Report for Programming Problem 3 – Bike Lanes

Team:

Student ID: 2018298731 Name João Pedro Pacheco Silva

Student ID: 2018297934 Name Luís Carlos Lopes Loureiro

1. Algorithm description

Para solucionar o problema, optámos por usar dois algoritmos estudados nas aulas teóricas: O Tarjan algorithm e o Kruskal Algorithm com Union-Find.

1.1 Circuit identification

O Tarjan algorithm encontra-se implementado na função “findCircuits” e foi usado para identificação dos circuitos presentes em cada cidade. A função “findCircuits” faz uma travessia dfs do grafo atribuindo um id a cada um dos seus nós e mantendo numa stack aqueles que já foram visitados. Para alem desse valor de dfs, é também atribuído a cada nó um valor low, que é determinado da seguinte forma:

- dfs[v] = id de travessia
- $\text{low}[v] = \min \begin{cases} \text{dfs}[v] \\ \text{dfs}[x], \forall\ x \text{ filho de v em que } \text{arco}(v,x) \text{ corresponde a uma backedge} \\ \text{low}[w], \forall\ w \text{ vizinho de v} \end{cases}$

Caso algum dos vizinhos do nó atual ainda não tinha os valores de dfs e low definidos (ou seja, ainda tenha sido visitado), é feita uma chamada recursiva sobre esse nó. Após percorrer todos os visinhos, a função vê se o nó atual é raiz de um Strongly Connected Component, sendo para isso necessário comparar os seus valores de dfs e low. Caso estes sejam iguais, o nó é raiz de um Strongly Connected Component e todos os nós inseridos recursivamente na stack após ele fazem parte desse mesmo Strongly Connected Component. Cada Strongly Connected Component com mais de 2 nós constitui depois um circuito.

1.2. Selection the streets for the bike lanes

O Kruskal Algorithm encontra-se implementado na função “getLane” e foi usado para determinar a minimum spanning tree dos circuitos e assim poder definir as pistas de ciclismo. Como o Kruskal Algorithm é um algoritmo guloso, é necessário fazer primeiro um ordenamento das conexões entre POIs com base nas suas distâncias. Para isso foi usado o algoritmo Merge Sort. Após o ordenamento, passa-se por cada conexão do circuito, da mais curta para a mais longa, escolhendo apenas as que não resultem num ciclo.

A verificação para evitar ciclos é feita criando um conjunto de sets nos quais os diferentes nós serão incluídos. Inicialmente todos eles pertencerão a sets diferentes, mas à medida que as conexões vão sendo adicionadas à arvore, faz-se também a união dos sets dos POIs que as compõem. O valor desses sets será o fator que determina se a conexão é ou não descartada. Caso o dois nós da conexão pertençam a sets diferentes, a conexão é adicionada à arvore, caso pertençam ao mesmo set a conexão é descartada.

2. Data structures

A representação usada para os grafos foi a Adjacency list, visto que o algoritmo se foca em percorrer todos os vizinhos de um dado nó e não em verificar se dois nós estão conectados.

O armazenamento dos valores de dfs, low, set e rank de cada nó foi feito usando um array de inteiros, no qual se usa o valor do nó como índice para aceder ao valor correspondente (funcionando assim como um hash map).

Por fim, os nós visitados foram guardados recorrendo a uma stack, juntamente com um array de inteiros chamado “onStack” de modo a reduzir a complexidade temporal na operação de pesquisa.

3. Correctness

A representação da Adjacency list para representação dos grados foi a escolha certa, uma vez que com a Adjacency matrix, sempre que precisássemos percorrer os vizinhos de um nó, teríamos de considerar todas as conexões possíveis e ver quais eram validas, deixando assim de ter uma complexidade linear no Tarjan Algorithm. O facto de usarmos o array “onSack” também foi uma escolha correta dado que se fizéssemos a verificação percorrendo a stack, também aumentaria a complexidade temporal.

4. Algorithm Analysis

A complexidade temporal do algoritmo para resolver o primeiro subproblema (Circuit identification) é linear em função do número de nós do grafo, dado que a função “findCircuits” é chamada uma vez para cada nó, a fim de determinar os valores de dfs e low correspondentes e é linear em função do número de conexões do grafo, uma vez que em cada chamada da função são percorridos todos os vizinhos desse nó. Assim temos O(|V| + |A|), sendo V o numero de nós e A o número de conexões.

A complexidade espacial do algoritmo para esse subproblema depende da complexidade espacial envolvida na representação do grafo e das estruturas usadas para armazenamentos dos nós visitados e valores de dfs e low. A representação do grafo é feita usando uma Adjacency list que tem uma complexidade espacial de |V| + |A| e o armazenamento dos restantes valores é feito usando arrays e stacks, que têm uma complexidade espacial de |V|. Assim temos: |V| + |A| + |V| ∈ O(|V| + |A|).

Para além das estruturas há também o espaço necessário devido à recursão da função “findCircuits”, mas como a profundidade máxima da árvore de chamadas recursivas é V, a complexidade em notação O-grande mantem-se igual.

A complexidade temporal do algoritmo para resolver o segundo subproblema (Selection the streets for the bike lanes) depende da complexidade temporal da operação de ordenamento e da complexidade temporal da operação de escolha das conexões.

Na segunda operação percorre-se as conexões do circuito e para cada uma procura-se os sets aos quais os seus POIs pertencem. Devido à aplicação do Path compression a procura do set leva log*(|W|) passos, sendo W o número de POIs do circuito.

$$\log^*(n) = \begin{cases} 0, \text{se } n \leq 1 \\ 1 + \log * (\log(n)), \text{caso contrario} \end{cases}$$

logo temos: |E| log*(|W|)) ≤ |E| · 5, sendo E o número de conexões desse circuito.

Em relação ordenamento usamos o algoritmo Merge Sort, que tem complexidade |E| log(|E|). E portanto ficamos com: E| log(|E|) + |E| · 5 ∈ O(|E|log(|E|)).

A complexidade espacial do algoritmo para esse subproblema é linear em função do número de conexões do circuito devido ao uso do Merge Sort e linear em função do número de POIs do circuito devido às estruturas para representação dos sets e para armazenamento da profundidade destes. Assim temos: O(|W| + |E|).

Considerando o pior caso em que todo o grafo é um Strongly Connected Component, O algoritmo na globalidade terá uma complexidade temporal de O(|V| + |A|log(|A|)) e uma complexidade espacial de O (|V| + |A|).

5. References

Slides das Aulas teóricas: ” Week08 - T - Graph Algorithms (Graph Traversal)”, “Week 10 – Graph Algorithms (MST and APs)” e “Week 11 – Graph Algorithms (SCC and MaxFlow))”