Ride Sharing: Partilha de Viagens

Conceção e análise de algoritmos

Abril 2019

2MIEIC06 - Grupo G, Tema 1

- Inês Rodrigues Roque de Lacerda Marques, *up201605542@fe.up.pt*
- Nuno Miguel Fernandes Marques, *up201708997@fe.up.pt*
- Ricardo Jorge de Araújo Ferreira, *ee03195@fe.up.pt*

1. Descrição

No âmbito da unidade curricular Concepção e Análise de Algoritmos do curso Mestrado Integrado em Engenharia Informática e Computação, será desenvolvida uma aplicação de partilha de viagens (car sharing).
Esta aplicação irá processar pontos de partida, recolha e chegada, marcados num mapa pelos vários utilizadores da aplicação. O objetivo principal de uma aplicação deste tipo é a utilização optimizada do automóvel, tentado evitar que o condutor viaje sozinho pela menor tempo possível. Pode-se assim, identificar dois perfis de utilizadores da aplicação, os condutores e os passageiros. Os condutores, possuem um veículo com um determinado número de passageiros e durante a sua deslocação irão passar pelos pontos de recolha de utilizadores que não conduzem. Todos os utilizadores devem registar na aplicação o seu local de partida, de chegada e indicar um intervalo de tempo. Os utilizadores condutores devem ainda indicar que possuem um veículo e qual o número de passageiros.
Com estas informações, caberá ao algoritmo da aplicação, gerar rotas desde o local de partida e de chegada dos condutores, passando pelos pontos de recolha e destino dos passageiros. As rotas devem ter em conta as restrições temporais de cada utilizador e tentar optimizar a utilização do veículo.
A aplicação será desenvolvida em C++, com recurso à API GraphViewer para se desenhar o grafo gerado para os trajectos. Os mapas utilizados serão extraídos da plataforma open street maps (https://www.openstreetmap.org).
Neste documento será apresentada a formalização deste problema e uma proposta de implementação.

2. Identificação e formalização do problema

O objectivo principal de um sistema de partilha de viagens é a utilização partilhada de um veículo entre indivíduos que se deslocam numa rota específica. A poupança associada a este sistema depende da correcta criação de grupos de viagem e rotas [1]. Uma descrição formal deste problema irá permitir definir o critério de decisão e descobrir uma solução óptima para o problema.
O problema será analisado considerando deslocações apenas num sentido, ou seja, um condutor poderá ter uma rota desde o seu ponto de partida inicial para o ponto de chegada e uma rota diferente quando regressar ao ponto de partida inicial, resultado das restrições apresentadas no ponto 2.3.

2.1 Dados de Entrada

U - Conjunto de utilizadores com informação para identificar e qualificar os diferentes utilizadores

- *Nome*
- *Morada*
- *Destino*
- *Hora de Partida*
- Hora de chegada
- Tolerância de tempo (Restrição temporal descrita em 2.3.3)
- *Fumador* - Identifica se o utilizador é fumador ou não (Exemplo de restrição social descrita no ponto 2.3.6)
- Carro - Se possui ou não
    - Capacidade do veículo

G <V_i, E_i>- Grafo dirigido pesado, composto por:

- V - vértices ou nós que identificam moradas num mapa, com
    - Morada
    - Coordenadas
    - AdjE - conjunto de arestas que saem do vértice
- E - Arestas que representam caminhos entre um vértice de origem e um vértice de destino, com:
    - W - Peso relativo ao tempo necessário para percorrer o caminho entre os vértices ligados pela aresta (retirado do open street maps)
    - Vertice de destino
    - Vertice de origem

i - Indice que identifica cada vértice e aresta

2.2 Dados de Saída

K - Caminho numa sequência ordenada de arestas a visitar na rota
O - Total de passageiros transportados na rota
St - Hora de partida final
Et - Hora de chegada final
S - Capacidade do carro

2.3 Restrições

2.3.1 Utilizadores

Dois tipos de utilizadores podem ser identificados neste problema, condutores e passageiros. O algoritmo deverá tentar agrupar utilizadores, para maximizar a ocupação do veículo durante o percurso. Cada grupo será constituído por passageiros e um condutor com diferentes pontos de partida e paragem na mesma rota.
Uma versão mais avançada do algoritmo poderá tratar condutores de uma forma mais flexível, podendo o condutor ser incluído num grupo já existente e ser convertido em passageiro ou então ser o condutor de um novo grupo. Esta flexibilidade será um extra do algoritmo, apenas implementado se houver possibilidade para tal.

2.3.2 Capacidade dos carros

A capacidade dos carros é uma restrição importante para formar os grupos de condutor e passageiro. A capacidade de cada carro não pode ser excedida e deve-se tentar ocupação máxima em todas as viagens.

2.3.3 Intervalos temporais e tempos de viagem

Cada utilizador deverá registar uma hora de partida e uma hora de chegada e um intervalo de tolerância para cada horário. O cálculo da rota deverá ter em consideração estas restrições e garantir que cada passageiro e cada condutor efectuam viagens dentro destas janelas temporais.
Na figura 1, apresenta-se uma definição da janela temporal:

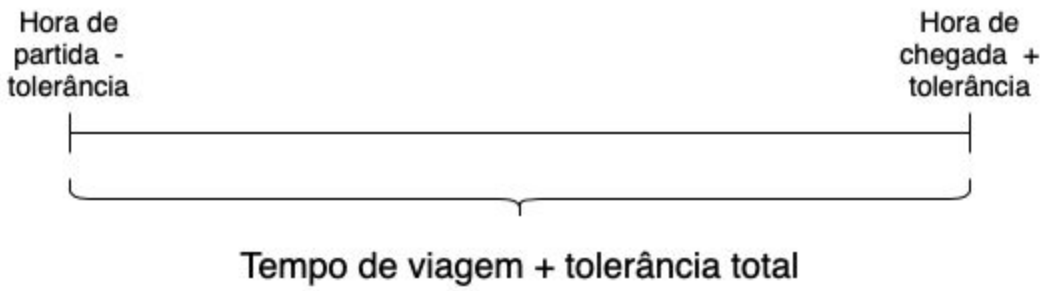

Figura 1: Esquema da janela temporal de cada trajecto.

2.3.4 Localizações (Pontos de partida e chegada)

Cada utilizador deverá escolher um ponto de partida e um ponto de chegada. No âmbito deste trabalho estas localizações serão dadas por coordenadas X e Y (nós num grafo) que serão marcadas num mapa e depois unidas com possíveis rotas (arestas num grafo) com peso dado pelo tempo de viagem.

2.3.5 Tempo total de viagem

O tempo e a sua minimização será utilizado como factor de qualidade. A distância não será utilizada, como sugerido noutros problemas deste tipo [2].
Para simplificação do problema serão consideradas velocidades médias constantes em todos os trajectos, o que pode ser convertido para tempo de forma proporcional. Para além disso, evita-se outras variáveis como trânsito e percursos em auto estrada, que iriam apenas introduzir maior complexidade sem grande valor para o âmbito deste projecto.
Desta forma a criação de um grupo de viagem deverá ter em consideração qual o caminho que reduza o tempo total. Tomando como exemplo o grafo da figura 3, onde se representa a rota de um condutor e dois possíveis passageiros, a rota preferencial seria a do passageiro A.

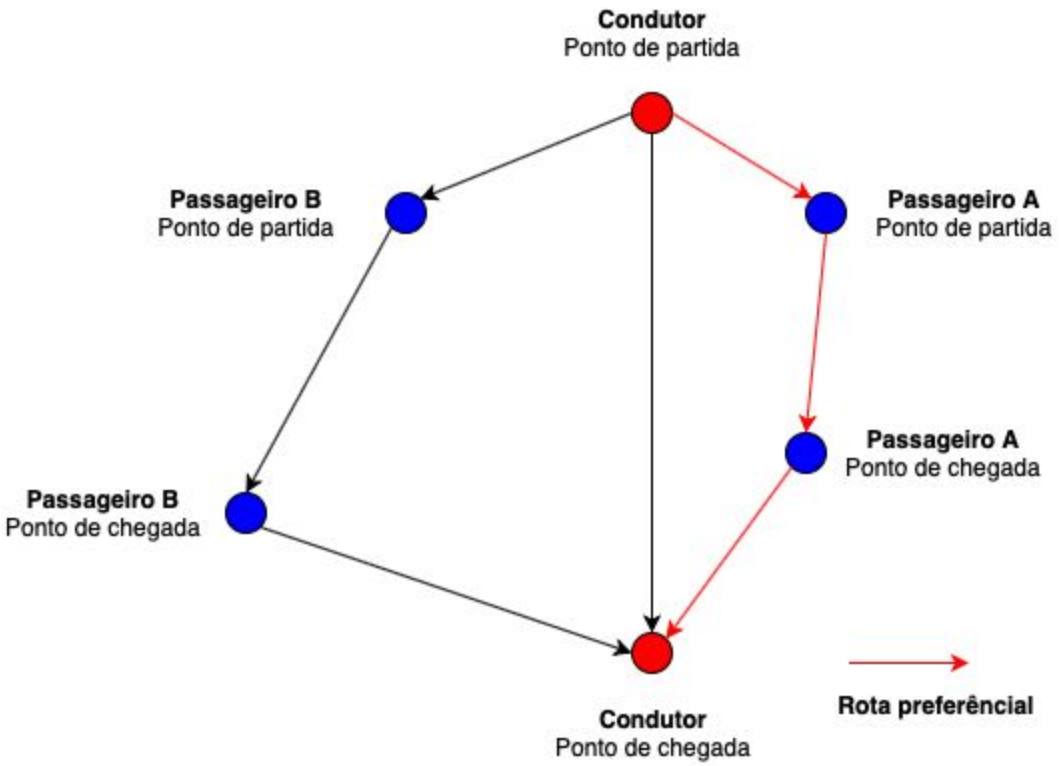

Figura 3: Escolha da rota preferencial com base na distância total.

2.3.6 Preferências sociais

Uma característica interessante deste tipo de problemas é juntar pessoas com algum grau de afinidade para tornar as viagens mais agradáveis. Vários critérios podem ser utilizados para definir proximidade entre os utilizadores [3], no entanto, neste trabalho será apenas utilizado o critério de fumadores e não fumadores.
Como funcionalidades extra podem ser considerados critérios como profissão ou género, mas estas opções, para evitar o aumento de complexidade, estão fora do âmbito deste projecto e apenas serão implementadas se tal se proporcionar durante a fase de desenvolvimento.

2.4 Função Objectivo

A solução óptima consiste em escolher caminhos onde se obtenha ocupação máxima dos veículos, se percorra a rota em menor tempo possível respeitando as restrições temporais de cada passageiro e conciliando as suas preferências sociais. A função objectivo consiste então em: Maximizar a função f(total de passageiros) e minimizar a função g(duração total da viagem), sendo f e g;

f = |O|

g = $\sum_{e \in K}$ W(e)

A função f será privilegiada em relação à função g.

3. Perspectiva de solução

3.1 Identificação das técnicas de concepção

Sendo este problema um problema de minimização de tempo de viagem e maximização do número de ocupantes, os algoritmos gananciosos representam as técnicas de concepção mais aplicáveis.
O algoritmo tentará em cada instante escolher a solução óptima, neste caso, trajecto com menor tempo e adicionar passageiros ao carro até obter a ocupação máxima, diminuindo o número de trajectos efectuados sem o máximo de ocupantes.
De uma forma geral, a técnica de concepção passará pelos seguintes passos:

1. Obter um conjunto de pontos de paragem válidos para o trajecto do condutor;
2. Uma função de seleção que escolherá o melhor próximo ponto de paragem, nomeadamente ligado pela aresta com o menor tempo e onde se possa adicionar um passageiro à viagem;
3. Uma função de viabilidade que determina se o ponto escolhido poderá efectivamente fazer parte da solução. Neste caso se se conseguem cumprir as restrições temporais e sociais do grupo;
4. Uma função objectivo que atribui um valor qualitativo à solução encontrada;
5. Uma avaliação para determinar se foi atingida a solução completa do problema, ou seja, condutor foi do ponto de partida até ao ponto de chegada, apanhando possíveis passageiros pelo caminho e deixando todos nos seus destinos dentro dos prazos estabelecidos.

Esta solução deve constantantemente avaliar o trajecto individual de cada passageiro, para validar se será possível levá-lo do ponto de partida ao de chegada no tempo estabelecido. Para estas análises intermédias, a aplicação de técnicas de divisão e conquista serão uma mais valia. O problema será dividido em subproblemas permitindo chegar a uma solução final de forma mais eficiente e que pode ser feita de forma paralela.

3.2 Principais algoritmos a implementar

Como este projeto irá fazer uso de um grafo pesado dirigido, o algoritmo ganancioso usado para encontrar o melhor caminho, de um vértice para todos os outros, será o algoritmo de Dijkstra usando uma fila de prioridades. No entanto, no âmbito deste projeto não se deve apenas calcular a rota mais rápida de um ponto A a um ponto B, mas também tentar ocupação máxima dos veículos. Enquanto o carro não estiver totalmente ocupado, passageiros que satisfazem os requisitos temporais terão que

ser considerados da seguinte maneira usando algoritmos gananciosos e/ou algoritmos de divisão e conquista:

1. Calcular a duração total da viagem do condutor sozinho.
2. Eliminar passageiros em que o seu ponto de recolha não cumpra restrições.
3. Calcular a duração da viagem do condutor com cada um dos passageiros, eliminar passageiros que não cumprem requisitos.
4. Selecionar passageiro que adicione menos peso à viagem, a não ser que existam restrições sociais.
5. Guardar o destino do passageiro, avançar para o seu ponto de recolha.
6. Se o carro não estiver cheio, verificar se existe um ponto de recolha mais perto que o destino do passageiro no carro, se existir repetir os passos anteriores mas agora com um passageiro já no caro. Caso contrário avançar para o destino. E então verificar os pontos de recolha até não ter mais passageiros possíveis ou a restrição não permitir mais passageiros.

Como consequência o algoritmo Dijkstra poderá ter que ser usado várias vezes para calcular uma rota tentando sempre o caminho mais rápido possível. Uma fila de prioridade será usada para guardar os passageiros a testar em cada ponto estando na cabeça passageiros sem restrições sociais. A conectividade da rota terá que ser verificada para evitar destinos inacessíveis.

4. Casos de utilização

A partilha de boleia tem um objectivo concreto que é relacionar condutores com passageiros à procura de viagem de forma a que todos possam chegar a um destino de forma eficiente e reduzindo custos.
Aplicações que implementam esta funcionalidade podem ser utilizadas de diversas formas:

- Juntar estudantes que se deslocam para a faculdade vindos de diferentes pontos da mesma cidade ou região;
- Juntar pessoas que se deslocam para o trabalho, que pode ser ou não o mesmo local de trabalho;
- Juntar pessoas a deslocarem-se para diferentes pontos de uma cidade ou região.

Estas aplicações têm ainda uma componente social e ambiental bastante elevada, servindo para aproximar pessoas com gostos semelhantes e optimizando a utilização de veículos nas estradas.

4.1 Funcionalidades

1. Os utilizadores podem-se registar na aplicação.
2. Os utilizadores podem registar carros na aplicação e definir a capacidade do mesmo, mostrando a sua disponibilidade para dar boleia a outros utilizadores.
3. Utilizadores sem carro, podem pedir boleia de um ponto A para um ponto B e definir restrições temporais para a partida e para a chegada.
4. Os utilizadores serão agrupados com base não só no seu trajecto mas também com base em preferências sociais, concretamente se é fumador ou não.
5. Obtenção de rotas rápidas priorizando a ocupação completa dos carros para uma maior poupança.
6. Apresentação das rotas no mapa com recurso a aplicações gráficas, GraphViewer e OpenStreetMaps.

5. Estruturas de dados utilizadas

A estrutura principal utilizada é um objeto que implementa o grafo com os pontos de interesse. Esta estrutura está representada pela classe “Graph” que implementa as seguintes estruturas:

- Vertex: Representa um nó ou vértice do grafo
- Egde: Representa uma aresta do grafo

Cada grafo possui uma lista (implementada por um std::vector em C++) com todos os Vertex do grafo, fornecendo também algumas funções utilitárias tais como:

- Encontrar um vertex: findVertex();
- Adicionar Vertex: addVertex();
- Adicionar uma aresta: addEdge();

Cada Vertex, por sua vez, guarda uma lista (implementada por um std::vector de C++) de todas as arestas que saem desse ponto. Esta estrutura tem ainda elementos úteis para a aplicação do algoritmo de caminho mais curto de Dijkstra, no qual se baseia este trabalho, são elas:

- dist: Guarda o custo mínimo para ir do ponto inicial até este ponto.
- path: Guarda qual o Vertex imediatamente antes no caminho entre o nó inicial e este ponto

A estrutura Edge, guarda o Vértice de destino da aresta, na variável dest e o peso associado à aresta na variável weight.

Outras estruturas auxiliares são utilizadas, são elas as classes User, OSMNode e OSMCollection.

Cada objecto da classe User, guarda informação sobre um utilizador da plataforma, seja ele um condutor ou um passageiro. Neste objecto, podem-se encontrar informações como: nome, ponto de partida, ponto de chegada e respectivos horários e tolerância e ainda a indicação se é fumador e se é condutor e quantos lugares tem disponível no carro.

A classe OSMNode guarda informação sobre cada um dos nós do grafo. É com esta estrutura que se criam os vertices do grafo. Cada OSMNode guarda um identificador único do nó, a posição (latitude e longitude) e o nome do ponto. Os vertices utilizados no grafo utilizam o identificador do OSMNode.

A estrutura OSMCollection funciona como uma classe utilitária para iterar sobre a lista de nós e de arestas. Nesta classe existe um mapa com pares id de nó e nó, que permite obter nós directamente pelo id, facilitando assim a consulta dos dados associados a cada nó. Existe também um vector com todas as arestas, que igualmente permite obter de forma simples as ligações de cada aresta. Estas duas colecções são mapeadas diretamente pelos ficheiros de dados e podem depois ser utilizados para gerar o grafo de forma rápida e simples.

6. Conectividade do grafo

A conectividade do grafo é testada usando um algoritmo de busca em profundidade, ou “Depth first search”. Para verificar se o grafo é fortemente conexo, ou seja para qualquer vértice existe um caminho de qualquer outro vértice, primeiro é feito uma busca em profundidade no grafo em que todos os vértices visitados são colocados num vector. O grafo é então invertido e o mesmo é feito, se o número de vértices de qualquer um dos vectores for inferior ao número de vértices no grafo nesse caso o grafo não é fortemente conexo.

No grafo usado na criação do programa, usando os dados do Porto fornecidos no moodle podemos ver que não é fortemente conexo. Para tentar minimizar a computação de caminhos que podem não gerar resultados, durante a fase de recolha de dados dos utilizadores é gerado o caminho mais curto entre o ponto de início e o ponto de fim indicados, para se verificar se tal caminho é possível e se pode ser feito dentro das restrições temporais. Se tal não for possível, o utilizador é rejeitado. Com este pré processamento inicial, garante-se que para cada utilizador está associado um caminho possível entre nós conectados.

O pré processamento faz ainda uma selecção de utilizadores com caminhos possíveis relativamente aos pontos de início e de fim dos condutores, o que faz com que só calcule rotas possíveis.

A informação sobre a conectividade do grafo está disponivel no menu “Graph Information”.

7. Casos de uso implementados

Este trabalho utiliza como dados um conjunto de nós retirados do open street maps, divididos por diversas cidades com dimensões diferentes, desde cidades mais pequenas com cerca de 3.000 nós a cidades maiores com quase 100.000 nós.
Para cada cidade é possível registar condutores e passageiros, escolhendo entre os nós disponíveis. Durante o registo o utilizador indica os dados que foram já descritos ao longo do relatório: Nome, pontos de partida e de chegada, horário de partida e de chegada, tolerância, número de passageiros por carro e se o utilizador é fumador ou não.

Com esta informação, é então possível juntar condutores e utilizadores à procura de boleia dentro de cada cidade, garantindo-se que todos chegam aos seus destinos dentro das restrições indicadas. Aplica-se ainda uma métrica de conexão social, tentando juntar utilizadores fumadores no mesmo veículo.

Havendo apenas um factor social em estudo neste trabalho, o caso de uso principal, consiste em optimizar a utilização dos recursos automóveis dentro da mesma cidade, ou seja, o conceito base de ridesharing, juntar condutores e passageiros em deslocação para locais dentro da mesma área. É também garantido que todos os utilizadores fazem as suas viagens dentro dos tempos indicados e tenta-se juntar pessoas que fumem.

Pedindo mais informação a cada utilizador, seria possível gerar rotas com pessoas com um maior grau de conectividade, evoluindo a aplicação para um conceito de ridesharing e de rede social, mas tal estava já fora do âmbito deste trabalho.

8. Algoritmos implementados (Pseudo-Código)

Durante o processo de registo de utilizadores é feito um algoritmo de caminho mínimo de Dijsktra com fila de prioridades, para verificar se o caminho pedido é viável. Desta forma faz-se uma primeira filtragem, reduzindo o número de pontos de interesse a considerar.
O algoritmo de Dijsktra implementado está esquematizado no pseudo código 1.

```
G = (V,E), V = vertices, E = edges
shortestPath(G, s), s=startNode belongs to G
foreach (v in V) {
  v->distance = INF
  v->path = NULL //nearest vertex
}
s->distance = 0;
Q <- empty //create empty priorityQueue
insert(Q, (s,0)) //Insert start node with key 0
While (!Q.empty()) {
  v <- Extract vertex with min-key from Q
  foreach (w in Adj(v)) { //Adj(v) = set of adjacent vertices of v
    if (w->distance > v->distance + weight(v,w)) {
      w->distance = v->distance + weight(v,w)
      w->path = v
      if (w not in Q) { //The previous distance of w was INF
        insert(Q, (w, dist(w)))
      } else {
        decrease_key(Q, (w, dist(w))) //w now has a better path
      }
    }
  }
}
```

Pseudo-Código 1: Algoritmo de Dijskstra com fila de prioridade

Com esta validação prévia obtém-se a garantia que todos os utilizadores possuem um caminho possível, evitando verificações futuras e aumento de complexidade no algoritmo principal. Para além disso, este cálculo pode ser reutilizado, retirando necessidade de computação do algoritmo.

Foram desenvolvidas duas abordagens para o problema, uma mais rápida em termos computacionais mas que não devolve a solução óptima e outra um pouco mais lenta mas que tenta otimizar tanto o número de pessoas no carro como minimizar o tempo de viagem. Os algoritmos: rideshareFast e rideshareBest serão apresentados de seguida.

O algoritmo rideshareFast, menos complexo e menos eficiente, é descrito pelo pseudo-código 2.

```
G = G(V,E), V = Vertexes; E = Edges
U = Set of users (both drivers and passengers)
d = Driver
T = final calculated travel time
P = Set of passengers to be selected by the algorithm (set empty)
rideShareFast(G, U, d, t, P) {
  Pot <- empty (set of potential passengers)
  totalTravelTime = 0
  foreach(u in U) {
     shortestPath(G, u->startNode) <- best route from user start to user
end
     shortestPath(G, u->endNode) <- best route from user end node
     if ( u can be added to trip) {
      insert(Pot, u)
     }
   }
   if ( Pot.empty() ) {
     return d->shortestPath, only drivers path
   }

   foreach(pot in Pot) {
      Res <- empty, list of nodes to visit
      append(P, pot) <- greedy adds the potencial passenger to the end of
the list
      pot->WAITING <- the passenger is waiting for the ride
      insert(Res, pot->startNode)
      foreach(res in Res) {
         shortestPath(G, pot->startNode)
         bestWeigth = 0;
         bestUser = NULL;
         for (pot in Pot)
            if (pot->WAITING && carNotFull ) {
              uPickUpTime = timeToFrom(pot->start, d->start)
```

```
              if(uPickUpTime < pot->earliestStart) {
                  pot->weigth =pot->earliestStartTime -
d->EarliestStartTime;
              }
              if (pot->weight < bestWeight) {
                 bestWeight = pot->weight
                 bestUser = pot
              }
         }

         if (bestUser == NULL) {
           removeAll(Res, pot->IN_TRAVEL) <- remove all potential already
in travel state
         } else {
           bestUser->path = empty
           if (bestUser->Waiting) {
             set(bestUser, IN_TRAVEL)
             bestUser->path = from(currentPos to bestUser->startNode)
             --carCapacity
           } elseIf (bestUser->IN_TRAVEL) {
             set(bestUser, ARRIVED)
             bestUser->path = from(currentPos to bestUser->endNode)
             ++carCapacity
           }
           if (bestUser.empty()) {
             res.clear()
             return;
           }

           totalTravelTime += bestWeight
         }
          append(RES, bestUser->path)
      }

      if (RES.empty() OR totalTravelTime + d->EarliestStart >
d->LatestArrivalTime) {
         remove_last(P)
      } else {
         d->path = bestUser->path
      }
   }
```

```
Return d->path;
```

Pseudo-Código 2: Algoritmo rideShareFast

Este algoritmo começa por criar uma lista de potenciais utilizadores que possam ser incluídos na viagem do condutor, atendendo à conectividade dos diferentes pontos de início e de fim e aos tempos estabelecidos para as viagens. Com esta lista, o primeiro elemento da lista é avaliado e se o caminho for viável o utilizador é adicionado ao carro e o seu caminho anexado ao caminho original. Os próximos elementos da lista já serão avaliados atendendo à presença dos utilizadores no carro. Os utilizadores vão sendo incluídos e removidos do carro durante todo o tempo de viagem do condutor. Este algoritmo é ganancioso, pois adiciona logo o primeiro elemento da lista, mesmo que este não seja o melhor.

O algoritmo rideShareBest, segue a lógica apresentada no pseudo-código 3.

```
G = G(V,E), V = Vertexes; E = Edges
U = Set of users (both drivers and passengers)
d = Driver
T = final calculated travel time
P = Set of passengers to be selected by the algorithm (starts empty)
rideShareBest(G, U, d, t, P) {
  Pot <- empty (set of potential passengers)

  foreach(u in U) {
     shortestPath(G, u->startNode) <- best route from user start to user
end
     shortestPath(G, u->endNode) <- best route from user end node
     if ( u can be added to trip) {
      insert(Pot, u)
     }
   }
   if ( Pot.empty() ) {
     return d->shortestPath, only drivers path
   }

   i = 0
   while (i == P.size()) {
      ++i
      bestTime = INF
```

```
      bestUser = NULL

   foreach(pot in Pot) {
      Res <- empty, list of nodes to visit
      insert(P, pot) <- greedy adds the first potencial passenger
      pot->WAITING <- the passenger is waiting for the ride
      insert(Res, pot->startNode)
      foreach(res in Res) {
         shortestPath(G, pot->startNode)
         bestWeigth = 0;
         bestUser = NULL;
         for (pot in Pot)
            if (pot->WAITING && carNotFull ) {
              uPickUpTime = timeToFrom(pot->start, d->start)
              if(uPickUpTime < pot->earliestStart) {
                  pot->weigth =pot->earliestStartTime -
d->EarliestStartTime;
              }
              if (pot->weight < bestWeight) {
                 bestWeight = pot->weight
                 bestUser = pot
              }
         }

         if (bestUser == NULL) {
           removeAll(Res, pot->IN_TRAVEL) <- remove all potential already
in travel state
         } else {
           bestUser->path = empty
           if (bestUser->Waiting) {
             set(bestUser, IN_TRAVEL)
             bestUser->path = from(currentPos to bestUser->startNode)
             --carCapacity
           } elseIf (bestUser->IN_TRAVEL) {
             set(bestUser, ARRIVED)
             bestUser->path = from(currentPos to bestUser->endNode)
             ++carCapacity
           }
           if (bestUser.empty()) {
             res.clear()
             return;
           }
```

```
           totalTravelTime += bestWeight
         }
          append(RES, bestUser->path)
      }

      if (RES.empty() OR totalTravelTime + d->EarliestStart >
d->LatestArrivalTime) {
         remove(P, pot)
      } else if (d->bestTravelTime < bestTime){
         bestPath = bestUser->path
         bestUser = pot <- users only added at the end
      }
      remove_last(P)
    }
    if (bestUser != NULL) {
       insert(P, bestUser)
       remove(Pot, bestUser)
    }
  }
  return bestPath;
}
```

Pseudo-Código 3: Algoritmo rideShareBest

O algoritmo rideShareBest testa todos os potenciais utilizadores para verificar qual o que origina o caminho mais eficiente e só depois é que o primeiro utilizador é escolhido. Depois de adicionado este utilizador à viagem é feita nova procura pelo próximo melhor passageiro.

9. Complexidade dos algoritmos

Os dois algoritmos apresentados obtêm resultados satisfatórios, no entanto, o rideShareFast é mais rápido a calcular a rota final. Isto deve-se à propriedade gananciosa deste algoritmo, pois um passageiro viável é logo adicionado à viagem. Isto reduz o número de ciclos de computação do algoritmo mas traduz-se em viagens não óptimas. Contudo, este aspecto não é necessariamente negativo, principalmente se o grafo em análise tiver muitos nós e muitos utilizadores. O algoritmo rideShareBest percorre a lista de utilizadores várias vezes, para perceber qual o passageiro que irá adicionar menos peso na viagem. Apesar de conduzir a um resultado mais otimizado que o algoritmo anterior, o tempo de cálculo é bastante superior. Este algoritmo é muito interessante para grafos de pequenas dimensões mas bastante custoso para grafos grandes. É essencial para este algoritmo haver uma filtragem prévia de utilizadores, para o algoritmo processar apenas passageiros que efetivamente possam ter uma possibilidade de ser incluídos na viagem.

Nas imagem seguinte é feita uma comparação de execução entre os dois algoritmos para um grafo de 10.765 nós, com 4 utilizadores:

```
rideshareFast
Path      -  Description
349907698 - Driver 'testDriver' departure
137994583 - User 'testJoao' pickup
129548927
137992808
137994720
137992807
137993129
129548930
349907649 - User 'testJoao' destination
129548929
473873465
129548927
137992808
137994720
137992807
137993129
473873457 - Driver 'testDriver' destination
Time: 52
Execution time(ms): 10
```

Imagem 4: Execução do algoritmo rideshareFast com 4 utilizadores

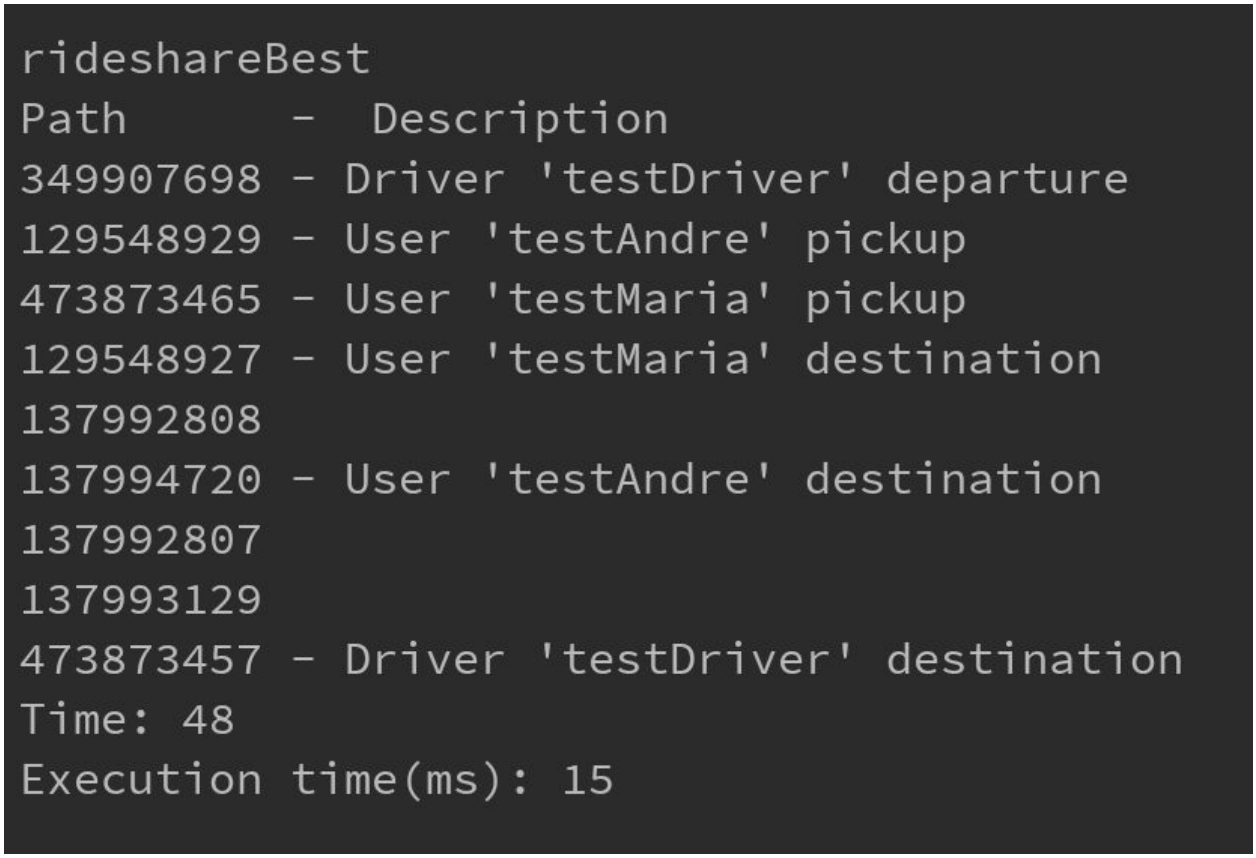

Imagem 5: Execução do algoritmo rideshareBest com 4 utilizadores

Pelas imagens é possível ver que o algoritmo rideshareFast foi bastante mais rápido, cerca de 30% mais rápido que o rideshareBest, mas o caminho obtido demora mais tempo e apenas um passageiro foi incluído na rota. O algoritmo rideshareBest apesar de mais lento, obteve um caminho mais curto e incluiu dois passageiros na viagem. É interessante analisar que no algoritmo rideshareFast o passageiro incluído não foi escolhido no algoritmo rideshareBest, pois iria originar um caminho menos eficiente.
Pode-se concluir que quanto mais nós existirem no grafo e mais utilizadores registados maior será a diferença entre o tempo de execução dos dois algoritmos e também a diferença de optimização das viagens originadas.

5. Conclusão

Neste trabalho foi feita a análise de um problema de partilha de viagens, com vista à optimização da utilização dos recursos automóveis. Foram analisadas várias restrições deste tipo de problema que permitiram uma melhor definição do objectivo final e a proposta de uma abordagem possível para a solução.

Durante a análise das restrições, foram ainda feitas algumas simplificações no problema geral, de forma a manter o problema com a dimensão correcta para o âmbito da disciplina. Nomeadamente, simplificou-se o cálculo de pesos para cada trajecto, assumindo que os carros viajam a velocidades constantes e que portanto a distância terá uma relação directa com o tempo, fazendo com que apenas seja necessário minimizar a variável tempo.

Das restrições apontadas para o problema pode-se destacar a restrição temporal (secção 2.3.3) que será fundamental para avaliar a viabilidade da inclusão de passageiros em carros. Foi ainda incluída uma restrição social, o facto de os condutores serem ou não fumadores, para se explorar o conceito de agrupamento de pessoas com base em preferências. Muitas outras preferências poderiam ter sido incluídas e até uma combinação de várias, como, género, língua, raça, gostos pessoais, profissão, etc. No âmbito deste trabalho utilizar uma é suficiente e mostra como se pode utilizar esta variável.

Este problema foi modelado como um grafo onde os vértices são os pontos de paragem e de chegada e as arestas os caminhos entre cada um desses pontos, apresentando como peso o tempo que demora a percorrer esse caminho. A solução do problema passará então por encontrar um trajecto que leve um condutor do seu ponto de partida para o ponto de chegada e que durante este caminho o condutor possa levar passageiros, de forma a que todos cheguem aos seus destinos dentro do tempo definido e tentando sempre que a ocupação do veículo seja máxima e o tempo de viagem curto. Isto traduz-se em poupança e preservação ambiental. Para atingir este objetivo, foram propostas técnicas de análise do problema baseadas em algoritmos gananciosos e de divisão e conquista implementados juntamente com o algoritmo de Djisktra com filas de prioridade. Foram implementados dois algoritmos, um que privilegia o tempo de execução e outro a qualidade dos caminhos finais originados.

O algoritmo com tempo de execução mais baixo, utiliza uma técnica gananciosa, escolhendo o primeiro passageiro compatível com a viagem do condutor. A inclusão deste passageiro na viagem pode levar a uma viagem mais longa e não proporcionar a que a ocupação máxima do veículo seja atingida, no entanto, devido a esta simplificação, o algoritmo apresenta uma solução num curto espaço de tempo e com reduzidos recursos computacionais. Esta simplificação é muito interessante para conjuntos de grafos e utilizadores de grandes dimensões. É também importante quando o acesso a recursos de computação é limitado.

O segundo algoritmo implementado, é mais exigente a nível de computação e demora mais tempo a fornecer a rota final, mas, a rota é optimizada tanto a nível de tempo de viagem como de número de ocupantes no carro.

5.1 Esforço dedicado

Todos os elementos do grupo estiverem envolvidos em todas as fases e decisões de desenvolvimento. O esforço foi igualitário.

6. Bibliografia

1. Nelson D. Chan, Susan A. Shaheen: **Ridesharing in North America: Past, Present, and Future** In *Transport Reviews*, Vol. 32, No. 1, 93-112, January 2012
2. Alan Erera, Martin **Savelsbergh: Dynamic Ride-Sharing: A Simulation in Metro Atlanta** In *Elsevier - Procedia Social and Behavioral Sciences* 17, 2011
3. Lukas Mohs: **Exploration of Ride-sharing Opportunities at the TUM Garching Campus** In *Master Thesis In Information Systems*, Technische Universitat Munchen, April 2018
4. R.Rossetti, L.Ferreira, L.Teófilo, J.Filgueiras, F.Andrade: **Slides das aulas teóricas de Concepção e análise de algoritmos**, 2018-2019
5. Timothy Brown: WEB(https://eng.lyft.com/matchmaking-in-lyft-line-9c2635fe62c4) , MatchMaking in Lyft Line
6. C. Chitra, P. Subbaraj: **Multiobjective optimization solution for shortest path routing problem**, 2010