Tema 6 - Supermercado ao Domicílio

Relatório Parte 1

Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

Turma 03 Grupo A:
Ricardo Silva - up201607780
Luís Oliveira - up201607946
Henrique Ferreira - up201605003

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

1 de Abril de 2018

Conteúdo

1 Contexto

A unidade curricular de Conceção e Análise de Algoritmos deu-nos a conhecer o mundo da Teoria do Grafos, as suas respetivas problemáticas e as suas diversas variantes consoante o objetivo predefinido.

Para melhorar a compreensão e exercitar a sua aplicação selecionámos o tema com o enunciado que se segue: *Uma rede de supermercados deseja inovar e passa a permitir que os seus clientes realizem as suas compras de supermercado pela Internet, sendo depois as mesmas entregues no domicílio do cliente. Para rentabilizar o serviço, o supermercado tenta agrupar o maior número de compras numa única viagem, distribuindo as compras para os clientes nas suas respetivas moradas. Elabore um sistema que permite ao supermercado avaliar diferentes estratégias de entregas ao domicílio. Os camiões poderão todos partir de um único supermercado, ou poderá haver vários supermercados a manterem o seu próprio veículo de entrega, procurando-se agrupar os clientes mais próximos de uma dada sucursal. Avalie a conectividade do grafo, a fim de evitar que clientes sejam atendidos por supermercados para os quais não haverá itinerário possível. Para além de tentar-se minimizar o itinerário de distribuição das compras, pretende-se também realizar o maior número de entregas numa única viagem, com a condição de terem todas as compras de ser entregues no mesmo dia em que foram realizadas.*

Várias questões são levantadas imediatamente: podemos considerar uma só rota de distribuição por supermercado? Ou será suficiente a aplicação do caminho mais curto entre dois vértices, para obter a solução ótima? É válido agrupar os clientes por áreas de influência de cada supermercado? Ou será, mais válido tirar partido da distância euclidiana, para obter a vantagem geométrica das redes viárias, já que estamos a referirmos a um mapeamento rodoviário?

Com o intuito de avaliar as diferentes estratégias que poderão ser implementadas este relatório debruça-se, numa primeira parte, sobre *Identificação do Problema*, onde se descreve algumas restrições e assunções tendo em vista a simplificação do problema. Segue-se a sua *Formulação*, em que se definem os dados de entrada e de saída, e demonstra-se a sua função objetivo. No quarto capítulo, descrevemos a *Solução Implementada* indicando os algoritmos usados. O quinto capitulo, *Análise Teórica*, incide sobre a análise de alguns algoritmos sobre a ótica da sua complexidade temporal e espacial. Depois, *Análise Empírica*, onde se demonstram os resultados obtidos. O capítulo oito *Manual de utilização* detalha as instruções e o modo de funcionamento do programa. Por fim, surgem as dificuldades, melhorias, conclusões e dic≪visão de tarefas, capítulos nove, dez, onze e doze respetivamente.

2 Identificação do problema

Na ótica de simplificação, decidimos partir do cenário mais elementar, ou seja, o cenário de existência de um supermercado e uma só rota. Considera-se uma rota o trajeto percorrido por um camião de entrega ao domicílio. Por sua vez, um supermercado com duas rotas, por exemplo, deve entender-se como um supermercado com dois camiões de distribuição. Para cada supermercado já

existe um conjunto de clientes definidos.

Descartou-se a capacidade do veículo e apenas debruçamos o trabalho prático sobre a minimização de distância percorrida entre a localização do(s) cliente(s) e a localização do(s) supermercado(s).

A minimização da distância, vulgo caminho mais curto, e do tempo de processamento são os objetivos primordiais, pois, do ponto de vista de um supermercado, a preocupação essencial é minimizar a distância percorrida, para assim, minimizar os seus custos de distribuição. O peso designado nas arestas dos futuros grafos é a distância entre os dois pontos no mapa.

Em suma, estamos perante um problemas de encaminhamento(routing), isto é, encontrar o melhor percurso numa rede viária, passando obrigatoriamente por pontos predefinidos, clientes no nosso caso, com retorno ao ponto de partida.

3 Formalização do problema

Os algoritmos implementados irão utilizar um conjunto de dados, traduzidos em vértices e arestas de uma área pré-definida, extraídos do *OpenStreetMaps* e parsados para ficheiros de edges, nodes e roads pela aplicação disponibilizada no moodle da unidade curricular. Por consequência, à partida, o grafo será um grafo não dirigido, com pesos positivos em cada aresta, fortemente conexo, denso e não completo, pois não existirão arestas de cada nó para todos os restantes.

A fim de construir a base do grafo, teremos de organizar a informação em termos de nós e arestas. Neste caso, os vértices do grafo serão, os cruzamentos entre cada rua, com as respetivas arestas. Estas constituem-se por um conjunto de informações sobre os nós de destino e os seus pesos. Deste modo, formalmente:

- G representa o grafo, a abstração do mapa de estradas;
- V representa o conjunto de vértices do mapa, especificamente, todos os possíveis cruzamentos;
- v_i representa um dado vértice;
- E representa o conjunto de arestas associado a cada vértice, ou seja, as estradas entre cada cruzamento no mapa;
- $E_{i,j}$ representa a aresta, isto é a estrada, que vai de v_i até v_j;
- x_i *e* y_i representam as coordenadas x e y no vértice i;
- $path_{v_i}$ representa o vértice que precede o vértice v_i , mais tarde, o conjunto destes será o caminho ótimo entre os dois vértices selecionados;
- $A_{i,j}$ é V que determina o caminho entre v_i e v_j;
- $dist_{v_i,j}$ define o peso de cada $E_{i,j}$, no nosso caso particular a distância do vértice v_i até a v_j;

Especificamente, a nossa função objetivo será a conjugação dos melhores resultados entre (1) e (2):

$$F = min \sum_{k=v_i}^{n} dist_{v_i,v_j} \tag{1}$$

$$F = min \sum_{k=v_i}^{n} O_{temporal} \tag{2}$$

3.1 Dados de entrada

Os ficheiros obtidos a partir do OpenStreetMaps, **nodes.txt**, **roads.txt** e **edges.txt** seguidos de indices de 1 a 6 variam nas suas quantidades de dados, para futura análise empírica, correspondendo a versões com cerca de, 650, 1.300, 2.500, 4.000, 57.000 e 146.000 nós, respetivamente,

Além destes, temos ainda outros três com dados por omissão para representar e testar supermercados, camiões(rotas) e clientes. Todos funcionam como fonte de informação para construir o grafo e para a aplicação calcular os percursos.

Segue-se o modo como cada informação é tratada no implementação:

- **nodes.txt**: Ficheiro de data de V com cinco colunas: número de identificação (*id*), longitude (coluna 2 e 3) e latitude (coluna 4 e 5). Por cada linha, existirá um v_i no grafo, formado pelo *id*, *(*x_i*,* y_i *)*, e um *E*.

- **roads.txt**: Ficheiro com três colunas: número de identificação, nome da rua, valor booleano. Por cada *id*, corresponderá a um $E_{i,j}$ que verifica se a aresta é de dois sentidos. Se for, cria um novo $E_{i,j}$ invertido e adiciona-o ao v_i respetivo.

- **edges.txt**: Ficheiro das arestas com três colunas, o número de identificação e o número de identificação do vértice em cada extremo. Por cada linha, será construído um $E_{i,j}$, que, por sua vez, será adicionado ao E do v_i com o peso calculado, e portanto, chegamos ao valor de $dist_{v_i,j}$.

- **clients.txt**: Ficheiro dos clientes fictícios com três colunas: número de identificação incrementado, nome do cliente, número do vértice identificativo da morada do cliente, nome da rua, número de associação ao respetivo supermercado. Esta será a lista de v_i do grafo por onde obrigatoriamente o caminho final, ou seja, o conjunto de $path_{v_i}$, irá de passar.

- **supermarkets.txt**: Ficheiro com uma lista de supermercados da área com três colunas: número de identificação, nome, e número de identificação do vértice. Por cada linha, será identificado o v_i de cada supermercado. Por outras palavras, dependendo do supermercado escolhido da lista predefinida, o supermercado escolhido será o nosso v_i de origem e o v_j de chegada, porque estamos a considerar rotas com ida e volta. Logo, no conjunto de $path_{v_i,v_j}$, $path_{v_i}$ e $path_{v_j}$ terão exatamente os mesmos números de identificação.

- **trucks.txt**: Ficheiro com uma lista de camiões com quatro colunas: número de identificação, nome, capacidade do camião e número de supermercado.

3.2 Dados de saída

Modo Consola: Depois de selecionado o algoritmo, a consola irá apresentar, para cada rota, por ordem de percurso, os números de identificação dos vértices, se cada vértice por onde a rota passa corresponder a um cliente ou supermercado será apresentado como tal e numa linha própria. Se tiver sido escolhida a comparação em vez de um algoritmo, a consola mostrará os dados comparativos de nós, distancia e tempo de execução.

```
number of nodes in path:27


rota 4/4 do supermercado 4/11

# Super 4 #
474685704->474685708->474546985->474546987->
->126617427->
 # Ruby Skinner #
474683167->475081801->2165604353->475082799->
->474676240->343646851->3162256074->474559375->122452429->
->474802735->2585545686->1216466398->474802741->
 # Yoko Cote #

# Super 4 #

# Super 4 #

total=814.946


> Consecutive Dijkstra's Algorithm execution time (ms): 12

total de nós rotas =202

total das rotas =3198 m
```

Figura 1: da rota predefinida 4, obtida pelo Dijkstra de Dijkstras, com indicação do número de nós da rota bem como dos totais

Modo Gráfico: Depois de selecionado o algoritmo, será aberta uma janela gráfica, vulgo graphViewer, com o mapa escolhido e uma rota de ida e volta nele desenhado, estando os supermercados identificados por pontos verdes, os clientes por pontos vermelhos e os restantes vértices por pontos azuis e setas a preto.

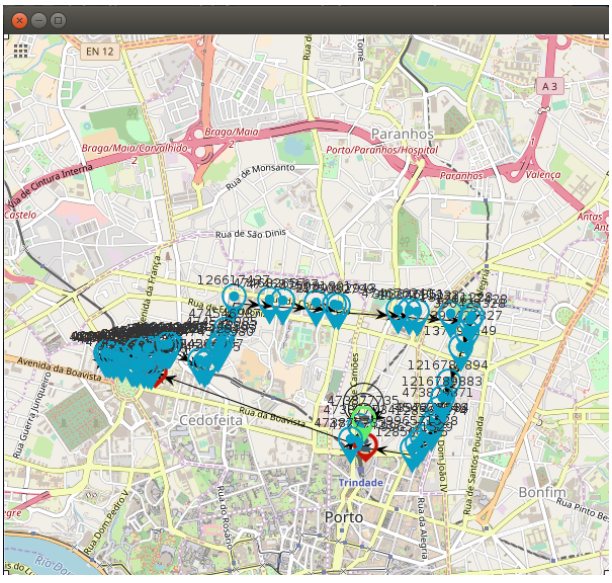

Figura 2: graphViewer da rota predefinida 1, obtida pelo Dijkstra de Dijkstras

4 Solução Implementada

Para a análise dos quatro cenários foram implementados três algoritmos, seguindo-se uma explicação mais exaustiva de cada um, com foco no modo de funcionamento.

4.1 Algoritmo Dijkstra Of Dijkstra

É o Algoritmo Dijkstra aplicado a cada etapa que constitui uma rota. Chamamos-lhe Dijkstra Of Dijkstras, pois a cada iteração calcula qual o cliente ainda não visitado (ou o supermercado, na última etapa) mais perto donde se está, obtém o caminho e passa à iteração seguinte até estarem todos os pontos visitados.

Uma **estratégia gananciosa**, que não garante o caminho ótimo, passando por todos os clientes até ao mais longínquo e que depois volta ao ponto de partida. Chega ao fim quando acabarem todas as paragens e volta novamente ao supermercado. Tem esse senão de fazer uma longa viagem de regresso. Tentamos minimizar esse problema com o método seguinte.

4.2 Algoritmo Bidirecional

Ir pelo caminho mais curto a partir do supermercado, segundo uma **estratégia gananciosa bidirecional**,voltando ao mesmo com união no ponto médio que acontecerá a metade do percurso.

A implementação procura, a cada iteração, qual o ponto mais próximo ainda não visitado, segundo duas direções e em ambas usando o algoritmo caminho mais curto de Dijkstra. O ponto inicial é o supermercado, para os dois semi-percursos que se vai tratar.

Nas iterações ímpares procura no percurso de ida qual o cliente seguinte, ainda não visitado, mais próximo indo desde o ponto de ida atual e insere numa fila.

Nas iterações pares procura no sentido de volta, por grafo inverso, qual o cliente, ainda não visitado, mais perto desde o ponto de regresso atual e insere numa pilha. Quando só houver um cliente não visitado, deixa de diferenciar entre iteração par ou ímpar, calcula uma etapa de ida até esse cliente em falta e outra desse até ao último cliente da volta, acrescenta ambas as etapas à fila e manda os conteúdos da pilha para a fila. Temos assim a rota desejada e com isto poupamos no tamanho do percurso.

4.3 Algoritmo de Floyd-Warshall

Neste método, utilizamos o algoritmo de Dijkstra para calcular as etapas e depois obtém-se o caminho dentro de cada etapa através do algoritmo de Floyd-Warshall.

Como se sabe, este algoritmo utiliza uma **estratégia de programação dinâmica** em que guarda os pesos do vértice A para o vértice B e da viagem inversa numa matriz de adjacências.

5 Análise Teórica

5.1 Algoritmo Dijkstra Of Dijkstra

Para fazer uma análise analítica temos que analisar primeiro o seu pseudocódigo abaixo ilustrado:

```
DoD (dijkstraOfDijkstras)
recebe inicio e vector de paragens
tem acesso a grafo(G<V,E>)

stack dijkstraLeg
vector leg
vector pathOrder
tmp ← 0
source ← inicio
next ← source
min, k = 0

Insert (pathOrder, source)
      |
while tamanho(paragens) do
      min ← ∞
      k ← 0
      DijkstraShortestPath(G,source)

      for each stop ∈ paragens do
            if [ tmp ← dist(source até stop) ] < min then
                  min ← tmp
                  next ← stop
                  k ← indice da stop
      leg ← getPath (source, next)
      Insert (pathOrder, leg)

      if tamanho(paragens) > 1 then
            source ← next
            Remove (paragens, k)
      else
            DijkstraShortestPath(G,paragens(0))
            leg ← getPath(paragens(0), inicio)
            Insert (pathOrder, leg)
            Remove (paragens, 0)

Return (pathOrder)

A limitação de area entre stops para agilizar Dijkstra teria de ser implementada
no dijkstraShortestPath
```

Figura 3: Dijkstra Of Dijkstra Pseudocode

O algoritmo de Dijkstra é conhecido por apresentar uma análise de complexidade temporal de $O\ (|V|^2)$ e de complexidade espacial de $O(|E|+|V|log(|V|))$. Assim, no nosso caso específico, podemos esperar uma complexidade temporal de $O\ (|V|^2)$, já que o fator de multiplicação de cada etapa (no ttoal dando Dijkstra x $2^{clientes-1}-1)não interfere$.

5.2 Algoritmo Bidirecional

A complexidade temporal pode ser expressa em $O(|V|+|E|)$, pois no pior cenário cada vértice e cada aresta será explorado. A complexidade espacial será de $O(|V|)$.

Para o Bidirecional temos o seguinte pseudo código:

```
Bidireccional (getOurPath)
recebe inicio e vector de paragens
tem acesso a grafo(G<V,E>) e a grafo invertido (IG<V,E>)

stack returnstack
vector pathOrder
vector leg
min,k = 0

source ← inicio
next,nextReturn, sourceReturn ← inicio
Insert (pathOrder, source)
iter ← 0
tmp ← 0
while tamanho(paragens) > 1 do
      k ← 0
      min ← ∞
      increase iter
      next = stops.at(0)

      if iter impar then DijkstraShortestPath(G,source)
      else DijkstraShortestPath(IG,sourceReturn)

      for each stop ∈ paragens do

            if [ tmp ← dist(source até stop) ] < min then
                  min ← tmp
                  k ← indice da stop

                  if iter impar then next ← stop
                  else  sourceReturn ← stop

      if iter impar then
            leg ← getPath (source, next)
            Insert (pathOrder, leg)
            source ← next
      else
            leg ← getPath (sourceReturn, nextReturn)

            for each node ∈ leg do
                  Insert (returnStack, node)

            sourceReturn ← nextReturn

      Remove (paragens, k)

DijkstraShortestPath(G,source)
leg ← getPath (source, paragens(0) )
Insert (pathOrder, leg)

DijkstraShortestPath(G, paragens(0))
leg ← getPath (paragens(0), inicio)
Insert (pathOrder, leg)

Delete (paragens)

Return (pathOrder)
```

Figura 4: Bidireccional Pseudocode

5.3 Algoritmo de Floyd-Warshall

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
let next be a |V| × |V| array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction ()
   for each edge (u,v)
      dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
      next[u][v] ← v
   for k from 1 to |V| // standard Floyd-Warshall implementation
      for i from 1 to |V|
         for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← next[i][k]

procedure Path(u, v)
   if next[u][v] = null then
       return []
   path = [u]
   while u ≠ v
       u ← next[u][v]
       path.append(u)
   return path
```

Figura 5: Floyd-Warshall Pseudocode (taken from wikipedia)

Usamos o código disponibilizado na resolução da ficha prática 5. À complexidade temporal do algoritmo em si $O(|V|)^3$ temos de somar o pré-processamento de cada etapa que corresponde à complexidade do Dijkstra ($O\ (|V|^2)$).

6 Análise Empírica

A análise empírica será feita em conjunção com o algoritmo Floyd- Warshall, implementado na aplicação, com o intuito de existir uma maior base comparativa e de análise.

O tempo significa o tempo de execução entre o início do algoritmo até à obtenção do resultado. A distância total percorrida é a distância em metros do caminho encontrado das quatros rotas, dos quatro supermercados, o nosso caso base. Cada rota tem um conjunto específico de clientes.Já o número de nós é o somatório dos nós que constituem o caminho ótimo encontrado.

Obtivemos a seguinte tabela de resultados:

Quantidade de dados		Tempo de execução (ms)			Tamanho dos percursos (nos)		
V	**E**	**DoD**	**bi**	**fw**	**DoD**	**bi**	**fw**
649	696	10	60	21423	198	118	166
1329	1450	42	245	195700	282	202	238
2543	2773	99	849	1300683	376	225	261
3980	4360	217	2095	4862145	400	273	310
57480	62110	11413	486843		404	273	
145980	155840	31022	3482214		404	273	

Figura 6: Total de tempo por número de Vértices

O dado mais interessante que nos é apresentado é o facto de o algoritmo Floyd-Warshall, demorar tanto tempo que nem sequer consegue ser usado para comparar com os outros dois algoritmos implementados desde o cenário mais básico com 649 vértices e 696 arestas. Um dado esperado e comprovado pela literatura de grafos. Nos mapas maiores nem sequer o tentamos correr.

Por outro lado, o o algoritmo Bidirecional aumenta a sua duração proporcionalmente à quantidade de dados envolvida, de acordo com a complexidade teórica, como podemos confirmar nas Figuras 6 e 7.

Provamos que o algoritmo Bidirecional é aquele que apresenta o caminho mais curto pelo numero de nós percorridos, conforme se pode comprovar na tabela e no gráfica de barras.Enquanto que o algoritmo Dijkstra of Dijkstra e o Floyd-Warshall têm o mesmo tamanho, ganhando o Dijkstra no seu tempo de execução que é o menor dos três.

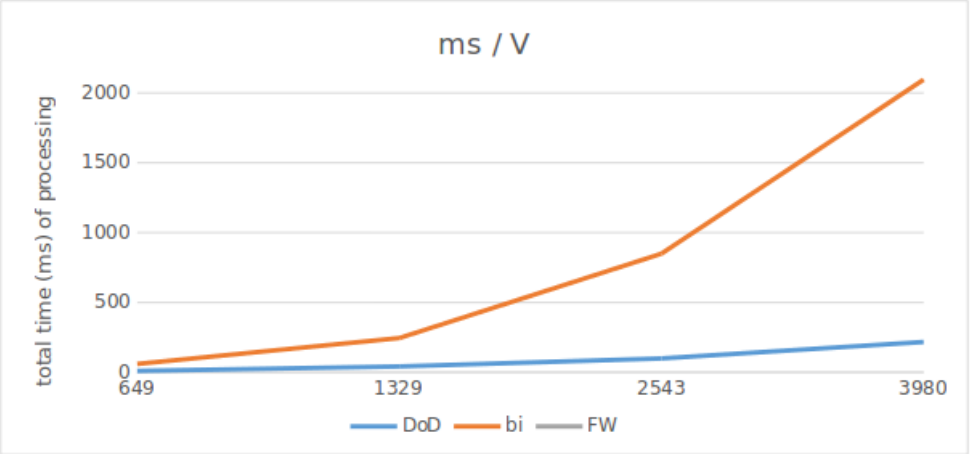

Figura 7: comparativo de tempo de execução (Floyd-Warshall bem acima dos 2000ms)

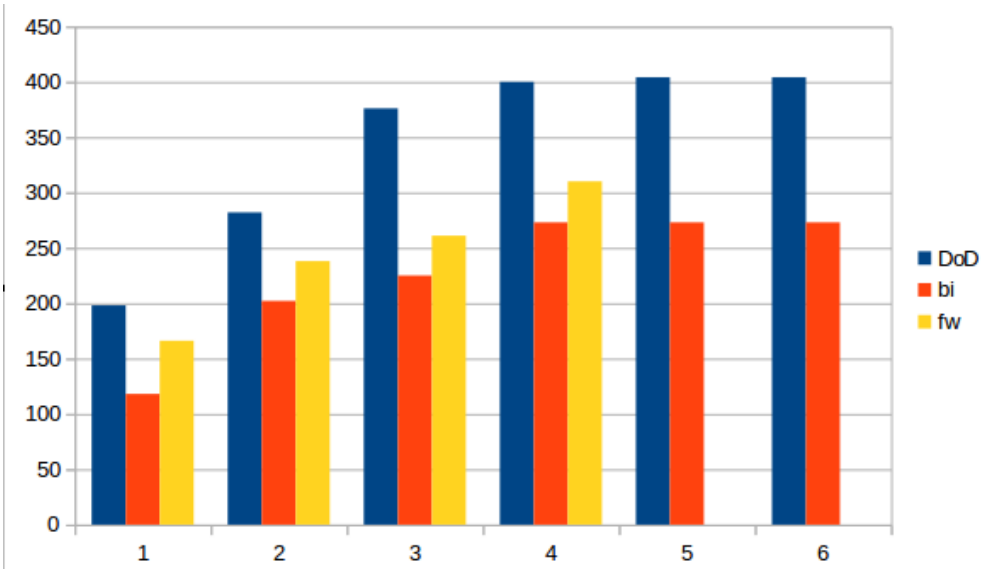

Figura 8: gráfico comparativo do número de nós

7 Estruturas de classes

- *Menu.h* Classe onde se encontra a implementação dos principais algoritmos e onde estão guardadas todas as sequências de dados. Contem os métodos relacionados com os algoritmos: DoD e DijkstraOfDijkstras, Bi, fw e getFW. (getOurRoute, relacionado com o bi está no Graph.h)
- *File.h* Classe que contém todas as funções de leitura de ficheiros txt, constrói os objetos das diferentes classes e guarda a informação na classe Menu.
- *Client.h* Classe com as informações dos clientes: número de identificação atribuído, nome, morada e número de supermercado mais perto do cliente e a distância.
- *Node.h* Classe para o cálculo dos vértices a partir da informação extraída dos ficheiros.
- *Road.h* Classe utilizada na leitura de ficheiros, e classifica cada estrada como um sentido ou dois sentidos.
- *RoadConnection,h* A classe representativa das arestas do nosso grafo, com um número identificativo, e os números de identificação dos vértices nos seus extremos.
- *Supermarket.h* Classe dos supermercados: número de identificação atribuído, nome, e o número do vértice onde está localizado.
- *Truck.h* Classe com número de identificação, nome da rota e supermercado associado. Contém o vetor dos seus clientes.
- *graphViewer.h*, *connection.h*, *edgetype.h*, *Graph.h* são ficheiros fornecidos na aula prática.

Por último temos o ficheiro *Utilities.h* com as funções de cálculo do tempo para efetuar as medições em cada cenário e método bem como auxiliares para a análise do grafo.

8 Manual de utilização

A primeira interação com o utilizador acontece na escolha do ficheiros a ser analisados pela aplicação.

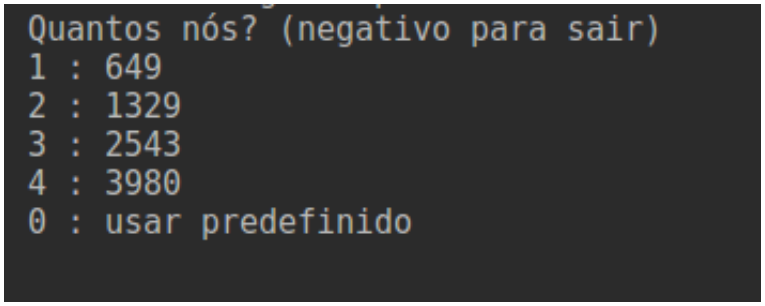

Figura 9: Menu inicial

As opções de menu de 1 a 4 correspondem ao número da nós que se deseja analisar. O maior número de nós é a opção 6 (na imagem era a 4) e o menor número a opção 1. A opção zero é o *default* apenas usada para efeitos de testes e de *debug*.

Após a seleção, aparecerá a informação da confirmação dos ficheiros lidos e as respetivas quantidades de dados lidas. Neste ponto, basta primir **Enter** para continuar.

A seguir especifica-se quantos clientes queremos usar para calcular rota:

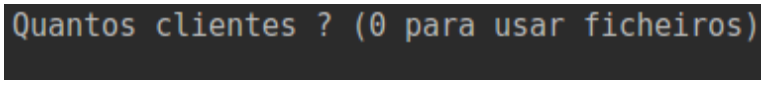

Figura 10: Interface número de clientes

Prima 0 para considerar todos os clientes predefinidos, ou seja, 22 clientes.

A segunda questão será o número de supermercados a considerar. Inserir zero para considerar supermercados guardados por omissão:

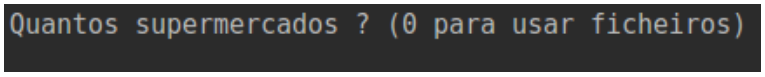

Figura 11: Interface supermercados

Se escolher quantos supermercados quer ser-lhe-á também perguntado quantas rotas quer por supermercado (camiões).

A penúltima escolha será a visualização de resultados na consola ou no graph-viewer.

Finalmente, na figura em cima a triagem do algoritmo desejado para proceder aos cálculos.

Se seleccionar Compare, obtem-se, na consola, uma análise comparativa pelo total das rotas calculadas em termos de nós e de tempo de execução. Se nesta sessão já tiverem sido corridos alguns dos métodos, estes não serão novemente

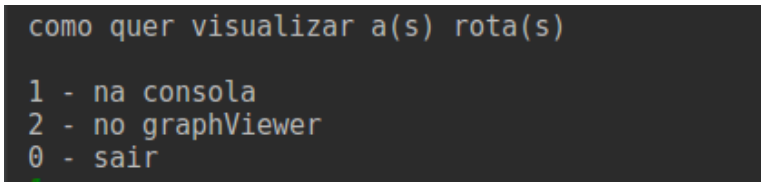

Figura 12: Interface supermercados

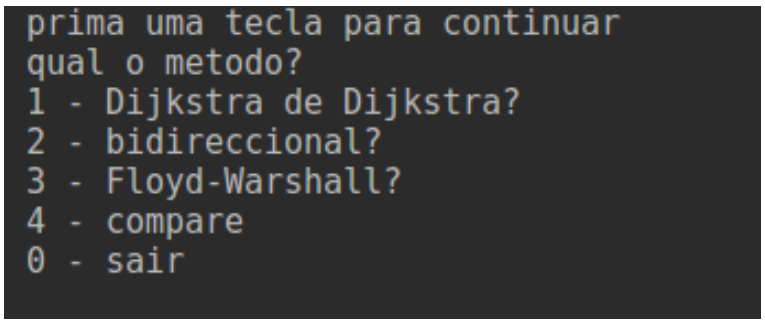

Figura 13: Algoritmos Implementados

corridos pois da primeira vez os dados ficaram guardados numa estrutura implementada para o efeito.

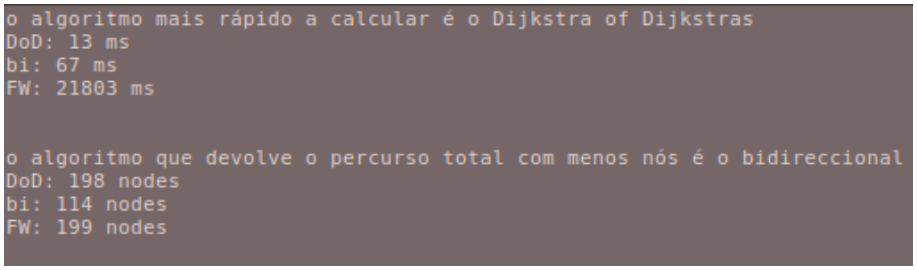

Figura 14: saída Compare para o mapa mais pequeno

Os resultados são extensos, pois conseguimos ver cada rota de camião por cada supermercado, os clientes visitados, e oferece-nos ao mesmo tempo a possibilidade de saber os seus tempos de execução, e também o número de nós que constituem o caminho ótimo em cada rota.

Ao escolher um resultado gráfico será aberta uma janela com a rota desenhado sobre o mapa escolhido.

```
number of nodes in path:26


rota 4/4 do supermercado 4/11

# Super 4 #
474685704->474685708->474546985->474546987->
->126617427->
 # Ruby Skinner #
474683167->475081801->2165604353->475082799->
->474676240->343646851->3162256074->474559375->122452429->
->474802735->2585545686->1216466398->474802741->
 # Yoko Cote #

# Super 4 #

# Super 4 #


> Consecutive Dijkstra's Algorithm execution time (ms): 14

total de nós rotas =198
```

Figura 15: saída com a rota 4 do mapa mais pequeno (com informação do total das 4 rotas

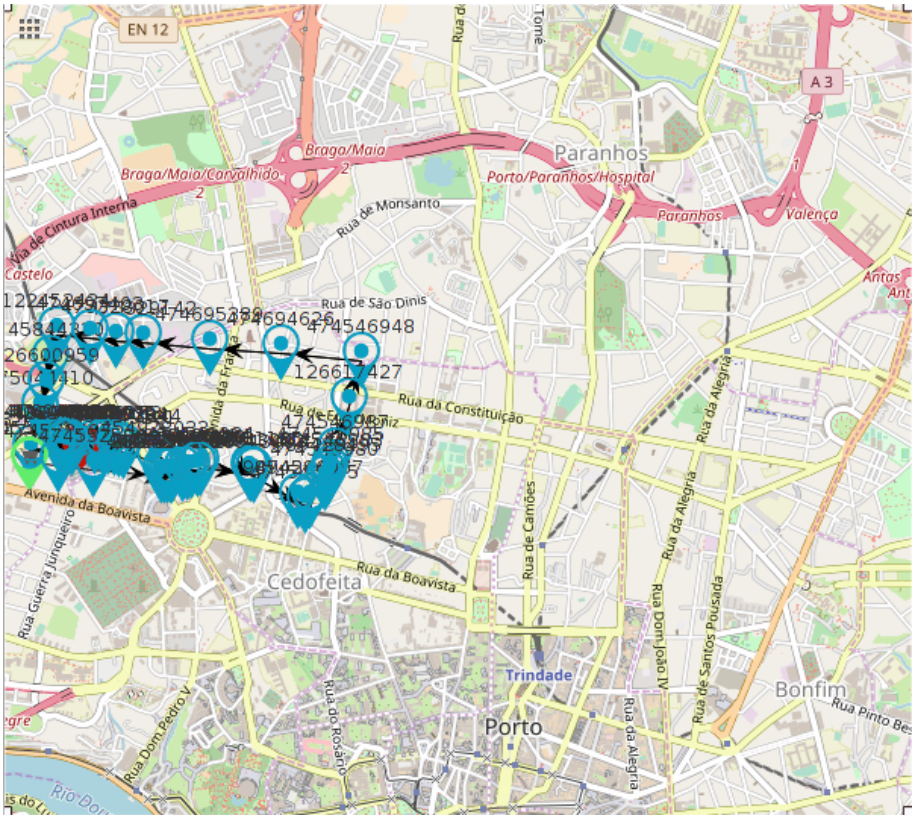

Figura 16: rota 2 do mapa mais pequeno calculada pelo algoritmo bidireccional

9 Dificuldades

A principal dificuldade encontrada, numa primeira fase, foi restringir o problema proposto em consonância com os objetivos pedidos e, assim, criar a base de trabalho.

Numa fase posterior encontramos uma certa dificuldade em trabalhar com os ficheiros de código fornecidos para a implementação, nomeadamente o *graphviewer.h* e o *graph.h*.

Achamos que as aulas teóricas e práticas de exposição do algoritmo de Dijsktra deveriam ser lecionadas um pouco mais cedo para nos dar um maior intervalo de consolidação da matéria.

10 Melhorias

Poderíamos ter melhorado a eficiência e reduzido o tempo de processamento de todos estes métodos se tivéssemos introduzido uma alteração no código do DijkstraShortestPath. Se o colocássemos a fazer o preenchimento de distancias não para todos os vértices do grafo mas por círculos concêntricos em cada origem com raios crescentes até encontrar um cliente dentro desses raios. Infelizmente não tivemos tempo para o implementar.

11 Conclusão

Através do código fornecido nas aulas práticas procuramos provar que tipo de grafo obtivemos. Ao contrário do que foi suposto no capítulo *Formalização do problema*, provou-se que o grafo arquitetado é na realidade um grafo esparso. No entanto, continua a ser um grafo não dirigido, não completo e fortemente conexo. Apenas os mapas maiores nos são apresentados como correspondendo a grafos densos, quando esperavamos o contrário

A análise empírica confirma as complexidades esperadas para os algoritmos, mesmo havendo um Dikstra de Dikstras, que conforme já explicado não aumenta a complexidade a $O(|V|^2)$, o mesmo que o algoritmo original de Dijsktra.

Em termos de resultados, o Dijsktra of Dijsktra é o mais rápido dos três testados, mas não dá o caminho mais curto, pois vai até ao cliente mais longe e calcula o caminho mais curto do último cliente até ao supermercado, o que por vezes não é a rota ótima. Por outras palavras, o algoritmo serve primeiro os seus clientes e só depois de eles estarem satisfeitos é que se preocupa em voltar ao supermercado. O Bidirecional devolve-nos, sem margem para dúvidas, o caminho mais curto, mas é mais consumidor de tempo. Para o Floyd-Warshall obtivemos numeros de nós parecidos com o Dijkstra. No entanto provou-se que demora muito tempo, conforme a complexidade teórica indica e por esse facto não se testou para mapas grandes.

12 Distribuição de Tarefas

A definição, implementação e distribuição das tarefas foi igualmente distribuída entre os três elementos que constituem o grupo de trabalho.

[FIM]