Ridesharing - Parte 2

Concepção e Análise de Algoritmos
Turma 1, Grupo D

David Luís Dias da Silva	up201705373@fe.up.pt
Joana Catarina Teixeira Ferreira	up201705722@fe.up.pt
Luís Pedro Pereira Lopes Mascarenhas Cunha	up201706736@fe.up.pt

Índice

1. Descrição do problema

Muitas pessoas têm como primeira opção para as suas viagens diárias o carro, cuja utilização abusiva provoca danos ao meio ambiente e leva ao esgotamento de recursos não renováveis. Uma maneira de melhorar esta situação, bem como diminuir o tráfego nas estradas, é adotar o *ridesharing*, uma prática em que pessoas que residem em zonas vizinhas e têm destinos em comum partilham o mesmo carro, com o objetivo de minimizar o número de carros em circulação utilizando apenas uma fração mínima da sua capacidade.

Desta forma, reduz-se o número de carros em circulação e o custo da viagem pode ser repartido pelos ocupantes, diminuindo o custo da viagem para quem disponibiliza o automóvel e fornecendo um meio de transporte confortável para os passageiros.

No âmbito da unidade curricular Concepção e Análise de Algoritmos, foi-nos proposto desenvolver uma aplicação que promova a adesão a esta prática, permitindo a condutores partilhar o seu carro, bem como a pessoas que procurem transporte encontrar boleia.

O problema pode ser abstraído como um problema de grafos, em que as arestas representam caminhos e os nós representam localizações geográficas num mapa, as quais podem corresponder a origens e destinos de pessoas.

A seguir descrevemos diferentes complexidades com as quais podemos encarar o problema, tendo em conta as suas várias nuances.

1.1 Um carro, destinos em comum, ignorando compatibilidades temporais

A primeira abordagem, que tem a menor complexidade das três abordadas, consiste em considerar que o carro e todas as pessoas da rede possuem o destino em comum e que não existem quaisquer restrições temporais, isto é, nem o condutor nem as pessoas que

recebem a boleia possuem uma hora específica para apanhar a boleia nem uma hora máxima para chegarem ao seu destino.

O carro, que possui certa capacidade máxima, e o seu condutor iniciam a sua viagem num vértice de origem. Aí o carro irá dirigir-se para o local em que se encontrar a pessoa mais perto de si e irá adicioná-la à boleia. Este processo repete-se até o carro ter a sua capacidade completa, pelo que deve usar o caminho mais curto até ao vértice de destino, que, neste caso, é o mesmo para todos os ocupantes.

1.2 Um carro, destinos em comum, tendo em conta compatibilidades temporais

A segunda abordagem consiste em adicionar as restrições temporais de cada passageiro às restrições já assumidas na primeira abordagem. Assim, cada passageiro terá uma hora de partida e de chegada máxima para a sua viagem, que deverá ser levada em conta quando proposta uma solução.

1.3 Um carro, passageiros com destinos diferentes do condutor, tendo em conta compatibilidades temporais

A terceira abordagem consiste em, para além das restrições adicionadas nas duas primeiras abordagens, dar a possibilidade a cada passageiro de ter um destino diferente do condutor, ou seja, para cada viagem, o condutor poderá ter de deixar os passageiros antes do seu destino final.

2. Formalização do problema

2.1 Dados de entrada

- $G_{inicial}(V_{inicial}, E_{inicial})$ - Grafo dirigido pesado sem pesos negativos :

  . $V_{inicial}$ - vértices que representam lugares num mapa de estradas. Cada vértice $v \in V_{inicial}$ contém as seguintes informações:
  - . **id** - identificador inteiro único para cada vértice do grafo;
  - . **morada** - morada completa do local representado pelo vértice;
  - . **coords** - coordenadas geográficas do vértice no mapa, sendo coords.lat a latitude e coords.long a longitude;
  - . **pessoas** - lista que contém as pessoas presentes numa determinada;
  - . **adj -** lista de arestas adjacentes. Isto é, $e \in$ v.adj $\Rightarrow$ e.v_i = v .

  . $E_{inicial}$ - arestas dirigidas que representam estradas/caminhos entre lugares do mapa. Cada aresta $e = (v_i, v_f) \in E_{inicial}$ contém as seguintes informações:
  - . **id** - identificador inteiro único para cada aresta do grafo;
  - . $\mathbf{v_i}$ **-** vértice de origem da aresta;
  - . $\mathbf{v_f}$ **-** vértice de destino da aresta;
  - . **vmax** - velocidade máxima permitida;
  - . **dist** - comprimento da estrada/caminho;
  - . **fTransito** - fator de trânsito. Fator que representa a intensidade de trânsito rodoviário numa estrada/caminho.
  - . **tempo** - representa o tempo que é necessário para ir do local representado por $\mathbf{v_i}$ até o vértice representado por $\mathbf{v_f}$. Este tempo pode ser obtido através de uma fórmula dependente dos valores de **vmax,dist** e **fTransito.**

- *Pessoas* - Lista de pessoas que necessitam de boleia . Sendo cada pessoa $p \in$ *Pessoas* , contém as seguintes informações:
  - . **id** - identificador inteiro único para cada pessoa da rede;
  - . **origem -** morada correspondente ao ponto onde a pessoa apanha a boleia
  - . **destino** - morada correspondente ao destino da pessoa;
  - . **horaPartidaMin** - hora mínima (horas e minutos) do dia a que a pessoa pretende apanhar a boleia;
  - . **horaChegadaMax** - hora máxima (horas e minutos) do dia a que a pessoa pretende chegar ao seu destino;
  - . **horaPartida** - hora em que a pessoa é apanhada na boleia atual (pode ter um valor indefinido).

- *Carros* - Lista de carros . Cada carro corresponde a uma pessoa disposta a dar boleia a pessoas que existam no grafo G_i . Um carro $c \in$ *Carros* possui as seguintes informações:
  - . **id** - identificador inteiro único para cada carro da rede;

. **idCondutor** - identificador inteiro do condutor do Carro;
. **numMaxPass** . **-** número máximo de passageiros que o carro pode levar;
. **numPass -** número atual de passageiros no carro.

- *idCarro* - Identificador do carro que irá efetuar o percurso no algoritmo. Portanto, ∃ c ∈ *Carros : c.id = idCarro .*

2.2 Dados de saída

- P - lista de pessoas (p ∈ *Pessoas*) a quem deve ser dada boleia.
- S - sequência de nós ($v \in V_{inicial}$) ordenada pela ordem que devem ser percorridos de modo a passar por todas as origens e destinos das pessoas a quem deve ser dada boleia (P). Se S(i) for o i-ésimo elemento de S, então *S(0).morada = Pessoas[Carros[idCarro].idCondutor].origem* e *S(n-1).morada = Pessoas[Carros[idCarro].idCondutor].destino* , em que n é o número de nós do caminho. A título de exemplo, se S = {v1,v5,v3,v1,v6 }, o carro começará na morada correspondente do vértice v1 e seguirá o caminho representado pela aresta (v1,v5) ∈ $E_{inicial}$, de seguida (v5,v3), (v3,v1), e (v1,v6), sendo v6 o vértice com a morada correspondente ao destino do condutor.
- S = { } ⇒ Não existe caminho possível entre a origem do condutor do carro e o seu destino ;
- P = { } ⇒ Não é dada boleia a nenhuma pessoa na rede. Neste caso, S irá resumir-se ao caminho mais curto entre a origem do condutor e o destino ou, como é dito no ponto anterior, à inexistência de um caminho entre os dois pontos.

2.3 Restrições

2.3.1 Dados de entrada

- ∀v1,v2 ∈ $V_{inicial}$: v1≠v2 ⇒ v1.id ≠ v2.id ;
- ∀v1,v2 ∈ $V_{inicial}$: v1≠v2 ⇒ v1.coords ≠ v2.coords ∧ v1.morada < v2.morada ;
- ∀e ∈ adj: adj ∈ $V_{inicial}$⇒ p ∈ *Pessoas;*
- ∀e1,e2 ∈ $E_{inicial}$: e1≠e2 ⇒ e1.id ≠ e2.id ;
- ∀e ∈ $E_{inicial}$ ⇒ $e.v_i \neq e.v_f$;
- ∀e ∈ $E_{inicial}$ ⇒ $e.v_i \in V_{inicial}$ ∧ $e.v_f \in V_{inicial}$;
- ∀e ∈ $E_{inicial}$ ⇒e.vmax > 0 ∧ e.dist > 0 ∧ e.fTransito > 0 ∧ e.tempo > 0 ;
- ∀e ∈ $E_{inicial}$ ⇒e.tempo = f(vmax,dist,fTransito);
- ∀p ∈ pessoas ⇒ p ∈ *Pessoas;*
- ∀p ∈ Pessoas ⇒ ∃v1,v2 ∈ $V_{inicial}$: v1.morada = p.origem ∧ v2.morada = p.destino ;
- ∀p ∈ Pessoas ⇒ p.origem ≠ p.destino ∧ p.horaPartidaMin < p.horaChegadaMax ;
- ∀p ∈ pessoas ⇒ p.horaPartida ≥ p.horaPartidaMin ∧ p.horaPartida < p.horaChegadaMax*;*
- ∀p1,p2 ∈ Pessoas : p1≠p2 ⇒ p1.id ≠ p2.id ;

- $\forall$ *c1,c2* $\in$ *Carros* : c1≠c2 $\Rightarrow$ c1.id ≠ c2.id ;
- $\forall$ *c* $\in$ *Carros* $\Rightarrow$ c.numMaxPass ≥ 0 $\wedge$ c.numPass ≥ 0 ;
- $\forall$ *c* $\in$ *Carros* $\Rightarrow$ c.numPass ≤ c.numMaxPass ;
- $\forall$ c $\in$ Carros $\Rightarrow$ $\exists$ p $\in$ Pessoas: c.idCondutor = p.id ;
- $\exists$ c $\in$ *Carros : c.id = idCarro .*

2.3.2 Dados de saída

- $\forall$ p $\in$ P $\Rightarrow$ p $\in$ Pessoas ;
- $\forall$ v $\in$ S $\Rightarrow$ v $\in$ V_i .

2.4 Função objetivo

A solução ao problema dado passa por dois objetivos que são o de minimizar o tempo total da boleia (1), bem como garantir que cada pessoa que participa na boleia, incluindo o condutor, chega ao seu destino dentro do limite máximo (2 e 3) estipulado por essa pessoa (p.horaChegadaMax):

(1) $\sum_{k=0}^{n-2}$ e(v_k , v_{k+1}).tempo : e $\in$ $E_{inicial}$, onde n é o tamanho de S e V_k é o k-ésimo elemento de S.

(2) $\forall$p $\in$ P, p.horaPartida + $\sum_{k=x}^{y-1}$ e(v_k , v_{k+1}).tempo ≤ p.horaChegadaMax: e $\in$ $E_{inicial}$, onde x é o índice do vértice inicial e y o índice do vértice final do trajeto de p em S, e V_k é o k-ésimo elemento de S.

(3) Pessoa[Carros[idCarro].idCondutor].horaPartida + $\sum_{k=0}^{n-2}$ e(v_k,v_{k+1}).tempo ≤ Pessoa[Carros[idCarro].idCondutor].horaChegadaMax: e $\in$ $E_{inicial}$, oonde n é o tamanho de S e V_k é o k-ésimo elemento de S.

3. Solução

3.1 Pré processamento do grafo

3.1.1 Obtenção da rede simplificada

O pré-processamento do grafo passa por encontrar um grafo simplificado (G_{simple}) do grafo inicial ($G_{inicial}$). No grafo simplificado é apenas mantida a informação relevante aos algoritmos implementados para a resolução do problema, para tal o grafo é constituído apenas pelos nós que contêm pessoas(origens) ou que são destinos de pessoas da rede. Cada nó possui uma ligação com todos os outros nós da rede, que tem custo igual ao custo total mínimo de deslocamento em $G_{inicial}$.

A obtenção de G_{simple} exige a aplicação de um algoritmo de cálculo do "caminho mais curto" entre todos os pares de vértices. Os algoritmos considerados para o efeito são o algoritmo de *Floyd-Warshall ,*que possui uma complexidade $\Theta(|V|^3)$, e o algoritmo de *Dijkstra* que deve ser aplicado uma vez para cada vértice (este algoritmo será referido como *All-Pair-Dijkstra*, ou *APD*, daqui para a frente).

A complexidade do algoritmo de *Dijkstra*, pode variar consoante a implementação usada para a fila de prioridade mutável usada pelo mesmo. Usando uma fila de prioridade comum, a complexidade do algoritmo é de $O((|V| + |E|).\log|V|)$, que a aplicada a todos os vértices se traduz numa complexidade total, para o algoritmo *APD*, de $O(|V|.(|V| + |E|) \log|V|)$. Por outro lado, usando uma fila de prioridade de *Fibonacci* (*Fibonacci Heap*), é possível reduzir a complexidade do algoritmo de *Dijkstra* para $O(|V|.\log|V| + |E|)$ e a complexidade total para $O(|V|^2 .\log|V| + |V|.|E|)$.

Para escolher o algoritmo apropriado é necessário efetuar uma análise a $G_{inicial}$, mais concretamente, à sua densidade. De facto, se o grafo for pouco denso, tal que $|E| \sim |V|$, a complexidade do algoritmo *APD* , pode ser aproximada por $O(|V|^2 .\log|V| + |V|^2)$ ou apenas $O(|V|^2.\log|V|)$. No entanto, no caso de um gráfico denso, em que $|E| \sim |V|^2$, a complexidade ficaria $O(|V|^2 .\log|V| + |V|^3)$. Podemos então concluir que para uma rede pouco densa, há benefícios em utilizar o algoritmo de *Dijkstra* e para uma rede densa é mais vantajoso utilizar o algoritmo de *Floyd-Warshall.*

Para este projeto, escolhemos assumir o pior caso de uma cidade densa, utilizando portanto, o algoritmo de *Floyd-Warshall.* Assim, no fim da aplicação do algoritmo, temos o grafo G_{simple} que, note-se, pode ser um grafo que apenas existe como uma parte da matriz gerada por *FW* e não como um grafo completamente independente em memória.

3.1.2 Redução de G_{simple}

A fase seguinte de pré-processamento do grafo passa por efetuar uma triagem às pessoas que se encontram na rede, de forma a retirar as pessoas que apresentam incompatibilidades incontornáveis com o condutor do carro.

A incompatibilidade mais óbvia é a existência de pessoas que têm uma hora máxima de chegada (horaChegadaMax) inferior à hora de partida mínima (horaPartidaMin) do condutor.

Outras pessoas da rede que são incompatíveis são aquelas em que a soma da sua hora de partida mínima com o caminho mais curto entre a origem do condutor e o destino da pessoa seja superior à sua hora máxima de chegada.

A triagem de pessoas pode implicar a remoção de nós do grafo simplificado G_{simple}, visto que os nós deste representam apenas os locais que possuem pessoas prontas a receber boleia e caso sejam removidas todas as pessoas associadas a um local o vértice deixa de ser relevante para G_{simple}.

A fase anterior dá-nos também informação acerca da conectividade entre elementos da rede, isto é, sobre a existência(ou não) de um caminho para um dado vértice. Assim, uma análise à matriz das distâncias gerada pelo algoritmo de *Floyd-Warshall* , mais concretamente, à coluna do vértice de origem, permite-nos averiguar se existem nós na rede que são inalcançáveis pelo condutor (distância infinita na matriz), pelo que estes nós devem ser removidos.

3.2. Solução

3.2.1. Abordagem 1.1 - Um carro, destinos em comum, ignorando restrições temporais

3.2.1.1 Algoritmo

Na abordagem mais simples , são escolhidas através do uso de uma fila de prioridade e de uma função de “compatibilidade” a ela associada os nós que possuem pessoas que são mais compatíveis com a boleia. O caro começa no seu nó de partida e constrói a fila de prioridade. Retira dela a pessoa mais compatível, adiciona-a à boleia e repete o processo até o carro estar cheio ou até não existirem mais pessoas compatíveis na área designada. A fila atribui a prioridade conforme a função ϕ.

3.2.1.2 A função $\phi(v)$

Para tentar minimizar o tempo total da viagem do carro, é utilizada a função ϕ que procura atribuir um valor numérico a cada nó $v \in V_{simple}$ que permita avaliar a sua compatibilidade com a boleia em questão. Para isso a função $\phi(v)$ depende de três fatores que têm pesos diferentes no resultado final da mesma.

O primeiro fator, é a distância (euclidiana) entre o nó atual do condutor e o nó em análise (v), de modo que a função irá privilegiar pessoas que estejam perto do condutor. O segundo fator, é a distância euclidiana entre o destino da pessoa em v e do destino do condutor, desta forma, serão escolhidas primeiramente pessoas que pretendam ir para locais próximos do destino do condutor. O último fator é a distância euclidiana entre **v** e o destino final do condutor, com o objetivo de, de forma semelhante ao algoritmo *A**, dar um incentivo à progressão na direção do nó de destino.

No entanto, dado a que os vértices de V_{simple} contêm, também, vértices de destino das pessoas os quais podem não conter ninguém, é necessário aplicar uma constrição $\phi(v)$ tal que um nó que não possua ninguém tenha um resultado inválido.

3.2.2. Abordagem 1.2 - Um carro, destinos em comum, tendo em conta compatibilidades temporais

3.2.2.1 Algoritmo fillCarGreedy

fillCarGreedy é um algoritmo *greedy* que, para um condutor com um destino específico, escolhe um caminho e pessoas a apanhar. Assim, inicializa-se a pesquisa pelo condutor e até ao carro estar cheio ou não haver mais pessoas que justifique juntar à viagem, seleciona a próxima pessoa a adicionar.

Para cada escolha é explorada uma parte limitada do gráfico (em que todos os vértices **v** distam menos que **dis** do vértice/pessoa atual). Todos os vértices que se situam nesta área são acrescentados a uma fila de prioridade **Q**, de acordo com a função ϕ.

3.2.2.2 Implementação

Num passo seguinte é escolhido o vértice com mais prioridade de Q. Se for possível passar por esse vértice e chegar a tempo ao destino final, então essa pessoa será acrescentada ao carro e o algoritmo começará novamente a explorar para esse vértice. Caso não seja possível, seleciona-se a próxima pessoa da fila, etc. Se a fila estiver vazia, então o condutor não deverá apanhar mais ninguém e deverá prosseguir imediatamente o destino.

O pseudo-código para este algoritmo encontra-se a seguir apresentado:

```
1.  function fillCarGreedy(G_simple , carro):
2.
3.      Priority_queue Q
4.      dis // distância máxima a procurar nas proximidades de um vértice
5.      vertAtual ← carro.condutor
6.      horaAtual ← carro.condutor.horaPartida
7.      dest ← carro.condutor.destino
```

```
8.      maxHoraChegada ←carro.condutor.horaChegada
9.
10.     while not carro.isFull():
11.         Q.makeEmpty()
12.
13.         for each v in Gsimple:
14.                 if dist[vertAtual, v] < dis and v.hasPerson and v.Person.notInCar:
15.                         Q ← v
16.                 endif
17.         endfor
18.
19.         while Q is not empty:
20.                 u ← Q.extractMin()
21.                 previsaoChegada ← horaAtual + temp[vertAtual, u] + temp[u, dest]
22.                 if previsaoChegada< maxHoraChegada and previsaoChegada<
    u.horaChegada:
23.                         horaAtual ← horaAtual + temp[vertAtual, u]
24.                         p ← maxϕPerson(u)
25.                         carro ← p
26.                         vertAtual ← u
27.
28.                         if u.horaChegada < maxHoraChegada:
29.                                 maxHoraChegada ←u.horaChegada
30.                          endif
31.
32.                         break
33.                 endif
34.         endwhile
35.
36.      if Q is empty:
37.         return carro
38.      endif
39.
40.      endwhile
41.
42.    return carro
```

As linhas 3-8 tratam de inicializar os valores iniciais das variáveis utilizadas e de inicializar a fila de prioridade(Q), que usa a função ϕ(v) para definir a prioridade. O vértice inicial é o vértice do condutor, que começa a boleia à hora indicada pela sua hora de partida. Neste caso, os destinos de todos os intervenientes são os mesmos e iguais ao do condutor (carro.condutor.destino). Enquanto o carro não se encontrar cheio (carro.isFull()), é populada a fila de prioridades(13-17) e é escolhido o novo elemento a fazer parte da boleia (linha 19 a 34). A linha 24 é usada para desambiguar o resultado para vértices que contêm mais que uma pessoa, retornando a pessoa que obtenha o maior valor de ϕ. Como todos os intervenientes vão para o mesmo lugar, apenas se tem de garantir que é possível chegar ao nó correspondente a esse local antes do tempo de chegada mínimo para todos os ocupantes do carro.

3.2.3. Abordagem 1.3 - Um carro, com destinos diferentes do condutor, tendo em conta compatibilidades temporais

3.2.3.1 Algoritmo greedy

Para encontrar uma solução para o problema também foi utilizado um algoritmo greedy, muito semelhante ao algoritmo visto em 1.2 mas com algumas alterações, nomeadamente na verificação das compatibilidades de modo a acomodar as restrições temporais em relação aos diferentes destinos.

Dado uma nova pessoa **p**, que se deseje adicionar à boleia, é necessário verificar se existe um caminho entre o vértice que contém **p** e o vértice final do carro passando anteriormente por todos os vértices de destino dos ocupantes que já se encontram no carro (excluindo o condutor).

A título de exemplo, dadas quatro pessoas ({A,B,C,D} sendo A o condutor) que já se encontrem dentro de um carro e uma quinta pessoa(E) sobre a qual se pretende averiguar a compatibilidade com a boleia, existem, no máximo, 3! = 6 caminhos possíveis (função *preverHora* apresentada em seguida). Se um dos caminhos considerados conseguir satisfazer os requisitos de tempo, a pessoa **p** é adicionada à boleia e o algoritmo é repetido se ainda existem lugares suficientes no carro.

Caso o carro seja completamente preenchido ou não exista mais nenhuma pessoa compatível no raio considerado, o algoritmo termina.

O pseudo-código para esta adaptação encontra-se a seguir apresentado:

```
43. function isSetPossible(hora, S, u, carro): //returns -1 if set not possible
44.    hora ←horaAtual +temp[u, S[0]]
45.
46.    for i in S-1:
47.       hora ← hora + temp[S[i].destino, S[i+1].destino]]
48.       if hora>S[i+1].horaChegada:
49.          return -1
50.       endif
51.    endfor
52.    hora ← hora + temp[S[S.size-1].destino, carro.condutor.destino]
53.
54.    if hora>carro.condutor.horaChegada:
55.      return -1
56.    else:
57.      return hora
58.    endif
59.
60.
61. function preverHora(carro, u, horaAtual): //returns -1 if not possible
62.
63.    minHora ← ∞
64.
```

```
65.    for every ordered set S in carro\carro.condutor where
    |S|=|carro\carro.condutor|:
66.
67.       hora = isSetPossible(horaAtual, S, u, carro)
68.       if (hora != -1 and hora < minTempo):
69.          minHora ←hora
70.       endif
71.     endfor
72.
73.    if minHora= ∞:
74.       return -1
75.    else:
76.       return minHora
77.    endif
78.
79.
80.
81. function fillCarGreedyDifferentDestinations(G_simple , carro):
82.
83.     Priority_queue Q
84.     dis // distância máxima a procurar nas proximidades de um vértice
85.     vertAtual ← carro.condutor
86.     horaAtual ← carro.condutor.horaPartida
87.     dest ← carro.condutor.destino
88.     maxHoraChegada ←carro.condutor.horaChegada
89.
90.     while not carro.isFull():
91.         Q.makeEmpty()
92.
93.         for each v in Gsimple:
94.                 if dist[vertAtual, v] < dis and v.hasPerson:
95.                         Q ← v
96.         endfor
97.
98.         while Q is not empty:
99.
100.                u ← Q.extractMin()
101.                p ← maxϕPerson(u)
102.                previsaoChegada ← preverHora(carro+p, u, horaAtual)
103.
104.
105.                        if previsaoChegada != -1:
106.                                horaAtual ← horaAtual + temp[vertAtual, u]
107.
108.                                carro ← p
109.                                vertAtual ← u
110.
111.                                if u.horaChegada < maxHoraChegada:
112.                                        maxHoraChegada ←u.horaChegada
113.                            endif
114.                            break
115.                    endif
116.            endwhile
```

```
117.
118.            if Q is empty:
119.               return carro
120.        endif
121.
122.     endwhile
123.
124.    return carro
```

Esta solução funciona de forma muito semelhante à vista no ponto anterior. A principal diferença encontra-se na linha 100 com o uso da função *preverHora* . Esta função itera segundo todas as permutações de ordens de visita das pessoas que já se encontram no carro usando a função *isSetPossible* (começando no vértice a testar e acabando no destino do condutor). Para além disso a função devolve o tempo da permutação de tempo mínimo. A solução do algoritmo é o vetor carro que corresponde a P (ver *2.2 Dados de saída*).

3.2.3.1 Destinos intercalados com as partidas

Para abranger casos em que existam destinos intercalados com partidas, ou seja, em que um condutor possa apanhar uma pessoa A, deixá-la no seu destino e em seguida apanhar uma pessoa B e levá-la até ao seu destino, pode se adaptar o algoritmo desenvolvido anteriormente.

Para isso cria-se uma *priority queue* **Q2** onde serão guardados os vértices a passar pelo carro em cada momento da exploração, com uma hora de partida e de chegada máxima associadas a cada um.

Q2 será ordenada da seguinte forma:

1. *O primeiro elemento da fila é o vértice de partida do condutor e o último o seu vértice de chegada*
2. *Garantido que se V2 é destino de V1, então V2 estará sempre depois de V1*
3. *Por ordem crescente de hora de chegada máxima.*
4. *Por ordem crescente de inserção*

Sempre que se queira verificar se uma nova pessoa poderá ser acrescentada ao carro, terá de ser feita a inserção em **Q2** do seu vértice de chegada e partida e verificar se, por ordem da fila, todas as restrições se mantêm. Isto é, começando na hora de partida do primeiro vértice que será sempre o condutor, acrescentando o tempo que demora a chegar ao segundo elemento da fila, verifica-se se chegaria lá até à hora máxima a que esse vértice está associado e assim sucessivamente. Se as restrições não forem cumpridas essa pessoa é retirada da fila e continuará a pesquisa.

Sempre que se insere uma pessoa, se o seu vértice de chegada ou partida já estiver na fila e se a pessoa tiver uma hora de chegada máxima a esse vértice menor que a atual, altera-se a hora do vértice, aumentando a sua prioridade. Se não, não se insere esse vértice.

Assumindo que não seria produtivo um condutor apanhar mais que *n* pessoas, sendo *n* o número de lugares do seu carro excluindo o condutor, em qualquer momento a fila teria um comprimento máximo de *2(n+1)*, caso em que o carro estaria preenchido com *n* pessoas e que cada uma teria um destino e uma partida diferentes.

3.2.4. Extrapolar dados de saída

No fim das soluções apresentadas é devolvido o vetor contendo as pessoas que incorporam a boleia (*P*). Este vetor encontra-se ordenado pela ordem que estas devem ser adicionadas à boleia. Para obter o vetor de nós correspondentes ao percurso do carro, é necessário usar G_{simple} e a matriz gerada pelo algoritmo *FW* para encontrar a sequência de nós que corresponde aos caminhos da matriz (*S*).

Uma otimização interessante à solução apresentada passaria por otimizar a sequência de nós obtida (S) para eliminar quaisquer repetições que levem à passagem desnecessária por determinados nós.

3.2.5. Equilíbrio entre otimização e custo computacional

Para encontrar uma solução que seja mais próxima da ótima, é possível adaptar estes algoritmos. Para isso basta que não se selecione somente a primeira opção da fila de prioridade. Assim, de forma recursiva, é possível explorar mais do que um caminho e no final selecionar o com menos custo. Quanto maior for o *n* número de possíveis próximos vértices a analisar, mais otimizada será a solução.

No entanto, é de notar que o aumento do *n*, implica um aumento polinomial da complexidade do problema, dado que se propaga nas iterações seguintes. Contudo, visto que todas as combinações são independentes, utilizando técnicas de computação paralela e dependendo do hardware, seria possível reduzir significativamente o tempo de computação. Por outro lado, usando programação dinâmica para impedir repetição de cálculos (como por exemplo da função $\phi(v)$), também seria possível melhorar o desempenho.

4. Casos de utilização e funcionalidades

Atualmente, o desenvolvimento acentuado da plataforma móvel proporciona uma base ideal para a implementação a solução dada de ridesharing. Como tal, o caso de utilização da solução apresentada passaria pela integração numa aplicação para *smartphone* que permitiria o registo de pessoas que precisem de boleia para o seu destino ou que, por outro lado, queiram dar boleia a outros para minimizar o impacto ambiental ou reduzir os custos da sua viagem.

A aplicação deve ter associada uma extensa rede de mapas de modo a conseguir satisfazer os pedidos dos seus utilizadores. A introdução da rede é feita pela própria aplicação e é alheia aos seus utilizadores.

Para simplificar a complexidade do grafo, nem todas os locais de um mapa são representados para um nó, pelo que a inserção de uma pessoa com uma dada morada implica a associação desta com o nó do grafo com a localização mais próxima da morada dada.

A fim de evitar aproximações muito grosseiras, o grafo deve ter uma "densidade" suficiente para modelar corretamente uma zona. Para tal, para além de ser usado um vértice para cada entroncamento, também as ruas muito extensa devem ser repartidas em conjuntos de vértices e arestas. Isto garante uma maior precisão de localização consoante as moradas inseridas pela pessoa.

A simplificação do grafo inicial descrita em 3.1.1 é alheia ao utilizador e pode ser mantida ao longo de diversas iterações da solução apresentada visto ser informação que é comum ao longo do tempo (exceto atualizações da rede em questão). No entanto, a simplificação referida em 3.1.2 já necessita ser repetida a cada iteração visto que nela são tratadas questões referentes à compatibilidade entre os intervenientes, as quais mudam em iterações diferentes (diferentes condutores e destinos).

A utilização da aplicação considera dois casos, o de uma pessoa que procura boleia e o de uma pessoa que, tendo carro próprio, procura oferecer boleia. Para isto, é permitido às pessoas inscreverem-se para procurar uma boleia, tendo para isto que dizer a origem e destino da viagem que pretende realizar, bem como as horas de partida e chegada. No caso de uma pessoa que esteja a oferecer boleia, deve-se inscrever com as mesmas informações do caso anterior, às quais deve acrescentar a lotação do carro. Neste caso, o utilizador é apresentado com um percurso detalhado que deve realizar com indicação dos locais onde deve apanhar e deixar outros utilizadores, que lhe permite chegar ao destino desejado cumprindo as restrições temporais que ele especificou.

Quando uma boleia que cumpra os requisitos é encontrada para um utilizador, este é notificado com as horas de partida de modo a ele estar no local indicado à hora certa.

5. Implementação

5.1 Estruturas de dados

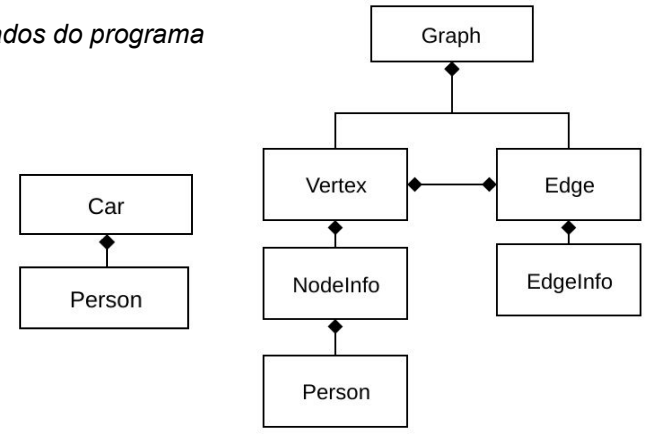
5.1.1 Os dados

A classe **Graph** guarda um vetor de vértices e uma matriz de adjacências com as distâncias entre os vértices (que é preenchida pelo algoritmo de *Floyd-Warshall*).

A matriz será usada no *fillCarGreedy* para aceder às distâncias entre vértices dado ter acesso O(1), enquanto que o vetor é usado no pré-processamento.

A redundância entre **Vertex** e **Edge** existe para diminuir tempos de acesso e procura nos algoritmos.

Fig. 1: Estrutura de dados do programa



5.1.2 Heap usada pelo *fillCarGreedy*

Como explicado anteriormente o *fillCarGreedy* necessita de uma fila de prioridade para a cada iteração escolher o melhor candidato. A seguinte função foi implementada:

$$\phi = currentToSubjectTime \times 10 + \frac{subjectToDriverDestDistance}{100} + subjectToDriverDestTime \times 10$$

Tal que:

- *Current: vértice a analisar na iteração*
- *Subject: vértice a colocar na heap*
- *DriverDest: vértice do destino final*

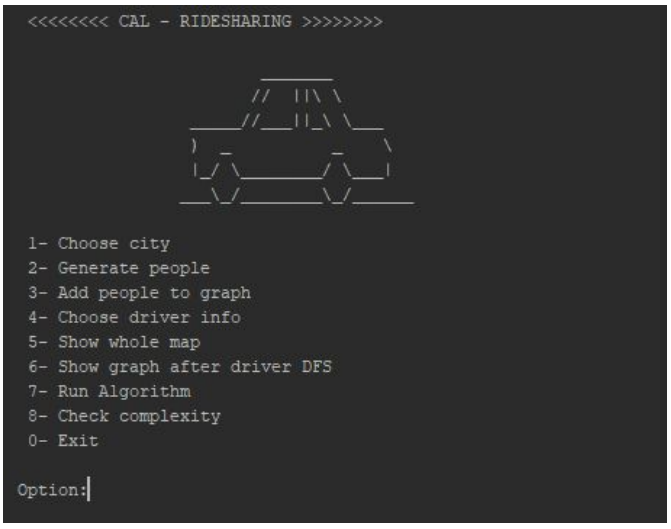
5.2 Conetividade dos grafos

Os grafos utilizados não são conexos, mas são pesados e bidirecionais. Para simplificar a adaptação dos algoritmos, assumimos que os grafos são direcionados, tendo 2 arestas por caminho (uma em cada sentido).

5.3 Discussão sobre os principais casos de uso implementados

Como nos foi sugerido pelo professor, focamo-nos no caso em que os destinos são comuns de modo a modelar a situação em que o algoritmo seria usado por pessoas que andam na mesma faculdade ou que trabalham na mesma empresa por exemplo.

Para tal desenvolvemos uma aplicação simples que permite a demonstração do nosso algoritmo. Nela, é possível escolher a importação de um dos mapas fornecidos e visualizá-lo sob a forma de grafo, bem como a geração de pessoas e a escolha de um condutor de modo a possibilitar a aplicação do algoritmo fillCarGreedy. No fim do algoritmo é possível ver no grafo o caminho que o condutor deve efetuar para apanhar as pessoas que lhe são designadas e chegar aos seus destinos comuns.



A primeira opção permite a escolha da cidade dentro dos mapas disponíveis (exemplo de input: “Fafe”).

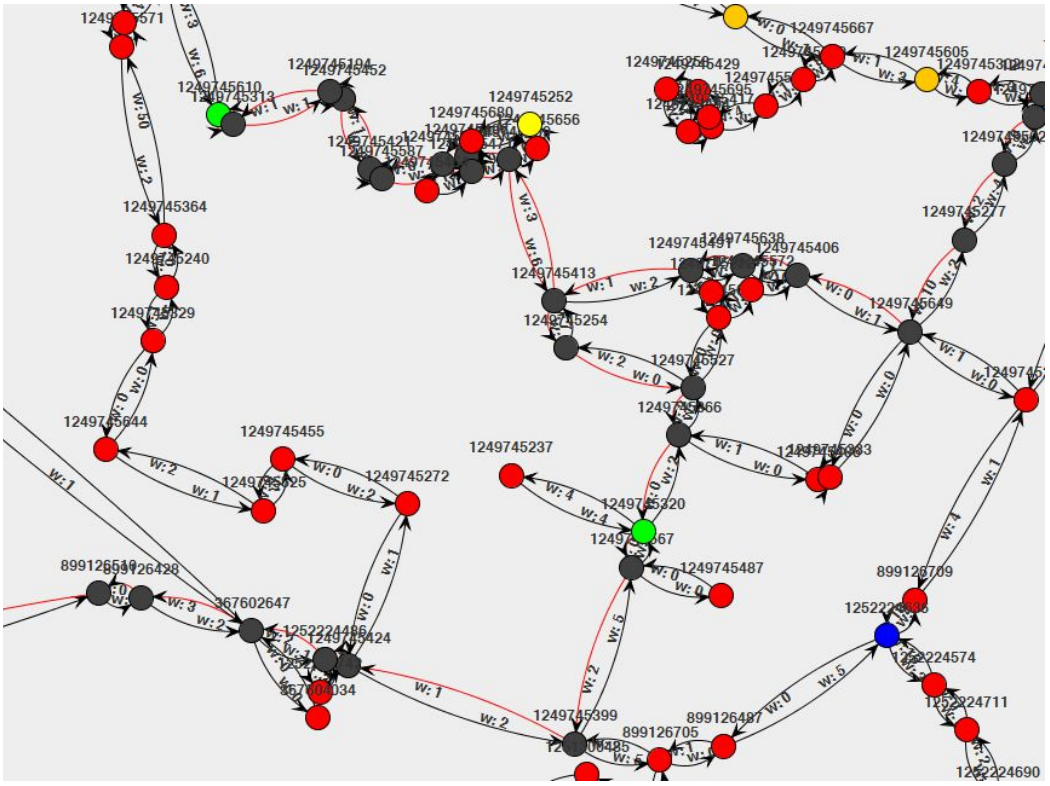
A segunda opção permite gerar pessoas que possui um destino em comum no mapa e que têm nós de partida, hora mínima de partida e hora máxima de chegada aleatórias.

A terceira opção adiciona as pessoas de um ficheiro gerado por 1 (pessoas_Fafe.txt por exemplo) a um grafo.

A quarta opção permite introduzir os dados do condutor: nome, id do nó de partida, id do nó de destino, hora de de partida e hora máxima de chegada. É de notar que o nó de destino é o mesmo das pessoas do ficheiro carregado caso esse ficheiro tenha sido gerado na mesma sessão, caso contrário, este id é introduzido pelo utilizador e deve coincidir com o id de destino das pessoas carregadas.

A quinta opção permite mostrar através do uso da biblioteca *“GraphViewer”* o grafo gerado. O peso das arestas do grafo representa o número de minutos que é necessário

para percorrer o caminho que une os dois nós. A cor dos nós possui significados variados: azul-claro representa nós sem nenhuma pessoa, azul-escuro nós com uma pessoa, amarelo com duas pessoas, laranja com três pessoa e vermelho para o restante número de pessoas.



A sexta opção permite efetuar uma *Depth-First-Search* a partir do nó de partida do condutor de modo a visualizar os nós que são alcançáveis a partir do mesmo. Caso o nó de destino do condutor não seja alcançável o programa termina.

A sétima opção executa o algoritmo *fillCarGreedy* e nas consequentes execuções da opção 5 irão existir nós de cores diferentes: a cor-de-rosa o nó de origem do condutor e a magenta o nó de destino, a verde os nós onde são apanhadas pessoas para a boleia e a cinzento os nós por onde o carro passa (as arestas do percurso do condutor também estão coloridas a vermelho). O algoritmo escreve na consola dados acerca do seu resultado e da sua terminação:

```
Empty queue - ArrivalTime: 9:06
Done.
Passengers:
Passenger id: -1, name: David minTime 7:00 maxTime 22:00 pickupTime 7:00
Passenger id: 1763, name: Ubouv minTime 7:49 maxTime 9:36 pickupTime 8:01
Passenger id: 3535, name: Ipivu minTime 8:15 maxTime 9:43 pickupTime 8:21
```

A oitava opção irá cronometrar sucessivas execuções do algoritmo exportanto os resultados para um ficheiro CSV de modo a efetuar o estudo empírico da complexidade.

É aconselhado a consulta do README enviado com o relatório para instruções sobre a utilização do programa.

5.4 Análise da complexidade temporal da solução

5.4.1 Análise Teórica

A complexidade temporal do algoritmo apresentado recebe contribuições das suas diferentes fases: o pré-processamento do grafo e o algoritmo executado no grafo simplificado.

5.4.1.1 Pré-processamento

Uma pesquisa em profundidade (DFS - *Depth First Search*) no grafo do mapa inicial G1(V1,E1)a partir do nó do condutor permite a eliminação dos nós inalcançáveis por parte do condutor. Uma DFS tem, no pior caso, uma complexidade de O(|V1| + |E1|) e a consequente construção do primeiro grafo simplificado (G2)implica uma complexidade acrescida de O(|V1| + |E1|). Somando obtemos uma complexidade de 2.O(|V1| + |E1|), que se pode simplificar em apenas O(|V1| + |E1|) .

Tendo o grafo simplificado G2(V2,E2), é efetuada uma segunda simplificação de modo a permitir ao algoritmo principal ter, com prontidão, os valores das distâncias mais curtas entre todos os pares de vértices e a remover pessoas incompatíveis com o condutor .

Para tal, é efetuado o algoritmo de *Floyd-Warshall* que possui uma complexidade aproximada de $O(|V2|^3)$. Com esta informação é construído o segundo gráfico simplificado G3(V3) que apenas possui nós que são origens de pessoas/condutor e o seu destino comum. Este grafo apenas possui os vértices e a matriz de adjacências gerada pelo algoritmo de FW. A construção tem complexidade temporal de O(|V2|) pois apenas implica adicionar os nós com pessoas de G2 à estrutura de G3.

A remoção de pessoas incompatíveis tem complexidade de O(|V2|.|P|),em que P é o número médio de pessoas por nó do grafo(é necessário percorrer todos os nós e todas as pessoas num dado nó a fim de verificar as compatibilidades), deixando a complexidade da fase de processamento com um total aproximado de $O(|V1| + |E1|) + O(|V2|^3) + O(|V2|) + O(|V2|.|P|)$.

As proporções entre a quantidade de nós de G1,G2 e G3 estão dependentes de vários fatores como a origem, o destino e a quantidade de pessoas no grafo inicial, bem como a sua dispersão pelos nós.

5.4.2.2 Algoritmo fillCarGreedy

O algoritmo fillCarGreedy possui uma complexidade que depende de vários fatores. De facto a procura dos vértices de destino e iniciais do percurso acarreta uma complexidade linear sobre o número de vértices do grafo (relembra-se que nesta fase o grafo é G3), O(|V3|) .

O interior do ciclo *while* das linhas 10-40 consiste na construção de uma fila de prioridade com os vértices do grafo e, de seguida, a extração de pessoas até ser encontrada uma pessoa compatível com a boleia. A construção da fila implica a adição dos nós do grafo e a definição da sua “prioridade”, obtendo-se uma complexidade de

O(log(|V3|).|V3|.O(ϕ)) em que O(ϕ) é a complexidade da função de prioridade. É de notar que esta complexidade é uma complexidade de pior caso, visto que apenas são adicionados vértices que se encontram dentro de um dado raio do vértice “atual”.

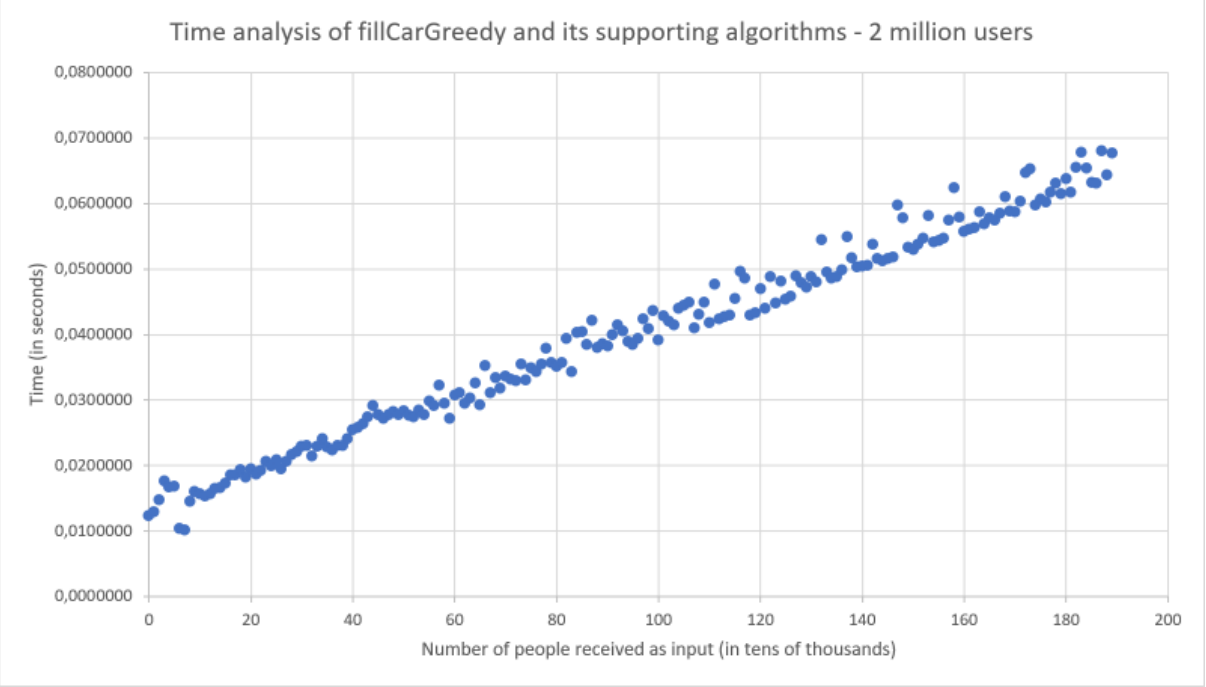
A extração de pessoas da fila possui uma complexidade de pior caso de O(|V3|.log(|V3|)), sendo o caso em que a única pessoa compatível é a que tem menos prioridade dentro do conjunto de vértices do grafo. As compatibilidades são regidas pelas restrições horárias das pessoas dos nós, pelo que esse também é um fator contribuinte para a complexidade do algoritmo.

Finalmente o ciclo *while* das linhas 10-40 é executado, no máximo, uma vez por cada lugar vago no carro, que dado a ser um valor constante, pode ser ignorado no estudo da complexidade.

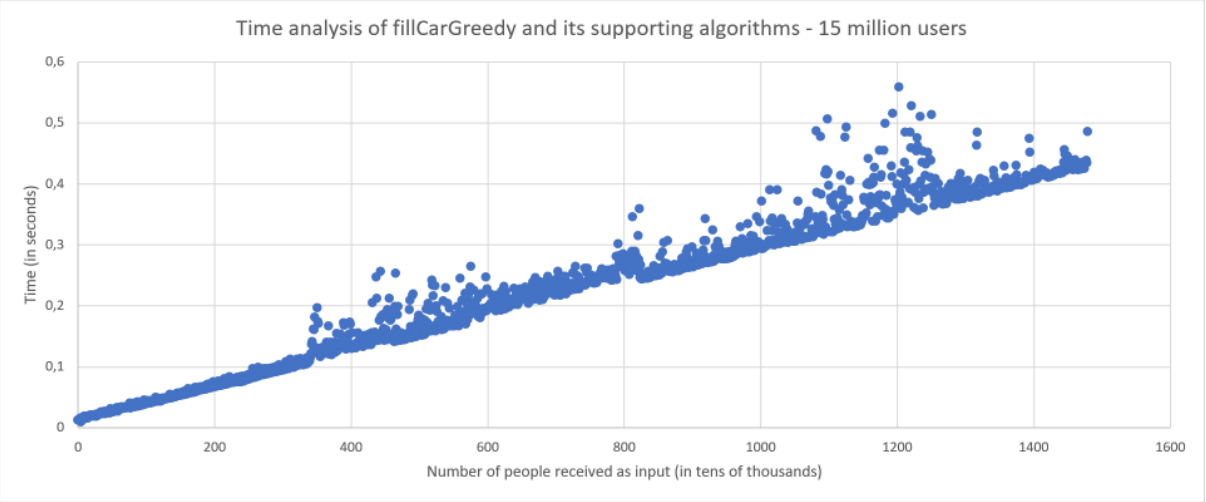
O algoritmo fillCarGreedy possui então uma complexidade total aproximada de O(|V3|) + O(log(|V3|).|V3|.O(ϕ)) + O(log(|V3|).|V3|), simplificado em O(log(|V3|).|V3|.O(ϕ)).

3.3.2 Análise empírica

Para verificar empiricamente a complexidade do algoritmo, fizemos uma análise à sua dependência em relação à quantidade de pessoas, dado que, no grafo que entra no algoritmo, os vértices são proporcionais a esta quantidade. Assim, geramos pessoas aleatórias de forma crescente (189 iterações, aumentando em 10 mil o número de pessoas a cada iteração), mas mantendo o mesmo condutor e o mesmo mapa. Obtivemos o gráfico em baixo que aponta para uma complexidade aproximadamente linear, com aproximadamente dois milhões de utilizadores.



Para verificarmos os resultados, passamos a um input de 15 milhões de utilizadores e obtemos o gráfico em baixo, que aponta novamente para uma complexidade linear, mas que tem indícios que demonstram que poderemos estar perante uma complexidade n.log(n).
Esta conclusão é coerente com a previsão teórica de O(log(|V3|).|V3|.O(ϕ)).



6. Conclusão

6.1 Dificuldades encontradas (1ª parte)

Ao longo da realização do trabalho foram encontradas diversas dificuldades, sendo que a principal foi a compreensão do que era pretendido com o tema e qual o problema concreto para o qual tínhamos que encontrar uma solução.

Tivemos ainda dificuldade em decidir quais os algoritmos abordados nas aulas que poderíamos aplicar para a resolução do nosso problema, devido à complexidade do problema bem como restrições inerentes ao tema, nomeadamente tolerâncias temporais e forçar que um caminho passasse em vértices onde se encontrassem passageiros.

O facto de termos escrito o relatório sem tentar implementar a solução apresentada também nos deixa relutantes em relação à qualidade dos resultados que a solução possa obter, pois não nos foi possível testar nem comparar diferentes abordagens.

Outra dificuldade encontrada passa pela escrita de pseudo-código que seja ao mesmo tempo fácil de compreender bem como consiga cobrir todos os pontos e casos extremos da solução pretendida.

6.2 Esforço dedicado pelos elementos do grupo (1ª parte)

O esforço dedicado nesta fase do projeto foi igual para todos os elementos no grupo. Inicialmente tratamos de idealizar a estrutura que seria usada para representar a informação respetiva ao problema e de seguida idealização de algoritmos capazes de atender às necessidades. Ao longo do tempo fomos aperfeiçoando os algoritmos com pequenas alterações sendo que todas as mudanças foram verificadas por todos os elementos do grupo.

6.3 Dificuldades encontradas (2ª parte)

Uma das maiores dificuldades encontradas foi a análise empírica deste problema, dada a grande quantidade de variáveis (mapa, pessoas, drivers, restrições temporais, etc) e a geração aleatória destes dados. De facto, foi difícil garantir que as variáveis (que não a em estudo) fossem constantes. Por outro lado, também foi custoso encontrar nos *datasets* um condutor que fizesse sentido, ou seja, não isolado e cujo nó destino fosse alcançável.

Em segundo lugar, tivemos uma limitação devida à capacidade das máquinas disponíveis. Por exemplo, para gerar o *dataset* de users para o segundo teste necessitamos de 1.85 GB, excluindo o espaço utilizado pelo programa em si.

Seria deveras interessante, no futuro, fazer uma análise mais aprofundada deste problema, nomeadamente estendendo o *dataset*, para verificar se a função de

complexidade temporal é puramente linear ou se começa a sentir o efeito da função logarítmica.

Durante o desenvolvimento tornou-se evidente uma das limitações da solução apresentada. Aquando da entrada de uma nova pessoa para o carro, o limite máximo para a chegada ao destino pode passar a ser regido pelo limite da pessoa (caso seja maior do que o limite existente do carro), assim, a entrada de uma pessoa compatível que reduza o limite limite em grande número as sucessivas tentativas de inserção.

6.4 Esforço dedicado pelos elementos do grupo (2ª parte)

A maioria da preparação do trabalho, nomeadamente a extração de dados para a criação de grafos, a geração aleatória de pessoas e a ligação com a interface foi realizada pelo David Silva. O restante desenvolvimento foi dividido igualitariamente pelos três membros da equipa.

6. Referências Bibliográficas

- “Introduction to Algorithms”, 3rd Edition, T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein., MIT Press, 2009
    . Chapter 19 - Fibonacci Heaps;
    . Chapter 22 - Elementary Graph Algorithms;
    . Chapter 24 - Single-Source Shortest Paths;
    . Chapter 25 - All-Pairs Shortest Paths.

- “Slides de Concepção e Análise de Algoritmos”, R. Rossetti, L. Ferreira, L. Teófilo, J. Filgueiras, F. Andrade, CAL, MIEIC, FEUP
    . Algoritmos em Grafos: Introdução;
    . Algoritmos em Grafos: Caminho mais curto (Parte I)
    . Algoritmos em Grafos:Caminho mais curto (Parte II)