Universidade do Porto

Faculdade de Engenharia

FEUP

Projeto de Concepção e Análise de Algoritmos:

RideSharing: partilha de viagens (tema 1) - Parte I

Grupo D , turma 3:

Ana Rita Fonseca Santos	201605240
Filipe Carlos de Almeida Duarte da Cunha Nogueira	201604129
Pedro Maria Passos Ribeiro do Carmo Pereira	201708807

(2018/2019 - 2º Semestre)

Índice

1. Descrição do tema

Neste trabalho, pretende-se implementar uma aplicação para suportar o conceito de *ridesharing*. Esta aplicação irá conciliar as moradas, destinos e restrições temporais de diferentes utilizadores e gerar a melhor rota desde a origem do condutor até ao seu destino. Os utilizadores estão espalhados pela cidade de origem e a aplicação irá calcular a melhor rota tendo em consideração todos os fatores mencionados.

Para enquadrar o projecto no âmbito desta unidade curricular, aproximando o problema de um do caixeiro viajante a aplicação irá revolver em torno de *ridesharing* entre cidades. O condutor antes de partir para a cidade destino, encontra o melhor caminho para recolher os utilizadores da cidade de partida. Tendo recolhido os mesmos avança para o destino, chegando lá calcula o melhor caminho para entregar os seus passageiros sendo que o destino final será o original do condutor. Para testes definiu-se o ponto de partida na cidade do Porto e o ponto de chegada em Fafe.

Este problema pode ser decomposto em cinco iterações.

1.1. Pré-cálculo

Extração e processamento do grafo das duas cidades em questão, gerado pelo OpenStreetMap, onde os nós representam cruzamentos ou fins de rua e as ruas arestas.

De seguida, guarda-se toda a informação relativa ao condutor e de todos os utilizadores envolvidos. Aqui é já estipulado certas informações importantes para a aplicação como a capacidade máxima do veículo, os pontos de partida e destinos, tanto do condutor, como dos utilizadores e as suas respectivas restrições de tempo.

Após a recolha dos dados do problema, vão se gerar dois novos grafos.
O primeiro grafo irá conter todos os pontos de partida tanto do condutor como dos utilizadores.
O segundo irá conter todos os pontos de entrega destes dois últimos.
A poda destes grafos é feita em dois momentos explicitados a seguir.

1.2. Seleção geográfica

Num primeiro momento, removem-se todos os utilizadores onde a sua localização inicial/final não permite ao condutor recolher-lo/entregá-lo. Ou seja, não existe nenhuma aresta (estrada) que ligue o ponto de recolha/entrega deste último com o resto do grafo (nó independente).

1.3. 1ª iteração: Seleção sem restrições temporais

Num segundo momento, calcula-se todos os utilizadores que respeitam todas as condições mencionadas em cima. É neste momento que se seleciona quais os utilizadores mais viáveis para fazer a viagem.

A selecção é efectuada através de um mapa de utilizadores.
Os utilizadores fora do veículo irão ser caracterizados em função dos seu locais de recolha e de entrega e das restrições da viagem que irão realizar.
Nesta fase ainda não são consideradas restrições temporais.

Desta forma é composto um mapa de utilizadores com os seguintes critérios: posição inicial e final do utilizador em relação às posições iniciais e finais do condutor. Este mapa está estruturado de forma a que um elemento do mapa representa o par nós origem e destino do utilizador, e a chave desse elemento representa o somatório da distância entre a origem do utilizador e o condutor, com a distância entre os destinos dos mesmos. As chaves de seguida são ordenadas de forma crescente.

1.4. 2ª iteração: Seleção com restrições temporais

Nesta iteração, irá ser feita uma melhor análise de compatibilidade. No momento de gerar a lista de prioridades mencionada na iteração anterior, a aplicação irá agora ter em conta todas as restrições dos utilizadores. Ou seja, para além dos critérios já considerados na iteração anterior, irá agora atribuir peso às restrições de tempo. Esta ação irá gerar uma fila de prioridades mais completa.
O programa remove primeiramente todos os utilizadores que apresentam uma hora de partida antes da hora de partida do condutor.
De seguida, verifica-se se o condutor tem tempo para efetuar a viagem com todos os utilizadores considerados até agora. Caso não tenha, remove-se o utilizador que apresenta menos tempo para efetuar a viagem. A verificação do condutor e a remoção do pior utilizador é efetuada em loop até o condutor tiver tempo de efetuar a viagem com os utilizadores que restam depois de cada remoção. Caso o condutor tenha tempo, o programa continua e é efetuado um loop onde em cada iteração remove-se o pior utilizador (o que tem menos tempo), até que todos os utilizadores restantes tenham tempo de efetuar a viagem.

1.5. Cálculo da rota

À luz deste mapa de utilizadores, os dois grafos anteriores (grafo com as origens e grafo com os destinos), irão então sofrer a segunda poda.
Isto consiste em manter um número equivalente de utilizadores viáveis e desejáveis nos grafos igual à capacidade do veículo. Todos os outros utilizadores são removidos. Ou seja, assumindo, por exemplo, que o veículo apenas tem capacidade para levar quatro utilizadores, são apenas selecionados os primeiros quatro utilizadores do mapa gerado, visto que são estes que apresentam um somatório das distâncias mencionadas anteriormente menores e //…..falar das restrições temporais.

Posteriormente à poda final dos dois grafos mencionados anteriormente, é possível executar o objectivo desta aplicação: calcular a rota mais curta que satisfaça as condições previamente referidas. A rota é calculada apenas considerando um veículo.

O utilizador que disponibiliza o veículo (condutor), vai passar por todos os utilizadores do grafo das origens e leva-os até aos seus respectivos destinos, acabando o seu percurso no seu destino.

2. Formalização do problema

2.1. Dados de entrada

Grafo G_Origem = (Vi, Ei) dirigido pesado, que consiste num conjunto de:

- vértices Vi, moradas válidas para utilizadores, cruzamentos em que os mesmos podem se encontrar e ser apanhados pelo condutor.
- arestas Ei, estrada entre duas moradas em que o peso respectivo de cada aresta representa o tempo que o veículo demora a atravessar a estrada.

Este grafo representa a cidade origem, onde todos os nós são potenciais pontos de recolha dos utilizadores.

Grafo G_Destino = (Vj, Ej) dirigido pesado, que consiste num conjunto de:

- vértices Vj, moradas válidas para utilizadores, cruzamentos em que os mesmos podem se encontrar e ser largados pelo condutor.
- arestas Ej, estrada entre duas moradas em que o peso respectivo de cada aresta representa o tempo que o veículo demora a atravessar a estrada.

Este grafo representa a cidade origem, onde todos os nós são potenciais pontos de recolha dos utilizadores.

Lista de utilizadores.
Cada utilizador Ui é caracterizado por:

- Si, morada de partida (Si $\in$ Vi)
- Dj, morada de destino (Dj $\in$ Vj)

- TolS, tempo de tolerância de partida
- TolD, tempo de tolerância de chegada
- TSi, tempo pretendido de partida
- TDi tempo pretendido de chegada

Condutor C (derivado de utilizador), representando o utilizador cujo percurso se quer otimizar, composto por:

- Atributos de utilizador
- N, capacidade máxima do veículo
- TSc, tempo pretendido de partida
- TDc tempo pretendido de chegada

2.2. Dados de saída

Grafo G1 = (Vf, Ef) dirigido pesado, em que Vf $\subset$ Vi e Ef $\subset$ Ei, representando apenas os nós e arestas de origem dos utilizadores válidos e co condutor.
Grafo G2 = (Vk, Ek) dirigido pesado, em que Vk $\subset$ Vj e Ek $\subset$ Ej, representando apenas os nós e arestas de destino dos utilizadores válidos e co condutor.

Condutor C: estado final do utilizador condutor, em que todos os nós dos grafos gerados foram visitados, sendo que o último nó é o nó que representa o destino do condutor. Faz parte da saída também o peso total do percurso do condutor que representa o caminho de menor custo que passa por todos os pontos uma vez dos dois grafos de saída.

2.3. Restrições

N >= 0, se o condutor não tivesse lugares disponíveis, não seria necessário selecionar os utilizadores mais viáveis e otimizar o seu percurso que seria direto.

Size(Condutor.currOcup) <= N, o condutor não poderá transportar mais passageiros que o máximo de lugares disponíveis no carro.

Restrições temporais de cada passageiro e do condutor. Condutor deve chegar à posição final dentro do limite de tempo, e deve-se assegurar que consegue transportar cada passageiro de acordo com as suas próprias restrições temporais.

2.4. Funções objetivo

A solução óptima para este problema é atingida quando o mapa de utilizadores referido anteriormente é construído de forma a que os dois últimos grafos gerados apenas contém os utilizadores mais viáveis e desejáveis para efectuar a viagem. Para tal é necessário minimizar a função **f** a seguir descrita na perspectiva de solução:

- f = Σ(wi*Ci)

Após uma boa implementação desta função, selecionar os utilizadores mais desejáveis que irão efetuar a viagem com o condutor.

Para atingir o potencial máximo de otimização do problema é necessário também que a função h, que representa o número de passageiros recolhidos, aproxime-se à capacidade máxima do carro (= N) e, simultaneamente, a função g, que representa o tempo total de viagem do condutor (onde t(e) representa o tempo de viagem de cada aresta), seja minimizada.

- h = |Condutor.currOcup|

- $g = \sum_{e \in E} t(e)$

Desta forma é encontrada a melhor solução que consiste em respeitar todas as restrições do problema enquanto se recolhe o maior número de utilizadores.

3. Perspectiva da solução

A solução deste problema terá como objectivo efectuar a melhor seleção dos utilizadores a recolher e minimizar o tempo de viagem do condutor. Ou seja, gerar a rota mais curta de acordo com as restrições de todos os utilizadores envolvidos.
Para atingir esta solução é necessário implementar certos algoritmos:

- No cálculo do mapa de utilizadores referido anteriormente, irá ser utilizado o seguinte algoritmo:
    - f = Σ(wi*Ci)
      f - valor unitário de prioridade do respectivo utilizador.
      Ci - critério a considerar.
      wi - peso do critério i.

- No cálculo do peso do critério distância entre o par de nós origem-destino do utilizador e o par de nós origem-destino do condutor, é utilizado o algoritmo de Dijkstra que apresenta uma complexidade de O((|V|+|E|) * log |V|). Onde |E| representa o número de arestas e |V| número de vértices.
  A aplicação deste algoritmo garante que é calculado o peso do caminho mais curto entre os nós considerados e é também utilizado para a seleção dos utilizadores mais desejáveis para efectuar a viagem.

- No cálculo do peso do critério tempo, foi implementado o raciocínio descrito em 1.4.

- No cálculo da rota mais eficaz que o condutor terá que efetuar é utilizado o algoritmo Held-karp para resolver o problema caixeiro viajante. Este algoritmo apresenta uma complexidade |||X|||| e garante que é encontrado o caminho mais curto entre a origem do condutor e o seu destino, passando por todos os pontos de recolha e destinos dos utilizadores.

4. Casos de utilização

- Criação do condutor;
- Criação de novos utilizadores;
- Listar utilizadores;
- Identificar melhores utilizadores candidatos;
- Calcular melhor caminho de recolha de utilizadores;
- Calcular melhor caminho de entrega de utilizadores;

5. Estruturas de dados utilizadas

Para representar as cidades utilizou-se a estrutura de grafo definida nas aulas práticas da unidade curricular. Para resolver o problema é instanciada uma classe *RideShare* que contém uma instância da classe *Driver*, dois grafos representando as cidades de origem e destino, dois vetores guardando os nós relevantes (instâncias da classe *Node*) em cada grafo e um vetor contendo todos os utilizadores (instâncias da classe *User*) considerados.

User: contém um inteiro que representa o código identificador para cada utilizador, duas estruturas do tipo tm (definido em <ctime>) representando o tempo esperado de chegada e partida do utilizador, dois inteiros que representam os identificadores dos pontos de partida e chegada (v. *Node*) do utilizador, e um valor booleano que indica se o utilizador também será um condutor.

Driver: extensão da classe *User* contêm todos os dados-membros da sua classe pai, dois inteiros que representam a capacidade máxima do veículo e a ocupação atual e ainda uma estrutura do tipo tm (definido em <ctime>) que representa o tempo esperado para concluir a viagem.

Node: representa um possível ponto de recolha ou entrega como o nó de um grafo. Contém um inteiro que representa o identificador do nó correspondente no grafo e uma estrutura do tipo *Coords* (que contém um inteiro para a posição em X e outro para a posição em Y), que representa as coordenadas do ponto no mapa.

6. Conectividade dos grafos gerados

Nesta aplicação são gerados quatro grafos distintos.

Os primeiros dois grafos representam a cidade de origem e de destino. Estes embora representando informações diferentes, ambos apresentam uma estrutura semelhante.
São grafos complexos, pesados, dirigidos e não conexos, visto que cada aresta apresenta um peso que representa a distância entre nós e existem nós isolados. Ou seja, nós onde não existe nenhuma aresta a ligá-los com o resto do grafo.
Estes grafos, como mencionado anteriormente, sofrem duas podas. É no segundo momento de poda que a conclusão sobre a conectividade dos grafos anteriores é atingida.
Embora não esteja implementado um algoritmo de conectividade, ao calcular a distância entre o condutor e os users, são encontrados nós (que representam locais de origem ou de destino), independentes.
Desta forma é possível inferir que este grafos são não conexos.

Finalmente, os dois últimos grafos (grafos de saída), representam os locais de origem e de destino do condutor e de todos os utilizadores viáveis.
Novamente, embora estes grafos representem informações diferentes, ambos têm uma estrutura semelhante.
Estes grafos são complexos, pesados, dirigidos e conexos, visto que todas as aresta apresentam um peso que representa a distância entre os nós que a delimitam e todos os nós estão ligados entre si.
Os grafos de saída apresentam apenas nós que estão ligados entre si, pois estes são resultantes da segunda poda, onde todos os nós independentes dos grafos resultantes da primeira poda são removidos (sendo os utilizadores dependentes destes nós removidos de consideração).
Desta forma, é possível inferir a conectividade destes últimos grafos como sendo conexos.

7. Apresentação dos algoritmos implementados (pseudocódigo)

- **Dijkstra**

```
function Dijkstra(Graph, source):
      dist[source] ← 0                              // Initialization

      create vertex set Q

      for each vertex v in Graph:
          if v ≠ source
              dist[v] ← INFINITY                    // Unknown distance
from source to v
          prev[v] ← UNDEFINED                       // Predecessor of v
```

```
        Q.add_with_priority(v, dist[v])


    while Q is not empty:                          // The main loop
        u ← Q.extract_min()                        // Remove and return
best vertex
        for each neighbor v of u:                  // only v that are
still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

- **Held-Karp**

```
function algorithm TSP (G, n)
  for k := 2 to n do
    C({k}, k) := d1,k
  end for

  for s := 2 to n-1 do
    for all S ⊆ {2, . . . , n}, |S| = s do
      for all k ∈ S do
        C(S, k) := minm≠k,m∈S [C(S\{k}, m) + dm,k]
      end for
    end for
  end for

  opt := mink≠1 [C({2, 3, . . . , n}, k) + dk,1]
  return (opt)
end
```

8. Análise de complexidade dos algoritmos (teórica)

Pior caso Held-Karp: (com ‘n’ nós)

Complexidade Temporal = O(2^{n}*n^{2})

Complexidade Espacial = O(2^{n}*n)

Pior caso Dijkstra: (com ‘v’ nós e ‘e’ arestas)

Complexidade Temporal = O(v^2 + e*log(v))

9. Principais dificuldades encontradas no desenvolvimento do trabalho

A primeira dificuldade encontrada foi a definição e compreensão dos requisitos do trabalho proposto. Após a fase de ilação de requisitos levantou-se outra questão. O âmbito do projeto proposto seria para os alunos estudarem o problema do caixeiro viajante, assim, foi preciso trabalhar a interpretação do enunciado até aproximar o problema ao anteriormente referido.

A nível de código, as maiores dificuldades foram sentidas na implementação das restrições temporais de cada utilizador e na implementação do Algoritmo de Held-Karp. Deve-se referir ainda que o grupo experienciou alguma tribulação proveniente da utilização do software GraphViewer.

10. Esforço dedicado por cada elemento do grupo

A escrita deste relatório e a implementação do código foi feita em conjunto, com todos os elementos do grupo presentes e contribuindo para um objetivo em comum, creemos que o trabalho e esforço de cada membro foi igual por esta razão.

Ana Rita Santos	-	33.3%
Filipe Carlos Nogueira	-	33.3%
Pedro Pereira	-	33.3%

11. Bibliografia

Material de estudo de Teoria de Grafos disponibilizado na página da UC de CAL e utilizado na realização deste relatório e do projeto em questão:

https://moodle.up.pt/pluginfile.php/227078/mod_label/intro/06.grafos1.pdf

https://moodle.up.pt/pluginfile.php/229148/mod_label/intro/07.grafos2.pdf

https://moodle.up.pt/pluginfile.php/231896/mod_label/intro/08.grafos3.pdf

https://moodle.up.pt/pluginfile.php/231896/mod_label/intro/09.grafos4.pdf

https://moodle.up.pt/pluginfile.php/237688/mod_label/intro/13.grafos8.pdf

https://moodle.up.pt/pluginfile.php/237688/mod_label/intro/14.grafos9.pdf

Held-Karp:
https://www.youtube.com/watch?v=cY4HiiFHO1o&t=937s

https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#Dynamic_programming_approach

Dijkstra:
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://www.quora.com/What-is-the-complexity-of-Dijkstras-algorithm