Report for Programming Problem 3 - Bike Lanes

Team: 2015236199_2017267408

Student ID: 2015236199 Name: Tiago José Rodrigues Menezes

Student ID: 2017267408 Name: Pedro Carreiro Carvalho

1. Algorithm Description

1.1. Circuit Identification

Inicialmente é recebida a informação sobre o grafo e esta é armazenada numa matriz de adjacência.

Para identificar os circuitos existentes é necessário compreender a definição dos mesmos, sendo esta, um conjunto de *POI (Points Of Interest)* nos quais, estando em qualquer *POI*, é possível chegar a qualquer outro *POI* do mesmo circuito. Ao analisar esta definição é facilmente notado que a definição desta é equiparável à de uma *SCC (Strongly Connected Component)*, substituindo apenas *POI* por vértice. Para resolver este problema utilizamos o *algoritmo de Tarjan*, que é utilizado para descobrir as *SCC* de um grafo.

Este algoritmo recebe como input o vértice atual a ser analisado. Para o funcionamento deste é utilizado um contador, que define o valor de *DFS (Depth First Search)* e o valor de *low* iniciais de cada vértice. Este contador é aumentado a cada chamada do algoritmo. São também utilizados 2 vetores que contêm os valores de *DFS* e *low* de cada um dos vértices do grafo e uma stack na qual vão estar os nós pertencentes à *SCC (Strongly Connected Component)* a ser descoberta na iteração atual. Quando a função é chamada é aumentado o contador e é igualado o valor *low* e *DFS* do vértice ao valor atual do contador e o vértice atual é colocado na stack. De seguida, para cada aresta com origem no vértice atual são verificadas duas condições: se o *DFS* do vértice destino da aresta não tem valor e se o vértice destino da aresta é pertencente à stack. Caso se verifique a primeira condição acima descrita é chamado o algoritmo com o vértice destino e, após a execução dessa chamada, é definido o valor de *low* do vértice origem com o mínimo entre o valor *low* atual (definido pelo contador quando o algoritmo foi chamado com o vértice origem) do vértice origem e o valor de *low* do vértice destino. Caso o vértice destino já pertença à stack é definido o valor de *low* do vértice origem com o mínimo entre o valor *low* atual do vértice origem e o valor de *DFS* do vértice destino. De seguida é verificado se o valor de *DFS* e o valor de *low* do vértice são iguais. Caso o sejam, significa que a *SCC* já foi descoberta na sua totalidade e que os vértices que formam a mesma estão todos na stack. Assim sendo é criado um novo vetor ao qual são adicionados todos os vértices na stack enquanto se procede à remoção dos mesmos desta e é adicionado este vetor a um array de vetores que

contém todas as *SCC* do grafo. Este algoritmo é chamado para todos os vértices do grafo que ainda não tenham sido visitados.

1.2. Selection the streets for bike lanes

Para decidir quais as ruas que definem o percurso de ciclovias é utilizada a função de Kruskal. Esta função utiliza uma abordagem greedy para criar a *MST (Minimum Spanning Tree)* de um grafo não direcionado. Uma *MST* é o subconjunto de arestas de um grafo que permitem que todos os vértices deste estejam conectados com o mínimo peso possível nas arestas. Neste problema, uma *MST* corresponde às ciclovias mínimas a serem construídas. Neste algoritmo é necessário ordenar as arestas por peso e, de forma a tornar essa operação mais simples, utilizamos a estrutura edge (Descrita na secção 2). Ao adicionar os edges para cada circuito é verificado se os dois vértices que formam a aresta fazem parte do circuito a ser analisado, garantindo assim que a aresta também pertence a esse circuito, e verificamos, caso existam duas arestas entre os mesmos vértices, qual é a de menor peso e adicionamos apenas essa à lista de arestas, garantindo assim a correção dos valores a serem analisados pelo algoritmo de Kruskal.

Inicialmente é definido um valor “t” que contém o valor total da *MST*. De seguida são criados os sets para todos os vértices (Estrutura descrita na secção 2). O set de cada nó, inicialmente, tem o seu rank a 0 e tem como seu pai o próprio nó. Após a criação dos sets são ordenadas as arestas do grafo, por ordem crescente do seu peso. Por cada uma dessas arestas, caso o vértice de origem e o vértice de destino não pertençam ao mesmo set, é adicionado o peso da aresta a “t” e são unidos os sets de cada um dos vértices. Para realizar esta união é utilizado o rank dos vértices. A root do set com o rank mais elevado fica como pai da root com o rank menor. Caso as roots tenham o mesmo rank apenas é aumentado o rank da root do set que contém o vértice destino. Este algoritmo é utilizado em cada um dos circuitos de forma a descobrir qual o circuito com a maior ciclovia e o tamanho total das ciclovias existentes.

2. Data Structures

Para armazenar o input é criada uma matriz de adjacência *adj [N] [N]* na qual é armazenada os vértices do grafo e as arestas do mesmo, incluindo o seu peso. Para o primeiro algoritmo são criados dois arrays, um array *DFS [N],* que contém o valor de *DFS* de cada nó do grafo, e um array *low [N]*, que contém o nó mais abaixo na *DFS* a que o nó do índice consegue chegar. Para armazenar as respostas às duas primeiras perguntas é utilizado um vetor que contém vetores de inteiros, sendo que cada um desses vetores contém os índices dos vértices que formam o subgrafo que é uma *Strongly Connected Component* do grafo inicial, sendo assim o número de *SCC* o número de vetores existentes e o maior *SCC* o tamanho do maior destes vetores. Para o segundo algoritmo são criadas duas estruturas: set e edge. Set contém o rank e o pai de um nó do grafo. Edge contém a origem, o destino e o peso de cada uma aresta do grafo. De seguida são criados dois vetores, um que contém os sets e um que contém

os edges, utilizado para armazenar os sets e os edges do grafo a ser analisado pelo algoritmo de Kruskal.

3. Correctness

Tendo obtido a pontuação de 200 no Mooshak iremos explicar o porquê da correção da nossa abordagem.

Para o primeiro problema é necessário identificar os subgrafos que contêm os vértices e arestas de forma a que cada vértice desse subgrafo consiga aceder a qualquer outro ponto do mesmo. Para obter esses subgrafos foi utilizado o algoritmo de Tarjan, utilizando o pseudocódigo fornecido pelo professor nas aulas para a implementação do mesmo, garantindo assim a correção da implementação do mesmo. Este algoritmo é o correto a ser utilizado nesta situação pelo que foi descrito na secção 1.1.

Para as duas últimas perguntas é necessário descobrir o conjunto de arestas, num grafo não direcionado, que formam o caminho mais curto de forma a que todos os vértices do grafo consigam aceder a qualquer outro vértice do mesmo. De forma a obtermos este caminho foi utilizado o algoritmo de Kruskal, usando o pseudocódigo fornecido pelo professor nas aulas, garantido assim a correção do mesmo. Este algoritmo é o correto a ser utilizado nesta situação pelo que foi descrito na secção 1.2.

4. Algorithm Analysis

Relativamente à complexidade temporal, o algoritmo de Tarjan tem uma complexidade de *O(V + A)*, sendo *V* o número de vértices do grafo e *A* o número de arestas do mesmo, pois cada nó é visitado apenas uma vez e o algoritmo considera cada aresta no máximo uma vez. O algoritmo de Kruskal tem uma complexidade de *O(A log (V))* pois entre todas as operações realizadas neste algoritmo o sort das arestas e a função *find* utilizada nos sets têm esta complexidade. No entanto no processamento de informação para o algoritmo de Kruskal é usada uma função com complexidade *O(SV * V)* sendo *SV* o número de vértices do circuito a ser analisado. Assim sendo, a complexidade temporal da nossa abordagem é a soma destas 3 complexidades.

Relativamente à complexidade espacial, para o algoritmo de Tarjan são utilizados 2 vetores de tamanho *V*, ou seja, *O(V)*, e é utilizada uma matriz de tamanho *V* por *V*, ou seja, *O(V^2)*. Assim sendo, para o algoritmo de Tarjan a complexidade espacial é de *O(V^2)*. Para o algoritmo de Kruskal é utilizado um vetor com os sets de cada vértice, *O(V)* e é utilizado um vetor para armazenar todas as arestas pertencentes ao circuito a ser analisado, *O(A)*. Concluindo, para o algoritmo de Kruskal a complexidade varia consoante o número de vértices e arestas do circuito, podendo haver mais arestas do que vértices ou o contrário.

5. References

Como referência foi apenas utilizado os slides fornecidos pelos professores .