Análise e Síntese de Algoritmos - 1º Projeto

Grupo 16
Afonso Gonçalves - 89399
Daniel Seara - 89427

1 Introdução

O primeiro projeto da cadeira de ASA teve como objetivo executar uma auditoria a uma rede. Esta rede poderá estar dividida em sub-redes, sendo que temos de realizar a auditoria a todas elas. Para tal é necessário calcular quais routers da rede, ao serem desligados, aumentariam o número de sub-redes.

O Input é dado com M+2 linhas. A primeira indica o número de routers da rede, a segunda o número (M) de ligações entre eles e as restantes representam as ligações (bidirecionais) entre estes, indicando os IDs dos routers em causa.

O Output é constituído por 4 linhas com, respetivamente, o número de sub-redes da rede fornecida, uma sequência ordenada do maior ID de cada sub-rede, o número de routers que quebram a rede e o número da maior sub-rede formada pela remoção desses routers.

2 Descrição da Solução

Decidimos usar a linguagem *C++* para a resolução deste problema, por ser uma linguagem eficiente, por ter uma boa biblioteca de estruturas de dados que consideramos necessárias e ainda por constituir um desafio começar a trabalhar com uma nova linguagem, tão usada e potente nos dias de hoje.

Identificámos que este problema pode ser resolvido como um problema de grafos: Cada router é representado por um vértice e as ligações entre estes pelas respetivas arestas. Os routers que podem quebrar a rede são os vértices de corte (AP) do grafo e as sub-redes os seus subgrafos.

Começámos por desenhar um algoritmo que identificava as pontes do grafo, uma vez que, removendo pelo menos um dos vértices onde esse arco incide, criaríamos novos subgrafos. Teríamos especial atenção aos vértices que apenas tivessem uma aresta, pois a sua remoção não aumentaria o número de subgrafos. Esta abordagem não contempla os vértices de articulação que não formam pontes, por isso mudámos de estratégia.

Em aula teórica e com alguma pesquisa na internet[1, 2] , aprendemos que o algoritmo de Tarjan permite encontrar os AP's dos grafos. Como esse algoritmo executa uma DFS, é possível calcular simultaneamente o número de subgrafos. Para além disso, ao visitar um subgrafo de cada vez, conseguimos saber o maior ID que lhe pertence.

Numa nova iteração do algoritmo, passámos a percorrer os vértices por ordem decrescente de ID, uma vez que estes serão os maiores do respetivo subgrafo. Deste modo, obtemos ainda os ID's máximos de cada subgrafo por ordem decrescente, não sendo necessário ordená-los.

Para calcular o número de vértices do maior subgrafo resultante da remoção de todos os vértices críticos encontrados, começámos por remover todos os AP's e as suas arestas e executar uma DFS. Rapidamente nos apercebemos que seria uma solução demasiado cara para o efeito: O uso de uma flag que indicasse em tempo constante se um vértice é um AP permite ignorá-lo durante a DFS, como se tivesse sido removido.

Com base na teoria de Grafos, sabemos também que é impossível para um grafo completo ter AP's. Deste modo, qualquer grafo com V vértices e $\frac{V(V-1)}{2}$ arestas terá necessariamente um subgrafo com ID igual a V e não terá pontos de articulação (pela natureza do problema, supomos que não há mais do que uma aresta entre dois vértices nem arestas de um vértice para si próprio).

3 Estruturas de dados empregues

Como neste algoritmo apenas será preciso inserir arestas e iterar por todas as adjacências dos vértices (sem procura), optámos por poupar em memória e representar o grafo usando um array de listas simplesmente ligadas (**std::forward_list<int>***). Para minimizar a complexidade, a inserção de novas arestas será sempre feita no início e a iteração do início para o fim.

Visto que cada vértice tem um ID, é possível aceder à informação necessária para a execução do algoritmo em tempo constante se ela for guardada em array. Preferimos guardar esses arrays na stack porque a alocação e o acesso a essa memória são mais rápidos. Contudo, para inputs muito grandes, obtivemos $SegmentationFault$, pelo que um desses arrays teve de ser guardado na heap.

4 Algoritmo & Análise Teórica

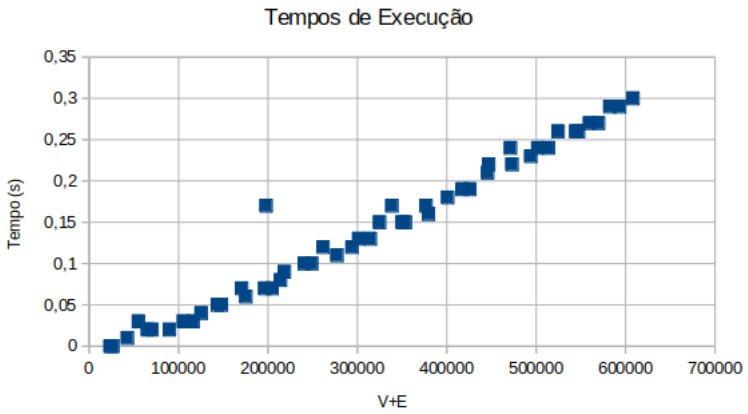
O nosso algoritmo tem complexidade linear, com pior caso $O(V+E)$ e melhor caso $\Omega(1)$:

1. **Main():**
2. Ler V, E $O(1)$
3. **If** E = $\frac{V(V-1)}{2}$: **then** Output() $O(1)$

```
4.  LerGrafo()                                                    O(E)
5.  For u ∈ V:                                                    O(V + E)
6.    If !visited(u):
7.      Incrementa número de subgrafos e guarda u
8.      Explora(u)
9.  maxSubgraphSize()                                             O(V + E)
10. Output()                                                      O(V)

11. Explora(u) (Função auxiliar para o Algoritmo de Tarjan)
12.  Inicialização das estruturas de informação                   O(V)
13.  For v ∈ Adj[u]:                                              O(E)
14.    If !visited(v):
15.      filhos(u)++ and pai(v) = u                               O(1)
16.      Explora(v)
17.      atualiza low(u)                                          O(1)
18.      If isAP(u) [1]:
19.        isAP(u)                                                O(1)
20.    Else If pai(u) != v:
21.      atualiza low(u)                                          O(1)

22. maxSubgraphSize(V)
23.  For v ∈ V:                                                   O(V + E)
24.    greater = 0
25.    If !visited(v) and !isAP(v):
26.      If subgraphSize(v) > greater:
27.        atualiza greater                                       O(1)
28.  return greater

29. subgraphSize(u):
30.  visited(u); size = 0                                         O(1)
31.  For v ∈ Adj[u]:                                              O(E)
32.    If !visited(v) and !isAP(v):
33.      size += subgraphSize(v)                                  O(1)
34.  return size
```

5 Demonstração de Resultados

O nosso algoritmo passa em todos os 16 testes propostos e nos testes disponibilizados. Apresentamos em baixo o gráfico de distribuição de 50 testes com número de vértices mais arestas entre os 10000 e os 600000 e o seu tempo de execução. Daqui se prova a linearidade da nossa solução.



É de notar que apesar de 1 ponto ter tido um tempo de execução fora no normal, todos os outros testes seguem uma distribuição linear. Apresentamos aqui alguns resultados concretos:

#Vértices	#Arestas	#Subgrafos	Tempo(s)
72111	71403	1060	0.04
176712	173663	4621	0.08
301250	297963	4899	0.17
437923	433496	6552	0.27
547066	544644	3619	0.35
603836	601467	3504	0.39

Referências

[1] Articulation Points (or Cut Vertices) in a Graph
`https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/`

[2] Articulation Points and Bridges
`https://www.hackerearth.com/practice/algorithms/graphs/articulation-points-and-bridges/tutorial/`