Relatório 1º Projecto de ASA

Grupo 154
Francisco Sousa 86416 & João Daniel Silva 86416

23 de Março de 2018

1 Introdução

No âmbito da cadeira de Análise e Síntese de Algoritmos foi-nos proposto desenvolver um projeto que consiste em identificar sub-redes regionais numa rede de distribuição de produtos, considerando as rotas atuais da mesma em que qualquer ponto de distribuição enviar produtos para qualquer outro ponto da rede regional. Isto é, se um ponto u da rede de distribuição tem uma rota para um ponto v e do ponto v também existe uma rota para o ponto u, então ambos os pontos fazem parte da mesma sub-rede regional.

Assim, representamos o problema como um grafo orientado, no qual aplicaremos um algoritmo de procura de componentes fortemente ligadas (SCC), especificamente o **algoritmo de Tarjan** apresentado na aula teórica.

2 Descrição da Solução

Neste grafo, os vértices correspondem a um ponto de distribuição e as arestas às rotas entre os mesmos.

Decidimos fazer a nossa solução em C++, para facilitar a implementação da stack e de algoritmo *sort*, e também para aprofundar o conhecimento nesta linguagem.

2.1 Estruturas

Como no projeto não é necessário adicionar ou remover vértices, o grafo foi representado como sendo um *array* de inteiros **grafD**, em que cada par corresponde a uma aresta. Para cada vértice cghegámos à conclusão que precisávamos de armazenar 5 propriedades:

1. índice da primeira ocorrência como origem de uma aresta no *array* grafo;
2. tempo de descoberta para o algoritmo de Tarjan;
3. menor valor de descoberta atingível por um arco para trás/cruzamento na sub-árvore (low);
4. informação se está contido na stack;
5. número da SCC em que está contido.

Para armazenar esta informação recorremos a um outro *array* de inteiros **tabelaV**, em que cada 5 elementos corresponde às propriedades de um único vértice.

Como no output é esperado que cada ligação entre scc's tenha como origem e destino o mínimo de cada scc, temos outro *array* que armazena isto **scc**.

Para facilitar o acesso às variáveis necessárias para a chamada recursiva do Tarjan, recorremos ainda a uma estrutura argumentos **args_struct**. Esta contém ponteiros para o grafo, stack, a tabela de vértices, tabela de mínimos de cada scc, além do tempo atual de visita e as dimensões do grafo.

2.2 Algoritmo

Começamos por ler do input o tamanho do grafo (número de vértices V e arestas E), para poder criar as estruturas **grafD** e **tabelaV**. De seguida, populamos o primeiro com o resto das ligações lidas. Procedemos à ordenação das arestas para ter a certeza que as com a mesma origem estavam juntas **grafO**, facilitando percorrer as adjacências de um vértice, atualizando simultaneamente a **tabelaV**. Para isto, utilizamos um *array* auxiliar **auxgrafo** de tamanho E que simula a ordem das ligações, necessitando de recorrer a uma função *lambda*, evitando colocar o grafo global, e que facilita a comparação de vértices. Aplicamos o Tarjan, atualizando a **tabelaV** com o número da scc a que o vértice pertence, e o *array* de minímos **scc** com o menor vértice de cada scc.

De seguida, substituímos ambos os vértices de cada ligação armazenados em **grafO** pelo menor valor da scc a que pertencem. Como isto atualiza as ligações, estas poderão não estar ordenadas. Assim, ordenamos de novo.

A substituição vai fazer com as ligações *(x,y)* entre scc's sejam identificáveis:

- Pares *(a,a)* representam ligações dentro da própria scc;

- Pares *(u,v) (u,v)* seguidos representam ligações repetidas entre scc's.

Assim, filtrando os dois casos acima, conseguimos uma lista de ligações únicas entre scc's. O tamanho máximo desta será o mínimo entre o número total de ligações E e a soma da progressão aritmética até ao número de scc's $\frac{n(n-1)}{2}$, que representa o número máximo possível de ligações entre scc's.

Com isto, conseguimos imprimir o output.

3 Análise

3.1 Teórica

A parte inicial do programa tem complexidade $O(E)$ na leitura do input e inicialização do grafo, e $O(V)$ na inicialização da tabela de vértices.

O algoritmo de ordenação utilizado foi o std::sort que garante complexidade de $O(Elog(E))$:), ao ordenar o **auxgrafo**. Isto ainda precisa de mais $O(E)$ para ordenar concretamente o grafo.

O algoritmo de Tarjan tem complexidade $O(V+E)$. A verificação da presença do vértice na stack é em tempo constante, pois temos esta informação armazenada na (tabelaV).

A substituição das ligações entre vértices por caminhos entre scc's demora $O(E)$, a sua reordenação adiciona tempo $O(Elog(E))$, e a filtragem tempo $O(E)$.

Imprimir o output demora no máximo $O(E)$.

Assim, a complexidade final do algoritmo é $O(V + ElogE)$.

3.2 Experimental

Apoiados pelo gráfico 1, podemos verificar que a execução do algoritmo é linear. Um pequeno desvio da linearidade é explicado pela aplicação do sort, em que considerámos teoricamente que a complexidade seria $O(V + ElogE)$.

O gráfico 2 mostra como a memória é alocada e libertada durante a execução do programa. O maior pico ocorre logo na leitura e armazenamento do input, diminui depois da ordenação do grafo, e tem um novo aumento devido à criação do *array* para guardar as ligações filtradas entre as scc's.

4 Referências

- Introduction to Algorithms, Third Edition: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein September 2009 ISBN-10: 0-262-53305-7; ISBN-13: 978-0-262-53305-8

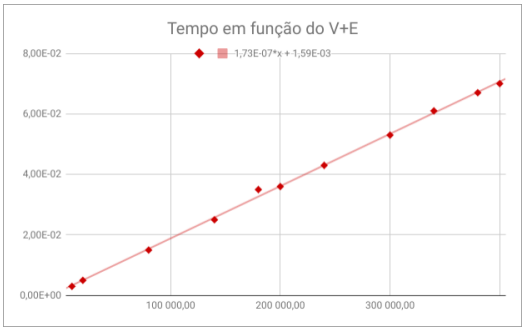

Figura 1: Variação do tempo em função do V+E

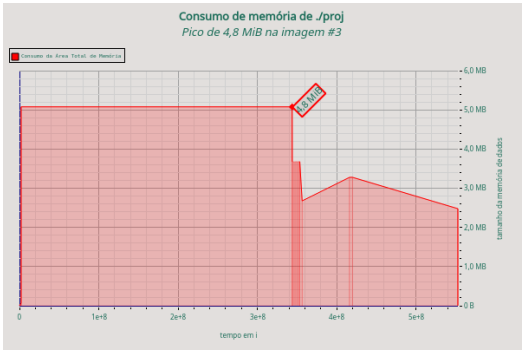

Figura 2: Variação da memória utilizada em função do tempo de execução

- https://www.geeksforgeeks.org/c-qsort-vs-c-sort/
- https://stackoverflow.com/questions/7627098/what-is-a-lambda-exp...
- https://stackoverflow.com/questions/936687/how-do-i-declare-a-2d-...