Relatório para o problema de programação 3 – Bike Lanes

Equipa:

N.º estudante: 2018295474 Nome Bruno Ricardo Leitão Faria

N.º estudante: 2018282583 Nome Diogo Alves Almeida Flórido

1. Descrição do Algoritmo

Para este problema ocorremos a dois algoritmos de grafos: o **algoritmo de Tarjan** para encontrar componentes fortemente conexas (circuitos) e o **algoritmo de Kruskal** para encontrar as árvores de extensão mínima para cada circuito (pistas de bicicleta).

1.1. Identificação de circuitos

```
Function Tarjan(v)
  low[v] = dfs[v] = t
  t = t + 1
  push(S, v)
  for each arc (v, w) ∈ A do
    if dfs[w] has no value then
      Tarjan(w)
      low[v] = min(low[v], low[w])
    else if w ∈ S then
      low[v] = min(low[v], dfs[w])
  if low[v] = dfs[v] then
    C = ∅
    repeat
      w = pop(S)
      push(C, w)
    until w = v
    push(Scc, C)
```

```
for each Vertex v:
    if v not in dfs:
        Tarjan(v)
```

Figura 1. Pseudocódigo do algoritmo utilizado para identificar circuitos.

1.2. Seleção das vias para as pistas de bicicletas

```
make_set
  for each vertex i ∈ V do
    set[i] = i
    rank[i] = 0

find(a)
  if set[a] ≠ a then
    set[a] = find(set[a])
  return set[a]

union(a,b)
  link(find(a), find(b))

link(a,b)
  if rank[a] > rank[b] then
    set[b] = a
  else
    set[a] = b
  if rank[a] = rank[b] then
    rank[b]++
```

```
Function Kruskal(G)
  lane_length = 0
  for each vertex v ∈ V do
    make_set(v)

  for each edge {u, v} ∈ E do
    if find_set(u) ≠ find_set(v) then
      lane_length = lane_length + edge.distance
      link(find_set(u), find_set(v))
  return lane_length
```

```
bike_lane_length = 0
set = vector(n, -1)
rank = vector(n, -1)
for each circuit:
    edges = []
    for vertex v in circuit:
        for each connection in circuit:
            if circuit.find(connection.B) != circuit.end():
                edges.push(connection)

        sort_by_distance(edges)
        bike_lane_length = Kruskal(circuit,edges)
        if bike_lane_length > longest_bike_lane_length:
            longest_bike_lane_length = bike_lane_length
        total_bike_lanes_length = total_bike_lanes_length + bike_lane_length
```

Figura 2. Pseudocódigo do algoritmo utilizado para identificar as vias para as pistas de bicicletas e obter os outputs 3 e 4.

2. Estruturas de Dados

Para este problema utilizamos uma struct **connection** constituída por inteiros e uma classe **Map** cujos atributos são inteiros e as estruturas **vetores de vetores de connection**, **vetores de inteiros/booleanos** e **vetores de sets** (não ordenados) **de inteiros.** A linguagem de programação utilizada foi o C++.

Struct connection: **int** POI_A, **int** POI_B, **int** distance -> a via que conecta o ponto de interesse POI_A ao ponto de interesse POI_B tem distance de distância.

Class Map:

Atributos: Para o Algoritmo de Tarjan: **int** t -> iniciado a 1. ID do vértice poço da rede de fluxo atua; **vector<int>** low -> low[v] contém o vértice com menor tempo de descoberta numa subárvore com raiz em v; **vector<int>** dfs -> dfs[v] contém o tempo de descoberta de v; **stack<int> S** -> stack temporária operada num contexto LIFO que vai contendo os vértices de cada componente fortemente conexa; **vector<bool> inStack** -> inStack[v] é true se v pertencer à componente fortemente conexa atual; **vector<unordered_set<int>> circuits** -> contém todas as componentes fortemente conexas do grafo cada uma com um set dos vértices que a constituem; **int largest_circuit_POIs_number** -> número de vértices da componente fortemente conexa mais longa (um dos outputs pedidos).

Para o algoritmo de Kruskal: vector<int> set -> set[v] contém o vértice filho de v na árvore de extensão mínima; **vector<int> rank** -> fator de comparação para decidir que vértice se torna pai do outro; **int longest_bike_lane_length** -> comprimento maior entre as árvores de extensão mínima (um dos outputs pedidos); **int total_bike_lanes_length** -> soma dos comprimentos de todas as árvores de extensão mínima (um dos outputs pedidos).

Métodos: public void findCircuits() -> chama o algoritmo de Tarjan para cada vértice do grafo; **public int getNumberOfCircuits()** -> devolve a resposta à primeira pergunta; **int public getNumberPOIsInLargestCircuit()** -> devolve a resposta à segunda pergunta; **void public findBikeLanes** -> chama o algoritmo de Kruskal para todos os circuitos descobertos com o findCircuits; **public int getLongestBikeLaneLength()** -> devolve a resposta à terceira pergunta; **public int getTotalBikeLanesLength()** -> devolve a resposta à quarta pergunta; **private void readConnections(int num_connections)** -> lê o input e regista conexões entre pontos de interesse; **private void Tarjan(int v) ->** calcula as componentes fortemente conexas do grafo através do algoritmo de Tarjan; **private bool compareConnections(const connection& a, const connection& b)** -> devolve true se a distância da conexão a for menor do que a da b; **private int find_set(int a)**, **private void link(int a, int b)** -> operações de melhoria ao algoritmo de union-find; **private int Kruskal(unordered_set<int> circuit, vector<connection> edges)** -> calcula a árvore de extensão mínima para o circuit dado; **private void notPossible(int q)** -> imprime 0 a todas as respostas.

3. Exatidão

Acreditamos que obtivemos os 200 pontos no Mooshak pelos seguintes motivos:

- O algoritmo produz os resultados corretos;
- O algoritmo de Kruskal foi otimizado através do algoritmo union-find;

4. Análise do Algoritmo

Algoritmo de Tarjan: O(V+A), sendo V o número de vértices e A o número de arestas.

Algoritmo de Kruskal: O(A*log(V)), sendo V o número de vértices e A o número de arestas.

Logo, visto que a nossa solução usa sequencialmente os algoritmos acima indicados, fica então com uma complexidade temporal de **O(A*log(V))**

Complexidade espacial: **O(n*m)** derivado do tamanho das estruturas necessárias aos algoritmos e da estrutura para armazenar o input.