**Análise e Síntese de Algoritmos**
**Relatório do 1º Projeto**
Grupo 95
89414 - Andreia Pereira | 89433 - Diogo Pacheco

**Introdução:**
O presente relatório aborda a solução encontrada para o problema proposto no 1º projeto da cadeira de Análise e Síntese de Algoritmos, do 2º semestre do ano letivo de 2018/2019.

O problema consiste em fazer uma auditoria a uma rede de routers, que poderá, ou não, estar dividida em sub-redes, e posteriormente retirar um conjunto de informações sobre essa mesma rede.

O *input* fornecido descreve a rede de routers e informa-nos do número de routers da rede (N), do número de ligações entre routers na rede e dessas mesmas ligações. Considera-se que cada router é identificado por um inteiro entre 1 e N, e que cada sub-rede é identificada pelo router com maior identificador a esta pertencente.

Através do input fornecido, o programa deverá calcular o número de sub-redes existentes na rede e os identificadores de cada uma das mesmas, o número de routers que, quando removidos individualmente, resultam no aumento de sub-redes e o número de routers da maior sub-rede formada após a remoção de todos os routers descritos anteriormente.

**Descrição da Solução:**
O programa foi implementado em C++.

Para a representação da rede recorremos a um grafo não dirigido sob a forma de uma lista de adjacências, no nosso caso um std::vector de std::vectors de inteiros. Este vetor de vetores, juntamente com outros dados como: o número de vértices, um vector que guarda os identificadores das SCC's e outro vetor que guarda os vértices que são pontos de articulação, foram mantidos numa classe Graph, que representa o grafo. Temos também um contador global de "tempo" chamado *d_time*.

Identificamos, neste caso, os routers como vértices do grafo, as ligações entre eles como arestas do grafo e as sub-redes como componentes fortemente ligados do grafo.

A deteção de vértices essenciais ao grafo, ou pontos de articulação, como são conhecidos, foi realizada através de uma variação do algoritmo de Tarjan. Um vértice é chamado ponto de articulação se, quando removido, aumenta o número de componentes fortemente ligados do grafo.

De um ponto de vista mais abstrato, a nossa ideia foi:

1. Inicializar os vetores auxiliares ao algoritmo de Tarjan.
2. Aplicar a cada vértice não visitado a função *visit* (posteriormente explicada).
3. Caso tenham sido encontrados pontos de articulação, “removê-los” do grafo, reinicializar os vetores *low, discovery* e *on_stack,* e repor *d_time* a 0.
4. Voltar a correr o algoritmo de Tarjan com a função *find_biggest_scc* para encontrar o tamanho da maior SCC após a “remoção” dos pontos de articulação.

A função *visit* é a função que efectivamente aplica o algoritmo de Tarjan. Esta função tem o intuito de encontrar as SCC’s existentes, os vértices que são pontos de articulação e um *preemptive_biggest*, que é o tamanho da maior SCC encontrada. Isto é útil pois dispensa a re-aplicação do Tarjan caso não haja pontos de articulação. Esta função consiste em:

1. Marcar o *low* e *discovery* do vértice com o valor atual de *d_time* e incrementar este último;
2. Adicionar o vértice à *stack*;
3. Para cada vértice adjacente, se não estiver visitado, aplicar a função *visit*;
4. Após a finalização do *visit* ao adjacente, atualizar o valor de *low* do vértice atual para o mínimo entre o atual *low* e o *low* do vértice adjacente;
5. Ainda dentro do caso de o vértice adjacente ser não visitado, verificar se o vértice atual é ponto de articulação, segundo as condições posteriormente apresentadas;
6. Caso o vértice adjacente não tenha sido visitado e estiver na *stack*, atualizar o *low* do vértice atual para o mínimo entre o *low* atual e o *discovery* do vértice adjacente;
7. Se o *low* e o *discovery* do vértice atual forem iguais, remover elementos da *stack* até encontrar o vértice atual. Este passo inclui encontrar o identificador da SCC e atualizar se necessário o *preemptive_biggest.*

Segundo o nosso algoritmo, um vértice é ponto de articulação quando se verifica uma das seguintes condições:

1. Quando o vértice é raiz de uma árvore DFS e tem mais de 2 vértices filhos;
2. Quando não é raiz de uma árvore DFS mas um dos vértices da sub-árvore de que é raiz não tem ligação com um vértice anterior a ele na procura DFS.

No nosso algoritmo, escolhemos usar um contador local a cada chamada recursiva da função *visit* para saber quantos filhos tem um vértice (*int children*) e um array de inteiros previamente alocado de tamanho N para guardar os predecessores de cada vértice (*int parent[N]*). De tal modo, identificamos o primeiro caso com a condição:

*parent[current] == NIL && children >=2*,

e o segundo caso com a condição:

*parent[current] != NIL && low[adjacent] >= discovery[current]*.

A função *find_biggest_scc* tem a mesma estrutura que a função *visit* mas tem como único intuito encontrar o tamanho da maior SCC. Para esta função é passada como argumento, para além do necessário, um vetor de booleanos (*is_ap[N]*), que nos permite ignorar os pontos de articulação, podendo assim calcular as SCC's resultantes da remoção dos mesmos sem ter de os chegar a remover, operação esta que seria muito mais custosa.

Para a ordenação dos identificadores dos SCC's, utilizamos a função *std::sort()* da biblioteca *algorithm.*

**Análise Teórica:**

Sendo V o número de vértices do grafo e E o número de ligações, temos:

- **Algoritmo Tarjan modificado:** O(V+E).
- **sort():** O(N logN), sendo N o número de sub-redes da rede.

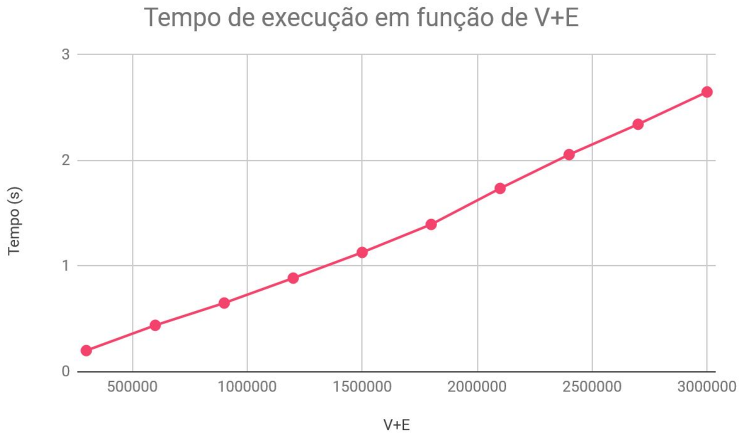
**Análise Experimental dos Resultados:**

O computador utilizado para a realização dos testes tem um processador Intel Core i7-7500U CPU @ 2.70GHz com 8GB de memória, a correr o Arch Linux.

Para podermos fazer uma análise mais extensa e expressiva, recorremos ao gerador de problemas disponibilizado pelos docentes. Corremos o nosso programa com os testes gerados e com o comando *time* para obter os tempos de execução de cada teste. Deste último, consideramos o tempo *real*.

Sobre os inputs gerados, é de notar que o argumento *#SubR* do gerador foi mantido com o valor constante de 100.

Após análise do gráfico abaixo (Tempo de execução em função de V+E), pudemos constatar que o algoritmo desenvolvido é, de facto, linear com V+E.




| V+E | Tempo (s) |
|---|---|
| 300000 | 0,201 |
| 600000 | 0,44 |
| 900000 | 0,651 |
| 1200000 | 0,887 |
| 1500000 | 1,13 |
| 1800000 | 1,395 |
| 2100000 | 1,736 |
| 2400000 | 2,056 |
| 2700000 | 2,343 |
| 3000000 | 2,649 |

**Referências:**


- Slides da cadeira, disponíveis no Fénix.
- https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/ (Artigo sobre pontos de articulação em grafos e como os encontrar).