Concepção e Análise de Algoritmos

Tema 3 - City Sightseeing: trilhas turísticas urbanas

28 de Maio de 2019
Grupo F, Turma 2

João Paulo Monteiro Leite **up201705312@fe.up.pt**

Márcia Isabel Reis Teixeira **up201706065@fe.up.pt**

Pedro Miguel Rodrigues Ferraz Esteves **up201705160@fe.up.pt**

Índice

Glossário

Grafo, G(V, E): par de conjuntos V e E, onde: V é um conjunto não vazio que representa os vértices do grafo; E é um conjunto de pares ordenados e= (v, w), com v w V, que representa as arestas (do inglês, *edges*) do grafo.

Grafo Conexo: grafo no qual há pelo menos uma aresta a ligar cada par dos seus vértices.

Grafo Fortemente Conexo: grafo no qual, para todos os pares de vértices (v, w), existe um caminho de v para w e também de w para v.

Grafo Dirigido: grafo cujas arestas são orientadas.

Grafo Pesado: grafo no qual as arestas têm um peso/custo associado.

POI: ponto de interesse.

UC: Unidade Curricular.

Introdução

Uma empresa de visitas turísticas em autocarros (city sightseeing) da cidade do Porto pretende desenvolver uma aplicação para determinar o melhor caminho que o autocarro turístico deve seguir para visitar certos pontos de interesse, consoante a preferência dos passageiros.

Esta aplicação será desenvolvida tendo por base os algoritmos lecionados ao longo da UC de Concepção e Análise de Algoritmos.

1. Divisão do Problema

O problema em questão pode ser dividido em três iterações, detalhadas a seguir.

1.1. 1ª iteração: escolha de caminhos consoante preferências, autocarro com capacidade ilimitada

Numa primeira fase, assume-se que o número de passageiros é sempre inferior à capacidade do autocarro, ou seja, o autocarro tem capacidade ilimitada.

Assim, o objetivo desta primeira iteração é simplesmente determinar o caminho mais curto entre vários pontos escolhidos pelos turistas de uma lista dada, a partir de um ponto de partida (ponto de encontro) até um ponto de chegada definidos a priori (dados de entrada). Só é possível realizar estes percursos, caso todos os pontos estejam ligados entre si, o que significa que estes devem fazer parte de uma componente fortemente conexa do grafo.

No caso de algum dos pontos turísticos ser inacessível, este é excluído do percurso.

1.2. 2ª iteração: agrupamento de turistas consoante semelhanças/proximidades das preferências, autocarro com capacidade ilimitada

Numa segunda fase, de forma a otimizar os percursos, agrupam-se os turistas consoante as semelhanças entre os seus pontos de interesse favoritos. Haverá assim uma divisão dos vários turistas por diferentes autocarros, cada um com um percurso próprio. Nesta fase considera-se ainda que o número de turistas não ultrapassa a capacidade dos autocarros.

1.3. 3ª iteração: agrupamento de turistas consoante semelhanças das preferências, autocarro com capacidade limitada

Na terceira iteração passa a ter-se em conta que os autocarros têm uma capacidade limitada. Assim, ao fazer a distribuição dos turistas pelos autocarros, deve ter-se em conta não só as suas preferências, como o facto de por vezes poder ser necessário dividir turistas com as mesmas preferências por autocarros diferentes. Assim, poderá haver autocarros diferentes com o mesmo percurso.

2. Formalização do Problema

2.1. Dados de Entrada

P[i]: lista de passageiros.
D[i]: lista de listas de destinos escolhidos pelos passageiros. A lista na posição i corresponde ao passageiro P[i].
A[i]: conjunto de autocarros disponíveis para as viagens turísticas. Cada autocarro é caracterizado por uma capacidade (ignorada nas primeiras iterações do problema), um conjunto de pontos a visitar, definidos a posteriori, com base nas escolhas dos passageiros e uma lista de passageiros, a preencher quando se inserem.
Pi: Ponto inicial da viagem.
Pf: Ponto final da viagem.
Gi = (Vi, Ei) - grafo dirigido pesado, com a rede da cidade, composto por:

- V - vértices (que representam pontos da cidade) com:
  - POI: booleano que indica se o vértice é um ponto de interesse
    - Nome: nome do ponto de interesse
    - Adj $\subseteq$ E - conjunto de arestas que partem do vértice
- E - arestas (que representam vias de comunicação) com:
  - w - peso da aresta (representa a distância entre os dois vértices que a delimitam)
  - ID - identificador único de uma aresta
  - dest $\in$ Vi - vértice de destino

2.2. Dados de Saída

Gf = (Vf,Ef) grafo dirigido pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei.
Af: sequência ordenada de todos os autocarros usados. Cada autocarro tem os seguintes valores:

- LP[i]: lista de passageiros a viajar neste autocarro
- P = {e $\in$ Ei | 1 $\leq$ j $\leq$ |P|} - sequência ordenada (com repetidos) de arestas a visitar

Pnv: lista de POIs não visitados, devido a imprevistos no percurso (obstrução de vias de trânsito, por exemplo).

2.3. Restrições

A. Sobre os dados de entrada:

- |P[i]| > 0, o número de passageiros tem de ser superior a 0.
- |D[i]| > 0, o número de destinos tem de ser superior a 0.
- |A[i]| > 0, o número de autocarros disponíveis tem de ser superior a 0.
- ($\sum_{i=1}^{i=|A[i]|}$ capacidade(|A[i]|)) ≥ |P[i]|, o número de passageiros tem de ser igual ou inferior ao somatório das capacidades dos autocarros.
- ∀ e ∈ Ei, w > 0, uma vez que o peso das arestas representa uma distância sendo que esta não pode ser negativa.
- Seja o conjunto C = {v ∈ Vi | v = Pi | v = Pf}, todos os elementos de C devem fazer parte do mesmo componente fortemente convexo de um grafo, de modo a garantir a possibilidade dos caminhos entre eles.
- ∀ e ∈ Ei, e deve ser utilizável pelo autocarro, caso contrário não é incluída no grafo Gi.

B. Sobre os dados de saída

Na sequência Af:

- ∀ a ∈ Af, |a.LP[i]| > 0, um autocarro usado deve transportar mais de 0 passageiros.
- |Af| ≤ |Ai|, pois não se podem usar mais autocarros do que os que estão disponíveis.
- $\sum_{a \in Af}$ |a.LP[i]| <= |P[i]|, o número total de passageiros transportados deve ser menor ou igual ao número de passageiros recebido inicialmente. O número de passageiros transportados pode ser menor que o inicial caso todos os POIs escolhidos por algum(ns) passageiro(s) estejam indisponíveis.

Na sequência P:

- Seja e_1 o primeiro elemento de P. É necessário que $e_1 \in$ Adj(Pi).
- Seja e_n o último elemento de P. É necessário que $e_n \in$ Adj(Pf).

Na lista Pnv:

- Se $\sum_{a \in Af}$ |a.LP[i]| < |P[i]|, então |Pnv| > 0, pois um passageiro não é transportado quando todos os POIs por si escolhidos estão indisponíveis.

2.4. Funções Objetivo

A solução ótima do problema passa por minimizar o número de autocarros usados e a distância total percorrida por todos, de forma a que todos os passageiros passem pelos pontos de interesse pretendidos.

Logo, o objetivo é a minimização das seguintes funções:

$f = |A|$

$$g = \sum_{a \in A} [\sum_{e \in P} w(e)]$$

O maior objetivo é tentar minimizar a função g, ou seja, tentar encontrar os melhores caminhos possíveis.

3. Perspetiva de Solução

Para resolver o problema proposto, iremos abordar vários algoritmos lecionados nesta UC.

Inicialmente usaremos o algoritmo de Tarjan, baseado numa pesquisa em profundidade, para encontrar as componentes fortemente conexas dos grafos. Serão mostrados todos os pontos no mapa, sendo os pontos em partes fortemente conexas (acessíveis) pintados a cor diferente dos restantes.

A resolução do problema do cálculo do caminho mais curto entre todos os POI (começando num ponto inicial e acabando num final), assumindo a capacidade do autocarro ilimitada, será feita recorrendo ao algoritmo de Dijkstra para encontrar o caminho mais curto.

A resolução do problema da divisão dos turistas pelos autocarros (minimizar o número de autocarros juntando os passageiros com mais pontos de interesse em comum) será feita recorrendo a um algoritmo baseado em fluxo máximo, ainda considerando autocarros de capacidade ilimitada.

Para a execução e tratamento do problema serão utilizados mapas da cidade do Porto extraídos do OpenStreetMaps (www.openstreetmap.org). Os pontos turísticos serão maioritariamente escolhidos do mesmo site.

Nas próximas subsecções, explicamos cada um dos algoritmos/abordagens utilizados, analisando a complexidade de cada um destes.

4. Estruturas de Dados usadas

4.1. Graph

O grafo é a principal estrutura de dados utilizada no trabalho, sendo o foco deste. Esta estrutura encontra-se implementada nos ficheiros Graph.h e Graph.cpp, e foi adaptada da implementação que é disponibilizada pelos docentes da UC para as aulas teórico-práticas. O grafo é usado para representar o mapa sobre o qual se farão os percursos, com vários pontos (vértices, Vertex) e ligações entre estes (arestas, Edge). Os vértices podem ou não representar um ponto de interesse. Esta estrutura de dados é essencial para a resolução do problema, pois permite uma representação/organização simples e intuitiva dos vários pontos e ligações entre si, bem como facilita a aplicação de vários algoritmos aos mesmos.

4.2. MutablePriorityQueue

A implementação da MutablePriorityQueue encontra-se no ficheiro MutablePriorityQueue.h (disponibilizado pelos docentes da UC para as aulas teórico-práticas, da autoria do Professor João Pascoal Faria). Esta estrutura de dados é usada na implementação do Algoritmo de Dijkstra. Para a implementação deste algoritmo é necessária uma fila de prioridades, de forma a processar sempre o vértice com distância mínima em relação ao atual; esta fila deve ser também mutável, pois pode ser necessário alterar a prioridade de um vértice na fila.

4.3. Bigraph

Foi implementado um grafo bipartido baseado na implementação do Graph. Esta estrutura de dados é utilizada no algoritmo que divide as pessoas pelos vários autocarros/percursos. Nesta parte do problema é adequado usar um grafo bipartido, pois um dos subconjuntos representa os vértices de pessoas e o outro os vértices dos pontos de interesse escolhidos pelas mesmas. As arestas que ligam uma pessoa a ponto de interesse indicam que esse ponto de interesse foi escolhido pela mesma.

4.4. unordered_set

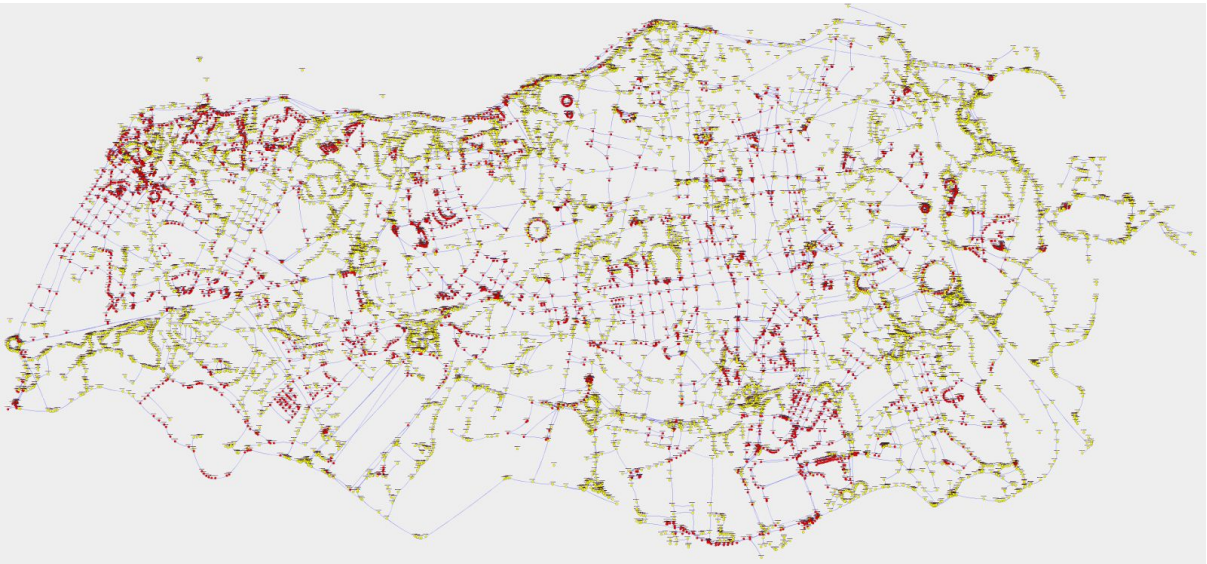
Foi usado o unordered_set da standard library de C++. Escolheu-se esta estrutura de dados para eliminar POIs repetidos no resultado final da função que divide as pessoas por autocarros de acordo com os seus POIs escolhidos. Para além desta aplicação, foi também usado o unordered_set na função que encontra o caminho mais curto entre o ponto inicial e final, passando por um conjunto de POIs. Neste caso, os POIs são guardados num unordered_set, pois são acedidos várias vezes no algoritmo de Dijkstra e a complexidade temporal média de operações sobre um unordered_set é O(1).

4.5. unordered_map

Foi usado o unordered_map da standard library de C++. O unordered_map é usado nas classes Graph e Bigraph, de forma a facilitar as operações de pesquisa e inserção nos mesmos, que desta forma têm complexidade temporal O(1), já que cada elemento (IDs ou Pessoas) tem uma key única.

5. Conectividade do Grafo

Para avaliar a conectividade do grafo, recorreu-se a uma implementação do algoritmo de Tarjan para encontrar componentes fortemente conexas em grafos dirigidos, algoritmo este baseado numa busca em profundidade. Do algoritmo retornam várias componentes fortemente conexas do grafo, sendo que ao avaliar a conetividade dos grafos fornecidos, podemos observar que não existem muitas zonas fortemente conexas.



Na imagem acima encontra-se a visualização do grafo que representa o mapa do Porto no GraphViewer. Os pontos a vermelho representam as zonas fortemente conexas, sendo estas, como se pode ver, normalmente isoladas e não muito abundantes.

6. Algoritmos de Solução

6.1. Algoritmo de Dijkstra

O Algoritmo de Dijkstra é um algoritmo ganancioso que calcula o caminho de custo mínimo entre vértices de um grafo, e garante a solução ótima em casos em que as arestas não têm pesos negativos.

Depois de escolhido um ponto inicial (a partir do qual serão calculados os melhores caminhos para todos os outros vértices), marca-se a distância desse vértice com 0 e a distância para todos os outros vértices com infinito.

De seguida, em cada iteração, seleciona-se um vértice atual. O vértice selecionado será o vértice não visitado mais próximo do inicial. A partir do vértice atual, atualiza-se a distância para todos os vértices não visitados diretamente conectados a este. Começa-se por somar o peso da aresta entre o vértice atual e o não visitado com o valor do vértice atual. O valor do vértice não visitado será atualizado para esta soma se a soma for menor que o valor atual do mesmo.

Quando todos os vértices adjacentes tiverem sido considerados, marca-se o vértice atual como visitado, seleciona-se o vértice atual seguinte, e repete-se o processo até que o vértice de destino esteja marcado como visitado.

O Dijkstra é utilizado para obter o caminho mais curto entre os pontos inicial e final do percurso, passando por todos os POIs. Primeiro, aplica-se o algoritmo ao vértice do ponto inicial, parando quando é encontrado um dos POIs do percurso (de forma a tentar que o próximo POI seja o mais próximo da origem), o qual é eliminado de um set de POIs, e insere-se num vetor de vértices (percurso) o caminho entre os dois. Continua-se a aplicar o algoritmo da mesma forma aos vários POIs até que o set esteja vazio: itera-se até encontrar o POI mais próximo, e esse será o próximo a ser processado. Por fim aplica-se o algoritmo a partir do último POI até chegar à origem, obtendo-se assim o caminho mais curto que passa por todos os pontos.

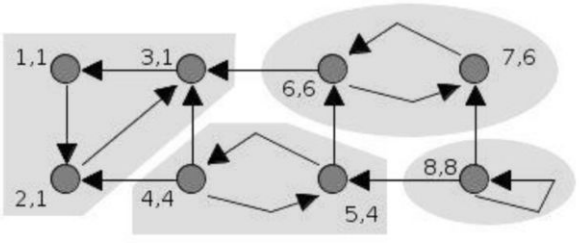
Pseudocódigo do Algoritmo de Dijkstra clássico (Adaptado dos slides da Unidade Curricular):

```
DIJKSTRA(G, s): // G=(V,E), s ∈ V
        for each v ∈ V do
            dist(v) := infinity
            path(v) := null
        dist(s) := 0
        Q  := empty // min-priority queue
        INSERT(Q, (s, 0)) // inserts s with key 0
        while Q != empty  do
            v := EXTRACT-MIN(Q) // greedy
            for each w ∈ Adj(v) do
                if (dist(w) > dist(v) + weight(v,w)) then
                    dist(w) := dist(v)+ weight(v,w)
                    path(w) := v
                    if w  Q then // old dist(w) was  infinity
                        INSERT(Q, (w, dist(w)))
                    else
                                    DECREASE-KEY(Q, (w, dist(w)))
```

Pseudocódigo da função que calcula o caminho mais curto que passa por todos os pontos:

```
SHORTEST_ROUTE(G, s, p[], f): // G=(V,E), s,f,p[i] ∈ V
        for each poi ∈ p
           insert(pois_set, p)
        while not found poi ∈ pois_set
           DIJKSTRA(s, pois_set)
        while pois_set not empty
           DIJKSTRA(last poi found, pois_set)
```

6.2. Algoritmo de Tarjan



O Algoritmo de Tarjan é um algoritmo com a finalidade de determinar componentes fortemente conexas num grafo. A ideia base consiste na realização de uma pesquisa em profundidade a partir de um nó arbitrário, e subsequentes pesquisas em nós ainda não descobertos, sendo cada nó visitado uma e uma só vez. Do algoritmo resultam vários conjuntos de nós, sendo que todos os nós do mesmo conjunto pertencem à mesma componente fortemente conexa do grafo.

Pseudocódigo do Algoritmo de Tarjan (tirado da wikipédia https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm):

```
algorithm tarjan is
  input: graph G = (V, E)
  output: set of strongly connected components (sets of vertices)

  index := 0
  S := empty stack
  for each v in V do
    if (v.index is undefined) then
      strongconnect(v)
    end if
  end for

  function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
      if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink  := min(v.lowlink, w.lowlink)
      else if (w.onStack) then
        // Successor w is in stack S and hence in the current SCC
          // If w is not on stack, then (v, w) is a cross-edge in the DFS tree
and must be ignored
```

```
            // Note: The next line may look odd - but is correct.
               // It says w.index not w.lowlink; that is deliberate and from the
original paper
              v.lowlink  := min(v.lowlink, w.index)
            end if
          end for

          // If v is a root node, pop the stack and generate an SCC
          if (v.lowlink = v.index) then
            start a new strongly connected component
            repeat
              w := S.pop()
              w.onStack := false
              add w to current strongly connected component
            while (w != v)
            output the current strongly connected component
          end if
        end function
```

6.3. Algoritmo de Agrupamento de Turistas

Neste algoritmo começa-se por dividir as pessoas e os pontos de interesse por elas escolhidos num grafo bipartido, em que as arestas que os ligam representam a escolha de um ponto de interesse por parte de uma pessoa. Escolhe-se o ponto de interesse escolhido por mais turistas e, caso o número de pessoas que o escolheram seja menor que a capacidade do autocarro, todas elas irão no mesmo autocarro que passará por todos os seus POIs. Caso contrário, só o número de pessoas que couberem no autocarro, serão escolhidas; esta escolha é feita pela ordem de entrada no grafo. As pessoas já alocadas para um autocarro são eliminadas do grafo, escolhe-se o próximo ponto escolhido por mais turistas e repete-se o processo até não faltar alocar mais turistas.

Pseudocódigo

```
dividePeople(bus_capacity):
B := Create bigraph
e := calculate initial number of edges
results := empty
while e > 0
        p := getPeopleForBus(e, bus_capacity)
        add p to results
        e := e - 1
return results

getPeopleForBus(e, bus_capacity):
while bus_capacity > 0 and e > 0
        mrp := search for most requested point
        for each mrp.edges
                add person to vperson
                bus_capacity = bus_capacity - 1
                for each person.edges
                        add poi to vpois
                e := e - 1
                if bus_capacity <= 0 or e <= 0
                        break
        for each unused edges
                erase edge
                e = e - 1
remove duplicates in vpois
return pair<vpois, vperson>
```

7. Discussão dos Casos de Utilização

Procedeu-se à criação de uma interface para permitir a utilização dos algoritmos criados.

Ao iniciar o programa o utilizador terá de escolher um grafo para importar (introduzindo o nome da cidade cujo mapa o grafo representa). De seguida, poderá importar um novo grafo, criar um turista ou importar turistas de um ficheiro.

Se escolher a opção de criar um turista, poderá num novo menu adicionar um POI à pessoa, ver os POIs conectados aos já escolhidos, ver todos os POIs conectados entre si (POIs das várias componentes fortemente conexas), adicionar a pessoa atual à opção, guardar a pessoa num ficheiro e continuar, voltar ao menu anterior ou, por fim, sair do programa.

Se escolher a opção de guardar num ficheiro e continuar, poderá, num outro menu, visualizar o grafo (esta opção abre uma nova janela em que se vê o grafo recorrendo ao GraphViewer e deve ser chamada sempre), dividir as pessoas e criar rotas para os autocarros (caso escolhida esta opção é pedida a capacidade dos autocarros), importar um novo grafo, ou sair do programa.

8. Análise de complexidade dos algoritmos (teórica e empírica)

8.1. Algoritmo de Dijkstra

O Algoritmo de Dijkstra clássico tem uma complexidade temporal teórica de O((V|+|E|) * log |V|) (sendo |V| o número de vértices do grafo e |E| o número de arestas). Na função onde é calculado o caminho mais curto que passa pelos vários pontos é usada uma versão do Dijkstra com esta complexidade temporal no pior caso (já que o algoritmo para de iterar quando encontra um ponto de interesse).

Esta função tem uma complexidade temporal diferente da do algoritmo de Dijkstra: devido a dois ciclos *for* que iteram pelos pontos de interesse e pelos vários sub-percursos entre os mesmos, e à função getPath() (disponibilizada pela docência da UC para as aulas teórico-práticas), todos estes com complexidade temporal O(N), a complexidade temporal teórica no caso médio da função que calcula o caminho mais curto e que utiliza o algoritmo de Dijkstra é O(N).

Após várias tentativas empíricas (para a cidade do Porto), para uma amostra pequena, o algoritmo de Dijkstra demorou 0.004s a terminar.

8.2. Algoritmo de Tarjan

O Algoritmo de Tarjan tem uma complexidade temporal teórica de O((V|+|E|)) (sendo |V| o número de vértices do grafo e |E| o número de arestas).

Após várias tentativas empíricas (para a cidade do Porto), em média, o algoritmo de Tarjan demorou em média 0.004s a executar.

8.3. Algoritmo de divisão de turistas por percursos de acordo com preferências

Este Algoritmo, realizado com base numa explicação do regente da unidade curricular, realizado num grafo bipartido de Pessoas e Pontos de Interesse, tem uma complexidade temporal teórica, no pior caso, de O((E|^2*|PE|*|IE|^2) ou O((E|*|B|*|PE|*|IE|^2) (sendo |E| o número de arestas do grafo bipartido, |PE| o número de arestas a partir de uma pessoa, |IE| o número de arestas a partir de um ponto de interesse e |B| a capacidade dos autocarros).

A função *dividePeople*, chama a função *getPeopleForBus* que atua sobre o grafo bipartido. A complexidade deste algoritmo é O((E|*|PE|*|IE|^2) ou O(|B|*|PE|*|IE|^2).

A complexidade do algoritmo *dividePeople* é tão grande uma vez que é chamada a função *getPeopleForBus* que por sua vez, para além de percorrer os vetores de arestas que partem dos vértices, chama funções da *STL* com complexidade O(N), tais como *vector::erase*.

Após várias tentativas empíricas (para a cidade do Porto), para uma amostra pequena, o algoritmo de agrupamento dos turistas demorou 0.003s a terminar.

Conclusão

Com a realização deste projeto, pretende-se ter mais prática na utilização e aplicação de algoritmos sobre grafos, de modo a produzir uma solução o mais ideal possível para o tema proposto (City Sightseeing: trilhas turísticas urbanas).

Nesta segunda fase, foram implementados algoritmos que iteram sobre o grafo, de modo a encontrar neste caminhos mais curtos entre um ponto de partida e um ponto de chegada, passando por vários POI pelo caminho. Sendo que para tal utilizamos uma adaptação do algoritmo de Dijkstra.

Foi-nos também pedido para agrupar os turistas com percursos semelhantes nos mesmos autocarros de modo a minimizar os recursos gastos (tentar preencher o máximo dos autocarros de modo a diminuir o seu número), tendo sido utilizada uma estratégia baseada em fluxo máximo, num grafo bipartido de turistas e POIs, em que o fluxo representa a capacidade, estratégia esta que nos foi sugerida e explicada pelo professor Rossetti.

A solução efetivamente implementada nesta entrega do projeto tem várias diferenças em relação à perspetiva de solução especificada no relatório da primeira entrega. Esta situação deve-se ao facto de termos percebido, em parte graças a sugestões do professor Rossetti, que várias das nossas ideias não eram as mais adequadas ao nosso problema, ou não foram possíveis de implementar no tempo que tivemos disponível.

Este relatório foi igualmente produzido pelos 3 membros do grupo, sendo que o João implementou o algoritmo de Tarjan para verificar as componentes fortemente conexas. A Márcia implementou e adaptou o algoritmo de Dijkstra para encontrar os caminhos mais curtos. O Pedro implementou o algoritmo utilizado no agrupamento de turistas por autocarro, baseado na ideia de fluxo máximo sugerida pelo regente. Para além disto, todos os membros contribuíram de igual forma para o desenvolvimento da interface e deste relatório.

Bibliografia

- Diapositivos da UC CAL (Concepção e Análise de Algoritmos) presentes no moodle – informações gerais sobre os algoritmos e imagem sobre algoritmo de Dijkstra
- Conceitos gerias sobre pesquisa em profundidade: https://en.wikipedia.org/wiki/Depth-first_search
- Conceitos de Grafos: http://www.inf.ufsc.br/grafos/
- Algoritmos em Grafos: https://www.ime.usp.br/~pf/algoritmos_em_grafos/
- Informações sobre Algoritmo de Dijkstra: https://en.wikipedia.org/wiki/Dijkstra's_algorithm
- Informações sobre a linguagem C++, utilizada nesta UC: http://www.cplusplus.com/
- Imagem a ilustrar o algoritmo DFS: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/?fbclid=IwAR2h1e_mdp5RFgsXjo0drmfS-Y2vwN9uRucXyYzFCfgwW8wpc3ke3-221eU
- Imagem a ilustrar a abordagem DAC: https://www.studytonight.com/data-structures/merge-sort?fbclid=IwAR2R6n_zchgvb-YEpCsagPzZpGdTVqfu2pxvMw4k5o-c59mWXqBOmN3F2kM