CITY SIGHTSEEING: TRILHAS TURÍSTICAS URBANAS
PARTE 2

Bruno Micaelo (up201706044@fe.up.pt)
Inês Alves (up201605335@fe.up.pt)
João Campos (up201704982@fe.up.pt)

CONCEPÇÃO E ANÁLISE DE ALGORITMOS - 26/04/2019

Introdução

Este trabalho tem como propósito servir como implementação de um sistema de planeamento de pequenas viagens turísticas em autocarros. Uma viagem de autocarro começa num certo ponto de encontro, onde os turistas se reúnem, e estes, por sua vez, têm vários pontos de interesse (POIs), que são locais da cidade que gostariam de visitar. O autocarro deve, então, tentar passar pelos pontos de interesse dos turistas que transporta, através do melhor caminho possível. Para isto, utilizar-se-ão vários algorítmos de forma a proporcionar uma solução ideal, que tanto minimize o número de autocarros utilizados pela empresa, como a distância total percorrida nas viagens, sendo preferida a redução da distância total, visto que uma menor distância equivale a um menor tempo de viagem para os turistas.

Na segunda parte deste trabalho focámo-nos em implementar os algoritmos propostos para atingir o nosso objetivo: obter os caminhos mais curtos para as viagens de cada grupo de turistas.

Formalização do problema

Dados de entrada

Gi = (Vi, Ei)
Grafo dirigido pesado, este é composto por:

- Vértices: são representativos de pontos da cidade, cada um tem:
    - id - Identificador do vértice;
    - name - Nome do ponto da cidade representado;
    - x - Componente X da coordenada geográfica do ponto;
    - y - Componente Y da coordenada geográfica do ponto;
    - adj - Vetor de arestas que partem do vértice.
- Arestas: são representativos das vias da cidade, cada uma tem:
    - id - Identificador da aresta;
    - name - Nome da via representada;
    - weight - Distância a percorrer na aresta;
    - dest - Vértice destino da aresta.

Ai

Vetor de autocarros, cada um composto por:

- id - Identificador do autocarro;
- cap - Capacidade do autocarro (sem contar com o motorista).

Pi
Vetor de passageiros, cada um composto por:

- id - Identificador do passageiro;
- name - Nome do passageiro;
- pois - Vetor de pontos de interesse (referências a Vértices).

S - Vértice inicial (tipicamente o ponto de encontro para os turistas).

Ti - Vetor de vértices finais (tipicamente é apenas um, correspondente ao ponto de encontro).

Dados de saída

Gf = (Vf, Ef)

Grafo dirigido pesado correspondente ao grafo inicial sem os vértices e arestas inacessíveis pelos autocarros.

Af

Vetor de autocarros utilizados, cada um composto por:

- pass - Vetor de referências aos passageiros que viajam no autocarro;
- path - Vetor de referências a arestas pelas quais o autocarro passa.

Restrições

Dados de entrada:

- O número de autocarros é superior a 0;
- O número de turistas é superior a 0;
- $\forall$ a $\in$ Ai, a.cap > 0;
- $\forall$ e $\in$ Ei, e.weight > 0;
- Todos os identificadores têm que ser únicos em:
  - Vértices;
  - Arestas;
  - Autocarros;
  - Turistas;
- S $\in$ Vi;
- Ti $\subseteq$ Vi;
- O grafo deve ser fortemente conexo;
- Todos os vértices não acessíveis pelos autocarros são ignorados.

Dados de saída

- Vf $\subseteq$ Vi, tendo em conta a remoção dos vértices inacessíveis reduz o conjunto de vértices original;
- Ef $\subseteq$ Ei, tendo em conta a remoção das arestas indisponíveis reduz o conjunto de arestas original;
- O tamanho de Af deve de ser menor ou igual ao tamanho de Ai, visto que os autocarros utilizados pertencem também ao conjunto de autocarros disponíveis.
- Para cada autocarro, o path deve ter como o primeiro elemento uma aresta proveniente de S, e o último elemento deve ter uma aresta cujo destino pertence a Ti (normalmente S).

Funções objetivo

De forma a ser utilizado o menor número de autocarros e a que estes percorram a menor distância possível entre pontos de interesse, pretendemos minimizar as seguintes funções objetivo, privilegiando a da função g em relação à da função f:

- f = |Ai| (minimização do número de autocarros);
- g = $\sum_{a \epsilon Af}$ [$\sum_{p \epsilon a.path}$ $w(p)$] (minimização da distância percorrida);

Solução do problema

Decompomos este problema em três iterações:

1. Autocarro com capacidade infinita, sem organização dos turistas

Numa primeira fase, o objetivo é apenas encontrar o menor caminho possível entre os vários pontos de interesse na viagem de autocarro, não se agrupando, então, os turistas pelos seus POIs (pontos de interesse) semelhantes. A viagem começa num ponto de encontro definido, num autocarro com capacidade de passageiros infinita, passando por todos os POIs dos turistas que transporta, sempre através do melhor caminho possível.

É, também, necessário que os vértices do grafo, que representam os pontos de interesse da cidade, façam parte da mesma componente fortemente conexa, ou seja, terão que existir caminhos de acesso a qualquer POI do grafo. Por exemplo, se o autocarro se encontrar num certo ponto de interesse, deve tanto poder chegar a qualquer outro POI, como regressar, se necessário, ao local inicial, seguindo vários caminhos do mapa.

Para além disto, é necessário ter em conta que algumas vias de comunicação da cidade podem estar inacessíveis (no caso de obras, por exemplo), procedendo-se à remoção das arestas que as representam.

Utilizamos, para garantir a melhor solução possível, o algorítmo de Dijkstra.

2. Autocarros com capacidades infinitas, com organização dos turistas

Neste caso, ao contrário da primeira iteração, os turistas irão ser distribuídos por vários autocarros, tendo, agora, em conta, os seus pontos de interesse, de modo a que cada autocarro tenha passageiros com POIs iguais ou próximos entre si no mapa. Assim, em cada viagem, haverá um menor número total de POIs, de forma a otimizar o caminho a percorrer. A capacidade do autocarro é, ainda, infinita.

Para isto, utilizamos uma implementação gananciosa para a distribuição dos turistas, tendo em consideração o algoritmo dos casamentos estáveis como base.

3. Autocarros com capacidades finitas, com organização dos turistas

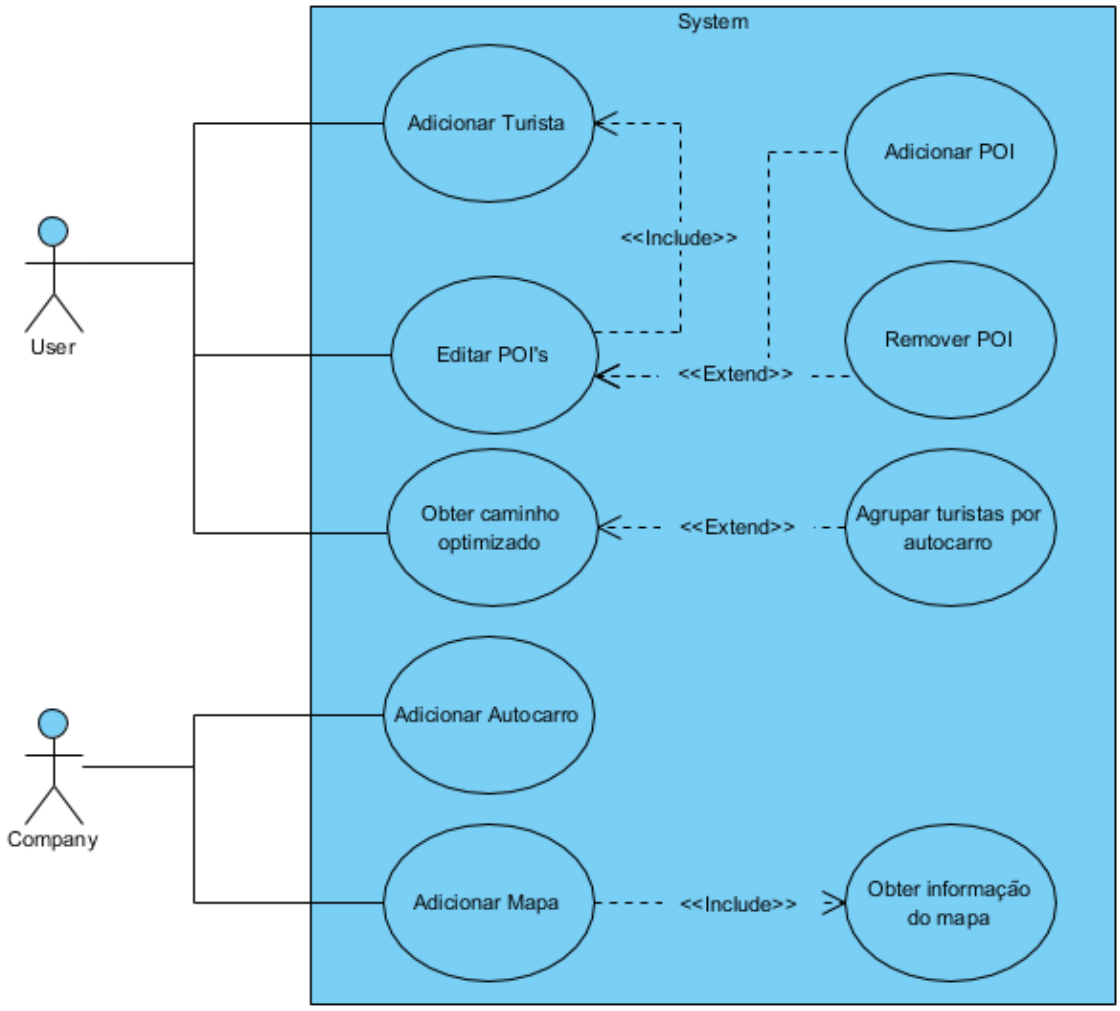
A terceira fase considera, finalmente, autocarros com capacidades finitas, encontrando a solução final para o problema. Neste caso, se não for possível transportar certo grupo de turistas por excederem a capacidade máxima de passageiros, terá de ser utilizado outro autocarro.

Pretende-se, então, minimizar tanto o número total de autocarros necessários como a distância total percorrida nas viagens, dando prioridade a esta última.

Para isto, utilizamos um algoritmo de retrocesso, alcançando a solução ideal através de sucessivas tentativas.

Casos de Utilização

Os seguintes casos de utilização têm como base a última implementação do algoritmo de minimização de caminhos e organização de turistas, sendo resumidos pelo seguinte diagrama.



Os casos uso do sistema envolvem dois atores, User e Company, representando respetivamente o utilizador (que adiciona um ou mais turistas e organiza as viagens) e a empresa responsável pelo sistema (que disponibiliza o transporte nas viagens).

Use Case 1	Add tourist
Actor	User
Fluxo básico	O utilizador especifica o id e o nome do turista a adicionar, podendo especificar de seguida os respetivos pontos de interesse.

Use Case 2	Add bus
Actor	Company
Fluxo básico	A empresa de transportes adiciona um autocarro ao conjunto de transportes disponíveis, especificando o seu id e capacidade.

Use Case 3	Check tourist
Actor	Company
Fluxo básico	Dá display da lista completa de turistas, com respetivos ids e nomes.

Use Case 4	Check bus
Actor	Company
Fluxo básico	Dá display da lista completa de autocarros, com respetivos ids e capacidades.

Use Case 5	Get path
Actor	User
Fluxo básico	O utilizador obtém o caminho otimizado tendo em conta a disponibilidade dos autocarros e a combinação dos interesses dos turistas.

Use Case 6	Get groups
Actor	User
Fluxo básico	O utilizador obtém o conjunto de autocarros com os respetivos grupos de turistas a transportar, considerando a compatibilidade entre os seus interesses.

Use Case 7	Generate a random tourist
Actor	Company
Fluxo básico	Gera um turista com id, nome e POIs aleatórios.

Estruturas de dados utilizadas

Utilizamos **vetores** para:

- guardar todos os turistas da viagem;
- guardar todos os autocarros da viagem;
- cada turista, guardar os seus pontos de interesse (POIs);
- guardar os IDs dos vértices dos caminhos obtidos pelo algoritmo de Dijkstra (função getPath()).

Utilizamos uma **mutable priority queue** para**:**

- guardar os vértices do caminho no algoritmo de Dijkstra.

Utilizamos um **unordered set** para:

- agrupar os turistas em grupos tendo em conta os seus pontos de interesse (POIs). Em vez de um vetor, escolhemos utilizar um unordered set, visto que este tem complexidade constante em pesquisa e inserção, tornando a criação e utilização dos grupos muito mais rápida.
- guardar os índices dos vértices no algorítmo de Kosaraju. O vantagem de um unordered_set é o facto de ter uma rápida procura e inserção de elementos, que acelera o programa.

Utilizamos um **map**:

- dado o ID de um grupo, devolve a compatibilidade deste com outro grupo.

Análise da conectividade do grafo

Naturalmente, os mapas de ruas têm certos vértices isolados ou ciclos separados, que não serão utilizados pelo algoritmo de procura de caminhos. Por essa razão, são removidos, aumentando a conectividade do grafo e eficiência do programa.

Para este fim, utilizamos uma versão do algoritmo de Kosaraju. Este consiste em percorrer todos os vértices do grafo e guardá-los numa stack, pela ordem contrária à de visita. De seguida, inverte-se as arestas do grafo e faz-se uma pesquisa em profundidade começando no vértices, que são retirados, por ordem, da stack.

Como no nosso mapa todas as arestas de um nó para outro têm uma irmã que faz o caminho oposto, não há necessidade de fazer o inverso do grafo.

O algoritmo devolve um grupo de índices de vértices fortemente conexos. Assim, repetimos este algoritmo começando em cada nó, selecionando como mapa final o grupo do índices de maior tamanho.

Algoritmos Implementados

Algoritmo de Kosaraju modificado:

```
S = {}
V = {}
max_V = {}
G = (v,e)

DFS_stack(start, S):
  v <- findVertex(start);
  visited(v) <- true;
  for each e in adj(v)
    if not visited( dest(e) )
      DFS_stack(id( dest(e) ), S);
  push(S, start);
DFS_vector(start, V):
  v <- findVertex(start)
  visited(v) <- true
  for each e in adj(v)
    if not visited( dest(e) )
      DFS_vector(id( dest(e) ), V)
  insert(V, v)

Kosaraju(start){
  unvisitAll(G)
  DFS_stack(start, S)
  while not empty(S)
    v <- top(S)
    pop(S)
    DFS_vector(v, V)

main():
  for each v in G
    Kosaraju(id(v))
    if size(V) > size(max_V)
      max_V <- V
```

Algoritmo de Dijkstra:

```
Q = {}
G = (v,e)

initsinglesource(origin):
  for each v in vertexSet
    distance(v) <- INF
    path(v) <- NULL
  s <- findvertex(origin)
  dist(s) <- 0
  return s

relax(v, w, weight):
  if (distance(v) + weight) < weight(w)
    distance(w) <- (distance(v) + weight)
    path(w) <- v
    return true
  else
    return false

dijkstra(origin):
  initsinglesource(origin):
  Q <- s
  while not empty(Q)
    v <- extractMin(Q)
    for each e of adj(v)
      old_dist <- dist( dest(e) )
      if(relax(v, e.dest, e.weight))
        if old dist = INF
          insert(Q, dest(e))
        else
         decrease-key(Q, dest(e))
```

Algoritmo de casamentos estáveis modificado:

```
getCompatibility(g1, g2):
  value <- 0
  for each poi1 in poi_set(g1)
    min <- INF
    v1 <- findVertex(poi1)
    for each poi2 in poi_set(g2)
      v2 <- findVertex(poi2)
      distance <- dist(v1,v2)
      if distance < min
        min = distance
    value <- value + min
  return value

getCompatibilities():
  for each g1 in group_set
    for each g2 in group_set
      c1 <- getCompatibility(g1, g2)
      c2 <- getCompatibility(g2, g1)
      c_min <- min(c1,c2)
      insert(compabilities(g1), c_min)
      insert(compabilities(g2), c_min)

merge(g1, g2, gSet):
  result <- {}
  for each t in g1
    insert(g, t)
  for each t in g2
    insert(g, t)
  for each c in compatibilities(g1)
    insert(compatibilities(g), c)
  for each c in compatibilities(g2)
    insert(compatibilities(g), c)
  for each c1 in compatibilities(g)
    c2 <- findDuplicate(compatibilities(g), c1)
    if not null(c2)
      erase( max(dist(c1), dist(c2) ) )
  for each c
```

```
organize():
  pair <- getBestCombination()
  g1 <- first(pair)
  g2 <- second(pair)
  g <- merge(g1,g2)
  if( size(buses) >= size(groups) )
    dist <- max(dist(g1), dist(g2) )
    if( (dist*2) < dist(g) )
      i <- 0
      while(i < size(groups) )
        bi <- buses[i]
        gi <- groups[i]
        put(bi, gi)
        i++
      return
  erase(groups, g1)
  erase(groups, g2)
  insert(group, g)
  for each gi in groups
    removeCompatibility(gi, g1)
    removeCompatibility(gi, g2)
  organize()

main():
  getCompabilities()
  organize()
```

Análise de complexidade dos algoritmos

Algoritmo de Kosaraju

Como o algoritmo consiste, essencialmente, em duas pesquisas em profundidade, o consumo de tempo é proporcional ao tamanho do grafo. Assim, a complexidade do algoritmo de Kosaraju é **O(V+E)**, onde N é o número de vértices e E é o número de arestas. Este algoritmo é repetido uma vez por vértices, por essa razão o seu desempenho no nosso programa é **O(V^2 + E*V)**.

Algoritmo de Dijkstra

O algoritmo de Dijkstra procura o caminho menos custoso entre dois pontos. No pior dos casos, este algoritmo tem uma complexidade temporal **O(N^2)**, onde N é o número de vértices. No nosso caso, ao utilizar uma fila de prioridade, a complexidade passaria a ser **O((|V|+|E|) * log |V|)**. Como o algoritmo é utilizado um número de vezes igual ao tamanho do vetor de pontos de interesse (POIs) a percorrer obrigatoriamente, a complexidade é **O((|V|+|E|) * log |V| * N)**, sendo, no pior caso, **O(N^3)**.

O algoritmo de Dijkstra, em média, demorou 1 milissegundo (de acordo com o output do programa) para o mapa mais pequeno.

Algoritmo de casamentos estáveis modificado

Utilizamos um algoritmo que tem como base o de casamentos estáveis, para, essencialmente, agrupar os vários turistas de acordo com os seus pontos de interesse (POIs). O algoritmo percorre o conjunto de grupos de turistas, reduzindo-o por um a cada recursão, até o conjunto de grupos ser adequado para ser juntado aos autocarros, tendo o corpo principal do algoritmo uma complexidade de **O(N * logN)**. No interior do corpo principal são chamadas outras funções que aumentam a complexidade do algoritmo: uma função que obtém o par de grupos mais eficiente (**O(N)**), uma função de merge para o par de grupos (**O(N)**), funções para remover as compatibilidades com os alvos do merge (**O(N)**) e um conjunto de funções de acesso e remoção que, por estarem a atuar sobre unordered sets e maps, têm complexidade constante (**O(1)**). No total, a complexidade deste algoritmo é **O(N^2 * logN)**.

Este algoritmo demorou, em média, de forma semelhante ao de Dijkstra, tanto para a compatibilidade como para o grouping, 1 milissegundo (de acordo com o output do programa) para o mapa mais pequeno e um conjunto pequeno de turistas com 3 POIs cada.

Conclusão

O objetivo deste trabalho é a criação de um algoritmo que permita organizar turistas por autocarros, de forma a permitir a visita de diferentes pontos de interesse numa dada área.

Ao realizar este relatório, tivemos em consideração a elevada complexidade da preferência de POIs pelos turistas. Decidimos, então, primeiramente, arranjar uma solução que não envolvesse esta parte do projeto. Mais tarde, construímos um algoritmo que categorizasse os turistas pelos seus gostos, de forma a distribuí-los pelo diferentes autocarros.

Quanto ao esforço realizado por cada membro do grupo, cada um trabalhou de forma igual, contribuindo para cada secção do trabalho.

Na segunda parte do projeto, focamo-nos na implementação dos algoritmos para chegar aos objetivos propostos. Conseguimos criar soluções para problemas individuais, mas infelizmente não conseguímos atingir o nosso objetivo de obter uma solução para a terceira fase, que considera as capacidades dos autocarros, ficando pela organização dos turistas, desprezando as capacidades, e a obtenção de caminhos.

Um dos problemas foi o facto de não sabermos trabalhar com o mapa, o que levou a uma dificuldade de testar os algoritmos de Dijkstra e Kosaraju numa grande escala, com muitos vértices e arestas.