A B A K Ó S
PUC Minas Instituto de Ciências Exatas e Informática



# Identificação de caminho eureliano, utilizando Fleury e identificação de pontes*

Model - Magazine Abakós - ICEI - PUC Minas

Breno Lopes Do Carmo[1]
Bruno Rodrigues Faria[2]
Lucas De Paula Vieira Santos[3]

**Resumo**

O artigo trata da identificação de caminho eureliano utilizando a identificação de pontes usando dois algoritmos, o algoritmo Naive e o (TARJAN, 1974), para que possa ser analisado a diferença entre esses algoritmos e o seu tempo de execução, para N quantidade de vértices. Diante disso, foram utilizados os algoritmos de identificação de pontes para identificar um caminho euleriano em um grafo qualquer, onde esses grafos quaisquer são criados de forma aleatória sendo eles, eulerianos, semi-euleriano e não euleriano, levando em consideração um grafo simples não-direcionado G=(V, E), em que V representa o conjunto de vértices e E o conjunto de arestas.

**Palavras-chave:** Grafos. Pontes. Busca. Euleriano. Complexidade. Naive. Tarjan.

*Identificação de caminho eureliano, utilizando Fleury e identificação de pontes

[1]Programador, E-mail:blcarmo@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.
[2]Programador, E-mail:bruno.faria@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.
[3]Programador, E-mail:lpvsantos@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.

## 1 INTRODUÇÃO

Atualmente a aplicação de grafos é cada vez mais importante para o mundo dos algoritmos e estrutura de dados, por ser ótimo para a resolução de problemas. Um dos problemas dos grafos é o custo que tem-se para aplicar alguns algoritmos em grafos muito grandes em quantidade de vértices (V) e arestas (E). Por isso o trabalho tem como intuito, abordar o tempo de execução da identificação de um caminho euleriano em um grafo aleatório de 100, 1000, 10000 e 100000 vértices, juntamente com a estratégia de identificação de pontes feitas pelo algoritmo Naive e (TARJAN, 1974), que serão utilizados no algoritmo Fleury para a escolha das arestas corretas.

## 2 DESENVOLVIMENTO

O projeto foi desenvolvido em Python (CPython), a qual é uma linguagem interpretada, devido isso o tempo de execução de alguns algoritmos pode demorar mais que as outras linguagens, como mostrado na figura 1, o objetivo de utilizar Python foi ter a possibilidade de ter uma comparação com os demais trabalhos em tempo de execução dos algoritmos.

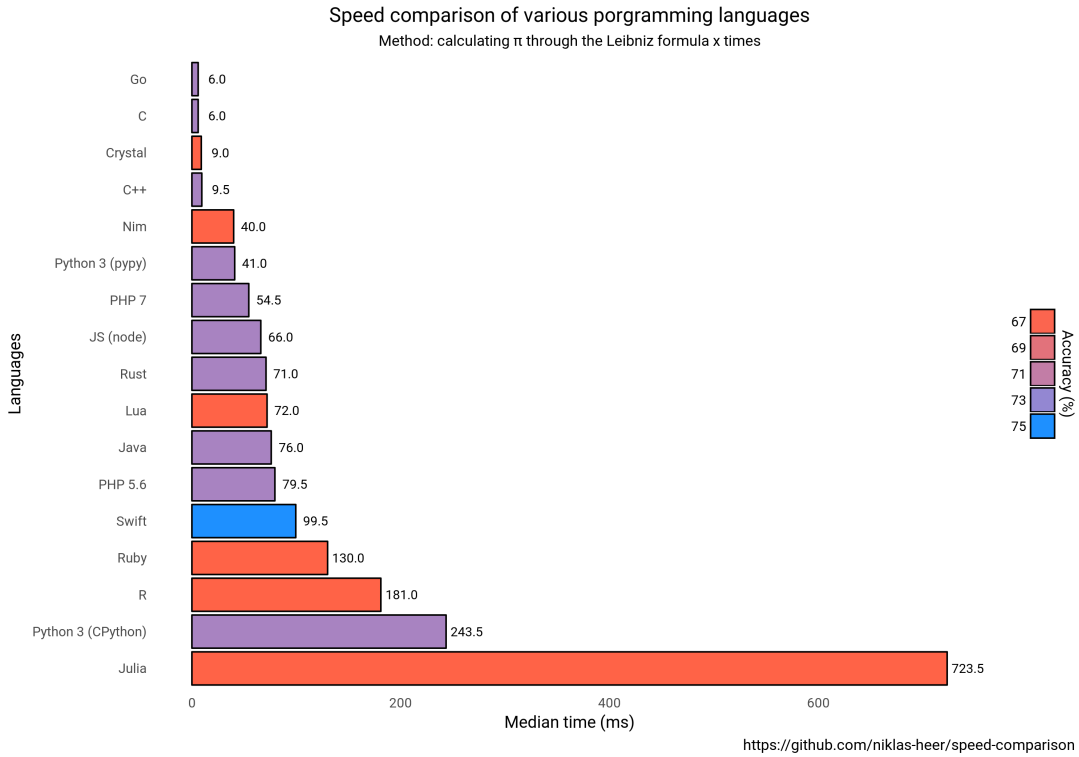


**Figura 1 – Comparação de velocidade entre as linguagens**

## 2.1 Criação dos grafos aleatórios

### *2.1.1 Grafo euleriano*

Um grafo euleriano é um grafo onde todos os seus vértices possuem grau par. Com isso, basta aplicar um algoritmo que ligue o vértice (i) no vértice (i + 1). Com isso o grafo será cíclico e todos seus vértices serão de grau 2, como mostrado na figura 2.

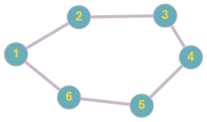


**Figura 2 – Grafo euleriano**

### *2.1.2 Grafo semi-euleriano*

Um grafo semi-euleriano é um grafo onde onde tem apenas dois vértices de grau ímpar. Com isso, para criar um grafo aleatório semi-euleriano bastava criar um grafo euleriano e sortear uma aresta para incidir em dois vértices, transformando o grau de 2 vértices do grafo para 3, como mostrado na figura 3.

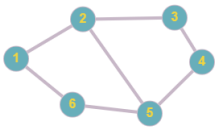


**Figura 3 – Grafo semi-euleriano**

### *2.1.3 Grafo não euleriano*

Um grafo não euleriano é um grafo onde onde tem mais de dois vértices de grau ímpar. Com isso, para criar um grafo aleatório não euleriano foi necessário criar apenas um grafo eureliano e sortear duas arestas no grafo, sem repeti-lás, para que assim o grafo tenho 4 vértices de grau ímpar, tornando assim o grafo em não eureliano, como exemplificado na figura 4.

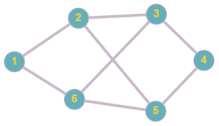


**Figura 4 – Grafo não euleriano**

## 2.2 Caminho euleriano - Fleury

Para implementação do algoritmo Fleury no qual consiste em buscar um caminho euleriano seja ele fechado ou aberto em um grafo, caso o grafo não seja eureliano ou semi-eureliano ele não vai conter um caminho euleriano, foi utilizado o material disponibilizado pelo Prof. Zenilton no Canvas, ferramenta de organização de materias da faculdade, mostrando na 5, onde pode-se usar como referência para implementar na linguagem Python:

1. se V(G) possuir 3 ou mais vértices de grau ímpar então **PARE**;
2. Seja G' = (V', E') tal que V' ← V(G)e E' ← E(G); // Inicializar grafo auxiliar
3. Selecionar vértice inicial $v \in V'$ (escolher v cujo grau seja ímpar, se houver)
4. enquanto $E' \neq \emptyset$ efetuar
   a. se $d(v) > 1$ então
      Selecionar aresta $\{v, w\}$ que não seja ponte em G'
   b. senão
      Selecionar a única aresta $\{v, w\}$ disponível em G'
   c. $v \leftarrow w$; $E' \leftarrow E' - \{v, w\}$; // Caminhar de v para w e eliminar aresta

**Figura 5 – Algoritmo Fleury - Prof. Zenilton**

### *2.2.1 Identificação pontes - Naive*

O algoritmo Naive faz uma busca muito custosa pelas pontes, já que ele percorre todos os vértices do grafo fazendo às seguintes operações: remove uma aresta da lista de adjacência do grafo, após isso faz uma busca em largura. Visto que, para cada vértice existente no grafo, ele deve ser visitado uma única vez e para cada aresta existente, ela também deve ser visitada uma única vez, para conferir se o grafo é conexo, onde se todos vértices forem visitados o grafo é conexo, caso contrário não é conexo, como mostrado na figura 6 de forma visual. Com isso, o algoritmo tem o custo de complexidade **O(E * (V + E))**.

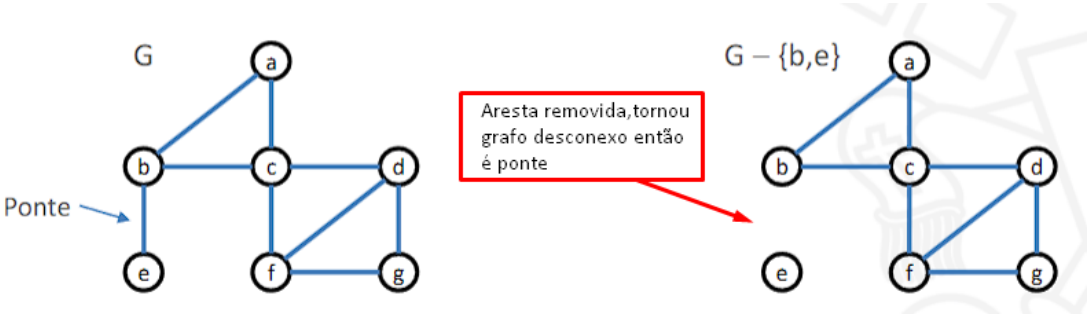


**Figura 6 – Identificação pontes (Naive) - Prof. Zenilton**

### *2.2.2 Identificação pontes - Tarjan*

Para implementação do algoritmo Tarjan foi utilizado da ideia do seguinte algoritmo retirado do artigo (TARJAN, 1974), a ideia da implementação pode ser mostrada na figura 7, onde o algoritmo consiste em achar uma árvore de abrângencia (WIKIPéDIA, 2022) utilizando o algoritmo de busca em profundidade, onde pode ser encontrado em tempo linear, onde começando a partir de um vértice arbitrário v , percorrendo os vizinhos dos vértices descobertos. Aplicando-se assim, um tempo para as variavéis **LOW[]** e **DISC[]**, sempre considerando o grafo como conexo com apenas 1 componente, tendo em vista isso, após a seguinte condição **disc[u] < low[v]** ter sido verdadeira durante a busca em profundidade a aresta **[u, v]** é uma ponte. Com isso, o artigo trata que o algoritmo possui uma complexidade **O(V + E)**, muito menos custoso.

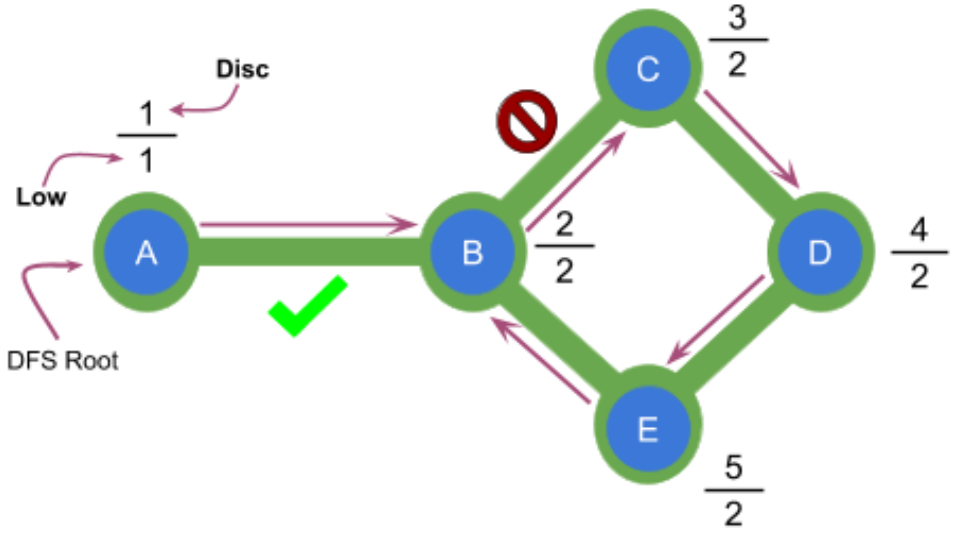


**Figura 7 – Tarjan (SEARLESER97, )**

## 3 ANALISE DE RESULTADOS

Todos os experimentos foram realizados em uma máquina:

- Processador i7-11390H - 3.4GHz - 2918Mhz - 4 núcleos
- Memória - 16GB - DDR4 - 3200MHz
- Windows 11
- SSD 500GB

### 3.1 Tabelas

A tabela 1 mostra todos os tempos de execução dos algoritmos Naive, o qual é um algoritmo mais custoso. Porém a tabela 2 possui todos os resultados da utilização algoritmo de Tarjan desenvolvido em 1974, onde o próprio autor aborda no artigo (TARJAN, 1974) que trata de um algoritmo de busca de pontes de forma mais eficiente. Vale resaltar que na tabela 1 alguns resultados foram escondidos com NA, pelo tempo de execução ser muito demorado, mais de 8 horas, sem obter resultado, além dos testes terem sido realizados com pelo menos 5 grafos para cada tipo de grafo, para se obter uma média do tempo estimado para execução do algoritmo. Com isso, a cada execução de uma aresta no algoritmo Fleury foi buscado as pontes após a remoção, fazendo com que o tempo do algoritmo Fleury seja mais demorado, por isso a baixo aborda o tempo médio de busca de pontes em vários grafos aleatórios gerados por código.

**Tabela 1 – Algoritmo Naive**

| Vértices | Grafo | Tempo (segundos) |
|---|---|---|
| 100 | Euleriano | 0000.0039 |
| 1000 | Euleriano | 0000.4889 |
| 10000 | Euleriano | 0127.1838 |
| 100000 | Euleriano | NA |
| 100 | Semi-euleriano | 0000.0159 |
| 1000 | Semi-euleriano | 0001.2655 |
| 10000 | Semi-euleriano | 0515.5800 |
| 100000 | Semi-euleriano | NA |
| 100 | Não-euleriano | 0000.0334 |
| 1000 | Não-euleriano | 0003.5124 |
| 10000 | Não-euleriano | 3680.2427 |
| 100000 | Não-euleriano | NA |

GitHub - zTrolly

**Tabela 2 – Algoritmo Tarjan 1974**

| Vértices | Grafo | Tempo (segundos) |
|---|---|---|
| 100 | Euleriano | 0000.0007 |
| 1000 | Euleriano | 0000.0400 |
| 10000 | Euleriano | 0003.4584 |
| 100000 | Euleriano | 0383.9894 |
| 100 | Semi-euleriano | 0000.0007 |
| 1000 | Semi-euleriano | 0000.0484 |
| 10000 | Semi-euleriano | 0007.9425 |
| 100000 | Semi-euleriano | 0765.4123 |
| 100 | Não-euleriano | 0000.0009 |
| 1000 | Não-euleriano | 0000.0548 |
| 10000 | Não-euleriano | 0050.2345 |
| 100000 | Não-euleriano | 4896.9488 |

GitHub - zTrolly

## 3.2 Discussão dos resultados

### *3.2.1 Algoritmo Naive*

A partir dos resultados obtidos através da execução dos algoritmos, pode-se observar que o algoritmo do Naive é muito mais custoso que o algoritmo do Tarjan, como pode ser explicado na figura 8, onde o algoritmo está no intervalo vermelho, fazendo com que o tempo de execução no eixo Y aumente conforme o número de arestas e vértices, mostrando que pelo algoritmo Naive apresentar diversas operações nas arestas e nos vértices, fazendo assim que a complexidade faça passar pelas arestas várias vezes, tendo um custo de complexidade muito alto, sendo ele O(E * (V + E)), isso explica o tempo elevado que execução, dependendo da quantidade de arestas e vértices que um grafo possui.

### *3.2.2 Algoritmo Tarjan*

O algoritmo do Tarjan é menos custoso, pois ele apresenta soluções que fazem diminuir consideravelmente o acesso as arestas, ou seja, o algoritmo evitou que as arestas precisassem ser visitadas mais de uma vez, fazendo assim que o custo seja linear ao aumento do números de vértices e arestas como mostrado na figura 8, ou seja, o algoritmo estaria no intervalo verde, onde ele passa por todas as arestas, mostrando assim que a complexidade do algoritmo é O(V + E).

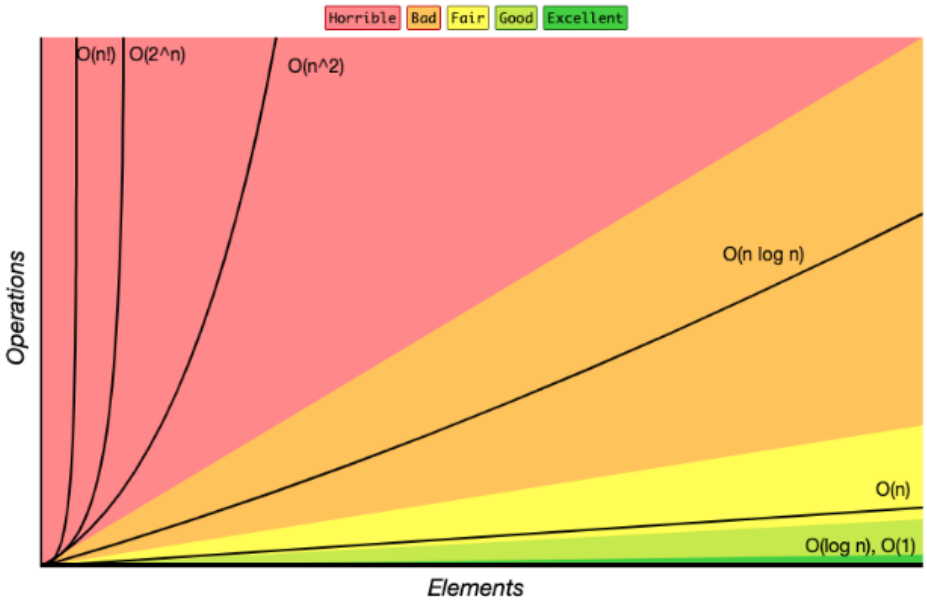


**Figura 8 – Gráfico Complexidade**

## 4 CONCLUSÃO

Pode-se concluir então que caso seja necessário implementar um algoritmo de identificação de pontes, por questão de tempo de execução o algoritmo de Tarjan é mais viável, por mais que seja de maior complexidade de implementação, já o algoritmo Naive é de mais fácil implementação, porém o custo de se ter uma fácil implementação vem no tempo, onde ele gasta um tempo longo para encontrar as pontes de grafos com muitos vértices e arestas.

## Referências

SEARLESER97. **Articulation points and bridges (Tarjan's Algorithm)**.

TARJAN, R. Endre. A note on finding the bridges of a graph. **Information Processing Letters**, Elsevier, v. 2, n. 6, p. 160–161, abr. 1974. ISSN 0020-0190.

WIKIPéDIA. **Spanning Tree**. 2022. Disponível em: <https://en.wikipedia.org/wiki/Spanning_tree#Spanning_forests>. Acesso em: 10 de outubro 2022.