Problema de Programação 3 - Bike Lanes

Student ID: 2016228735 Name: Afonso Matoso Magalhães
Student ID: 2017251509 Name: Leandro Pais

Maio 2021

1 Algorithm Description

O trabalho pode ser divido em duas partes distintas, primeiramente temos as duas primeiras questões que resolvemos utilizando o algoritmo de *tarjan*. E depois temos as restantes questões que foram resolvidas utilizando o algoritmo de *kruskal* abaixo estão as respetivas explicações.

1.1 Circuit Identification

No algoritmo de *tarjan*, atravessamos o grafo utilizando *dfs* à medida que o fazemos marcamos cada nó como visitado e temos um contador que vai marcando cada nó com um id e metemos o os dfs e low desse nó igual ao id. Posteriormente fazemos push para a pilha desse nó.
Se o nó ainda não tiver sido visitado é feita a chamada recursiva para os vizinhos.
Depois de voltar desta recursividade atualizamos o valor de low para o mínimo entre ele próprio e o vizinho. Depois verifica se o vizinho está na stack e se estiver significa que este faz parte do SCC e atualiza novamente o valor de low para o mínimo entre os dois.
Por fim, depois de todos os vizinhos percorridos, verificamos se o nó atual tem low igual a dfs e se for o caso estamos perante a raiz do SCC e assim, podemos então fazer pop de todos os nós. E é neste ponto que colocamos todos os SCC's numa matriz de circuitos.

1.2 Street selection for bike lanes

No algoritmo de kruskal temos:
array set[] - guarda o conjunto a que pertencem os vértices (um vertice a cujo pai é b tem o set de b - set[a] = b).
array rank[] - altura de um set (set mais pequeno junta-se sempre ao maior).
array num_verts[] (não faz parte do algoritmo mas nos usamos) - para guardar o número de vértices num set, de modo a quando juntarmos 2 sets, somarmos o número de vértices de cada set para o índice do vértice correspondente à raiz da árvore com maior rank. No fim de tudo, termos logo a soma das arestas da

lane no índice do vértice correspondente à raiz da MST.
make_set() - inicializa arrays rank[] a 0, set[] com os vértices respetivos e num_verts[] com 1 (pois todos os sets ao inicio têm um vertice que é o próprio vértice)
find_set(a) - vai buscar o set de um vértice (caso seja diferente do próprio vértice vai verificar se o set está correto, por exemplo quando se juntam sets de vários vértices só se atualiza o da raiz e não os dos filhos)
link() - juntar 2 sets, junta-se sempre a árvore com menor rank à com maior rank. Caso os ranks sejam iguais, incrementa-se 1 ao da segunda árvore.
get_lane_comps() - algoritmo de kruskal com union find e melhorias dos slides, em que percorre todas as arestas já ordenadas e caso os sets dos vértices de uma aresta sejam diferentes, a distancia é adicionada à lane_sum e são unidos os sets (árvores). Assim que o número de vértices na árvore MST é o suficiente, então retorna-se a soma das arestas dessa lane.

2 Data Structures

No que toca às estruturas de dados utilizados, temos as mais simples que são as matrizes, utilizamos várias, e como exemplo temos a matriz de adjacência e a matriz de adjacência com pesos utilizadas para descrever o grafo.
Implementámos também uma pilha para os vértices que é utilizada no algoritmo de *tarjan*.
E por fim, temos uma estrutura *edge* onde, como o nome indica, guardamos informação sobre as ligações adjacentes no grafo. Esta estrutura está enquadrada numa lista ligada é depois necessária para o algoritmo de *kruskal*.

3 Corretcness

Tanto o tarjan como o kruskal são algoritmos que foram abordados na aula e como tal sabemos que ambos estão corretos. Quanto às provas em si, existem várias abordagens possíveis de encontrar online para nós interessa-nos apenas que ambos os algoritmos são completos e corretos e como tal funcionam para qualquer grafo.
Assim, como as respostas obtidas resultam da aplicação de dois algoritmos corretos podemos afirmar que a solução é também correta.

4 Algorithm Analysis

Como as respostas pedidas envolvem dois algoritmos diferentes importa explorar a complexidade temporal separadamente.
Assim, temos que para as duas primeiras perguntas, como utilizamos o algoritmo *tarjan* que corre em tempo linear mas é chamado por um algoritmo *dfs* utilizando uma matriz de adjacência, podemos esperar uma complexidade temporal $\mathcal{O}(V+E)$, onde V é números e vértices e E é o número de arestas. Quanto a complexidade espacial temos $\mathcal{O}(n^2)$ devido à matriz de adjacência.

Quanto às restantes perguntas como é utilizado o algoritmo de *kruskal* onde o runtime é dominado pela organização das arestas e como estamos utilizar a implementação C do *qsort* que tem complexidade temporal $\mathcal{O}(ElogE)$ onde E é o número de arestas, assim no fim, para o *kruskal* temos $\mathcal{O}(ElogV)$. No que toca à complexidade espacial podemos novamente aproximar a $\mathcal{O}(n^2)$.

References

[1] Não utilizámos outros fontes para além do que foi fornecido nas aulas.