Faculdade de Engenharia da Universidade do Porto

Relatório Projeto CAL

Parte 1

RideSharing: partilha de viagens

11 abril 2018

Grupo 10 – Turma 2

Gonçalo Santos	201603265	up201603265@fe.up.pt
João Vieira	201603190	up201603190@fe.up.pt
Susana Lima	201603634	up201603634@fe.up.pt

Conteúdo

Introdução

No âmbito da unidade curricular de Conceção e Análise de Algoritmos do 2º ano do MIEIC foi proposta a criação de uma aplicação que gerencie a partilha de viagens (*RideSharing*) entre passageiros e condutores e algoritmos que a sustentem da forma mais eficiente possível usando os conhecimentos adquiridos nas aulas.

Neste relatório são descritos o problema e a melhor estratégia encontrada para a sua resolução.

Explicação do problema

O tema do projeto incide sobre *RideSharing*, partilha de viagens cujo objetivo seria a otimização do recurso automóvel, evitando que este se desloque com apenas um ocupante.

A ideia básica por detrás deste conceito é a partilha de automóveis por diferentes pessoas cuja origem e destino de viagem coincida parcial ou totalmente com as de um condutor.

O programa desenvolvido permite o cálculo de percursos mais curtos que permitam maximizar o número de ocupantes de um veículo entre uma origem e um destino, tendo em consideração as restrições de tempo impostas tanto pelo condutor como pelos possíveis passageiros, bem como as horas de chegada aos pontos de encontro.

Foi considerada também a familiaridade entre passageiros, ou seja, pessoas que viagem em conjunto.

Descrição do problema

O problema em questão foi decomposto em quatro iterações:

1ªiteração: percurso mais rápido

Numa primeira fase, o objetivo tratou-se simplesmente em calcular a rota mais curta entre a origem e o destino do condutor. Deste modo, os passageiros transportados seriam aqueles cujos caminhos mais curtos (entre as origens e destinos dos mesmos) estivessem contidos de forma parcial ou total no trajeto mais rápido do condutor.

É importante notar que o percurso só pode ser calculado caso existam vias de ligação, diretas ou indiretas entre a origem e o destino considerados.

2ªiteração: percurso mais rápido que maximize o número de passageiros sem restrições temporais e de capacidade

Numa segunda fase, o cálculo do caminho teve em consideração o número de pessoas a transportar, sendo que o objetivo passou a ser o caminho mais curto que permitisse o transporte do máximo número de pessoas.

É de notar que a maximização do número de passageiros foi privilegiada sobre a minimização do tempo de viagem.

A capacidade do veículo do condutor bem como qualquer tipo de condicionamentos temporais foram desprezados.

3ªiteração: percurso mais rápido que maximize o número de passageiros com restrições temporais e de capacidade

Numa terceira fase, foram consideradas algumas restrições, nomeadamente a capacidade do veículo em cada ponto de recolha de passageiros, o limite de tempo, imposto pelo condutor e pelos passageiros, de chegada ao destino e as horas de comparência nos pontos de encontro.

Assim sendo a rota calculada teria de ter uma duração menor ou igual ao limite de tempo definido pelo motorista. Do mesmo modo, só seriam recolhidas pessoas cujo limite de tempo imposto não fosse excedido.

Relativamente ao limite de capacidade, caso o automóvel se encontrasse completamente ocupado ou não tivesse lugares disponíveis para acomodar todos os elementos de um grupo os passageiros não seriam transportados.

4ªiteração: alargamento do critério de escolha de passageiros

Numa quarta fase, a escolha de passageiros foi alargada de modo a que pessoas cujo trajeto mais curto não estivesse contido no itinerário do condutor passassem a ser transportada caso os pontos de origem e de destino estivessem contidos, tendo sempre em consideração as restrições impostas

Formalização do problema

O programa implementado permite, de uma forma geral, resolver o problema de *ride sharing* recebendo um ponto de partida, com a respetiva hora, e um ponto de chegada, um limite de tempo máximo para o percurso a efetuar, atendendo à capacidade do veículo, caso o utilizador se trate de um condutor ou o número de pessoas a viajar em grupo, caso o utilizador seja um passageiro.

Dados de entrada

Pi – Sequência de passageiros que aguardam transporte, sendo Pi(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- nump - Número de pessoas com quem viaja
- srcp - Ponto de partida
- destp - Ponto de chegada
- hip - Hora de partida
- tlp - Limite de tempo

Ci – Sequência de condutores disponíveis, sendo Ci(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- cap - Capacidade do automóvel que conduz (excluindo o lugar do condutor)
- srcc - Ponto de partida
- destc - Ponto de chegada
- hic - Hora de partida
- tlc - Limite de tempo

Gi(Vi,Ei) – grafo dirigido pesado composto por:

- V - vértices - representam todos os pontos de encontro
- E - arestas - representam vias de ligação entre os pontos (estradas, ruas) com:
    - wt - corresponde ao tempo de viagem entre os dois vértices que a delimitam
    - wp - corresponde ao número de pessoas que aguardam transporte entre os dois vértices que a delimitam

- w - peso da aresta - valor pesado entre o tempo de viagem e o numero de possíveis passageiros a aguardar transporte

As origens e destinos dos passageiros e condutores estão contidos em V.

Dados de saída

Gf(Vf,Ef) – grafo dirigido em que Vf e Ef têm os mesmos atributos que Vi e Ei, embora os valores dos mesmos podem ser diferentes. No entanto, é de referir que Ef diferencia- se de Ei caso tenham sido transportados passageiros (remoção dos passageiros que aguardavam transporte).

Cf – sequência ordenada de todos os condutores utilizados sendo Cf(i) o seu i-ésimo elemento. Cada um com:

- I - sequência de vértices a visitar entre o vértice de partida e o de chegada, correspondente ao itinerário do condutor
- tp - Número de pessoas transportadas ao longo do percurso
- hfc - Hora de chegada

Pf – sequência de passageiros, sendo Pf(i) o seu i-ésimo elemento. Pode diferir de Pi caso parte (ou todos) dos passageiros que aguardavam transporte tenham sido transportados. Cada um com:

- I - sequência de vértices visitados, caso o passageiros tenha sido transportado
- pc - posição atual, entre srcp (a pessoa não foi transportada) e a destp (a pessoa foi transportada até ao seu destino) inclusive
- hcp - hora atual, pode ou não ser igual a hip (a pessoa não foi transportada)

Restrições

Sobre os dados de entrada

- $\forall i \in [1 ; |Ci|]$, cap(Ci(i)) > 0, visto a capacidade de um veiculo se tratar de um número inteiro positivo e , no contexto do problema, não interessar um automóvel que apenas possa transportar o condutor (capacidade = 0).
- $\forall i \in [1 ; |Pi|]$, nump(Ci(i)) > 0, uma vez que o número de pessoas (representativo do número de elementos de um grupo) é um número

inteiro positivo maior que zero, visto um grupo com zero elementos não fazer sentido no problema em questão.

- Seja P = Pi U Ci, ∀ i ∈ [1 ; |P|], 18 <= idade(P(i)) <= 70, intervalo considerado relativamente realista no contexto apresentado.
- ∀ i ∈ Ei, wt > 0, dado que representa o tempo entre dos pontos de um mapa
- ∀ i ∈ Ei, wp >= 0, dado que representa o número de pessoas que aguardam transporte entre dois pontos de um mapa. Pode não existir nenhuma pessoa a aguardar transporte entre dois determinados pontos (wp = 0).
- ∀ i ∈ Ei, w > 0, consequência das duas restrições anteriormente apresentadas, uma vez que é calculado em função de wt e wp.

Sobre os dados de saída

- ∀ i ∈ [1 ; |Ci|] , i ∈ [1 ; |Cf|], ou seja, todos os condutores presentes inicialmente também estão presentes no fim, mas com diferentes valores nos atributos
- |Ci | = |Cf |, pelo motivo apresentado em cima
- ∀ i ∈ [1 ; |Pi|] , i ∈ [1 ; |Pf|], ou seja, todos os passageiros presentes inicialmente também estão presentes no fim, mas com diferentes valores nos atributos, caso sejam transportados
- |Pi | = |Pf |, pelo motivo apresentado em cima
- ∀ i ∈ [1 ; |Vi|] , i ∈ [1 ; |Vf|], o que significa que todos os vértices presentes no grafo inicial estão presentes no grafo final, embora possam diferir no valor de certos atributos
- |Vi| = |Vf| pelo motivo apresentado em cima
- ∀ i ∈ [1 ; |Ei|] , i ∈ [1 ; |Ef|], o que significa que todas as arestas presentes no grafo inicial estão presentes no grafo final, embora possam diferir no valor de certos atributos
- |Ei| = |Ef| pelo motivo apresentado em cima

Funções objetivo

A solução ótima encontrada passa por minimizar o tempo de viagem e maximizar o número de passageiros transportados durante a mesma.

f - $\sum_{c \in C} [\sum_{e \in I} wt(e)]$

g - $\sum_{c \in C} c(tp)$

No programa implementado foi privilegiada a maximização do número de passageiros sobre a minimização do tempo, como é descrito na explicação do algoritmo implementado.

Descrição da solução

Estruturas de dados utilizadas

Representação de um grafo adequado ao problema

Vertex

A classe Vertex representa um vértice de um grafo. No contexto apresentado representa uma via de ligação entre dois pontos do mapa. Cada vértice é caracterizado por:

- vertexId – identificador único de um vértice. Este é atributo tem relevância na implementação do GraphViewer
- lastVertexId – permite a incrementação estática do vertexId cada vez que um novo vértice é construído. Armazena a informação do último vertexId criado
- info – informação única de um vértice. Funciona como outro tipo de identificador
- x – coordenada x do vértice. Relevante na implementação do GraphViewer
- y – coordenada y do vértice. Relevante na implementação do GraphViewer
- adj – vetor de apontadores para as arestas adjacentes ao vértice
- notConnected – vetor de apontadores para os vértices que não estão ligados a este vértice
- distance – soma total dos pesos das arestas entre o vértice original e o vértice atual
- time – tempo total entre o vértice de partida e o vértice atual
- previous – apontador do vértice do qual se partiu para se chegar ao vértice em questão
- queueIndex – auxiliar que permite a utilização de *MutablePriorityQueue*[1]
- pickedUp – vetor de apontadores para os passageiros que foram apanhados na aresta anterior (durante o algoritmo)

Edge

- dest – apontador para vértice de destino desta aresta
- weight – peso da aresta
- numP – número de pessoas que aguardam transporte na aresta
- waiting – vetor de apontadores para os passageiros que aguardam transporte na aresta

[1] Classe criada pelos docentes de CAL

- edgeId – identificador único de uma aresta. Este atributo tem especial relevância na implementação do GraphViewer

Graph

- vertexSet – vetor de vértices pertencentes ao grafo

Gestão de passageiros e condutores

Person

A classe Person representa, como o nome indica, uma pessoa no contexto do problema. Como tal, uma pessoa ("person") é caracterizada por:

- name – nome da pessoa
- age – idade da pessoa, valor compreendido entre 18 e 70 anos, intervalo considerado apropriado no contexto apresentado
- startTime – hora de chegada ao ponto de partida
- currentTime – hora "atual"
- timeLimit – tempo máximo imposto de chegada ao destino, este tempo tem de ser superior a zero

Driver

A classe Driver representa um condutor. Esta classe deriva da classe Person, uma vez que um condutor é uma pessoa sendo que possui os atributos que a caracterizam. Cada condutor é contém:

- capacity – capacidade de transporte de passageiros do veículo que conduz
- passengers – vetor que armazena apontadores para os passageiros transportados ao longo do percurso
- passengersPickedAt – *multimap* que permite identificar que passageiros foram apanhados em certos pontos do percurso. A chave desta estrutura é um apontador para o vértice em questão e o valor mapeado é um apontador para o passageiro
- passengersDroppedAt – *map* que permite identificar que passageiros foram deixados em que pontos do caminho. A chave desta estrutura é um apontador para o vértice em questão e o valor mapeado é um apontador para o passageiro
- path – lista de apontadores para os vértices percorridos desde o ponto de origem até ao destino, corresponde ao itinerário percorrido
- transportedPassengers - número total de passageiros transportados

- source – representa o ponto de partida
- destination - representa o ponto de chegada

Passenger

A classe Passenger representa um passageiro no contexto apresentado. Esta classe deriva da classe Person uma vez que que um passageiro é uma pessoa pelo que possui os mesmos atributos. Cada passageiro é caracterizado por:

- numPassengers – número total de passageiros a serem transportados. Permite considerar a familiaridade entre pessoas que desejem viajar em grupo.
- path – lista de apontadores para os vértices percorridos desde o ponto de origem até ao ponto atual, corresponde ao itinerário percorrido
- source – apontador para o vértice de origem, corresponde ao ponto de partida
- destination – apontador para o vértice de destino, corresponde ao ponto de chegada
- pos – apontador para o vértice que corresponde à posição atual do passageiro
- prevPos – apontador para o vértice que corresponde à posição anterior do passageiro
- dropped – booleana que permite identificar se o passageiro foi deixado num ponto ou não
- picked – booleana que permite identificar se o passageiro foi recolhido ou não
- infoSource – representa a informação relativa ao ponto de partida
- infoDestination – representa a informação relativa ao ponto de chegada

RideShare

A classe RideShare é a entidade que permite fazer a gestão dos condutores e dos passageiros. Esta é caracterizada por:

- name – nome da entidade
- passengers – unordered_set que armazena apontadores para os passageiros que aguardam transporte
- drivers – unordered_set que armazena apontadores para os condutores
- graph – grafo que representa o mapa no qual os percursos vão ser determinados

Algoritmos implementados

Análise da conetividade

De forma a impedir a entrada e (tentativa de) cálculo de caminhos impossíveis, é analisada a conectividade do grafo, através de uma pesquisa em profundidade. Apenas é feita a distinção entre conexo e não conexo, sendo que não consideramos muito relevante se acaba ou não por ser fortemente conexo, uma vez que este cenário seria extremamente difícil de encontrar no mundo real.

Cálculo do caminho mais rápido que permita levar mais passageiros

O algoritmo principal no cálculo dos passageiros que acompanham os condutores e os caminhos percorridos é uma versão modificada do algoritmo de Dijkstra, desenhada para além de minimizar o tempo gasto no percurso condutor, maximizar o número de viajantes no carro ao longo do percurso. Este algoritmo tem em conta a origem e o destino dos condutores e dos passageiros, e várias outras restrições anteriormente explicadas. Os vértices são guardados numa *min-priority queue* ordenados pelo tempo à origem do condutor, por ordem crescente. Antes de visitar as arestas de cada vértice a capacidade do veículo naquele momento é recalculado, evitando assim a sua sobrelotação. Em cada aresta, calcula-se o número de passageiros que têm condições de entrar no carro. Este valor é usado no cálculo da distância ao próximo vértice (sendo *u* o vértice atual e *v* o seguinte):

$$altDist(v) = dist(u) + \frac{dist(u,v)}{passengersPicked(u)^2 + 1}$$

Caso este valor seja menor que a anterior distância de v, o limite de tempo do condutor não seja ultrapassado e o carro não esteja sobrelotado, os valores de *v* são atualizados, a chave de *v* na *min-priority queue* é decrementada e os passageiros são acrescentados ao condutor.

Por fim, quando o algoritmo chega ao nó de destino, o caminho percorrido é visitado recursivamente, os passageiros transportados são acrescentados a uma lista de passageiros e contabilizados. O local em que cada passageiro foi deixado pelo condutor é também atualizado.

Pós-Processamento

Nesta fase, são analisados todos os pontos de referência do percurso já previamente calculado e verificadas todas as suas arestas por eventuais passageiros que sejam candidatos à boleia já calculada. A prioridade é dada sempre a passageiros que viagem em maior número, tornando a viagem do condutor e passageiros previamente selecionados, mais vantajosa, completando o mais possível os lugares vagos do veículo. Importa referir que, mesmo assim, são tidas em conta eventuais restrições temporais que possam existir.

Análise da solução

No pior caso, a complexidade do algoritmo descrito é de O(D* (E + V + P)logV). A complexidade espacial é O(D + E + V + P), uma vez que são usadas referências para os objetos ao longo do algoritmo.

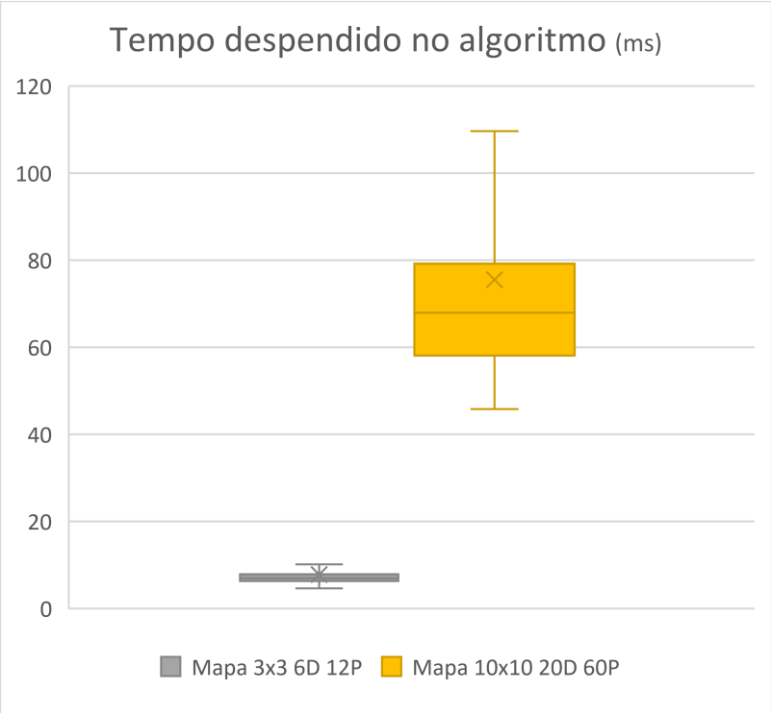

Figura 1 - Tempo despendido no algoritmo em função do mapa, número de passageiros e condutores

A solução apresentada não é ótima, uma vez que não é capaz de, por exemplo, deixar de apanhar um grupo que não lhe encha o carro para mais à frente, apanhar outro grupo que preencha a totalidade do carro. Não sendo ótima, apresenta uma complexidade temporal bastante aceitável. Num caso prático, a solução seria implementada a nível local e não é esperado que existam pontos de coleta em cada rua/esquina, pelo que o número de vértices não seria muito grande.

O algoritmo pode ser caracterizado por apresentar caminhos flexíveis, uma vez que o condutor não faz necessariamente o seu caminho mais curto da origem ao destino, podem haver desvios em função do número de passageiros, *multi-hop*, isto é, um passageiro pode fazer parte do seu percurso com um condutor e o resto/outra parte com outro e suporta múltiplos passageiros, isto é, mais do que um grupo de passageiros por veículo

Casos de utilização

A plataforma implementada permite simular o processo de gestão de um mapa e geração de percursos de *ride sharing* para diferentes condutores e passageiros. Em particular, o programa apresenta as seguintes opções e funcionalidades:

- Escolher um mapa
- Carregar os dados do mapa, dos condutores e dos passageiros a partir de ficheiros selecionados consoante a escolha do mapa
- Visualizar o mapa
- Verificar a conetividade do grafo que representa o mapa
- Adicionar novos pontos ao mapa, isto é, vértices ao grafo
- Adicionar novas vias ao mapa, isto é, arestas ao grafo
- Adicionar novos passageiros
- Adicionar novos condutores
- Guardar as mudanças efetuadas
- Calcular os percursos para todos os condutores
- Visualizar a informação dos percursos de todos os condutores
- Selecionar um condutor e visualizar a informação do seu percurso
- Selecionar um condutor e visualizar o seu percurso no mapa
- Selecionar um passageiro e visualizar a informação do seu percurso
- Selecionar um passageiro e visualizar o seu percurso no mapa

Esforço dedicado por cada elemento do grupo

A realização do projeto em questão foi possível devido ao empenho e esforço conjunto de todos os elementos do grupo, de uma forma organizada e sincronizada de modo a que não houvesse grande disparidade de contribuição entre os diferentes elementos. Cada elemento focou-se mais no módulo com que tinha mais à vontade, mas no geral o trabalho foi divido equitativamente entre todos.

Conclusão

O programa implementado permite resolver, de uma forma geral, o problema apresentado, cumprindo o propósito do trabalho. O algoritmo concebido mostrou-se eficiente, apresentando sempre tempos e resultados satisfatórios, face aos dados fornecidos.

A realização deste trabalho permitiu, não só aprofundar os conhecimentos adquiridos nas aulas teóricas e práticas, como também mostrar a aplicabilidade de certos algoritmos apresentados na unidade curricular de CAL na resolução de problemas atuais e presentes na vida quotidiana de muitas pessoas.

Referências Bibliográficas

Masoud, Neda, and R. Jayakrishnan. 2017. "A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system." *Transportation Research Part B: Methodological* 106: 218 - 236.

Apêndice A: Pseudocódigo do Algoritmo

```
function dijkstraPeopleDistancePath(Graph, source, destination, driver):
  create vertex set Q
  create vertex list Path
  create passenger list Passengers
  for each vertex v in Graph:
    v[distance] <- INFINITY
    v[previous] <- UNDEFINED
    v[pickedUp].clear()
    insert v in Q

  source[distance] <- 0
  source[time] <- 0
  ending <- destination

  while Q is not empty:
   u <- Q.extractMin()

  numberPeopleTransported <- 0
  if (u == ending)
    while (u[previous] != UNDEFINED)
      add u to Path

      for each Passenger p in u[pickedUp]:
        add p to Passengers
        add p to driver[passengers]
        p[currentTime] <- temp[time]

      if (!p[dropped])
        add p to driver[droppedAt]
        p[dropped] = TRUE
        numberPeopleTransported <- numberPeopleTransported + p[numberPassengers]

    add u to Path
    driver[PeopleTransported] <- numberPeopleTransported + driver[PeopleTransported]

    return Path, Passengers, numberPeopleTransported
  alreadyPicked <- 0
  for each Passenger p in driver[passengersPickedAt]:
    if (p[destination] != p[position] && p[destination] != u)
      alreadyPicked <- alreadyPicked + p[numberPassengers]

  for each neighbor v of u:
    lastAlreadyPicked <- alreadyPicked
    driver[currentTime] <- u[time]

    numberPassengersPicked <- 0
    create Passenger list temporaryPassengersPicked

    for each Passenger p in Edge(u, v):
      if (p[position] == u
          && (alreadyPicked + p[numberPassengers]) <= driver[capacity]
          && p[currentTime] <= driver[currentTime]
          && driver[currentTime] < p[startTime] + p[timeLimit])

        if (!p[picked])
          lastAlreadyPicked <-lastAlreadyPicked + p[numberPassengers]

        numberPassengersPicked <- numberPassengersPicked + p[numberPassengers]
        add p to temporaryPassengersPicked

      alt <- u[distance] + length(u, v) / (pow(numberPassengersPicked, 2) + 1)
      time <- u[time] + length(u, v)

    if (lastAlreadyPicked <= driver[capacity] && time <= driver[timeLimit]
```

```
65.         && alt < v[distance])
66.       v[distance] <- alt
67.       v[time] <- time
68.       v[previous] <- u
69.       v[pickedUp] <- temporaryPassengersPicked
70.
71.       Q.decreaseKey(v);
72.       for each Passenger p in temporaryPassengersPicked:
73.         p[previousPosition] <- p[position]
74.         p[position] <- v
75.
76.       if (m[picked])
77.         erase p from temporaryPassengersPicked
78.       else
79.         m[picked] <- TRUE
80.     add temporaryPassengersPicked to driver[passengersPickedAt]
```