

Trabalho Prático de Grafos

LibGrafos

Juliana P. G. da Cunha¹, Pedro Henrique M. de Oliveira²

¹Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Curso de Engenharia de Software

Abstract. *This article aims to document the results achieved by the students in developing a Python library for graph manipulation. The work is structured in three parts: (1) creation of a library for graph manipulation; (2) implementation of algorithms for detecting bridges in graphs, with a comparative analysis of the execution time between the approaches used; and (3) development of features for reading and saving graphs to files, allowing their visualization in the Gephi software. The remainder of the work consists of the report and presentation of the results obtained.*

Resumo. *Este artigo tem como objetivo documentar os resultados alcançados pelos alunos no desenvolvimento de uma biblioteca em Python para a manipulação de grafos. O trabalho está estruturado em três partes: (1) criação de uma biblioteca para manipulação de grafos; (2) implementação de algoritmos para a detecção de pontes em grafos, com análise comparativa do tempo de execução entre as abordagens utilizadas; e (3) desenvolvimento de funcionalidades para a leitura e salvamento de grafos em arquivos, permitindo sua visualização no software Gephi. O restante do trabalho consiste no relatório e na apresentação dos resultados obtidos.*

1. Introdução

Grafos são modelos matemáticos utilizados para representar relações entre objetos. Essas estruturas consistem em nós (vértices) conectados por ligações (arestas). O estudo de grafos une cientistas, acadêmicos e desenvolvedores de software interessados em entender essas relações, analisar seus comportamentos, como manipulá-las e aplicá-las no cotidiano.

Diversas representações podem ser categorizadas como grafos, desde que expressem a relação entre objetos, mesmo que, em alguns casos, essa relação seja considerada inexistente. A imagem a seguir ilustra uma rede de contatos, que pode ser considerada um grafo por seus contatos (vértices) e ligações entre eles (arestas).

A origem dos estudos sobre esses modelos datam o século XVIII, com os estudos do matemático suíço Leonhard Euler (1707- 1783). Seus trabalhos logo se tornaram influentes e essas estruturas passaram a ser amplamente usadas em diversos campos do conhecimento. Na tecnologia, por exemplo, são utilizadas para modelar redes computacionais e otimizar o tráfego de dados entre dispositivos. Nas ciências biológicas, grafos são utilizados em estudos filogenéticos para analisar a evolução das espécies e para modelar interações entre proteínas. Já nas ciências sociais, são usados para estudar redes de relacionamentos sociais, entre outras aplicações.



Figure 1. Social Network Analytics

Reconhecendo a relevância desses estudos para a sociedade, especialmente para o contexto acadêmico, os autores deste artigo decidiram criar a LibGrafos, uma biblioteca para manipulação de grafos desenvolvida em Python. Esta biblioteca foi desenvolvida com base nos conhecimentos adquiridos pelos autores durante as aulas ministradas pelo professor Leonardo Vilela Cardoso, docente da disciplina de grafos no curso de Engenharia de Software da PUC-MG. O objetivo deste artigo, portanto, é documentar o processo de desenvolvimento, as funcionalidades implementadas e os resultados alcançados com o uso deste software.

2. Metodologia

Este projeto foi dividido em três partes diferentes:

1. Criação de uma biblioteca para manipulação de grafos;
2. Implementação de algoritmos para a detecção de pontes em grafos, com análise comparativa do tempo de execução entre as abordagens utilizadas;
3. Desenvolvimento de funcionalidades para a leitura e salvamento de grafos em arquivos, permitindo sua visualização no software Gephi.

Sendo um grupo de dois integrantes, optou-se pelo desenvolvimento cooperativo, no entanto, a delegação de responsabilidades ficou da seguinte forma:

Parte 1

- Representação de grafos utilizando Matriz de Adjacência - Juliana
- Representação de grafos utilizando Matriz de Incidência - Pedro
- Representação de grafos utilizando Lista de Adjacência - Ambos
- Criação de um grafo com X vértices (o número de vértices deve ser inserido pelo usuário) - Juliana
- Criação e remoção de arestas - Pedro
- Ponderação e rotulação de vértices - Juliana
- Ponderação e rotulação de arestas - Pedro
- Checagem de adjacência entre arestas - Pedro
- Checagem de adjacência entre vértices - Juliana
- Checagem da existência de arestas - Juliana
- Checagem da quantidade de vértices e arestas - Pedro

- Checagem de grafo vazio e completo - Pedro
- Checagem de conectividade em simplesmente conexo, semi-fortemente conexo e fortemente conexo - Ambos
- Checagem de quantidade de componentes fortemente conexos com Kosaraju - Ambos
- Checagem de ponte e articulação - Pedro

Parte 2

(*Fleury-Tarjan e Fleury-Naive*) - Desenvolvida, testada e analisada conjuntamente.

Parte 3

- Juliana

2.1. Parte 1

No início do projeto, foram definidas seis classes principais para estruturar o sistema:

1. **Classe Vértice:** responsável por armazenar informações como rótulo, peso, arestas de entrada e saída, além do grau do vértice.
2. **Classe Aresta:** contém atributos como rótulo, peso, vértice de início e vértice de fim.
3. **Classe Grafo:** gerencia informações gerais do grafo, como direcionamento (direcionado ou não direcionado), uma lista de vértices, uma lista de arestas, um identificador único e instâncias para cada uma das representações (lista de adjacência, matriz de adjacência e matriz de incidência).
4. **Classe GrafoListaAdjacencia:** implementa a representação do grafo com base em uma lista de adjacência, contendo listas de vértices e arestas.
5. **Classe GrafoMatrizAdjacencia:** implementa a representação do grafo utilizando uma matriz de adjacência.
6. **Classe GrafoMatrizIncidencia:** implementa a representação do grafo com base em uma matriz de incidência.

Optou-se pela definição dessas seis classes para promover um melhor encapsulamento do código. Dessa forma, cada entidade é responsável por gerenciar suas próprias informações: os vértices e arestas armazenam dados de nós e ligações, enquanto a classe Grafo é responsável pela gestão desses elementos e atualiza suas representações conforme necessário. Todos os métodos básicos de manipulação estão na classe Grafo, exceto aqueles que lidam diretamente com a manipulação específica das listas e matrizes de representação.

Esse modelo, além de promover uma divisão clara de responsabilidades, permite que as representações do grafo sejam mantidas atualizadas de forma consistente. Assim, qualquer modificação no grafo é refletida automaticamente em suas representações por meio da própria classe Grafo.

A seguir, apresentaremos uma visão geral das principais funcionalidades da Parte 1, detalhando o processo de seu desenvolvimento.

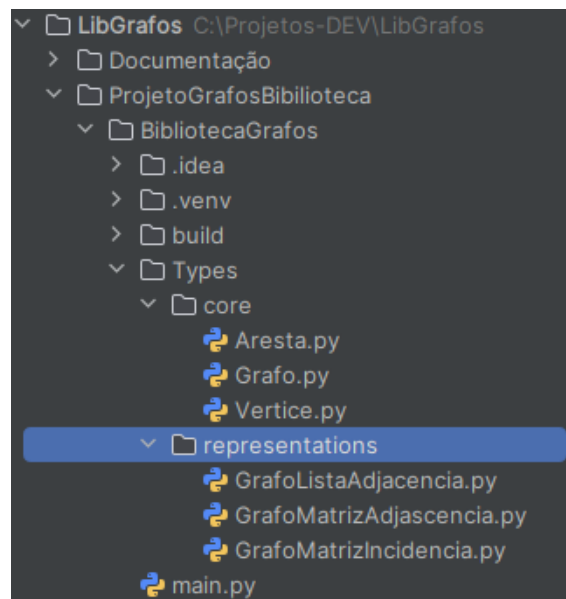


Figure 2. Estrutura biblioteca

2.1.1. Manipulação de Grafos

Entre os métodos básicos de manipulação, estão aqueles que permitem moldar a estrutura do grafo, como adicionar e remover vértices e arestas e gerar um grafo aleatório a partir de um número especificado de vértices. Essas operações são realizadas diretamente pela classe 'Grafo', que, na maioria dos casos, recebe como parâmetros instâncias de vértices e arestas.

Após a manipulação do grafo em si, que consiste quase sempre na alteração das listas 'vertices' e 'arestas', é necessário ajustar as representações. Um exemplo disso é a função responsável por adicionar vértices, como ilustrado a seguir:

2.1.2. Ponderação e Rotulação

Há duas formas de atribuir pesos e rótulos a vértices e arestas: (1) diretamente por meio do construtor desses objetos; e (2) utilizando funções que verificam a existência do objeto no grafo e, caso ele esteja presente, atualizam seus valores. Essa segunda abordagem facilita ao usuário a atualização de atributos de vértices ou arestas já inseridos no grafo. Em caso do objeto buscado não ser encontrado, é retornado um erro.

2.1.3. Verificação de Adjacência

A verificação de adjacência entre vértices e arestas é realizada por meio de duas abordagens. Para a adjacência entre vértices, utiliza-se a matriz de adjacência do grafo, que é uma matriz $n \times n$ (onde n representa o número de vértices). Nessa matriz, os valores são definidos como 0 para vértices não adjacentes e 1 para vértices adjacentes, permitindo consultas rápidas e diretas.

No caso da adjacência entre arestas, utiliza-se uma matriz de incidência, que é uma matriz $n \times m$ (onde n é o número de vértices e m o número de arestas). Essa matriz facilita a verificação da relação entre arestas.

2.1.4. Conectividade

A conectividade de um grafo pode ser verificada de diferentes formas. Ele pode ser classificado como simplesmente conexo, semi-fortemente conexo ou fortemente conexo, dependendo das condições que satisfaz. As funções a seguir verificam essas propriedades de conectividade.

A função `checar_se_simplesmente_conexo`, contida na classe `Grafo`, é responsável por verificar se um grafo é simplesmente conexo. Um grafo é simplesmente conexo se ele for conectado. Em outras palavras, não há vértices isolados ou grupos de vértices desconectados no grafo. A função funciona da seguinte maneira:

- Utilizamos uma busca em profundidade (DFS) para visitar todos os vértices.
- Um conjunto `visitados` é usado para rastrear os vértices já explorados.
- Após a DFS, verifica-se se todos os vértices foram visitados, garantindo que o grafo é conectado

A função `checar_se_semifortemente_conexo`, contida na classe `Grafo`, verifica se um grafo é semi-fortemente-conexo. Um grafo semi-fortemente conexo é um grafo direcionado em que, ao transformá-lo em não-direcionado, o grafo resultante é conexo. A função funciona da seguinte maneira:

- A conectividade é verificada usando DFS, mas considerando tanto as arestas de saída quanto as de entrada para simular um grafo não-direcionado.
- Todos os vértices acessíveis de qualquer vértice inicial são rastreados em `visitados`.
- Após a DFS, se todos os vértices foram visitados, o grafo é semi-fortemente conexo.

A função `checar_se_fortemente_conexo`, contida na classe `Grafo`, verifica se um grafo é fortemente conexo. Um grafo é fortemente conexo se, para qualquer par de vértices u e v :

1. Existe um caminho direcionado de u para v .
2. Existe um caminho direcionado de v para u .

A função funciona da seguinte maneira:

- A conectividade é verificada em duas etapas:
 1. A partir de um vértice inicial, uma DFS explora todas as arestas de saída. Isso verifica se todos os vértices são alcançáveis a partir do vértice inicial.
 2. A partir do mesmo vértice inicial, outra DFS explora todas as arestas de entrada, verificando se todos os vértices podem alcançar o vértice inicial.
- Se em qualquer etapa nem todos os vértices são visitados, o grafo não é fortemente conexo.
- Caso contrário, o grafo é fortemente conexo.

2.1.5. Quantidade de Componentes Fortemente Conexos

Para retornar a quantidade de componentes fortemente conexos, optou-se por utilizar o algoritmo de Kosaraju. Esse método é uma abordagem eficiente para determinar o número de componentes fortemente conexos (CFCs) em um grafo dirigido. Ele utiliza duas passagens de busca em profundidade (DFS) e tem complexidade $O(V + E)$, onde V é o número de vértices e E é o número de arestas. O método funciona da seguinte maneira:

Em um primeiro momento, é realizada uma DFS no grafo original:

- A cada vértice explorado, ele é adicionado ao conjunto `visitados`.
- Após visitar todos os seus vizinhos, o vértice é empilhado na lista `ordem_finalizacao`.
- O objetivo dessa etapa é determinar a ordem de finalização dos vértices, que será usada na segunda passagem para processá-los em ordem inversa.

Após a primeira DFS, o algoritmo utiliza as arestas de entrada (`get_arestas_de_entrada()`) para simular o grafo transposto. A transposição do grafo inverte todas as arestas, permitindo que os componentes fortemente conexos sejam explorados em isolamento.

Após a transposição:

- Os vértices são processados na ordem inversa de sua finalização, conforme armazenado em `ordem_finalizacao`.
- Para cada vértice ainda não visitado, uma nova DFS é realizada, explorando o grafo transposto.
- Cada componente fortemente conexo encontrado é armazenado em uma lista temporária `componente`, que é posteriormente adicionada à lista `componentes`.

Por fim, o número de componentes fortemente conexos é dado pelo tamanho da lista `componentes`, que armazena todos os subgrafos do grafo em questão.

Esse algoritmo executa duas buscas em profundidade, uma no grafo original e outra no grafo transposto, além de percorrer as arestas. Assim, a complexidade total é $O(V + E)$, onde V é o número de vértices e E é o número de arestas.

2.1.6. Checagem de Pontes e Articulação

Para identificar pontes e pontos de articulação em um grafo, utilizamos o algoritmo de Tarjan com base na execução de uma busca em profundidade (DFS). O algoritmo explora as propriedades de conectividade e tempo de descoberta dos vértices para determinar arestas e vértices críticos, cujo comportamento pode desconectar o grafo.

Uma ponte é uma aresta que, ao ser removida, aumenta o número de componentes desconexos do grafo. A detecção é baseada em duas métricas calculadas durante a DFS:

Tempo de descoberta (*discovery time*): Momento em que cada vértice é visitado pela primeira vez. Tempo mínimo alcançável (*low time*): Menor tempo de descoberta alcançável por um vértice diretamente ou indiretamente. O algoritmo compara o *low time* de um vértice filho com o *discovery time* de seu vértice pai na árvore de DFS. Se

o *low time* do filho é maior, significa que não existe outro caminho conectando os dois, caracterizando uma ponte.

Um ponto de articulação é um vértice que, ao ser removido, desconecta parte do grafo. A detecção considera se o vértice é a raiz da árvore de DFS e possui mais de um filho. Se não é raiz, mas o *low time* de algum de seus filhos é maior ou igual ao seu *discovery time*. Isso indica que o subgrafo do filho depende exclusivamente do vértice atual para permanecer conectado ao restante do grafo, tornando-o assim um vértice articulação.

No algoritmo implementado para as detecções citadas acima, são armazenados os tempos de descoberta e menor tempo alcançável. As listas de adjacência do grafo foram iteradas para executar a DFS de forma iterativa, garantindo robustez e controle explícito sobre o estado da pilha. Foram adicionadas condições específicas para identificar e armazenar as pontes e os pontos de articulação encontrados.

2.2. Parte 2

Nesta etapa, buscamos implementar dois métodos para detecção de pontes em grafos eulerianos, bem como utilizar o Algoritmo de Fleury para encontrar ciclos eulerianos. O problema das pontes em grafos é crucial, pois envolve identificar arestas cuja remoção desconecta o grafo, sendo uma tarefa relevante em aplicações que requerem a identificação de caminhos ou ciclos eulerianos.

2.2.1. Construção de Grafos Eulerianos

Para criar os grafos eulerianos com o número desejado de vértices, foi desenvolvido o método apresentado na Figura 3. Esse método assegura que o grafo gerado atenda às propriedades de um grafo euleriano, ou seja, que todos os seus vértices possuam grau par.

```
def criar_grafo_euleriano(quantidade_vertices): 1 usage: 4 pedro
    grafo = Grafo(id="grafogrande", bidirecional=False)

    vertices = []
    for i in range(quantidade_vertices):
        vertice = Vertice(peso_vertices+str(i), rotulo=f"V{i}")
        vertices.append(vertice)
        grafo.adicionarVertice(vertice)

    for i in range(quantidade_vertices - 1):
        grafo.adicionarAresta(Aresta(vertices[i], vertices[i + 1], rotulo=f"E{i}", peso=1))

    grafo.adicionarAresta(
        Aresta(vertices[quantidade_vertices - 1], vertices[0], rotulo=f"E{quantidade_vertices - 1}", peso=1))

    return grafo
```

Figure 3. Método para criação de grafos eulerianos

2.2.2. Detecção de Pontes

Foram implementadas duas abordagens para identificar as pontes no grafo:

- Método ingênuo (naive): testa a conectividade do grafo removendo sistematicamente cada aresta e utilizando uma busca em profundidade ou largura para verificar se o grafo permanece conectado.
- Método baseado no Algoritmo de Tarjan: uma abordagem eficiente que determina as pontes utilizando propriedades de busca em profundidade com tempo linear em relação ao número de vértices e arestas.

2.2.3. Execução e Comparação

Após implementados os métodos de detecção de pontes, utilizamos o Algoritmo de Fleury para encontrar ciclos eulerianos no grafo, empregando ambas as estratégias de detecção. Os tempos de execução de cada abordagem foram medidos e comparados para grafos com diferentes tamanhos: 100, 1.000, 10.000 e 100.000 vértices.

A execução das comparações foi organizada conforme ilustrado na Figura 4.

```
if __name__ == "__main__":  
    # menu()  
  
    quantidade_vertices = 100  
  
    print(f'Construindo um grafo com {quantidade_vertices} vértices...')  
    start_time = time.time()  
    grafo_grande = criar_grafo_grande(quantidade_vertices)  
    print(f'Grafo construído em {time.time() - start_time:.5f} segundos.')  
  
    print("Executando Fleury com Tarjan...")  
    start_time = time.time()  
    resultado_tarjan = grafo_grande.fleury_tarjan()  
    print(f'Algoritmo Fleury com Tarjan executado em {time.time() - start_time:.5f} segundos.')  
  
    print("Executando Fleury com Naive...")  
    start_time = time.time()  
    resultado_naive = grafo_grande.fleury_naive()  
    print(f'Algoritmo Fleury com Naive executado em {time.time() - start_time:.5f} segundos.')
```

Figure 4. Processo de obtenção e comparação de resultados

Com essa metodologia, buscamos não apenas avaliar a eficiência dos dois métodos de detecção de pontes, mas também explorar suas implicações na aplicação do Algoritmo de Fleury em grafos eulerianos de diferentes dimensões.

2.3. Parte 3

Após a conclusão das partes 1 e 2, a parte 3 envolve a visualização dos grafos gerados. Isso é possível por meio da função 'criar arquivo grafo graphml', presente na classe Grafo. Essa função gera um arquivo .graphml nomeado conforme o identificador (id) do grafo, que é definido no construtor.

Para visualizar o grafo, o usuário precisará baixar o software Gephi, que permite a visualização de grafos. Ao abrir o arquivo no Gephi, será possível visualizar uma representação do grafo instanciado.

3. Resultados

Nesta seção, serão apresentados os resultados obtidos após o desenvolvimento de cada uma das etapas. Para validar os resultados, foi criado um menu, acessível pelo arquivo 'main.py', que contém todas as opções, conforme ilustrado na imagem a seguir. Caso o usuário prefira, também é possível testar a biblioteca por meio da manipulação manual dos objetos.

3.1. Resultados Parte 1

Após a implementação dos métodos da primeira parte, conforme a metodologia especificada, foi possível verificar a funcionalidade da criação e manipulação de grafos direcionados e não direcionados, com a capacidade de adicionar vértices e arestas, tanto manualmente quanto de forma aleatória. Além disso, foi possível atribuir valores e pesos a esses componentes. Também foi confirmada a geração consistente das listas e matrizes

de adjacência, bem como das matrizes de adjacência dos grafos. Outros métodos para verificar conectividade, adjacência de vértices e arestas, grafos vazios e completos, estão funcionando. Resultados das representações abaixo:

Escolha uma opção: 4

Matriz de Adjacência:

	V0	V1	V2	V3
V0	0	1	1	1
V1	0	0	1	1
V2	0	0	0	1
V3	0	0	0	0

Figure 5. Exibição Matriz de Adjacência

24. Verificar adjacência entre arestas

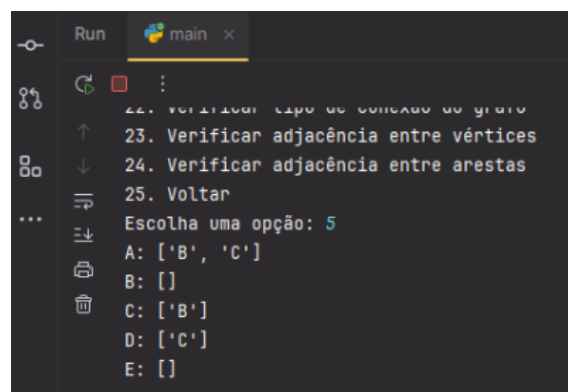
25. Voltar

Escolha uma opção: 6

Matriz de Incidência:

	A-B	A-C	C-B	D-C
A	-1	-1	0	0
B	1	0	1	0
C	0	1	-1	1
D	0	0	0	-1
E	0	0	0	0

Figure 6. Exibição Matriz de Incidência



```
Run main x
22. Verificar tipo de conexão do grafo
23. Verificar adjacência entre vértices
24. Verificar adjacência entre arestas
25. Voltar
Escolha uma opção: 5
A: ['B', 'C']
B: []
C: ['B']
D: ['C']
E: []
```

Figure 7. Exibição Lista de Adjacência

3.2. Resultados Parte 2

Nesta sessão, serão comparados os desempenhos de cada um dos métodos implementados na parte dois do projeto, segundo a metodologia descrita.

3.2.1. Desempenho Observado

Os resultados obtidos demonstraram que o método naive apresentou um crescimento exponencial no tempo de execução conforme o número de vértices no grafo aumentava, tornando-se inviável para grafos maiores. Por outro lado, o método baseado no Algoritmo de Tarjan manteve um desempenho eficiente mesmo para grafos de maior escala, como esperado.

3.2.2. Comportamento do Método Naive

O método naive, responsável por identificar pontes no grafo, opera removendo sistematicamente cada aresta e verificando a conectividade do grafo restante. Essa abordagem, embora funcional, é altamente ineficiente devido a dois fatores principais:

1. **Complexidade da Verificação de Conectividade:** Para cada aresta removida, a função realiza uma busca em profundidade (DFS) para determinar se o grafo permanece conectado. Essa operação possui complexidade $O(V + E)$, onde V é o número de vértices e E , o número de arestas.
2. **Iteração sobre Todas as Arestas:** Para identificar todas as pontes, o método testa cada aresta individualmente, resultando em uma complexidade combinada de $O(E \cdot (V + E))$. À medida que o número de vértices aumenta, o número de arestas também cresce (em grafos densos, $E \approx V^2$), amplificando drasticamente o custo computacional.

Desempenho do Algoritmo de Tarjan Em contraste, o método baseado no Algoritmo de Tarjan apresentou desempenho muito superior devido à sua complexidade linear $O(V + E)$. Ele utiliza uma única busca em profundidade para calcular os tempos de descoberta (`discovery_time`) e valores mínimos (`low_time`) de cada vértice, identificando pontes de maneira eficiente. Por essa razão:

Impacto na Execução do Algoritmo de Fleury Ao aplicar o Algoritmo de Fleury, a detecção de pontes influencia diretamente a eficiência do processo. O tempo de execução do método naive tornou-se um gargalo significativo, enquanto a abordagem baseada no Tarjan permitiu a execução fluida do Fleury mesmo em grafos maiores.

Limitações Observadas Para grafos com 100.000 vértices, o sistema enfrentou limitações de memória ao tentar construir o grafo, o que indica que a implementação utilizada necessita de otimizações adicionais para suportar instâncias tão grandes. A mensagem de erro observada foi:

3.3. Resultados observados

```
C:\Projetos-DEV\trab-grafos\pythonProject1\.venv\Scripts\python
Construindo um grafo com 100 vértices...
Grafo construído em 0.00207 segundos.
Executando Fleury com Tarjan...
Algoritmo Fleury com Tarjan executado em 0.00103 segundos.
Executando Fleury com Naive...
Algoritmo Fleury com Naive executado em 0.01061 segundos.
```

Figure 8. Resultado ao executar com 100 vértices

```
C:\Projetos-DEV\trab-grafos\pythonProject1\.venv\Scripts\python.exe
Construindo um grafo com 1000 vértices...
Grafo construído em 0.17830 segundos.
Executando Fleury com Tarjan...
Algoritmo Fleury com Tarjan executado em 0.03257 segundos.
Executando Fleury com Naive...
Algoritmo Fleury com Naive executado em 7.20439 segundos.

Process finished with exit code 0
```

Figure 9. Resultado ao executar com 1.000 vértices

Table 1. Resultados de Tempo de Execução

Nº de Vértices	Método Naive	Método Algoritmo Tarjan
100	0.01061	0.00103
1.000	7.20439	0.03257
10.000	<i>Erro de memória</i>	2.34835
100.000	<i>Erro de memória</i>	<i>Erro de memória</i>

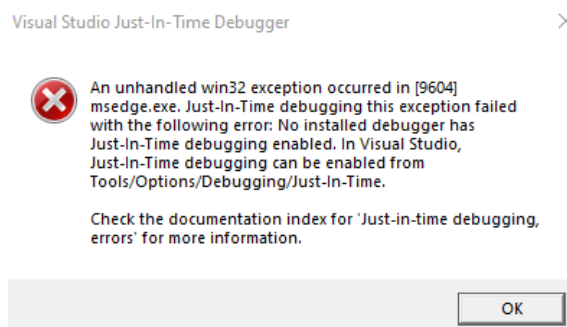


Figure 10. Resultado ao executar com 100.000 vértices

```
C:\Projetos-DEV\trab-grafos\pythonProject1\.venv\Scripts\python.exe C:\Projetos-DEV\trab-grafos\Biblioteca\Biblioteca\main.py
Construindo um grafo com 10000 vértices...
Grafo construído em 18.5320 segundos.
Executando Fleury com Tarjan...
Algoritmo Fleury com Tarjan executado em 2.34835 segundos.
Executando Fleury com Naive...
```

Figure 11. Resultado ao executar com 10.000 vértices

3.4. Resultados Parte 3

Para demonstrar os resultados da Parte 3, a seguir estão duas imagens de dois grafos direcionados, o primeiro [Figura 12], contendo 15 vértices e o segundo [Figura 13] contendo 30 vértices.

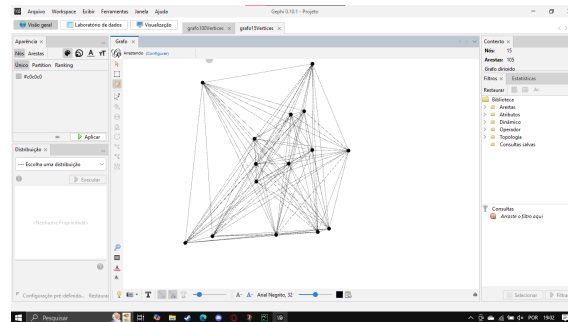


Figure 12. Grafo 15 vértices

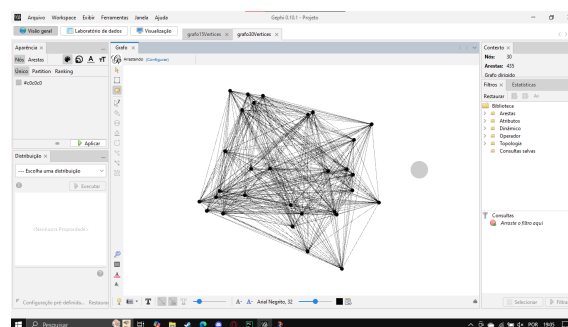


Figure 13. Grafo 30 vértices

4. Conclusão

Com o desenvolvimento da LibGrafos, foi possível não apenas consolidar os conhecimentos adquiridos em sala de aula, mas também desenvolver uma ferramenta que pode ser expandida e adaptada para diferentes necessidades dentro da área de teoria dos grafos. A LibGrafos, portanto, representa um avanço significativo para estudantes que buscam uma maneira prática e eficiente de trabalhar com grafos em Python, ao mesmo tempo em que abre portas para novas implementações e melhorias no futuro.

Como projeções futuras, os desenvolvedores pretendem se aprofundar na linguagem Python, explorando novas técnicas e abordagens para otimizar a eficiência da biblioteca e deixar o código mais limpo. Além disso, busca-se tornar a biblioteca ainda mais flexível e escalável, atendendo a um público maior e a aplicações mais complexas no campo da teoria dos grafos.

5. References

[Nanawati 2019]

References

Nanawati, S. (2019). Social network analytics (with a case study in python). Accessed: 2024-11-24.