Trabalho Pratico 1 - TCG

Arthur Henrique Tristão Pinto [1], **Luis Henrique Ferreira Costa** [2], **Samuel Correia Pedrosa** [3]

[1]Pontifícia Universidade Católica de Minas Gerais
R. da Bahia, 2020 - Lourdes, Belo Horizonte - MG, 30160-012

Abstract. *This work addresses the problem of identifying bridges in simple undirected graphs, which are edges whose removal disconnects the graph. Two distinct algorithms were implemented and analyzed for this purpose: a naïve method, which checks the graph's connectivity after removing each edge individually, and Tarjan's algorithm. Furthermore, both bridge-finding strategies were integrated into Fleury's algorithm to find Eulerian paths. The performance of these approaches was evaluated through computational experiments, measuring the average execution time on random graphs of varying orders (100, 1,000, 10,000, and 100,000 vertices) and classifications (Eulerian, semi-Eulerian, and non-Eulerian), allowing for a comparative analysis of the efficiency between the strategies.*

Resumo. *Este artigo descreve o problema da identificação de pontes em grafos simples não-direcionados, arestas cuja remoção desconecta o grafo. Foram implementados e analisados dois algoritmos distintos para esta finalidade: um método naïve, que verifica a conectividade do grafo após a remoção de cada aresta individualmente, e o algoritmo de Tarjan. Adicionalmente, as duas estratégias de detecção de pontes foram integradas ao algoritmo de Fleury para a busca de caminhos eulerianos. A avaliação de desempenho das abordagens foi realizada por meio de experimentos computacionais, medindo o tempo médio de execução em grafos aleatórios de diferentes ordens (100, 1.000, 10.000 e 100.000 vértices) e classificações (eulerianos, semi-eulerianos e não eulerianos), permitindo uma análise comparativa da eficiência entre as estratégias.*

1. Contextualização

Algoritmos baseados em grafos são ferramentas fundamentais em diversas áreas da computação, auxiliando na resolução de inúmeros problemas complexos. Este trabalho se concentra em um problema específico da teoria dos grafos: a identificação de pontes em um grafo simples e não-direcionado.

1.1. Estrutura do relatorio

Este artigo está organizado em cinco seções principais. A primeira seção introduz o problema da identificação de pontes em grafos, estabelecendo os objetivos do trabalho. A segunda seção apresenta as definições formais de grafos, pontes e tipos de grafos(eulerianos,semi-eulerianos e não-eulerianos). A terceira seção detalha a metodologia empregada, incluindo o ambiente experimental, a geração dos grafos aleatórios e a implementação dos algoritmos propostos. A quarta seção analisa os resultados obtidos nos testes de desempenho. Finalmente, a quinta seção apresenta a conclusão do trabalho, sintetizando os resultados e as observações.

2. Definições Formais

2.1. Grafos

Um grafo simples não-direcionado, denotado por G, é formalmente definido como um par $G = (V, E)$. Nesse par:

- V representa o conjunto de vértices.
- E representa o conjunto de arestas.

2.2. Pontes

G=(V,E). Uma ponte é definida como uma aresta cuja remoção torna o grafo desconexo, ou seja, aumenta o número de componentes conectados. A capacidade de determinar as pontes existentes em um grafo é crucial para diversas aplicações, como na busca por caminhos e ciclos eulerianos.

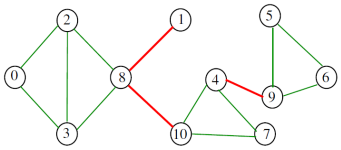

Figura 1. Na figura, as arestas vermelhas são exemplos de pontes.

2.3. Grafo Euleriano

Um grafo G é dito **euleriano** se ele possui um **ciclo euleriano** (ou **circuito euleriano**). Um ciclo euleriano é um caminho fechado que visita cada aresta do grafo exatamente uma vez, começando e terminando no mesmo vértice.

Condição formal: Um grafo conexo $G = (V, E)$ é euleriano se, e somente se, todo vértice $v \in V$ possui **grau par**.

- Matematicamente: $\forall v \in V, \deg(v) \equiv 0 \pmod 2$.

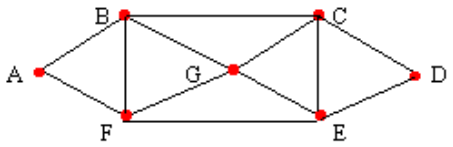

Figura 2. Exemplo de um grafo euleriano

2.4. Grafo Semi-Euleriano

Um grafo G é dito **semi-euleriano** se ele não possui um circuito euleriano, mas possui um **caminho euleriano**. Um caminho euleriano é um caminho aberto que visita cada aresta do grafo exatamente uma vez, começando e terminando em vértices diferentes.

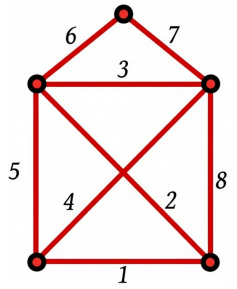

Figura 3. Exemplo de um grafo Semi-Euleriano

2.5. Grafo Não-Euleriano

Um grafo G é dito **não-euleriano** se ele não possui um caminho euleriano.

Condição formal: Um grafo conexo $G = (V, E)$ é não-euleriano se, e somente se, ele possui **mais de dois vértices de grau ímpar**.

3. Metodologia

3.1. Ambiente e Ferramentas

Para a implementação dos algoritmos e a condução dos experimentos, o ambiente de desenvolvimento foi composto pelo Java Development Kit (JDK) para compilação e execução do código-fonte. O desenvolvimento e a edição dos códigos foram realizados no editor Visual Studio Code, enquanto o versionamento e a colaboração no projeto foram gerenciados através da plataforma GitHub.

3.2. Geração dos Grafos

Para a avaliação experimental, foi desenvolvido um gerador de grafos em *Java* capaz de produzir instâncias aleatórias e conectadas, classificadas como **eulerianas**, **semi-eulerianas** ou **não-eulerianas**. O código do gerador baseia-se em três etapas principais:

1. **Construção de um grafo base conexo.**
   Inicialmente é criado um *grafo base*, garantindo que o grafo seja conectado e que todos os vértices possam ser alcançados em uma busca. Esse grafo contém $(N - 1)$ arestas. Em seguida, são adicionadas arestas aleatórias até que o número total atinja o valor desejado M, evitando duplicatas e laços.

2. **Cálculo dos graus dos vértices.**
   Após gerar a estrutura base, é calculado o grau de cada vértice. A partir dessa informação, é possível identificar os vértices de grau ímpar e ajustar o grafo conforme o tipo desejado:
   - **Euleriano:** todos os vértices devem ter grau par. Para isso, os vértices de grau ímpar são conectados em pares, tornando seus graus pares.
   - **Semi-Euleriano:** exatamente dois vértices devem possuir grau ímpar. O gerador verifica a quantidade de ímpares e adiciona ou remove arestas de forma a manter apenas dois vértices ímpares.
   - **Não-Euleriano:** o grafo deve possuir um número de vértices ímpares diferente de zero e de dois (no mínimo quatro ímpares). Quando necessário, o algoritmo força a criação de dois pares adicionais de vértices ímpares conectando vértices de grau par.
3. **Escrita em arquivo.**
   Por fim, o grafo é gravado em um arquivo texto contendo na primeira linha o número de vértices N, e nas linhas seguintes as arestas (u, v). O formato é compatível com os programas implementados para execução dos algoritmos de Fleury.

3.3. Implementação dos Algoritmos de Identificação de Pontes

A estratégia de busca para todas as implementações desenvolvidas neste artigo foi a busca em profundidade (DFS). A escolha desta abordagem foi diretamente inspirada pelo artigo de [Tarjan 1972]

3.3.1. Método Naive

1. **Teste de Conectividade (`buscaEmProfundidadeIterativaNaive`):**
   - O método central para verificar a conectividade é uma busca em profundidade (DFS) iterativa.
   - A função percorre um componente do grafo e retorna o número total de vértices que conseguiu alcançar. Este valor é usado como métrica para determinar se o grafo permaneceu conectado.
2. **Algoritmo Principal (`pontesNaive`):**
   Esta função implementa o fluxo da abordagem Naïve, testando cada aresta.
   (a) **Linha de Base:** Primeiro, uma DFS é executada no grafo original para contar o número de vértices alcançáveis (`verticesEncontraveis`), que serve como referência.
   (b) **Iteração sobre Arestas:** O algoritmo itera sobre uma lista de arestas únicas do grafo.
   (c) **Processo "Remover-Testar-Restaurar":** Para cada aresta `(u, v)`, o ciclo é executado:
      - **Remover:** A aresta é removida do grafo.
      - **Testar:** Uma nova DFS conta os vértices alcançáveis (`alcancados`).
      - **Comparar:** Se `alcancados < verticesEncontraveis`, a aresta é uma ponte.
      - **Restaurar:** A aresta é adicionada de volta ao grafo.

Ao final, a função retorna a lista de todas as pontes encontradas.

3. **Fluxo de Execução (`main`):**
    - O método `main` gerencia a execução: lê o grafo, invoca a função `pontesNaive` e imprime os resultados.

3.3.2. Metodo Tarjan

Metodo feito baseado no artigo de [Tarjan 1974]

1. **Estruturas de Dados e Inicialização:**
    - **Grafo:** Representado por uma lista de adjacências (`ArrayList<Integer>[]`).
    - **Tabela de Busca (`TabelaBusca`):** Classe que armazena os metadados da busca para cada vértice, contendo `TD` (Tempo de Descoberta), `TT` (Tempo de Término), `pai` e `low` (*low-link*).
2. **Algoritmo Principal (`buscaEmProfundidadeIterativaTarjan`):**
    - **Travessia DFS Iterativa:** A função inicia uma busca em profundidade a partir de um vértice `vInicial`, controlada por uma `Stack` e iteradores.
        - **Aresta de Árvore:** Ao encontrar um vértice não visitado, seus valores `TD` e `low` são definidos e ele é empilhado.
        - **Aresta de Retorno:** Se uma aresta leva a um vértice já visitado (que não seja o pai), o valor `low` do vértice atual é atualizado com o `TD` do vizinho.
        - **Propagação do `low`:** Ao finalizar a análise de um vértice, seu valor `low` é propagado para seu pai na árvore DFS.
    - **Verificação da Ponte:** Após a conclusão da DFS, a função utiliza os valores calculados para verificar se a aresta `(v1, v2)`, fornecida como parâmetro, satisfaz a condição de ponte de Tarjan: `tabela[v2].low > tabela[v1].TD` (assumindo que `v1` é o pai de `v2`).
3. **Fluxo de Execução (`main`):**
    - O método `main` lê o grafo, inicializa as estruturas e invoca a função de busca para analisar uma aresta específica.

3.4. Implementação de Fleury

3.4.1. Fleury com Naive

1. **Verificação da Pré-condição Euleriana:**
    - O algoritmo valida se o grafo possui no máximo dois vértices de grau ímpar, condição necessária para a existência de um caminho euleriano.
2. **Inicialização:**
    - Uma cópia do grafo original é criada para permitir a remoção de arestas sem alterar a entrada.
    - Um vértice inicial apropriado é selecionado (um de grau ímpar, se houver) e o caminho é iniciado.
3. **Construção Iterativa do Caminho:**

- Um laço é executado enquanto houver arestas. A escolha da próxima aresta é governada pela regra de Fleury, com a verificação de pontes realizada pelo método `ehPonteNaive`:
  - **Método de Verificação:** Para cada aresta candidata, a função `ehPonteNaive` a remove temporariamente e executa uma busca em profundidade (DFS) para contar o número de vértices alcançáveis. Se essa contagem for menor que a do grafo com a aresta, ela é uma ponte.
  - **Seleção:** O algoritmo escolhe a primeira aresta que, segundo o teste, **não** é uma ponte. Se todas as opções forem pontes, a primeira da lista é selecionada.

4. **Atualização e Conclusão:**
   - A aresta escolhida é permanentemente removida da cópia do grafo, o novo vértice é adicionado ao caminho e a posição atual é atualizada. O processo se repete até que todas as arestas sejam consumidas.

3.4.2. Fleury com Tarjan

1. **Verificação da Pré-condição Euleriana:**
   - O algoritmo primeiramente verifica se o grafo possui no máximo dois vértices de grau ímpar. Caso contrário, um caminho euleriano é impossível e a execução é interrompida.
2. **Inicialização:**
   - Uma cópia do grafo original (`G_prime`) é criada para que as operações de remoção de arestas não afetem a estrutura original.
   - Um vértice inicial é selecionado, priorizando um de grau ímpar, se existente. O caminho resultante é inicializado com este vértice.
3. **Construção Iterativa do Caminho:**
   - Um laço é executado enquanto houver arestas no grafo-cópia. A cada passo, a partir do vértice atual `v_atual`, a próxima aresta é escolhida seguindo a regra de Fleury:
     - **Caso 1 (Aresta Única):** Se há apenas uma aresta disponível, ela é selecionada.
     - **Caso 2 (Múltiplas Arestas):** O algoritmo testa cada aresta candidata com a função `buscaEmProfundidadeIterativaTarjan`. A primeira aresta que **não** for uma ponte é escolhida. Se todas forem pontes, a primeira da lista é selecionada como último recurso.
4. **Atualização e Conclusão:**
   - Após a escolha, a aresta é removida do grafo-cópia, o novo vértice é adicionado ao caminho e a posição atual é atualizada.
   - O processo se repete até que todas as arestas sejam percorridas. Ao final, a função retorna a lista de vértices que compõem o caminho euleriano.

4. Testes

4.1. Estratégia de Avaliação

Nesta seção são apresentados os resultados experimentais obtidos na execução dos algoritmos *Fleury_Naive* e *Fleury_Tarjan*. Os testes foram realizados em grafos com diferentes números de vértices ($N = 100, 1\,000, 10\,000$ e $100\,000$), cada um tendo aproximadamente em média N*1.5 número de arestas com o objetivo de comparar o tempo de execução entre as duas abordagens.

Para cada valor de N, foram gerados e testados **12 grafos distintos**: **4 eulerianos**, **4 não-eulerianos** e **4 semi-eulerianos**. Os tempos médios e os desvios padrão apresentados a seguir correspondem à média dos resultados obtidos nesses 12 grafos, medidos em milissegundos.

4.2. Resultados Quantitativos

Tabela 1. Tempo médio e desvio padrão (em milissegundos) para os algoritmos *Fleury_Naive* e *Fleury_Tarjan*.

Algoritmo	Vértices	Média (ms)	Desv. Padrão (ms)
Fleury_Naive	100	10.50	9.58
Fleury_Naive	1 000	108.08	82.3
Fleury_Naive	10 000	8 008.17	5 873.81
Fleury_Naive	100 000	1 400 803.83	1 051 130.70
Fleury_Tarjan	100	9.58	8.62
Fleury_Tarjan	1 000	80.25	68.2
Fleury_Tarjan	10 000	3 923.42	3 043.51
Fleury_Tarjan	100 000	770 652.58	596 471.87

4.3. Visualização dos Resultados

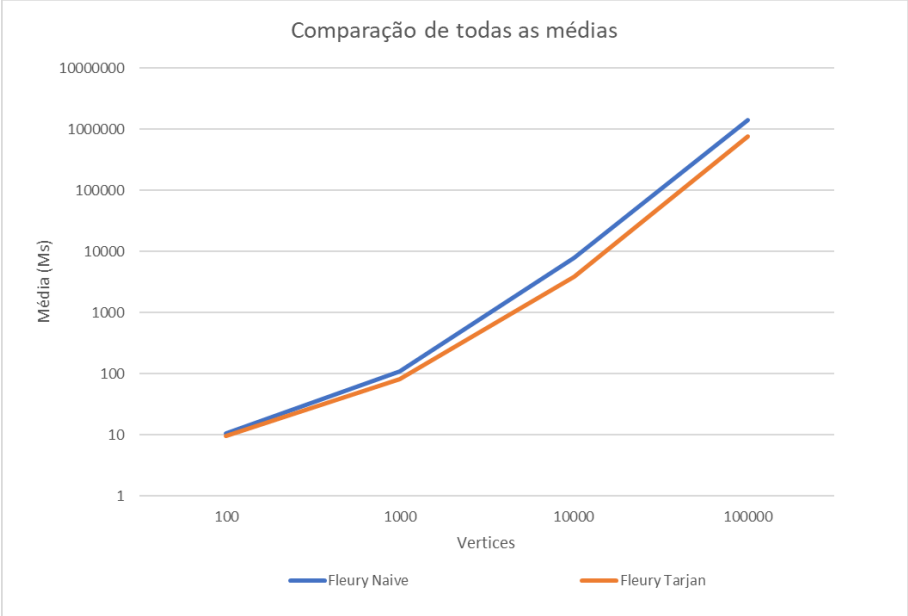

Figura 4. Crescimento do tempo médio de execução (ms) conforme o número de vértices.

4.3.1. Comparação para N = 100

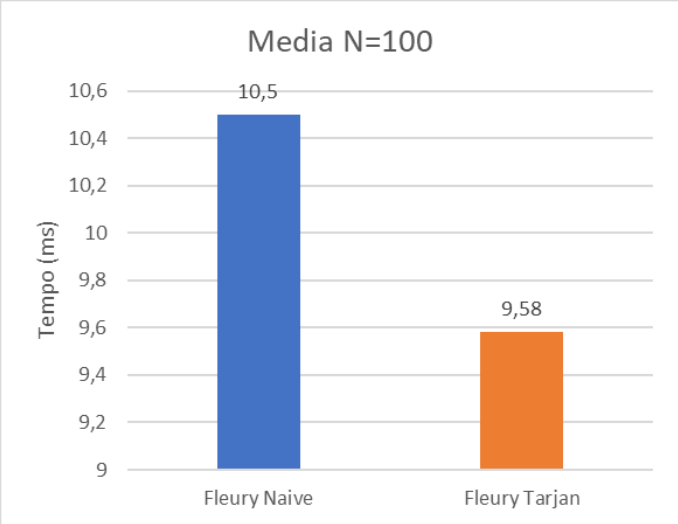

Figura 5. Comparação da média de tempo entre os algoritmos para $N = 100$.

4.3.2. Comparação para N = 1 000

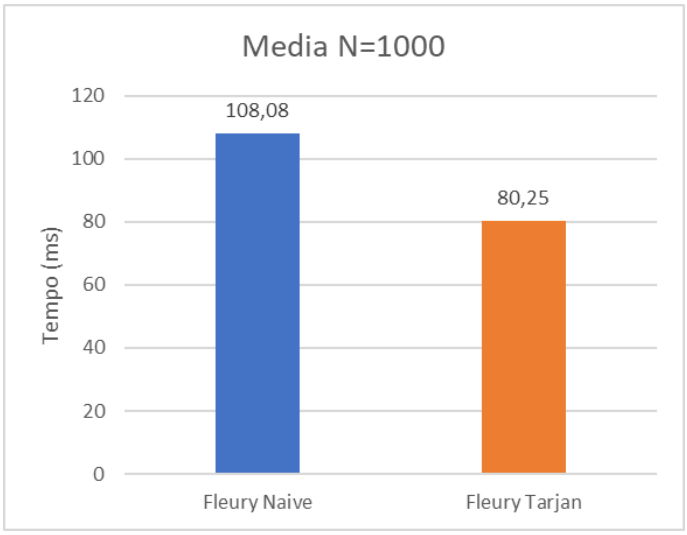

Figura 6. Comparação da média de tempo entre os algoritmos para $N = 1\,000$.

4.3.3. Comparação para N = 10 000

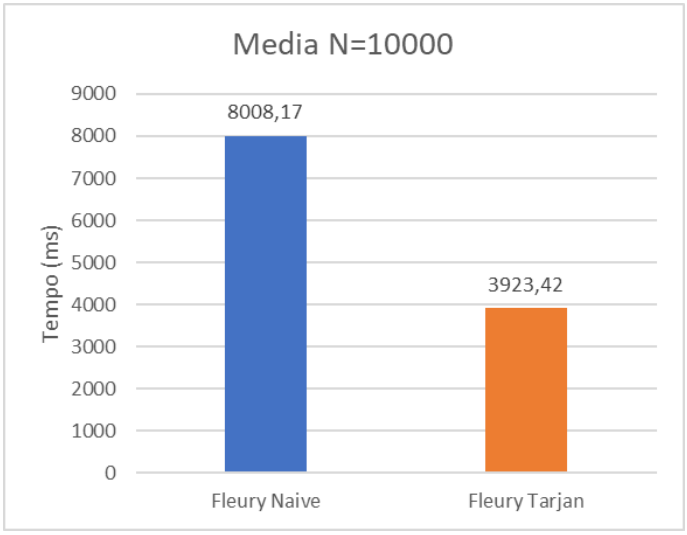

Figura 7. Comparação da média de tempo entre os algoritmos para $N = 10\,000$.

4.3.4. Comparação para N = 100 000

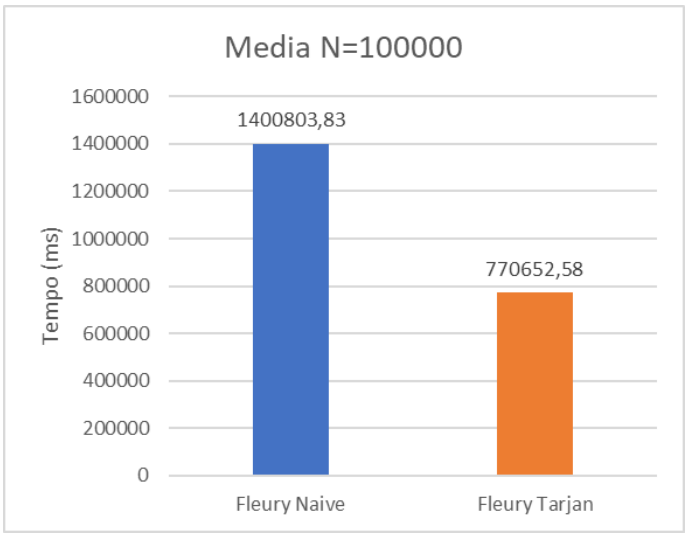

Figura 8. Comparação da média de tempo entre os algoritmos para $N = 100\,000$.

5. Discussão de Resultados

Após os testes realizados, observa-se que o algoritmo **Fleury_Tarjan** apresenta desempenho consistentemente superior ao **Fleury_Naive**, especialmente à medida que o número de vértices dos grafos aumenta.

Para grafos pequenos (100 vértices), a implementação **Fleury_Tarjan** (9,58 ms) já se mostra ligeiramente mais eficiente que a **Fleury_Naive** (10,50 ms). No entanto, à medida que o número de vértices cresce, a vantagem do algoritmo otimizado com Tarjan torna-se significativamente mais evidente. Para 10 000 vértices, a versão com Tarjan (3 923 ms) é cerca de duas vezes mais rápida que a Naive (8 008 ms). O cenário mais expressivo ocorre com 100 000 vértices, em que o **Fleury_Tarjan** (770 652 ms) executa em aproximadamente metade do tempo do **Fleury_Naive** (1 400 804 ms), confirmando sua melhor escalabilidade.

Os altos desvios-padrão observados resultam das diferenças estruturais entre os grafos *eulerianos*, *semi-eulerianos* e *não eulerianos*, que demandam quantidades distintas de verificações e travessias. Essa variação reflete a diversidade de complexidade das instâncias testadas, e não uma inconsistência dos algoritmos.

Referências

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160.

Tarjan, R. E. (1974). A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161.