Report for Programming Problem 3 - Bike Lanes

Team:

Student ID: 2018285621 | Name: Nuno Marques da Silva

Student ID: 2018285632 | Name: Pedro Tiago dos Santos Marques

1. Algorithm description

O algoritmo inicia com a leitura do *input* e começa a construir a estrutura de dados *adj* (lista de adjacências, fig. 3), que contém as conexões de um ponto de interesse e os seus pesos/distâncias.

1.1. Circuit identification

Para identificar os circuitos, foi utilizado o algoritmo de *Tarjan*. Este identifica os vértices de componentes fortemente conectadas, ou seja, sub-grafos de tamanho máximo onde existe um caminho possível de um vértice para todos os outros.

O algoritmo permite-nos passar de uma lista de adjacências para uma árvore DFS, isolando os ciclos do grafo. Um circuito corresponde a uma sub-árvore da árvore DFS.

No contexto do problema, permite-nos isolar os circuitos a partir das conexões de cada ponto de interesse.

```
Function Tarjan(v, inStack)
  low[v]=dfs[v]=t
  t = t + 1
  push(S, v)
  for each arc {w, d} ∈ adj[v] do
    if dfs[w] = -1 then
      Tarjan(w, inStack)
      low[v] = min(low[v],low[w])
    else if inStack[w] then
      low[v] = min(low[v],dfs[w])
  if low[v] = dfs[v] then
    auxSolution = ∅
    repeat
      w = pop(S)
      inStack[w] = false
      push(auxSolution, w)
    until w = v
```

Fig. 1 – Algortimo de Tarjan

O algoritmo (fig. 1) começa por percorrer linearmente todos os vértices da lista de adjacências para cada ponto de interesse, ignorando os já visitados (usa uma *stack*), à procura de circuitos. Vai guardando um contador do nível da profundidade do ponto de interesse atual do circuito a testar “t” e vai atualizando as estruturas “low” e “dfs”, sendo “dfs[ponto de interesse atual]” o nível de profundidade “t” da primeira vez que este ponto foi descoberto e “low[ponto de interesse atual]” o menor nível de profundidade a encontrar no sub-circuito do ponto de interesse atual. Deste modo, se o nível de profundidade atual (dfs[ponto de interesse atual]) for igual ao menor nível de profundidade que é possível encontrar percorrendo pelos circuitos dos pontos

de interesse adjacentes ao atual, significa que há um circuito, ou seja, através de um ponto, consigo avançar e chegar ao ponto de partida.

Conseguindo reconhecer os pontos de interesse de vários circuitos de uma cidade podemos responder às duas primeiras perguntas.

1.2. Selection the streets for the bike lanes

Para identificar as *bike lanes* criámos uma lista de adjacência nova (adjKruskal, fig. 4) com os pontos de interesse do circuito, as suas conexões e distâncias. Como estes novos caminhos são bidirecionais, podemos usar o algoritmo de Kruskal para encontrar a *minimum spanning tree*, ou seja, uma árvore de peso mínimo que conecta todos os vértices do circuito.

O algoritmo de *Kruskal* (fig. 2) é um algoritmo *greedy* que de um grafo conexo gera uma *minimum spanning tree.* No contexto do problema, permite-nos transformar um circuito noutro circuito equivalente cujas distâncias são mínimas. Por exemplo, para realizar A → C tanto podemos fazê-lo diretamente com um custo de 10 ou passando por B com um custo total de 8; ora, na *minimum spanning tree* apenas vai aparecer a passagem A → B → C, ignorando A → C, pois essa é a que tem o menor custo. Note-se que agora não interessa a direção das ligações entre os pontos de interesse.

Para cada circuito, o algoritmo começa por ordenar os pontos de interesse por ordem estritamente crescente consoante a distância e depois faz as ligações entre os vários pontos de interesse, garantindo a menor distância para cada ciclovia. Assim que todos os pontos têm as ciclovias devidas atribuídas ficamos com uma espécie de árvore DFS simplificada.

```
Function kruskal()
  len = 0
  make_setKruskal()
  sort adjKruskal by distance in increasing
order
  for each edge {d,{u,v}} in adjKruskal do
    if findKruskal(u) ≠ findKruskal(v) then
      len = len + d
      unionKruskal(u, v)
  return len
```

Fig. 2 – Algortimo de Kruskal

Como não é relevante as ligações que fazem parte da árvore, apenas incrementamos uma variável ‘len’ do peso total da *spanning tree*.

2. Data structures

As estruturas de dados principais usadas são as listas de adjacência “adj” e “adjKruskal”. Estas guardam as ligações possíveis entre dois pontos de interesse e a sua distância e seguem os seguintes esquemas:

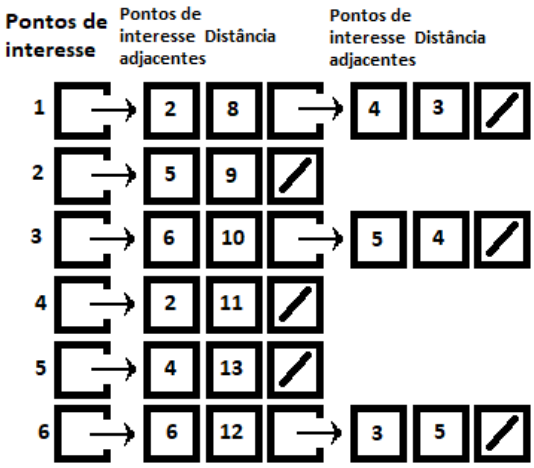

Fig. 3 – Lista de adjacência *adj*

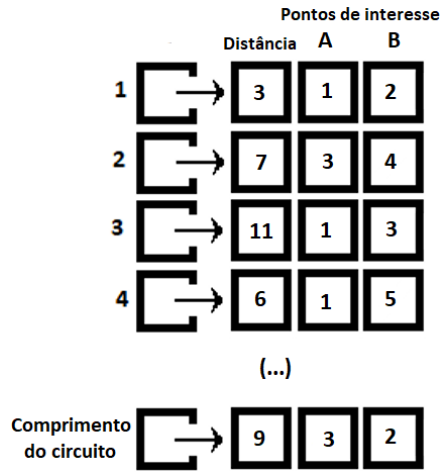

Fig. 4 – Lista de adjacência *adjKruskal*

3. Correctness

O algoritmo de Tarjan permite-nos obter um conjunto dos pontos de interesse que constituem um circuito. Sabendo que um circuito tem no mínimo 2 pontos de interesse, então é possível responder às duas primeiras perguntas de forma correta. Note-se que utilizámos um *array* de *booleans* (inStack, fig. 1) para saber de maneira instantânea se um dado elemento estava na pilha, não sendo necessário percorrer a mesma constantemente. O algoritmo de Kruskal permite obter o comprimento da *minimum spanning tree* de um determinado circuito. Sabendo que este circuito provém do algoritmo de Tarjan, então é possível responder às duas últimas perguntas.

4. Algorithm Analysis

O algoritmo de *Tarjan* tem complexidade temporal O(n+m), sendo “n” o número de pontos de interesse e “m” o número total de ligações entre pontos de interesse. Já quanto à complexidade espacial, o algoritmo é O(4*n+m).

O algoritmo de *Kruskal* tem complexidade temporal O(log(n)*m), sendo, mais uma vez, “n” o número de pontos de interesse e “m” o número total de ligações entre pontos de interesse. Já quanto à complexidade espacial, o algoritmo é O(2*n+m).

5. References

-SIMÕES, Marco [2021]. "Estratégias Algorítmicas 2020/21 Week 10 { Graph Algorithms (MST and APs)” [Apresentação PowerPoint], Coimbra, última consulta a 26 de maio de 2021

-SIMÕES, Marco [2021]. "Estratégias Algorítmicas 2020/21 Week 11 { Graph Algorithms (SCC and MaxFlow)” [Apresentação PowerPoint], Coimbra, última consulta a 26 de maio de 2021