

# Documentação do Trabalho Prático 02 da disciplina de Algoritmos I

Victor Hugo Lima do Nascimento<sup>1</sup>

<sup>1</sup>Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte – MG – Brasil

vhln10@ufmg.br

## 1. Introdução

Esta documentação detalha e explica um algoritmo proposto para resolver o problema de gerenciamento de voos durante a pandemia. Uma companhia aérea realizava voos de ida e volta regularmente para diversos países do mundo, porém devido à pandemia teve que cancelar algumas rotas e percebeu que um passageiro partindo de um aeroporto poderia não chegar a outro. Por isso é necessário saber quantas rotas devem ser adicionadas para que exista um caminho entre todos os aeroportos. Neste trabalho é proposto uma solução para esse problema usando o algoritmo de Tarjan para formar as componentes fortemente conectadas (CFC) e com elas criar um grafo direcionado acíclico (GDA). Esta documentação é composta por um seção de implementação onde será explicada como o algoritmo funciona e foi pensado para resolver o problema. Em seguida há uma seção de instrução de compilação e execução para que o leitor possa testá-lo localmente. Depois é apresentada uma análise de complexidade abordando o pior caso do algoritmo. Por fim há uma conclusão do autor.

## 2. Implementação

O código está contido em um único arquivo. Sua primeira função é chamada de Matriz, que recebe um string contendo o nome do arquivo .txt e cria uma matriz de adjacência a partir dele. Com essa matriz usamos a função Componente e CFC, baseadas no algoritmo de Tarjan, para encontrar as componentes fortemente conectadas.

O algoritmo de Tarjan utiliza a propriedade de que os vértices de CFC formam uma subárvore na árvore DFS do grafo. No algoritmo um DFS é executado nos vértices e as subárvores que formam uma CFC são removidas e adicionadas a um vetor de vetores chamado adj. Mais detalhadamente, dois valores count(a) e low\_link(a) são mantidos para cada um dos vértices. count(a) é o valor do contador quando o nó 'a' é explorado pela primeira vez. low\_link(a) armazena o count(a) mais baixo acessível de 'a' que não faz parte de outra CFC. Conforme os vértices são explorados, eles são colocados em uma pilha. Os filhos não explorados de um vértice são explorados e low\_link(x) é atualizado de acordo. Um nó é encontrado quando low\_link(u) == count(u) é o primeiro nó explorado em seu componente fortemente conectado e todos os nós acima dele na pilha

são retirados e atribuídos a uma mesma CFC.

Dentro da Main temos dois conjuntos de for's que são diferenciados com um comentário acima do código como Conjunto 1 e Conjunto 2. No Conjunto 1 os dois primeiros for's tem a função de andar por todo o vetor de vetores chamado de adj. Cada elemento de adj é um CFC e dentro dela temos os vértices do grafo original que formam essa CFC. Ou seja, adj[1] tem um vetor que contém os vértices do grafo original que formam o vértice 1 na CFC. Essa parte do código trata de verificar quais vértices do grafo original tinham uma aresta para outro vértice. Então procuramos os 1s da matriz de adjacência e quando achamos significa que há uma ligação entre o vértice n que está na componente i com o vértice y. Salvamos y no vetor Proc.

Então no Conjunto 2 procuramos em qual CFC o vértice y está. Se essa CFC é diferente da que o vértice n está, então as duas CFCs devem manter essa aresta no GDA. Isso é feito no vetor de vetores adjNEW[i]. Cada elemento dele é um vetor para onde o vértice i tem uma aresta apontando. Em seguida é computado o grau de saída e entrada de cada vértice do GDA.

O restante do código trata de usar os graus de saída e entrada para solucionar o problema final. Construindo exemplos mais simples a mão e analisado-os, nota-se que para o grafo original ser fortemente conectado tem-se que adicionar ao GDA um total de  $\max(|X|, |Y|)$  arestas, onde X é o conjunto de vértices com grau de entrada igual a zero, e Y é o conjunto de vértices com grau de saída igual a zero. Dessa forma chegamos ao resultado correto.

O programa foi testado localmente e retornou a resposta correta para todos os exemplos fornecidos em menos de 1 segundos. Configuração da máquina onde foi testado:

- Sistema Operacional do seu computador: Linux Ubuntu 20.04.2 LTS;
- Linguagem de programação implementada: C++;
- Compilador utilizado: gcc version 9.3.0;
- Dados do seu processador: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz;
- Quantidade de memória RAM: 8 GB;

### **3. Instruções de compilação e execução**

Utilizando a pasta Makefile que foi enviada basta acessar seu diretório pelo VS Code, por exemplo, e digitar no terminal o comando make. Surgirá então um executável na pasta bin. Com o diretório na pasta bin (digitando cd bin no terminal) e com o arquivo .txt a ser testado também na pasta bin pode-se digitar ./tp02 nome\_do\_arquivo.txt e o programa irá retornar o resultado correto.

## 4. Análise de complexidade

Considere  $V$  o número de vértices do grafo e  $E$  o número de arestas.

Função Matriz - Complexidade temporal:  $O(V^2)$ , devido aos dois for's aninhados.

Função Matriz - Complexidade espacial:  $O(V^2)$ , devido a alocação dinâmica de uma matriz  $n \times n$ .

Vamos analisar a função Componente e CFC como a mesma, pois elas são usadas em conjunto.

Função Componente e CFC - Complexidade temporal:  $O(V+E)$ , pois a função depende de um DFS e visita cada vértice apenas uma vez, fazendo um trabalho constante em cada um deles.

Função Componente e CFC - Complexidade espacial:  $O(V)$ , pois esse é o espaço máximo que a pilha pode precisar alocar.

Conjunto 1 - Complexidade temporal:  $O(V^3)$ , devido aos três for's que rodam todos sobre o número de vértices.

Conjunto 1 - Complexidade espacial:  $O(V.E)$ , pois no pior caso o vetor Proc terá  $V$  elementos e todas as CFC terão somente um vértice.

Conjunto 2 - Complexidade temporal:  $O(E.V^3)$ , devido aos quatro for, porém apenas três iteram sobre  $V$ , já que o pior caso de Proc está ligado ao número de arestas  $E$ .

Conjunto 2 - Complexidade espacial:  $O(V+E)$ , pois adjNEW tem  $V$  vetores e cada um tem  $E$  elementos.

Restante do código - Complexidade temporal:  $O(V)$ , devido aos for's, nenhum aninhado, que iteram sobre  $V$ .

Restante do código - Complexidade espacial:  $O(1)$ , pois não criam nenhum espaço, apenas alteram o que já existe.

Complexidade temporal total:  $O(E.V^3)$

Complexidade espacial total:  $O(V^2)$


## 6. Conclusão

Este trabalho lidou com o problema de encontrar quantas conexões uma rede aérea precisa adicionar para que um passageiro saindo de qualquer aeroporto consiga chegar em qualquer outro. Com o método implementado chegamos sempre nas respostas corretas e em tempo razoável.

Fico muito feliz com o resultado desse trabalho prático, pois achei poucas

referências na internet de como resolver e cheguei a pensar em desistir do trabalho quando os problemas aumentaram. Tive muitos problemas de segmentation fault ao tentar utilizar vetor de vetores, mas no final deu tudo certo. Acho que eu deveria ter feito uma lista de adjacência em vez da matriz, mas como deixei para fazer o código perto do prazo não foi possível fazer uma solução mais complexa. Mesmo assim acho que o resultado obtido foi satisfatório uma vez que conseguimos as respostas corretas rapidamente.

## References

 [Tarjan's Strongly Connected Component \(SCC\) Algorithm \(UPDATED\) | G...](#)