**Análise e Síntese de Algoritmos**

1º Projeto 2015/2016 | Grupo 022 | 78974 | 82448

# Descrição do Projeto

O problema abordado neste projeto corresponde a encontrar as chamadas “pessoas fundamentais” numa rede. Suponhamos a existência de 3 pessoas, A, B e C, tal que A conhece B e B conhece C. A transmissão de informação entre A e C faz-se através de B. Se esta for removida, deixa de haver essa passagem de informação. Generalizando, quando uma pessoa fundamental é removida, a transmissão de informação deixa de existir entre duas áreas de uma rede. Representando esta situação através de um grafo conexo não dirigido, em que pessoas são representadas por vértices e ligações entre elas por arestas, o problema resume-se a encontrar os pontos de articulação do grafo. A remoção de um ponto de articulação do grafo causa a separação do grafo inicial em dois grafos conexos separados.

# Implementação do Programa

O programa foi implementado na linguagem C, apenas com a utilização das bibliotecas <stdio.h> e <stdlib.h>.

## Implementação da estrutura de dados

O problema acima descrito foi implementado com a utilização de grafos não dirigidos, representados por listas de adjacências.

A implementação da lista de adjacências foi feita com base num vetor primário para representar cada um dos vértices, e uma lista simplesmente ligada para representar os arcos.

São definidas as operações de inicialização do grafo, criação de arcos, e inserção dos arcos.

## Algoritmo de procura de pontos de articulação

A ideia base para o algoritmo é a utilização de uma procura em profundidade (DFS). Numa árvore DFS, um vértice **u** é pai de **v** caso **v** seja descoberto por **u**. Um vértice **u** é um ponto de articulação se:

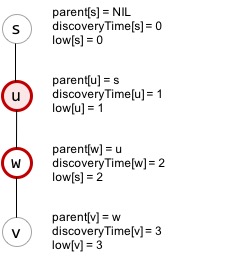
1. É a raiz da árvore DFS e tem pelo menos 2 filhos;
2. Não é a raiz, mas não existe nenhum vértice na subárvore em que **u** é raiz que tenha um arco de retorno para um dos antecessores do vértice **u**.

Para se verificar os pontos de articulação do grafo faz-se então uma passagem em profundidade do grafo com 3 vetores auxiliares:

* Um vetor **parent[]** onde **parent[u]** guarda o predecessor de u;
* Um vetor **discoveryTime[]** para guardar o tempo de descoberta de cada vértice;
* Um vetor **low[]** em que **low[u]** representa o menor tempo de descoberta de qualquer vértice que está ou na subárvore de raiz em **u** ou ligado a um vértice nessa subárvore por um arco de retorno.

Cada vértice tem também um contador dos seus filhos. Caso o vértice seja a raiz da árvore DFS e o contador tenha 2 ou mais filhos, estamos perante o primeiro caso de pontos de articulação.

Para o segundo caso, por cada nó **u** da árvore, precisamos de averiguar se não existe nenhum descendente de **u** que encontre um vértice **v** com um tempo de descoberta inferior a **u**. Na Figura 1 existe um arco de retorno de **w** (descendente de **u**) até um antecessor de **u**, portanto não existe nenhum descendente de **u** com valor de **low** igual ou superior ao tempo de descoberta de **u**. Neste exemplo, **u** não é um ponto de articulação. Na Figura 2, **u** é um ponto de articulação uma vez que, se for retirado, perde-se a ligação entre **s** e **w/v**. O **low** de **v** e de **w** é superior ao tempo de descoberta de **u**. Neste caso, **w** também é um ponto de articulação.

Fazer esta computação para todos os vértices é fácil: segue-se pela lista de adjacências, mantendo os valores mínimos que podem ser atingidos a partir de cada vértice no vetor **low[]**. Para arcos da árvore, fazemos a computação recursivamente; para arcos de retorno, usamos o tempo de descoberta do vértice adjacente no vetor **discoveryTime[]**.

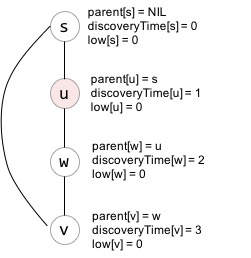


Figura 2– u é um ponto de articulação (w também).

Figura – u não é um ponto de articulação.

À medida que os pontos de articulação vão sendo encontrados, a variável **apcount** é incrementada, armazenando o número total de pontos de articulação do grafo. **min** e **max** são definidos estaticamente e atualizados em tempo de execução com os pontos de articulação mínimo e máximo.

## Função main

A função main lê do stdin o número de vértices com que é necessário inicializar o grafo e o número de arestas, que corresponde ao número restante de linhas de input que serão lidas. Cada uma destas linhas é então transformada numa aresta e adicionada ao grafo. Terminado este passo, corre o algoritmo de busca de pontos de articulação, terminando por enviar para o stdout o número de pontos de articulação, o ponto mínimo e o máximo.

Note-se que os valores enviados para a estrutura de dados para os valores das arestas são **u-1** e **v-1**. Isto deve-se ao facto de o programa esperar como input **N** vértices de valores de **1** a **N**, e a estrutura de dados interna funcionar com **N** vértices de valores **0** a **N-1**.

# Análise Teórica

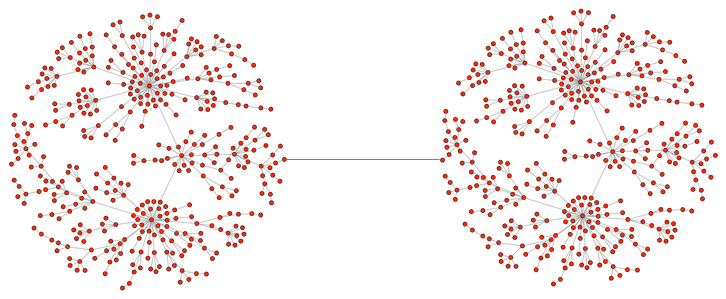
Foi escolhida como estrutura de dados para a implementação deste problema um grafo representado como lista de adjacências. Tomando **V** como o número de vértices e **E** como o número de arcos, numa estrutura deste tipo, a complexidade de inicializar um grafo vazio é de **O(|V|**), sendo que a posterior inserção de todas as arestas tem um custo de **O(|E|)**. Assim sendo, a complexidade total da construção de um grafo dado como input é de **O(|V|+|E|)**.

Este programa é uma modificação a uma pesquisa em profundidade (DFS – *Depth First Search*) que envolve adicionar alguns testes e operações de tempo constante, pelo que tem uma complexidade **O(|V|+|E|)** para listas de adjacência. A complexidade descrita, associada a uma DFS, deriva de serem percorridos todos os vértices uma só vez (garantido pelo ciclo for na função **GRAPHsearch()** e pelas verificações da ausência de tempo de descoberta), e todas as arestas uma só vez (garantido pelo ciclo for na função recursiva **searchArticulationPoints()**).

A correção do algoritmo é consequência de se usar uma DFS para descobrir propriedades do grafo. A representação do grafo afeta a ordem da procura, mas não afeta os resultados, uma vez que os pontos de articulação são uma característica do grafo, e não da forma como o representamos ou procuramos. Como o output representa os valores mínimos e máximos dos pontos de articulação, e não envolve imprimir todos estes, vai ser sempre igual, independentemente do vértice inicial. (Caso fossem impressos todos os pontos de articulação, a ordem seria diferente consoante o vértice inicial). Como é óbvio, qualquer árvore DFS é simplesmente outra representação do grafo, pelo que todas têm as mesmas propriedades.

# Avaliação Experimental dos Resultados

Para proceder à avaliação da eficiência do programa, foram criadas duas estruturas diferentes de casos de teste de input com tamanhos definidos.

Na construção do Gráfico 1, a ideia foi duplicar um grafo fornecido como input e juntar os dois grafos densos obtidos por um arco, como se mostra na figura 1. O primeiro caso corresponde ao input do 5º teste disponibilizado aos alunos, com 30000 vértices e 58783 arestas, somando um total de 88783 vértices+arestas (V+E). O segundo, terceiro e quarto caso contêm, respetivamente, **2x(V+E+1)-1**, **3x(V+E+1)-1** e **4x(V+E+1)-1** relativamente ao primeiro caso. Para aumentar ainda mais a dimensão do input, criámos ainda casos com **6x(V+E+1)-1** e **8x(V+E)-1**, somando este último um total de 710271 vértices e arestas. O Gráfico 1 relaciona os casos de teste descritos com o tempo de execução do programa para encontrar pontos de articulação no grafo, medido no terminal com o programa *time*. Como se pode constatar pela observação do gráfico, a linha de tendência é linear, o que nos confirma a complexidade linear O(|V|+|E|) do algoritmo criado.

Para a construção do Gráfico 2, fixou-se o número de vértices e variou-se apenas o número de arestas em potências de 2, obtendo grafos progressivamente mais densos. A soma obtida foi de, incrementalmente, **V+2\*E, V+4\*E, V+8\*E**, **V+16\*E**, **V+32\*E** e **V+64\*E**, contabilizando este último um total de 3762112 vértices+arestas! Mais uma vez, a linha de tendência é linear, o que nos confirma a complexidade linear O(|V|+|E|) do algoritmo criado.

Figura

Gráfico - Estrutura de teste 1

Gráfico - Estrutura de teste 2

# Referências bibliográficas

SEDGEWICK, Robert; **Algorithms in C;** 1997; Addison-Wesley Publishing Company