1º Projeto de ASA - Relatório

Introdução

Os objetos do problema (*routers* e ligações) podem ser representados numa linguagem de programação como nós e arcos, respetivamente. Podemos modelar o problema usando o conceito de grafos/subgrafos não dirigidos e, assim, utilizar algoritmos de pesquisa em grafos para determinar o **número de subredes** e o **maior identificador de cada subrede**. Os routers que, se removidos, aumentam o número de subredes correspondem ao conceito de **pontos de articulação** de um grafo. O último output é obtido determinando o **tamanho do maior subgrafo (sem os pontos de articulação)**. Decidimos utilizar essa representação na nossa solução e, assim, utilizar algoritmos de pesquisa em grafos para determinar os *outputs* requeridos.

Escolhemos implementar a nossa solução na linguagem C++.

Descrição da solução

Excetuando os inputs/outputs, a nossa solução divide-se em 2 partes :

- Realização de uma 1ª procura em profundidade (DFS) sobre o grafo, para determinar os três primeiros *outputs*
- Realização de uma 2ª DFS para determinar o último *output*

A função **getGraphData** é a função principal do programa que dado um grafo, calcula os dados descritos no problema.

Após várias inicializações, é realizada a 1ª DFS, que percorre o grafo, calculando para cada nó **u** o seu tempo de descoberta, **d[u]**, e o valor do *low*, **low[u]**, definido como o menor valor entre:

- o tempo de descoberta de **u** na pesquisa DFS.
- o tempo de descoberta dos nós **w** tais que existe um descendente de **u** que tem um arco para trás para **w** (na árvore DFS).

Esta procura é executada por chamadas sucessivas da função **getAP** (*get articulation points*) que, dado um vértice **u**, atualiza os seu valores de **d** e **low** e é chamada recursivamente sobre os vizinhos de **u** que ainda não foram visitados (**d[u] == NIL**).

O número de subgrafos (**numGroups**) é calculado através da contagem do número de vezes que a função **getAP** tem de re-invocada num novo vértice, por todas as suas chamadas recursivas terem terminado (o que significa que o subgrafo atual não tem mais nós por explorar).

O maior identificador de cada subrede (vetor **groupIDs**) é calculado simplesmente guardando o ID do primeiro nó visitado em cada *restart* da função **getAP**. Este método funciona, pois os vértices são percorridos do maior identificador para o menor, logo, da cada vez que a função **getAP** recomeça, o próximo nó por visitar será o de maior ID de todos os que ainda não foram visitados, o que implica que seja o de maior ID do seu subgrafo (independentemente do subgrafo a que pertença).

Os pontos de articulação (array de booleanos **isAP**) são calculados pela função **getAP** da seguinte forma:

- Se **u** é a raiz do seu subgrafo (**parent == NIL**) e tem mais de um filho na árvore DFS (**children > 1**), **u** é um ponto de articulação. **(1)**
- Se **u** não é a raiz do seu subgrafo e tem um filho **v** tal que **low[v] >= d[u]**, v é um ponto de articulação. **(2)**
- Caso contrário, **u** não é um ponto de articulação.

A propriedade **(1)** verifica-se devido ao facto de a pesquisa ser em profundidade primeiro. Se **u** é a raiz da árvore DFS e tem mais de um filho (**v1**, **v2**, …, **vn**) significa que não for possível encontrar, durante a DFS, um caminho de **v1** para **v2** (por exemplo) que não passe por **u**, o que implica que a remoção de **u** tornaria o grafo desconexo.

A propriedade **(2)** advém da definição do valor *low*. Se **u** não é a raiz, e tem um filho **v** tal que **low[v] >= d[u]**, então todos os arcos para trás que partem de **v** e dos seus descendentes têm como destino nós **w** com **d[w] >= d[u]**, ou seja, vértices descobertos depois de **u** (ou o próprio **u**). Isto significa que é impossível encontrar um caminho de **v** para os ascendentes de **u** sem passar por **u**, pelo que **u** é ponto de articulação.

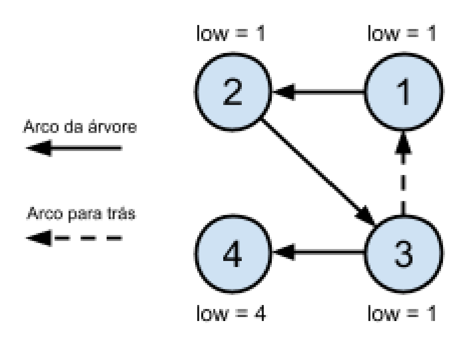


Fig.1 - *Low* (exemplo)

No exemplo da Fig.1, o nó 3 é um ponto de articulação, pois tem um filho - o nó 4 - tal que **low[4] >= d[3]** (4 >= 3).

O tamanho do maior subgrafo sem os pontos de articulação (**maxGroupSize**) é calculado na função **findMaxGroup** fazendo uma nova DFS em todo o grafo, mas não visitando os pontos de articulação (pontos **u** com **isAP[u] == true**), e contando o número de vértices visitados entre cada *restart* da função (**currentGroupSize**).

Análise teórica

Vamos determinar a complexidade da nossa solução em função do número de nós (N) e arcos (M) do grafo, usando a notação assintótica.

A leitura do *input* tem complexidade O(M), pois consiste na leitura de dois valores singulares e M arestas. A apresentação do *output* tem complexidade O(N), pois consiste em três valores singulares e, no máximo, N identificadores de vértices (no caso extremo de não haver arestas).

```
for (int v: adj[u])
{
    if (d[v] == NIL)
    {
        getAP(adj, v, u, data);
        children ++;
        if (low[v] >= d[u] && parent != NIL)
            isAP[u] = true;
        if (low[v] < low[u])
            low[u] = low[v];
    }
    else if (v != parent && d[v] < low[u])
        low[u] = d[v];
}
```

Fig.2 - Ciclo *for* da função **getAP**

A função **getGraphData** tem declarações e inicializações de complexidade total O(N) e chamadas sucessivas às funções **getAP** e **findMaxGroup**.

A função **getAP** tem, antes e depois do ciclo for (na Fig.2), várias instruções cuja complexidade não depende do tamanho de N ou M (O(1)). Uma vez que a função é chamada, no total, N vezes, estas instruções contribuem com O(N) para a complexidade da solução. No ciclo *for*, para cada nó, serão percorridas todas as suas adjacências, logo, cada adjacência de cada nó será percorrida uma única vez. Visto que cada arco liga dois nós, o bloco de código dentro do ciclo for será executado 2*M vezes, o que implica uma complexidade O(M). Logo, a função **getAP** tem complexidade O(N + M).

A função **findMaxGroup**, pelo mesmo raciocínio, no máximo (caso não haja pontos de articulação), também percorre uma única vez cada vértice e duas vezes cada aresta, pelo que tem complexidade O(N + M).

Logo, a função **getGraphData** tem complexidade O(N + M).

A função **main** executa as operações de *input*/*output* e chama a função **getGraphData**, pelo que a sua complexidade é O(N + M).

As estruturas de dados que utilizámos são todas, no máximo, lineares no tamanho dos dados que guardam. Nomeadamente a complexidade espacial da lista de adjacências **adj** é O(M) e a dos *arrays* **d**, **low** e **isAP** é O(N) tal como a do vetor **groupIDs**.

Avaliação experimental

Apresentamos aqui os resultados da avaliação da duração da execução e da memória utilizada pelo nosso programa com inputs de 10000*x nós e 20000*x arcos (com 1 <= x <= 50). Para calcular o tempo de execução realizámos 50 testes para cada input e calculámos a sua média. Para calcular a memória, realizámos apenas um teste para cada um dos 50 inputs.

Fig. 3 - Tempo de execução em função do número de nós e arcos

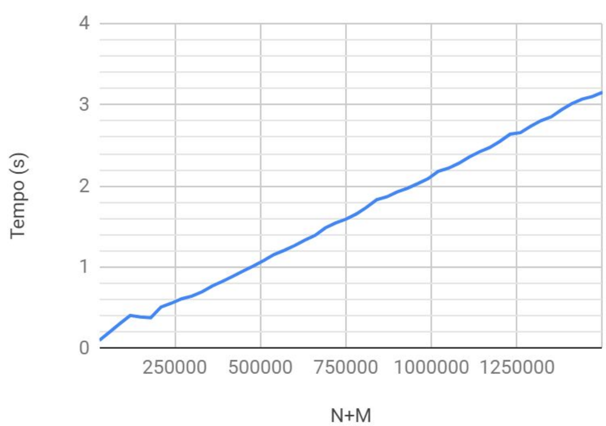
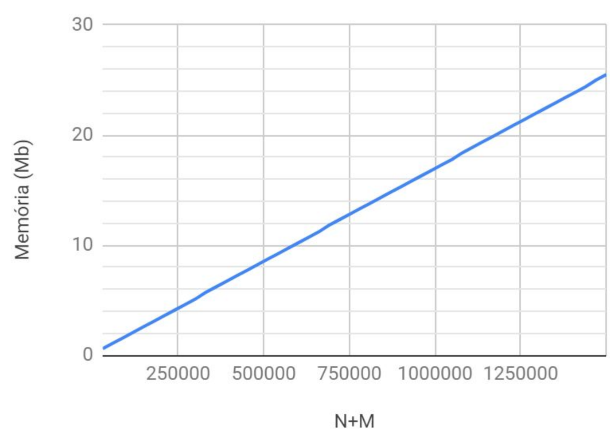


Fig. 4 - Memória em função do número de nós e arcos



Os resultados experimentais apoiam a nossa análise teórica, porque ambos os gráficos mostram uma variação linear no número de nós e arcos.

Referências:

https://en.wikipedia.org/wiki/Biconnected_component
https://walkccc.github.io/CLRS/Chap22/Problems/22-2/