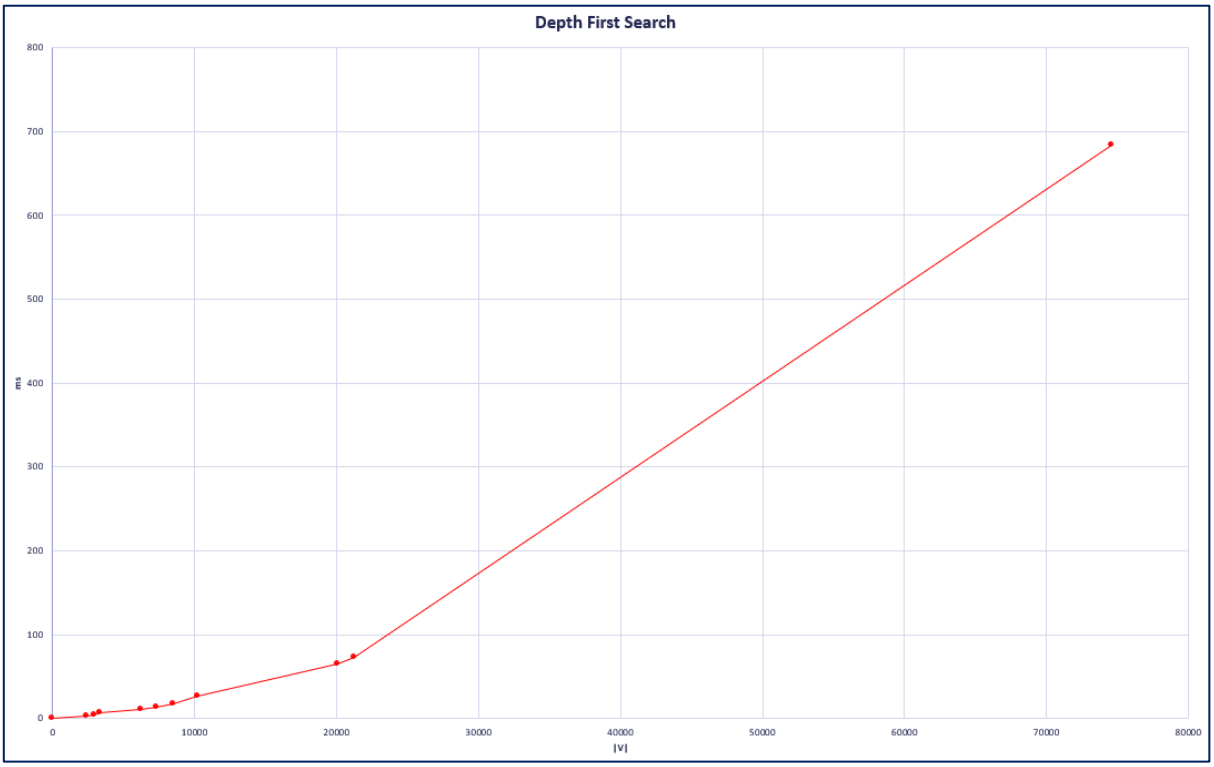
Tanto o pseudocódigo como a complexidade teórica dos algoritmos implementados já foram abordados na parte 1 deste trabalho, pelo que, por uma questão brevidade, não redundância e pensando no leitor, apenas faremos a análise da complexidade temporal empírica a partir dos resultados obtidos. De notar que os tempos não podem ser comparados em valor absoluto entre algoritmos uma vez que diferentes algoritmos são corridos sob diferentes condições (o computador pode variar por exemplo). Os tempos serão medidos em milissegundos podendo mudar a variável independente.

Busca em Profundidade

Este algoritmo não sofreu qualquer alteração em relação ao planeado, por isso, consideramos apenas relevante apresentar os resultados frutos dos testes efetuados.

Em baixo encontra-se o gráfico com os valores obtidos a partir de 10 dos 11 mapas-grafos disponibilizados (não foi corrido em Portugal devido à sua dimensão), num mesmo computador. Os grafos não são, contudo, aleatórios e há um salto grande no número de vértices de Coimbra para Lisboa, fazendo com que os resultados não sejam categóricos.





Como podemos ver, há de facto uma relação mais ou menos linear entre o número de vértices e o tempo (ms) que este algoritmo demora, principalmente ignorando Lisboa, corroborando a análise teórica do último relatório. Existem diversos factores que podem contribuir para a oscilação destes valores, pelo que foi corrido o mesmo algoritmo em cada mapa é feita uma média. No segundo grafo vemos uma discrepância maior, contudo o algoritmo é tanto mais eficiente quanto menor for o número de vértices atingíveis a partir do que definimos como inicial, sendo este factor fruto de maior variação em cidades maiores.

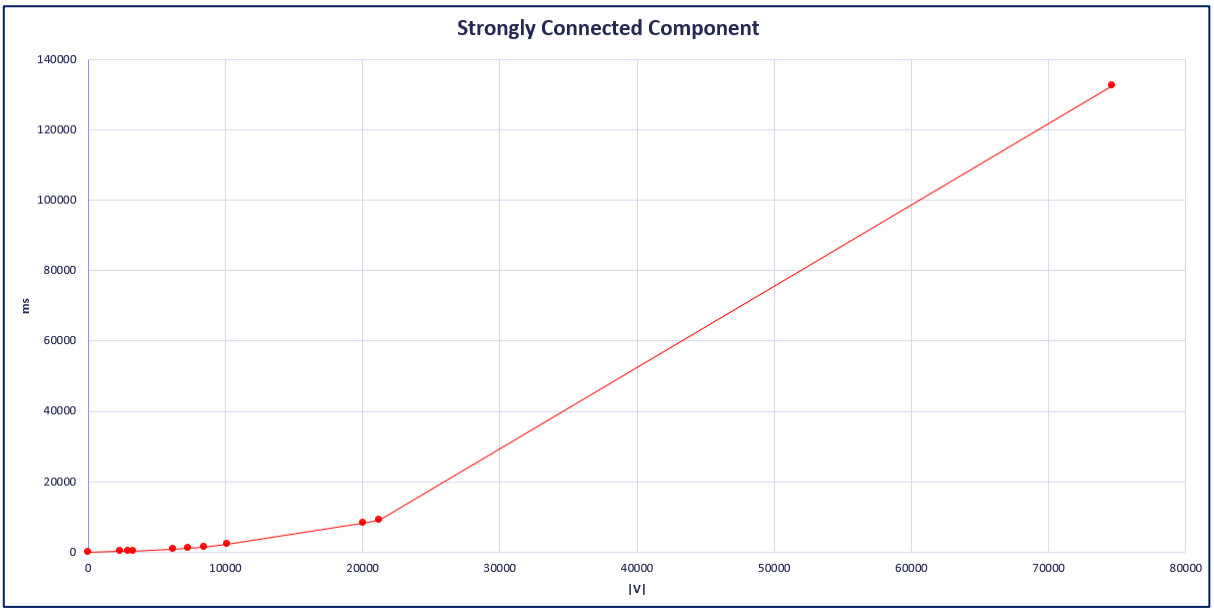
Componentes Fortemente Conexas

Quando não dirigido, este algoritmo não passa de uma simples **DFS** pelo que a sua análise seria redundante.

Porém, em grafos dirigidos a análise do componente fortemente conexa é mais complexa levando à necessidade de uma **DFS** inicial, inversão do grafo (uma vez que o não guardamos antecedentes em cada vértice) e uma segunda **DFS**. Este algoritmo foi apenas descrito no capítulo sobre as Perspectivas de Solução, no pré-processamento, pelo que consideramos relevante apresentar o seu pseudo código.

```
SCC(G, V):
    vector<Vertex> regular_solution = DFS(G, V);
    G_inv = invert(G)
    vector<Vertex> inverted_solution = DFS(G_inv, V)

    vector<Vertex> solution = intersect(regular_solution, inverted_solution);

    return solution;
```

Quanto à complexidade temporal, do algoritmo anterior sabemos que a sua complexidade é **|V|**, pelo que, ao ser feita duas vezes, equivale a **2|V|**. A inversão do grafo é feita simplesmente por percorrer todos os vértices (**|V|**) e suas respectivas arestas(**|E|**) trocando as direções das mesmas. Conclui-se portanto que se trata de uma complexidade de **|V| + |E|**. Por último, a interseção dos dois grafos consiste em percorrer a DFS do grafo normal e comparar com todos os vértices da DFS do grafo invertido. Esta operação pode variar muito a sua complexidade dependendo das soluções encontradas por cada DFS e poder-se-ia ter encontrado uma forma mais eficiente de a fazer para minimizar estes custos. Contudo, os resultados foram bastante satisfatórios e, como os teste eram a maior parte das vezes corridos em versão não dirigida, este acabou por não ser um dos pontos que demos mais importância.



Algoritmos Bidirecionais

Apenas corremos testes no dijkstra e A* nas suas versões bidirecionais, correndo cada uma das respectivas invocações até ao ponto de entrega em threads separados. A diferença que existira do modo não dirigido para o dirigido é apenas entre inverter ou duplicar o grafo (para evitar que dois threads diferentes estejam a alterar os mesmo vértices, já que estes são passados por apontadores), por isso, consideramos irrelevante a comparação de ambos e realizaram-se testes apenas em grafos com arestas bidirecionais (além de que obter componentes fortemente conexas de elevadas dimensões revelou-se um processo complicado como já explicado).

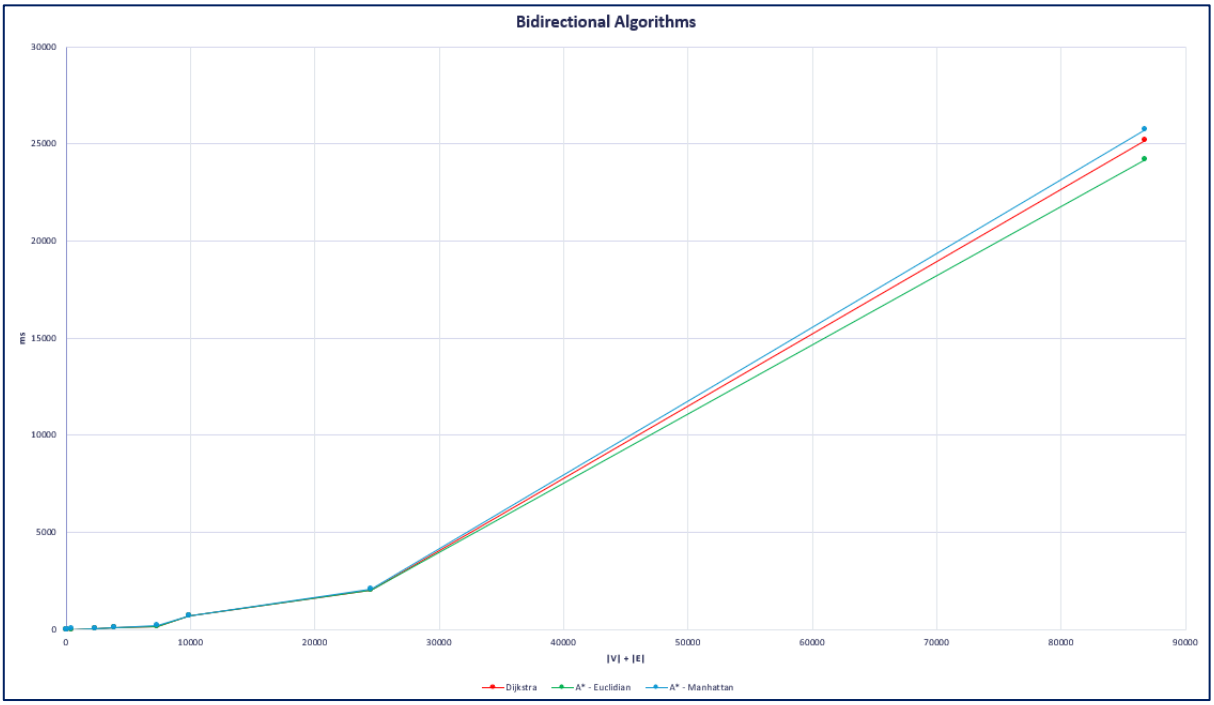
De notar que tanto as análises teóricas como os pseudo códigos já foram apresentados numa entrega anterior, pelo que apresentaremos apenas o resultado empírico da sua complexidade temporal dos diferentes algoritmos.

Optou-se por uma comparação destes 3 algoritmos, porque teoricamente as suas complexidades temporais são iguais apenas variando os tempos pela implementação de heurísticas (distância euclidiana ou de manhattan) que mantêm a pesquisa dirigida no caso do A*. Assim encontram-se a seguir o gráfico com os resultados em função da soma do número de arestas com o número dos vértices.

Como podemos observar, os resultados são bastante semelhantes em grafos pequenos, começando a ser possível notar uma maior diferença nos grafos de maior dimensão.

Esta pouca discrepância entre resultados deve-se não só ao facto de todas as versões encontrarem-se já otimizadas devido ao uso de threads, mas também devido aos caminhos reduzidos e já bastante direcionados que têm de fazer - apenas se corre cada invocação do algoritmo em metade do percurso e os grafos, principalmente das cidades mais pequenas, quase só apresentam um caminho possível quando reduzidos às suas componentes fortemente conexas.

Conseguimos concluir que a heurística da distância de Manhattan (a azul) não é adequada ao mapa das estradas portuguesas (foi pensado com vista a funcionar em cidades modernas) obtendo resultados piores que o Dijkstra (a vermelho). Com a heurística da distância euclidiana (verde) conseguimos obter realmente resultados ligeiramente melhores, sendo mais notórios em grafos com mais densos e com mais vértices.



Algoritmo de Floyd-Warshall

A implementação deste algoritmo começou com a realizada nas aulas práticas, adaptada à classe Grafo utilizada, mais concretamente as características das arestas. Ao testar o seu desempenho com os mapas disponibilizados, rapidamente se tornou claro do quão moroso este algoritmo era. Para uma componente fortemente conexa de 1270 nós, o tempo de execução era de cerca de 2 minutos e 31 segundos, o que, tratando-se de um exemplo relativamente pequeno, é longe de ser ideal.

Com o objetivo de encontrar possíveis otimizações, o código original foi alterado de modo a que:

Os 2 ciclos *for* iniciais de inicialização das matrizes (distâncias e caminhos) apenas percorressem metade das matrizes, isto é, o primeiro ciclo matinha-se de i = [0, n], sendo n o número de vértices do grafo, partindo o segundo de j = [i, n]. Isto porque, no estado inicial, ambas as matrizes são simétricas, pelo que, o valor da célula (i, j) é exatamente igual ao da célula (j, i).

Na verificação dos caminhos mais curtos, ou seja, dentro dos 3 ciclos for, existem valores de i, j e k que não necessitam de processamento, poupando-se tempo ao não ter que aceder aos seus valores na matriz de distâncias. Isto acontece quando i = j, uma vez que se referem ao mesmo vértice e não havendo arestas com peso negativo, a distância na matriz de adjacências será sempre 0 (valor colocado na inicialização), e quando k = j (ou i), dado que, a expressão **dist(i, j) > dist(i, k) + dist(k, j)** torna-se equivalente a **dist(i, j) > dist(i, j) + dist(j, j)**, igual a **dist(i, j) > dist(i, j)**.

Feitas as alterações, obteve-se um melhoramento de cerca de 5 segundos em relação ao

tempo anterior, utilizando exatamente os mesmo dados. Mesmo assim, o tempo de execução é demorado, contudo, em relação a grafos dirigidos, otimizações significativas teriam de incidir nas estruturas de dados usadas para guardar os valores das matrizes.

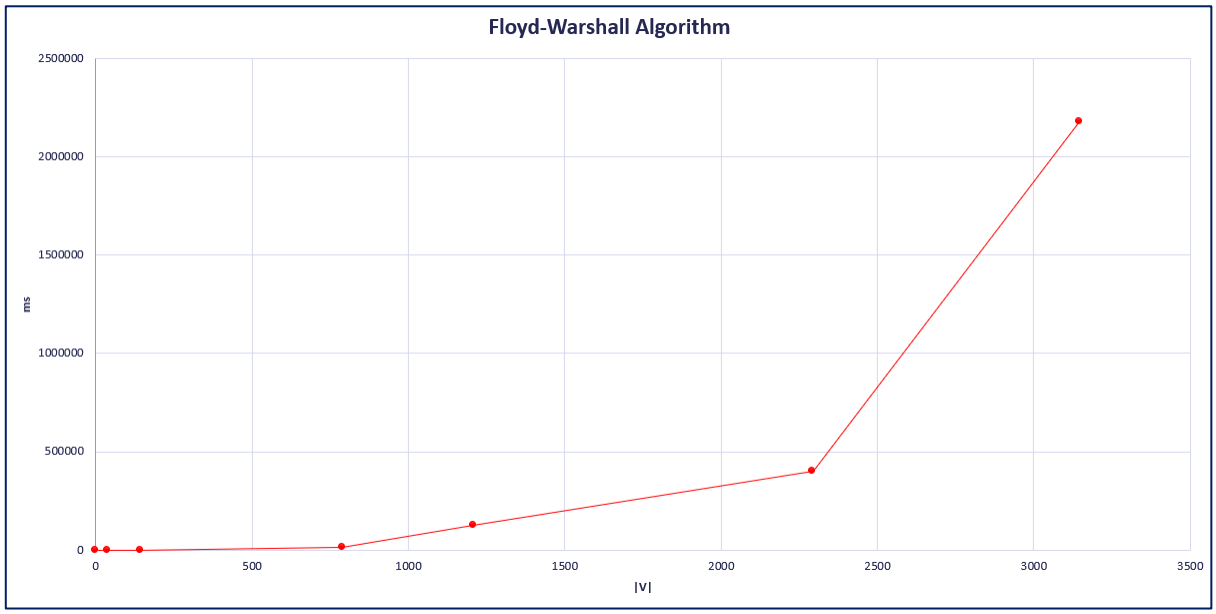
No entanto, há otimizações notáveis a serem feitas no caso em que escolhemos carregar os grafos no modo não dirigido, e, como não se previu esta possibilidade no primeiro relatório, passamos a explicar a adaptação que fizemos ao algoritmo.

Num grafo não dirigido a matriz de distâncias será simétrica, uma vez que, se a distância mais curta entre i e j é de n, ao existir um caminho de i para j, e sendo as arestas bidirecionais, o mesmo caminho pode se usar de j para i, e terão igual distância. Este raciocínio não pode ser aplicado à matriz de caminhos, contudo, se o caminho mais curto entre i e j é igual ao caminho mais curto entre j e i, mas invertido, podemos concluir que, se o valor na matriz de predecessores de i para j é igual ao de i para k, o valor de j para i será igual ao de j para k. Sabendo isto, podemos percorrer apenas metade das matrizes, de uma forma semelhante ao que foi feito na inicialização.

Utilizando exatamente o mesmo grafo dos teste anteriores e assumindo que se trata de um grafo não dirigido, o tempo de execução diminuiu para aproximadamente 1 minuto e 10 segundos, cerca de 46% do tempo original medido, o que constituí uma melhoria significativa. É fácil de perceber o porquê da drástica descida observada com as otimizações. Tomando como exemplo o grafo utilizado para testar o algoritmo, sendo constituído por 1270 nós, as respetivas matrizes terão 1612900 células (1270x1270). Usando a primeira implementação, o algoritmo percorre a totalidade da matriz 2 vezes, sendo que na segunda é necessário realizar operações de leitura em todas as iterações. No entanto, aplicadas as otimizações e assumindo um grafo não dirigido, o algoritmo só realiza 805180 iterações ((1612900 - 2x1270) / 2), cerca de 50% das iterações anteriores.

Quando à analise da complexidade temporal, a nossa implementação percorre 1 vez metade da matriz **$(|V|^2 / 2)$** e de novo comparando aproximadamente metade de todos os pares de vértices **$(|V|^2)$** com todos os possíveis vértices intermédios **$(|V|)$**, resultando numa complexidade temporal de **$O(|V| \times |V|^2 + |V|^2 / 2)$** equivalente a **$O(|V|^3)$**. Equivale à complexidade teórica do algoritmo, mantendo-se também a complexidade espacial.

O gráfico deste pré-processamento isolado (ainda sem correr o vizinho mais próximo) em grafos não dirigidos em milissegundos em função do número de vértices apresenta-se abaixo.



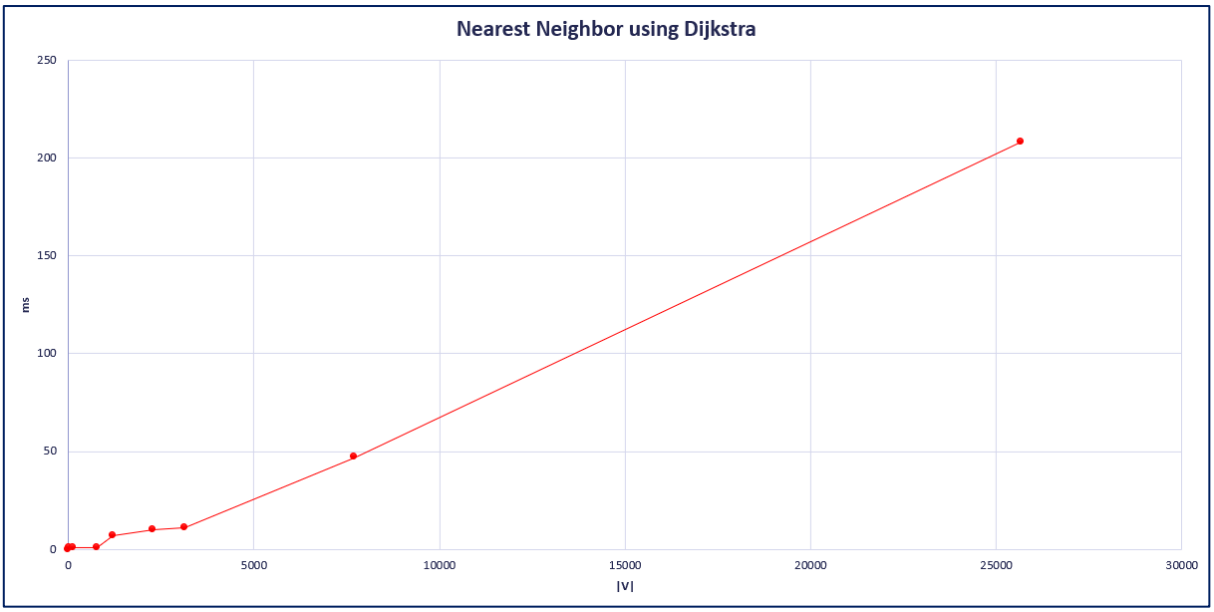
Como podemos ver há um crescimento exponencial com o tamanho do grafo (razão pela qual só se testou este algoritmo para os 6 mapas-grafos mais pequenos).

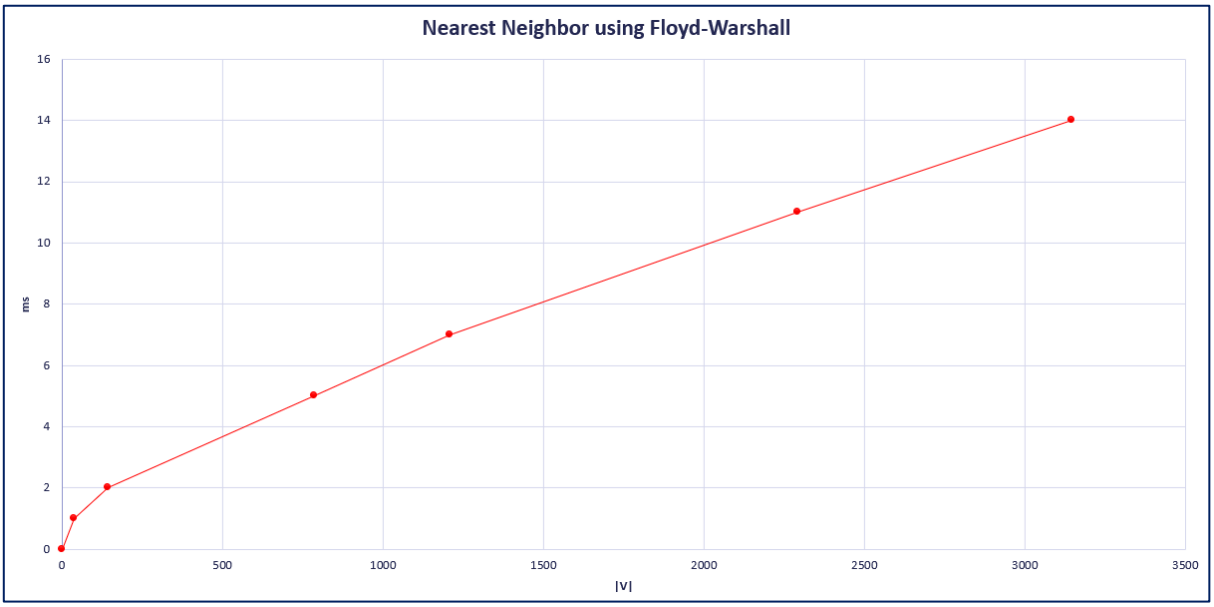
Algoritmo do Vizinho mais Próximo

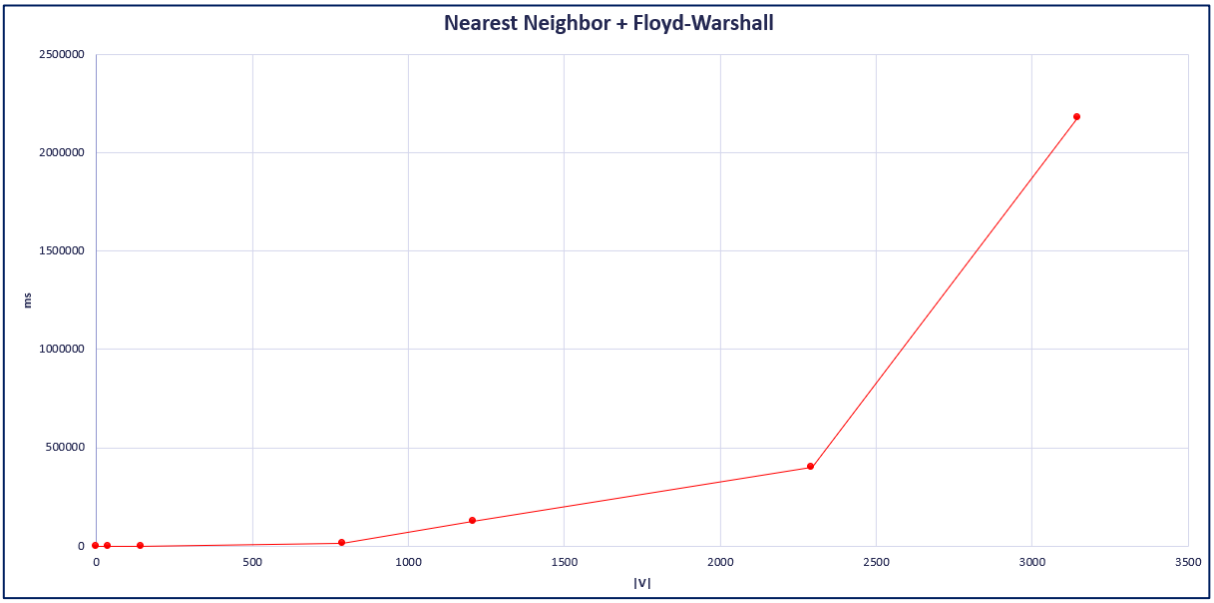
Conseguimos implementar as duas versões que tínhamos proposto e explicado no relatório anterior (já existindo, por isso, tanto análise da complexidade teórica, como pseudo-código).

Com os resultados práticos concluímos que se assumirmos o Floyd-Warshall como pré-processamento, os resultados deste serão melhores (temporalmente) que a versão que se serve do A* (utilizamos este algoritmo devido aos seus melhores resultados face ao Dijkstra), já que esta precisa de calcular os vários caminhos. Contudo, ao avaliarmos o tempo total, facilmente nos apercebemos que se quisermos correr apenas uma vez estes testes a melhor solução acaba por ser a 2ª, visto que o tempo total de Vizinho mais Próximo com Floyd-Warshall é muito superior a esta.

Os gráficos vêm em função do número de vértices dos gráficos e foram feitos com um mesmo número de encomendas (4) aleatoriamente escolhidas na zona fortemente conexa.


Nearest Neighbor using Dijkstra
ms
|V|
0
50
100
150
200
250
0
5000
10000
15000
20000
25000
30000


Nearest Neighbor using Floyd-Warshall
ms
|V|
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000
2500
3000
3500

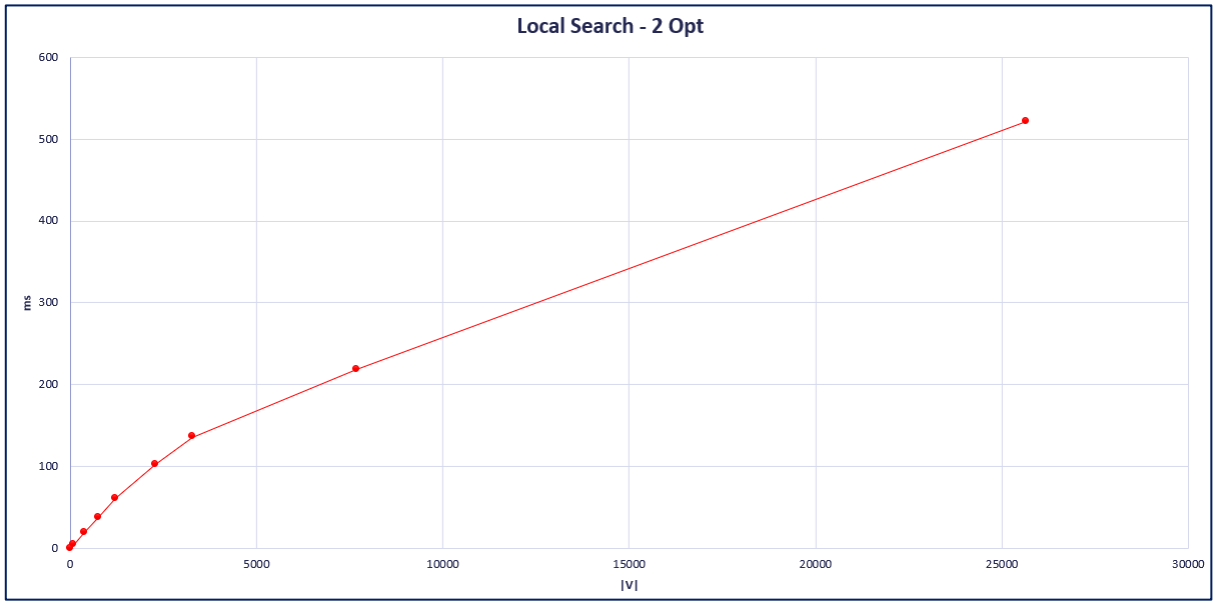

Nearest Neighbor + Floyd-Warshall
ms
|V|
0
500000
1000000
1500000
2000000
2500000
0
500
1000
1500
2000
2500
3000
3500

Nestes dois grafos percebemos que se ignorarmos o processamento do Floyd-Warshall (segundo gráfico são os resultados de apenas o Nearest Neighbor recorrendo às distâncias previamente calculadas), este algoritmo torna-se bastante rápido, pelo que se a central dispuser de um espaço suficiente para armazenar a tabela necessária pode ser uma solução a considerar. Contudo, no terceiro vemos a explosão de tempos se tomarmos em conta o pré-processamento (junto destes, os tempos do NN isolado tornam-se desprezáveis).

Algoritmo de Pesquisa Local – 2 Opt

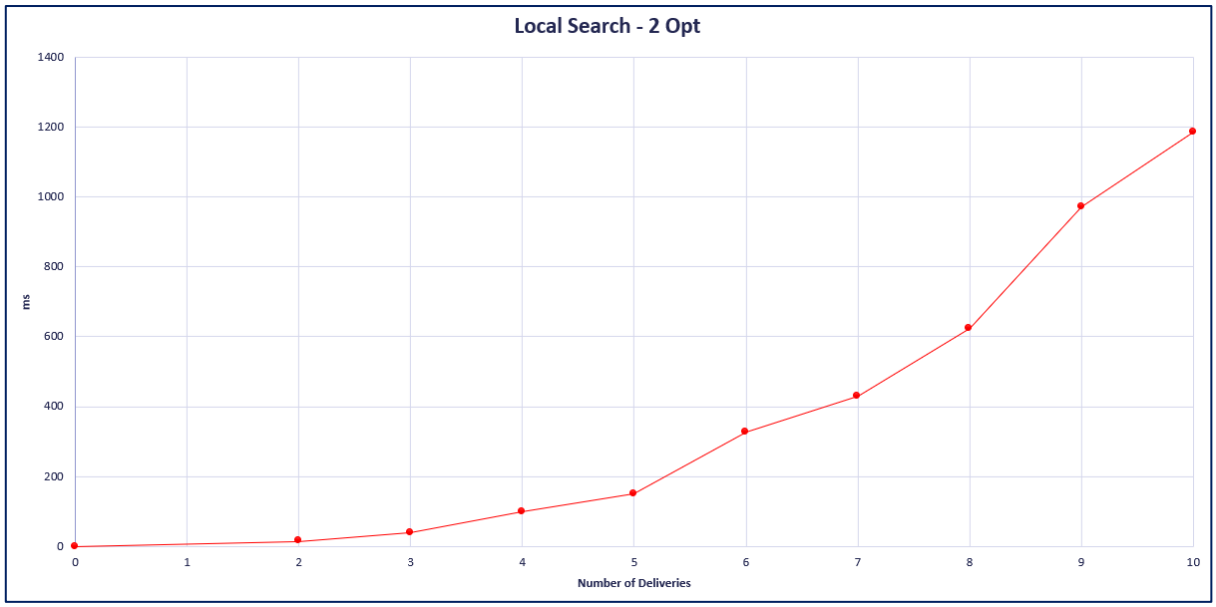
Foi com este algoritmo que conseguimos os melhores resultados nesta segunda versão do problema. Partindo de uma solução criada pela união do vértice inicial às diversas encomendas pela ordem de chegada e terminando na garagem, aplicamos o opt-2 como descrito no pseudocódigo e vamos testando se houve melhoramento do peso do caminho total.

Os resultados foram melhores não só no custo total, mas também, e principalmente, no tempo que demora, como podemos observar pelo gráfico a seguir apresentado.

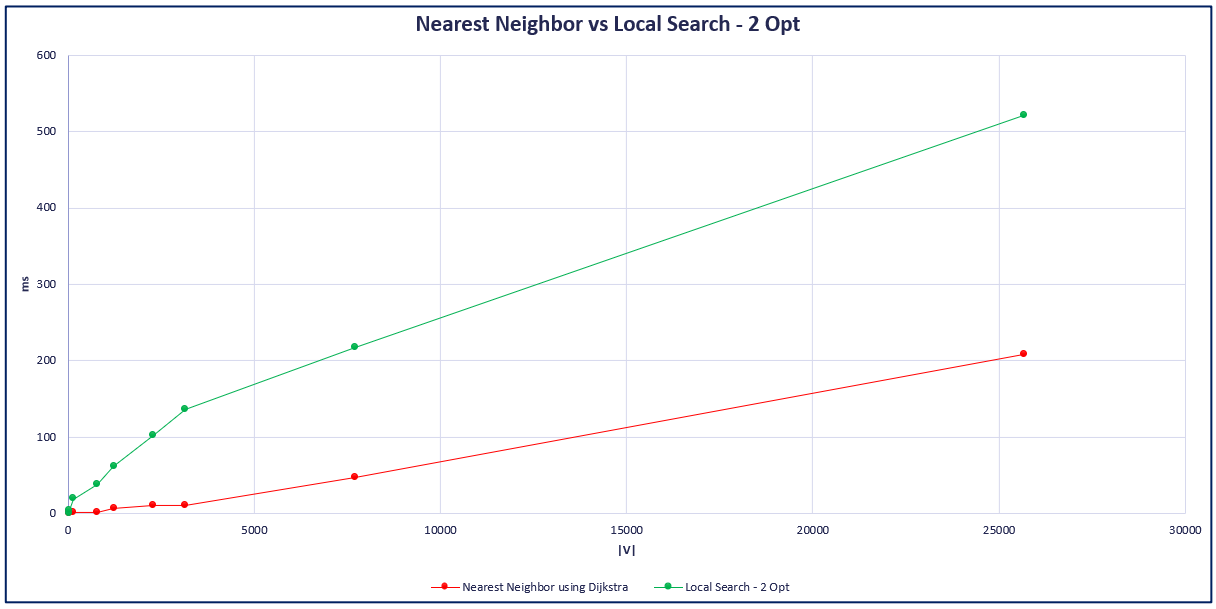


Como podemos observar, este algoritmo é muitíssimo mais rápido do que o NN (nearest neighbor) com FW (floyd-warshall) para um mesmo número de encomendas, num grafo de tamanho muito superior (aproximadamente 25000 vs perto de 3000) além de que encontra uma solução muito mais próxima do ótimo (ou de um ótimo local pelo menos).

No gráfico seguinte conseguimos ver como é que este algoritmo escala com o aumentar do número de entregas no mapa de Maia reduzido a uma zona fortemente conexa de aproximadamente 3000 vértices. Conseguimos ver que mesmo com mais do dobro das entregas, os resultados continuam a ser muito melhores que o vizinho mais próximo com Floyd-Warshall (o tempo de ambos somado).



O NN versão com Dijkstra é, de facto, mais rápido que o Opt-2 (pelo menos em mapas de dimensão mais reduzida e com menos entregas), contudo é utilizada uma aproximação grosseira para definir qual a próxima aresta a visitar - a distância euclidiana entre dois pontos não consecutivos pode não ser a menor distância entre estes, pois o caminho pode ter de dar uma volta grande para lá chegar - fazendo com que as soluções encontradas pelo algoritmo de busca local acabem por ter um caminho com menor custo que é precisamente o objetivo da empresa, uma vez que as perdas temporais não são significativas.



Último Problema

O algoritmo usado para a resolução do último problema consiste em aplicar algoritmo do vizinho mais próximo adaptado para o problema de camiões com capacidade limitada repetidamente, sendo que o utilizador pode escolher o método de cálculo das distâncias, tendo como opções usar a distância euclidiana ou aplicar o algoritmo de Floyd-Warshall.

Como tentativa de melhorar o algoritmo, usamos uma fila de prioridades para organizar os camiões disponíveis por tamanho, de forma a que aquele com maior capacidade seja o primeiro a ser usado no algoritmo.

A adaptação do algoritmo do vizinho mais próximo consiste em: sempre que é calculado o caminho mais curto para uma entrega, esta é adicionada ao camião que está a ser processado e o seu volume é subtraído à capacidade do camião. Caso, após esta subtração, a capacidade do camião fique negativa ou igual a zero, o algoritmo termina, retornando o caminho que o camião terá de fazer para entregar as encomendas que lhe foram atribuídas.

Dada esta adaptação, tudo o que o algoritmo geral tem de fazer é executar o vizinho mais próximo com a capacidade do camião que estiver no topo da fila de prioridades e passar ao próximo camião assim que o anterior estiver cheio, enquanto ainda houver entregas por distribuir.

Como a estratégia para a resolução deste problema passa por aplicação dos algoritmos estudados nos anteriores subcapítulos diversas vezes consideramos que o estudo desta abordagem como algoritmo isolado não era da maior relevância.

Uma otimização que poderíamos e deveríamos ter feito era uma ligeira modificação no vizinho mais próximo, ou seja, se ele não conseguisse pôr a encomenda do vértice seguinte no camião continuaria a percorrer as encomendas por ordem de menor distância para ver se alguma caberia e só em caso negativo é que se dirigiria para uma garagem.

```
//G = (V, E) - Grafo
//src = Vertice inicial, dest= vertice final
//deliveries= entregas com volume e destino, truck=camioes com capacidades
CalcTruckPaths(G, src, dest, deliveries, trucks):

    while(!deliveries.empty()) {
        truckCapacity = trucks.top().getCapacity();

        truckPath = NearestNeighbor(G, src, dest, deliveries, truckCapacity);

        trucks.top().setPath(truckPath);

        deliveries.removeDelivered();

        trucks.pop();
    }
```

Conclusão

Consideramos que, devido ao estudo prévio para a primeira parte e a alguns algoritmos já implementados nas aulas prática, revelou-se mais tranquila do que a primeira, sendo ainda a principal dificuldade conjugar a quantia de trabalhos proposta pelas diferentes disciplinas.

Conseguimos aplicar todos os algoritmos que nos tínhamos proposto, fazendo ainda otimizações em alguns deles. No entanto, algumas tarefas (como a conectividade) acabaram por ser prejudicadas devido aos dados fornecidos, obrigando a tornar o problema menos realista através da inserção de arestas bidirecionais em todas as ruas.

Quanto às conclusões dos algoritmos em si já foram sendo apresentadas ao longo do último caminho, mas conseguimos perceber que de um modo geral nem sempre é o mesmo testes em grafos de laboratório (complexidade teórica) e em grafos reais, já que estes tendem a obedecer a algumas regras que os grafos aleatórios não fazem (importância de alguns vértices em relação por exemplo).

Na primeira variante do problema conseguimos perceber e extrapolar que o A* com distância euclidiana acaba por ser o melhor, não só porque se adapta um pouco melhor à geometria das ruas portuguesas (enquanto o de Manhattam está mais pensado para cidades modernas), como também fornece uma direcionalidade apenas presente parcialmente no Dijkstra devido à ajuda dos grafos por representarem mapas reais. Contudo, os nossos resultados não são conclusivos ao ponto de podermos afirmar sem sombra de dúvidas qual o melhor em todas as situações.

Na segunda variante, caso a empresa de distribuição consiga fazer um pré-processamento de todos os pontos do grafo, o Floyd-Warshall seguido do vizinho mais próximo é uma solução a considerar, mas foi realmente o Opt-2 que encontrou de longe os melhores resultados, pelo que o seu uso seria aconselhado devido a ser o que tem um melhor compromisso entre solução encontrada e tempo que demora.

Notamos na última versão que o algoritmo idealizado previamente, não chega a um resultado com que nos possamos conformar, pelo que exigiria um melhoramento da estratégia de resolução (como o proposta), uma adaptação do Opt-2 para este caso com vista a procurar otimizar as soluções encontradas ou mesmo uma mudança completa da abordagem ao problema.

A distribuição do trabalho foi feita equitativamente até porque não seria possível acabar o trabalho atempadamente se tal não tivesse acontecido.

Notas

Sobre a compilação: é aconselhado a compilação com recurso às flags -lws2_32 e -std=c++11 para o uso da biblioteca dos threads adaptada e descarregada da internet - créditos no ficheiro mingw.thread.h na pasta Utilities. Pode ser necessário alterar a definição da flag _WIN32_WINNT para uma versão mais recente para que o programa corra sem problemas.