Análise e Síntese de Algoritmos
2017-2018
1ºProjecto

Introdução

O objetivo deste projecto é o desenvolvimento de uma solução eficiente que ao receber como input dado número de vértices (representativos dos pontos da rede de distribuição) e arestas,(ligações entre os mesmos pontos) calcula o:

1. Número de componentes fortemente ligadas (ligações entre estes pontos).
2. Número de ligações entre as componentes fortemente ligadas(explicitando-as no output do programa).

Proposta de solução

O algoritmo concebido tem por base 4 passos:

1. Recolha de input e criação de um grafo dirigido a partir deste (utilizando uma lista de adjacências).
2. Aplicação duma DFS usando o algoritmo de Tarjan para agrupar os vértices que pertencem à mesma componente.
3. Descobrir o número total e quais as ligações entre as SCC.
4.Ordenação e display do output.

Desenvolvimento da solução

Estrutura de dados e representação.

Foi escolhido para representar o grafo uma lista de adjacências(**uma lista de listas ligadas**) onde cada índice da lista representa um vértice do grafo e cada vértice numa sub-lista representa uma aresta. A introdução de novas ligações tem custo O(1)(pois as inserções ocorrem no início duma sub-lista).

Pode deduzir-se que vértices que possuam a sua sub-lista vazia, formam uma SCC que consiste apenas neles próprios.

Esta representação é utilizada nos passos 2 e 3 da proposta de solução(no passo 2 na altura de escolher o novo vizinho do vértice a ser iterado para aplicar *tarjanVisit()** e no passo 3 quando estiver a ser iterado um vértice que pertence a uma SCC de modo a saber quais são os seus vizinhos (para detectar as ligações entre as SCC).

Outras estruturas relevantes são 2 vectores de vectores(std::vector< std::vector<int> >):

Pedro Esteves, n°83541
José Carvalho, n°83495

1.O primeiro, (*vertexsBySCCs*) é usado no passo 2 para quando estão a ser geradas as SCCs conseguir agrupar os vértices que pertencem à mesma SCC num sub-vector.

2.O segundo *(allConnecs)* é usado no passo 3 para guardar em cada sub-vector as ligações entre SCCs que irão posteriormente ser ordenadas no passo 4.
Para os passos 2,3 e 4 para além do grafo e das estruturas previamente descritas foram utilizados std::vector e arrays nativos da linguagem C++.

**tarjanVisit()* é a função do programa que executa a DFS onde é aplicado o algoritmo de Tarjan.

Suporte Teórico:

A propriedade de **invariância da Stack** do algoritmo de Tarjan apresentado nas aulas e aplicado no projeto(passo 2 da Prosposta de Solução) consiste em:

-Sempre que um vértice é visitado, ele é adicionado no topo da stack, porém quando a chamada recursiva do tarjanVisit() retorna os vértices não são necessariamente removidos da stack, isto apenas acontece se não existir **pelo menos um** path no grafo que parta do último vértice adicionado ao anterior a este na stack.

A invariância da Stack colabora com outra propriedade mencionada na bibliografia como **bookkeeping.** Esta última consiste em:

-Após um vértice ser visitado, este vértice apesar de poder não visitar alguns dos seus vizinhos (porque entretanto estes também foram visitados), compara o seu valor lowkey com o valor lowkey do seu vizinho já visitado e atualizará o seu caso o do seu vizinho seja menor.

Isto é fulcral porque a condição que ativa o processo de "popping" dos vértices da stack apenas é verdadeira se o *discovery time* de um vértice for igual ao seu valor *lowkey*.

Bibliografia:
https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Implementação e complexidade do algoritmo

Para além do grafo cuja complexidade temporal de construção é O(V+E), existem outras estruturas, que têm de ser inicializadas para o passo 2 na proposta de solução ser executado:

1.Array booleano para saber se os vértices foram ou não visitados.
2.Array booleano para saber se os vértices estão na stack.
3.Array de ints que simula a stack.
4.Array de ints com o discovery time dos vértices.
5.Array de ints com os lowkeys(ou *low values*) dos vértices.

Pedro Esteves, n°83541
José Carvalho, n°83495

A complexidade temporal de inicialização destas 5 estruturas é majorada por O(V).

A DFS à qual vai ser aplicada o algoritmo de Tarjan tem complexidade temporal de O(V+E) pois cada vértice do grafo (V) vai ser considerado para a DFS e caso invoque o *tarjanVisit()* todos os seus vizinhos(E) vão ser ponderados para uma chamada recursiva.

No terceiro passo, no que toca à deteccção de ligações a complexidade temporal é O(V+E) pois a estrutura *vertexsBySCCs* vai conter exatamente todos os vértices, organizados por componente que vão ser iterados(V), e para cada um desses vértices os seus vizinhos(E) também vão ser percorridos com o objetivo de registar as SCC destes caso elas sejam diferentes da SCC a ser iterada.

No quarto passo, a estrutura *allConnecs* que contém as ligações entre as SCCs é ordenada com std::sort() da biblioteca <algorithm> que aplica uma versão do algoritmo QuickSort cuja complexidade temporal é O(E logE) chamada "IntroSort" que faz tracking do nível de recursão e como tal evita o pior caso do QuickSort que é na verdade $O(V^2)$ mudando o algoritmo de ordenação para um HeapSort caso se aperceba que o pior caso é muito provável.

Bibliografia: https://en.wikipedia.org/wiki/Introsort

Pode-se concluir que a complexidade temporal desta solução é (E logE) sendo majorada pelo processo de ordenação que ocorre no quarto passo.

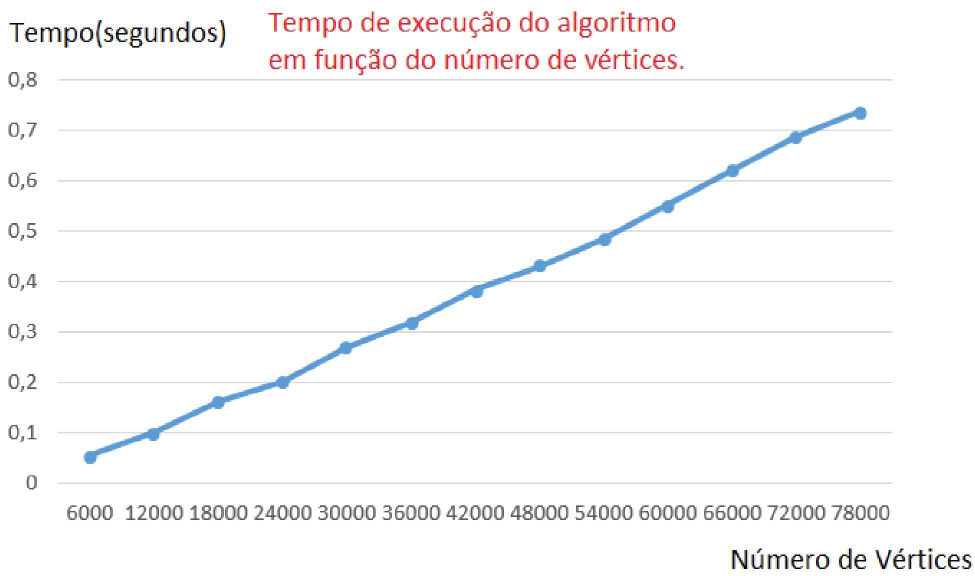
A complexidade espacial desta solução é O(V+E) pois trivialmente se percebe que a construção do grafo tem esta complexidade espacial O(V+E). A estrutura *vertexsBySCCs* é O(V) pois contém exatamente todos os vértices separados por SCC(não há vértices repetidos nesta estrutura) e a estrutura *allConnecs* é O(V+E) pois existe um vetor para cada SCC(no pior caso, número de SCCs= E) cujo o conteúdo são *ints* que representam as ligações para as outras SCCs (no pior caso, o número de ligações é E).

Avaliação experimental dos resultados

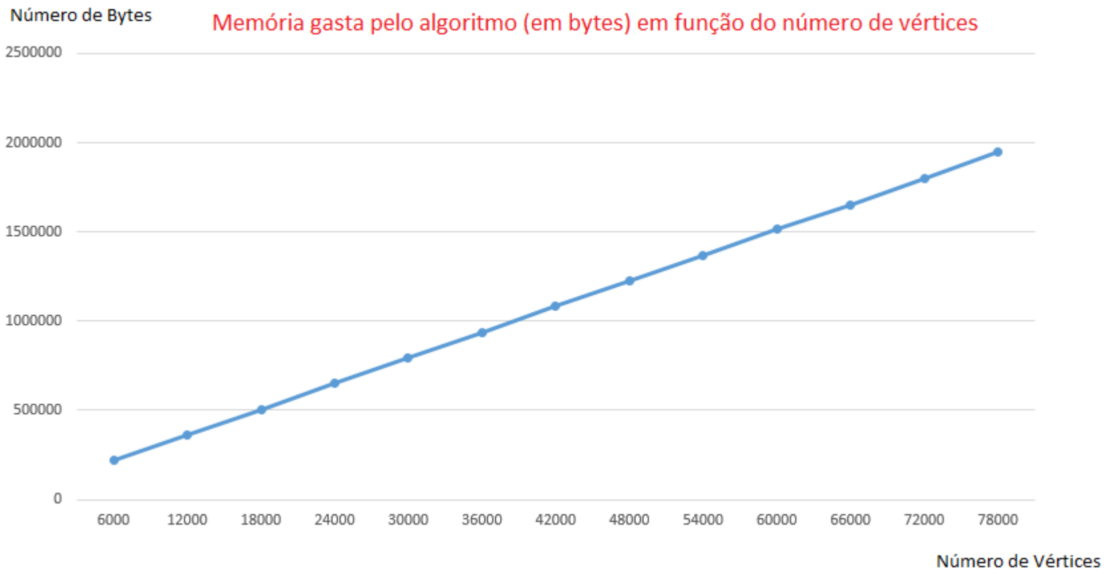
Aqui estão apresentados os resultados de execução do algoritmo para números sucessivamente maiores de vértices e de arcos do grafo**(sendo que para cada teste o número dos vértices e arestas é igual).**

No que toca ao gráfico da **complexidade temporal** para cada ponto no gráfico foram gerados 3 *inputs* diferentes(a partir do gerador de testes disponível na página da cadeira) e cada um foi executado 5 vezes, sendo registada a média dos 15 testes para cada ponto.

Para o gráfico **da complexidade espacial**, foram gerados para cada ponto 3 inputs diferentes e calculada a sua média (cada um dos inputs foi executado apenas uma vez, pois a memória utilizada pelo programa não varia consoante os testes).
Os testes foram executados num ThinkPad Carbon X1, 2nd gen.



Verfica-se empiricamente que a **complexidade temporal** do algoritmo em função do número de vértices e arcos do grafo é O(E logE).



Verfica-se empiricamente que a **complexidade espacial** do algoritmo em função do número de vértices e arcos do grafo é O(V+E).

Pedro Esteves, n°83541
José Carvalho, n°83495