Trabalho Prático Grafos

**Alfredo L. Vieira, Bruno E. G. de Azevedo, Davi J. Ferreira,
Estêvão de F. Rodrigues, Vinicius S. de Oliveira**

[1]Pontifícia Universidade Católica de Minas Gerais (PUC-MG)
R. Dom José Gaspar, 500 - Coração Eucarístico, Belo Horizonte - MG, 30535-901

Resumo. *Este projeto acadêmico consiste no desenvolvimento de uma biblioteca para manipulação de grafos, com três etapas principais. Na primeira, serão implementadas representações de grafos por Matriz de Adjacência, Matriz de Incidência e Lista de Adjacência, além de funções básicas como criação e remoção de arestas, ponderação e rotulação, checagem de conectividade, pontes e articulações. A segunda etapa envolve a implementação de dois métodos para identificação de pontes: uma abordagem ingênua e o algoritmo de Tarjan, seguido pela aplicação do Algoritmo de Fleury para encontrar caminhos eulerianos, comparando o desempenho em grafos de diferentes tamanhos. Por fim, a terceira etapa adiciona funcionalidade de leitura e exportação de grafos em formatos compatíveis com o Gephi, permitindo a geração de visualizações. O trabalho será documentado em um relatório conforme o modelo SBC, incluindo análise dos resultados e apresentações presenciais e em vídeo.*

1. Responsabilidades

As responsabilidades no projeto foram distribuídas entre os integrantes da equipe, conforme detalhado abaixo:

- **Carregar grafo a partir de arquivo JSON:** Responsável: Davi.
- **Carregar grafo a partir de arquivo CSV:** Responsável: Alfredo.
- **Criar grafo aleatório baseado em seu número de vértices:** Responsável: Estêvão.
- **Adicionar vértice:** Responsável: Alfredo.
- **Adicionar aresta:** Responsável: Alfredo.
- **Exibir grafo:** Responsável: Davi.
- **Alterar vértice:** Responsável: Bruno.
- **Alterar aresta:** Responsável: Bruno.
- **Deletar vértice:** Responsável: Alfredo.
- **Deletar aresta:** Responsável: Alfredo.
- **Checar adjacência entre vértices:** Responsável: Bruno.
- **Checar adjacência entre arestas:** Responsável: Bruno.
- **Checagem da existência de arestas:** Responsável: Bruno.
- **Checagem da quantidade de vértices e arestas:** Responsável: Bruno.
- **Checagem de grafo vazio e completo:** Responsável: Equipe.
- **Checagem de articulações (por busca em profundidade):** Responsável: Estêvão.
- **Exibir matriz de adjacência:** Responsável: Alfredo.
- **Exibir matriz de incidência:** Responsável: Alfredo.

- **Verificar conectividade:** Responsável: Vinicius.
- **Verificar conectividade (naive):** Responsável: Vinicius.
- **Exibir lista de adjacência:** Responsável: Equipe.
- **Verificar componentes fortemente conectos com Kosaraju:** Responsável: Davi.
- **Encontrar pontes (Tarjan):** Responsável: Davi.
- **Exportar CSV:** Responsável: Alfredo.
- **Executar Fleury:** Responsável: Bruno.
- **Executar Fleury com Tarjan:** Responsável: Bruno.
- **Encontrar pontes:** Responsável: Vinicius.

2. Estruturas de Dados

O sistema desenvolvido para manipulação de grafos foi estruturado com base em três componentes principais: vértices, arestas e a classe principal de grafos. Cada elemento foi cuidadosamente projetado para suportar operações robustas e flexíveis, garantindo modularidade e reutilização de código.

2.1. Vértices

A classe `Vertice` representa os nós do grafo, sendo caracterizada por:

- **Rótulo**: Identificador único que distingue cada vértice.
- **Peso**: Atributo opcional que pode ser usado em algoritmos ponderados.
- **Arestas**: Lista de arestas conectadas a esse vértice, permitindo rastrear diretamente suas conexões.

Métodos auxiliares, como `adicionar_aresta`, garantem que novas conexões sejam registradas de forma eficiente e consistente.

2.2. Arestas

A classe `Aresta` representa as conexões entre vértices, com os seguintes atributos:

- **Rótulo**: Identificador único da aresta.
- **Peso**: Valor associado à conexão, essencial para grafos ponderados.
- **Vértices**: Uma tupla que armazena os dois vértices conectados por essa aresta.

Funções auxiliares, como `alterar_aresta`, permitem modificar os atributos de uma aresta existente, enquanto `listar_arestas` facilita a inspeção e interação com o conjunto de arestas.

2.3. Grafo

A classe `Grafo` é o núcleo do sistema, encapsulando tanto a estrutura quanto as operações dos grafos. Suas principais características incluem:

- **Direcionamento**: Os grafos podem ser direcionados ou não, afetando diretamente a conectividade e as operações realizadas.
- **Coleções de Vértices e Arestas**: Estruturas internas para armazenar os elementos do grafo.
- **Operações Fundamentais**:
  - Adição e remoção de vértices e arestas.
  - Checagem de adjacência e conectividade.
  - Geração de representações, como matrizes de adjacência e de incidência.

Essas funcionalidades foram projetadas para trabalhar de forma integrada com a estrutura do grafo, permitindo operações complexas de forma eficiente.

2.4. Extensibilidade e Exportação

O sistema também suporta a leitura e exportação de grafos em formatos como CSV, viabilizando a integração com ferramentas externas, como o Gephi. Essa funcionalidade assegura que os grafos manipulados possam ser visualizados e analisados com ferramentas especializadas, promovendo maior utilidade e aplicabilidade prática.

Essa organização modular facilita a manutenção e a extensão do sistema, garantindo flexibilidade para suportar casos de uso variados.

3. Funcionalidades do Sistema de Manipulação de Grafos

O sistema desenvolvido apresenta uma ampla gama de funcionalidades para a manipulação de grafos. Cada funcionalidade foi projetada para atender a diferentes necessidades na análise e modificação de grafos, proporcionando flexibilidade e robustez. Abaixo, descrevemos cada uma delas de forma detalhada.

3.1. Carregar Grafo a Partir de Arquivo JSON

A funcionalidade permite carregar grafos diretamente de arquivos no formato JSON. Isso possibilita a reutilização de grafos previamente salvos e facilita a integração com outras ferramentas. O arquivo JSON deve conter informações sobre os vértices (rótulo e peso) e as arestas (rótulo, peso, e vértices conectados).

```
grafo = carregar_grafo_json("grafo.json")
```

3.2. Carregar Grafo a Partir de Arquivo CSV

Essa funcionalidade permite carregar grafos a partir de arquivos CSV, outro formato amplamente utilizado para armazenamento de dados. O usuário deve informar se o grafo é direcionado e fornecer o nome do arquivo CSV. Essa abordagem é útil para integração com sistemas que geram dados em formato tabular.

```
grafo = grafo.ler_grafo_from_csv("grafo.csv")
```

3.3. Criar Grafo Aleatório Baseado no Número de Vértices

O sistema pode gerar grafos aleatórios com um número definido de vértices. Isso é especialmente útil para testes e simulações. O usuário especifica o número de vértices, e o sistema conecta os vértices de forma aleatória.

```
grafo = grafo.criarGrafo()
```

3.4. Adicionar Vértice ao Grafo

O sistema permite a adição de vértices com rótulo e peso definidos pelo usuário. Essa funcionalidade é essencial para expandir um grafo existente.

```
grafo.adicionar_vertice("V1", 10)
```

3.5. Adicionar Aresta ao Grafo

O sistema também suporta a adição de arestas entre vértices. O usuário define o rótulo, peso e os vértices conectados. Caso os vértices não sejam válidos ou não existam, o sistema retorna um erro.

```
grafo.adicionar_aresta("A1", 5, vertice1, vertice2)
```

3.6. Exibir Grafo

A funcionalidade permite visualizar o grafo atual, incluindo os vértices e as conexões existentes. Essa representação é útil para inspecionar rapidamente a estrutura do grafo.

```
print(grafo)
```

3.7. Alterar Vértice do Grafo

O sistema possibilita a alteração do rótulo e peso de um vértice existente. O usuário seleciona o vértice a partir de um índice, insere os novos valores e o grafo é atualizado.

```
grafo.alterar_vertice(vertice, "NovoRótulo", 20)
```

3.8. Alterar Aresta do Grafo

A funcionalidade permite modificar o rótulo e o peso de uma aresta. O usuário escolhe a aresta e fornece os novos valores para atualização.

```
grafo.alterar_aresta(aresta, "NovaAresta", 15)
```

3.9. Deletar Vértice do Grafo

É possível remover um vértice do grafo, incluindo todas as arestas conectadas a ele. O usuário seleciona o vértice pelo índice, e a operação é realizada.

```
grafo.remover_vertice(vertice)
```

3.10. Deletar Aresta do Grafo

Semelhante à funcionalidade anterior, o sistema permite a remoção de arestas específicas, selecionadas pelo índice.

```
grafo.remover_aresta(aresta)
```

3.11. Checar Adjacência Entre Vértices

Essa funcionalidade verifica se dois vértices estão diretamente conectados por uma aresta. O resultado informa se há ou não adjacência.

```
grafo.checar_adjacencia_vertices(vertice1, vertice2)
```

3.12. Checar Adjacência Entre Arestas

Verifica se duas arestas compartilham pelo menos um vértice em comum, indicando adjacência entre elas.

```
grafo.checar_adjacencia_arestas(aresta1, aresta2)
```

3.13. Checar Existência de Arestas

Essa funcionalidade verifica se há arestas no grafo. Caso não existam arestas, o grafo é considerado vazio nesse aspecto.

```
grafo.checar_arestas()
```

3.14. Checar Quantidade de Vértices e Arestas

O sistema conta e exibe a quantidade de vértices e arestas presentes no grafo, auxiliando na análise da estrutura.

```
print(f"Vértices: {len(grafo.vertices)}, Arestas: {len(grafo.arestas)}"
```

3.15. Checar Grafo Vazio e Completo

Permite verificar se o grafo é vazio (sem vértices ou arestas) ou completo (todos os vértices conectados a todos os outros).

```
grafo.checar_grafo()
```

3.16. Checar Articulações

Essa funcionalidade identifica vértices cuja remoção desconecta o grafo. O método é baseado em busca em profundidade.

```
grafo.checar_articulacoes()
```

3.17. Exibir Matriz de Adjacência

Exibe a matriz de adjacência do grafo, mostrando as conexões entre os vértices. Pode ser usada para grafos direcionados ou não.

```
grafo.exibir_matriz_adjacencia()
```

3.18. Exibir Matriz de Incidência

Exibe a matriz de incidência, representando as relações entre vértices e arestas.

```
grafo.exibir_matriz_incidencia()
```

3.19. Verificar Conectividade

Verifica se o grafo é conectado, ou seja, se há caminhos entre todos os pares de vértices.

```
grafo.verificar_conectividade()
```

3.20. Verificar Componentes Fortemente Conexos com Kosaraju

Aplica o algoritmo de Kosaraju para encontrar componentes fortemente conectados em grafos direcionados.

```
kosaraju(grafo)
```

3.21. Encontrar Pontes (Tarjan)

Aplica o algoritmo de Tarjan para identificar pontes, ou seja, arestas cuja remoção desconecta o grafo.

```
pontes = encontrar_pontes_tarjan(grafo)
```

3.22. Exportar Grafo para CSV

Permite exportar a estrutura do grafo para um arquivo CSV, facilitando a interoperabilidade com outras ferramentas.

```
grafo.gerar_csv("grafo_exportado.csv")
```

3.23. Executar Algoritmo de Fleury

Executa o Algoritmo de Fleury para encontrar um caminho Euleriano no grafo, se ele existir.

```
resultado = fleury(grafo)
```

3.24. Executar Algoritmo de Fleury com Tarjan

Combina o Algoritmo de Fleury com o algoritmo de Tarjan para melhorar a identificação de caminhos Eulerianos em grafos.

```
resultado = fleury_com_tarjan(grafo)
```

4. Classe Grafo

A classe `Grafo` foi desenvolvida para representar estruturas de grafos e fornecer métodos que permitem a manipulação eficiente de vértices e arestas. Nesta seção, apresentamos as principais funcionalidades e métodos implementados na classe.

4.1. Definição da Classe

A classe `Grafo` possui três atributos principais:

- **vertices**: Uma lista que armazena todos os vértices do grafo, instâncias da classe `Vertice`.
- **arestas**: Uma lista que armazena todas as arestas do grafo, instâncias da classe `Aresta`.
- **direcionado**: Um valor booleano que indica se o grafo é direcionado (`True`) ou não direcionado (`False`).

```
class Grafo:
    def __init__(self, direcionado: bool):
        self.vertices = []         # Lista de vértices no grafo
        self.arestas = []          # Lista de arestas no grafo
        self.direcionado = direcionado  # Determina se o grafo é direci
```

4.2. Métodos da Classe

4.2.1. Adicionar Vértices e Arestas

Os métodos `adicionar_vertice` e `adicionar_aresta` permitem adicionar vértices e arestas ao grafo. Cada vértice possui um rótulo único e um peso opcional, enquanto as arestas conectam dois vértices e podem ser ponderadas.

```
def adicionar_vertice(self, rotulo: str, peso: int = 0):
    vertice = Vertice(rotulo, peso)
    self.vertices.append(vertice)
    return vertice

def adicionar_aresta(self, rotulo: str, peso: int, vertice1: Vertice, v
    aresta = Aresta(rotulo, peso, vertice1, vertice2)
    self.arestas.append(aresta)
    vertice1.adicionar_aresta(aresta)
    vertice2.adicionar_aresta(aresta)
    return aresta
```

4.2.2. Remoção de Vértices e Arestas

Os métodos `remover_vertice` e `remover_aresta` permitem remover elementos do grafo. Ao remover um vértice, todas as arestas conectadas a ele também são excluídas.

```
def remover_vertice(self, vertice: Vertice):
    if vertice in self.vertices:
        for aresta in list(vertice.arestas):
            self.remover_aresta(aresta)
        self.vertices.remove(vertice)

def remover_aresta(self, aresta: Aresta):
    if aresta in self.arestas:
        self.arestas.remove(aresta)
        aresta.vertices[0].arestas.remove(aresta)
        aresta.vertices[1].arestas.remove(aresta)
```

4.2.3. Verificação de Adjacência

A classe permite verificar a adjacência entre vértices e arestas, identificando se há conexões diretas ou compartilhamento de vértices.

```
def checar_adjacencia_vertices(self, vertice1: Vertice, vertice2: Verti
    return any(
        aresta for aresta in self.arestas if (
            aresta.vertices == (vertice1, vertice2) or
            (not self.direcionado and aresta.vertices == (vertice2, ver
        )
    )

def checar_adjacencia_arestas(self, aresta1: Aresta, aresta2: Aresta) -
    return bool(set(aresta1.vertices) & set(aresta2.vertices))
```

4.2.4. Exibição do Grafo

O método `__str__` fornece uma representação textual do grafo, exibindo os vértices e arestas presentes.

```
def __str__(self):
    vertices_str = ", ".join([v.rotulo for v in self.vertices])
    arestas_str = "\n".join([str(a) for a in self.arestas])
    return f"Grafo (Direcionado: {self.direcionado}):\nVertices: {verti
```

4.2.5. Análise Estrutural do Grafo

A classe oferece métodos para verificar características estruturais do grafo, como se ele é vazio, completo, ou contém pontos de articulação.

```
def checar_grafo(self):
    if len(self.vertices) == 0:
        return "Este grafo está vazio!"
    for vertice in self.vertices:
        conexoes = {aresta.vertices[1] for aresta in self.arestas if ar
            {aresta.vertices[0] for aresta in self.arestas if aresta.ve
        if len(conexoes) != len(self.vertices) - 1:
            return "O grafo não é completo!"
    return "O grafo é completo!"
```

4.2.6. Representação em Matrizes

A classe implementa métodos para gerar matrizes de adjacência e incidência, tanto para grafos direcionados quanto não direcionados.

```
def exibir_matriz_adjacenciaND(self):
    matrizAdj = []
    for vertice1 in self.vertices:
        linha = []
        for vertice2 in self.vertices:
            linha.append(1 if (vertice1 != vertice2 and
                any((aresta.vertices == (vertice1, vertice2) or
                aresta.vertices == (vertice2, vertice1)) for aresta in
        matrizAdj.append(linha)
    return matrizAdj
```

5. Classe Vertice

A classe `Vertice` é responsável por representar os vértices de um grafo, incluindo suas propriedades e conexões. Ela oferece métodos para gerenciar os rótulos, pesos e as arestas associadas a cada vértice, permitindo uma manipulação direta e eficiente.

5.1. Estrutura da Classe

A classe `Vertice` possui os seguintes atributos:

- **`rotulo`**: Identificador único do vértice, representado por uma string.
- **`peso`**: Valor numérico associado ao vértice, utilizado em análises ou algoritmos.
- **`arestas`**: Lista de arestas conectadas ao vértice.

5.2. Métodos Principais

5.2.1. `__init__`

O método construtor inicializa os atributos do vértice, configurando o rótulo, peso e uma lista vazia de arestas conectadas. Por exemplo:

```
vertice = Vertice("V1", 5)
```

5.2.2. `adicionar_aresta`

Permite adicionar uma aresta ao vértice. Isso mantém um registro das conexões associadas a ele, facilitando operações como busca de adjacências.

```
vertice.adicionar_aresta(aresta)
```

5.2.3. `__str__`

Retorna uma representação textual do vértice, incluindo seu rótulo e peso, ideal para depuração e exibição.

```
print(vertice)
# Saída: "Vértice V1 (Peso: 5)"
```

5.3. Funções Relacionadas

Além dos métodos internos, há funções auxiliares para manipulação e visualização de vértices:

5.3.1. `listar_vertices`

Essa função exibe todos os vértices disponíveis, juntamente com seus índices. É útil para seleção de vértices em listas maiores.

```
listar_vertices(grafo.vertices)
# Exemplo de saída:
# 0: Vértice V1 (Peso: 5)
# 1: Vértice V2 (Peso: 10)
```

5.3.2. `alterar_vertice`

Permite modificar o rótulo e o peso de um vértice selecionado pelo usuário. Caso nenhuma alteração seja necessária, o usuário pode deixar os campos vazios para manter os valores existentes.

```
alterar_vertice(grafo.vertices)
# Fluxo interativo:
# Digite o índice do vértice que deseja alterar: 0
# Novo rótulo para o vértice V1 (deixe vazio para manter): V1.1
# Novo peso para o vértice 5 (deixe vazio para manter): 8
# Vértice atualizado: Vértice V1.1 (Peso: 8)
```

6. Classe Aresta

A classe `Aresta` é responsável por representar as conexões entre os vértices de um grafo. Ela encapsula informações sobre os vértices conectados, o peso associado à aresta e fornece métodos para manipulação e exibição dessas informações.

6.1. Estrutura da Classe

A classe `Aresta` possui os seguintes atributos:

- **`rotulo`**: Identificador único da aresta, representado por uma string.
- **`peso`**: Valor numérico associado à aresta, utilizado para representar custo, distância ou outra métrica.
- **`vertices`**: Uma tupla contendo os dois vértices conectados pela aresta.

6.2. Métodos Principais

6.2.1. `__init__`

O método construtor inicializa os atributos da aresta, configurando o rótulo, peso e os vértices conectados. Por exemplo:

```
aresta = Aresta("A1", 10, vertice1, vertice2)
```

6.2.2. `__str__`

Retorna uma representação textual da aresta, incluindo os vértices conectados, o rótulo e o peso. Ideal para depuração e exibição.

```
print(aresta)
# Saída: "Aresta A1: de V1 para V2 com peso 10"
```

6.2.3. `__repr__`

Método auxiliar que retorna a mesma saída do método `__str__`, garantindo que as representações em listas ou outros contextos sigam o mesmo formato.

6.3. Funções Relacionadas

Além dos métodos internos, há funções auxiliares para manipulação e visualização de arestas:

6.3.1. `listar_arestas`

Essa função exibe todas as arestas disponíveis, juntamente com seus índices. É útil para seleção de arestas em listas maiores.

```
listar_arestas(grafo.arestas)
# Exemplo de saída:
# 0: Aresta A1: de V1 para V2 com peso 10
# 1: Aresta A2: de V2 para V3 com peso 5
```

6.3.2. `alterar_aresta`

Permite modificar o rótulo e o peso de uma aresta selecionada pelo usuário. Caso nenhuma alteração seja necessária, o usuário pode deixar os campos vazios para manter os valores existentes.

```
alterar_aresta(grafo.arestas)
# Fluxo interativo:
# Digite o índice da aresta que deseja alterar: 0
# Novo rótulo para a aresta A1 (deixe vazio para manter): A1.1
# Novo peso para a aresta 10 (deixe vazio para manter): 15
# Aresta atualizada: Aresta A1.1: de V1 para V2 com peso 15
```

7. Classe Algoritmo

A classe `Algoritmo` reúne a implementação de algoritmos fundamentais para o processamento e manipulação de grafos, abordando problemas como identificação de componentes fortemente conectados, detecção de pontes, e a busca de caminhos eulerianos. Esta seção descreve os métodos principais implementados e apresenta os tempos de execução para alguns desses algoritmos.

7.1. Métodos Implementados

7.1.1. `kosaraju`

O algoritmo de Kosaraju é usado para identificar componentes fortemente conectados em grafos direcionados. Ele opera em três etapas principais:

1. Realiza uma busca em profundidade (*Depth First Search* - DFS) para calcular os tempos de término de cada vértice.
2. Constrói o grafo transposto, invertendo as direções das arestas.
3. Executa uma nova DFS no grafo transposto, na ordem decrescente dos tempos de término.

Exemplo de uso:

```
componentes = kosaraju(grafo)
for i, componente in enumerate(componentes):
    print(f"Componente {i+1}: {[v.rotulo for v in componente]}")
```

7.1.2. `fleury`

O algoritmo de Fleury encontra um caminho ou ciclo euleriano em grafos que atendem às condições necessárias (conectividade e, no máximo, dois vértices de grau ímpar). Ele utiliza as seguintes etapas:

- Verifica se o grafo é conexo.
- Seleciona um vértice inicial, preferencialmente um de grau ímpar.
- Percorre as arestas, evitando remover pontes sempre que possível.

Exemplo de uso:

```
caminho = fleury(grafo)
print(f"Caminho Euleriano: {caminho}")
```

7.1.3. `e_ponte`

Este método verifica se uma aresta é uma ponte, ou seja, se sua remoção desconecta o grafo. Ele realiza a remoção temporária da aresta, verifica a conectividade do grafo e a restaura ao final.

Exemplo de uso:

```
is_ponte = e_ponte(grafo, aresta)
print(f"Aresta {aresta.rotulo} é uma ponte: {is_ponte}")
```

7.1.4. `encontrar_pontes_tarjan`

Este método implementa o algoritmo de Tarjan para detectar pontes em grafos. Ele utiliza uma busca em profundidade (DFS) para calcular identificadores e valores baixos (*low*) para cada vértice, determinando as pontes de forma eficiente.

Exemplo de uso:

```
pontes = encontrar_pontes_tarjan(grafo)
print(f"Pontes encontradas: {pontes}")
```

7.1.5. `fleury_com_tarjan`

Este método combina o algoritmo de Fleury com o de Tarjan para melhorar a escolha de arestas ao construir um caminho euleriano. Ele detecta pontes utilizando Tarjan e evita removê-las, sempre que possível.

Exemplo de uso:

```
caminho = fleury_com_tarjan(grafo)
print(f"Caminho Euleriano com Tarjan: {caminho}")
```

7.2. Desempenho dos Algoritmos

Os algoritmos `fleury` e `fleury_com_tarjan` foram avaliados em grafos com diferentes números de vértices. As tabelas abaixo apresentam os tempos de execução observados.

7.3. Conclusão dos Resultados de Desempenho

Os resultados apresentados nas Tabelas 1 e 2 evidenciam diferenças significativas no desempenho dos algoritmos `fleury` e `fleury_com_tarjan` conforme o número de vértices no grafo aumenta.

O algoritmo `fleury`, embora eficiente para grafos menores, apresenta um crescimento exponencial no tempo de execução à medida que o número de vértices aumenta. Isso se deve à necessidade de verificação de conectividade a cada remoção de aresta, o que torna o método computacionalmente caro em grafos maiores.

Por outro lado, o algoritmo `fleury_com_tarjan` demonstra uma melhoria significativa no desempenho devido à utilização do algoritmo de Tarjan para detecção de pontes, reduzindo drasticamente o custo computacional. Esse comportamento é evidente nos tempos reportados, especialmente em grafos com maior número de vértices, onde a diferença de desempenho entre os dois algoritmos se torna ainda mais acentuada.

Em resumo, enquanto o algoritmo `fleury` é adequado para grafos menores, o `fleury_com_tarjan` se destaca como a escolha ideal para aplicações que envolvem grafos maiores, onde a eficiência é um fator crítico.

Table 1. Tempo de execução do algoritmo de Fleury com base no número de vértices

Número de Vértices	Tempo de Execução (segundos)
100	0.0287
1000	8.8765
10000	3465455.1563
100000	4183688469.4756

Table 2. Tempo de execução do algoritmo de Fleury com Tarjan com base no número de vértices

Número de Vértices	Tempo de Execução (segundos)
100	0.0061
1000	0.4554
10000	62.1044
100000	8084.4882