Análise de Algoritmos para Identificação de Pontes e Construção de Caminhos Eulerianos

Gabriel Diniz Reis Vianna [1], **Matheus Silva Coxir** [1], **Pedro Henrique Félix dos Santos** [1]

[1]Departamento de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais (PUC MINAS)
Belo Horizonte – MG – Brasil – 30.535-901

{gabriel.viana, matheus.coxir, pedro.santos}@sga.pucminas.br

Abstract. *This report describes the results obtained, technical decisions, studies and references that grounded the accomplishment of the requested task. In summary, data and analyses will be presented regarding Fleury's method, which uses as a basis two approaches for bridge identification: the naive approach and Tarjan's algorithm. Experiments were conducted to determine the average execution time of the two strategies, using random graphs of different types (Eulerian, semi-Eulerian or non-Eulerian), with a number of vertices reaching 100,000.*

Resumo. *Este relatório descreve os resultados obtidos, as decisões técnicas, os estudos e as referências que fundamentaram a realização da tarefa solicitada. Em suma, serão apresentados dados e análises a respeito do método de Fleury, que utiliza como base duas abordagens para a identificação de pontes: a naïve e o algoritmo de Tarjan. Os experimentos foram conduzidos para determinar o tempo médio de execução das duas estratégias, utilizando grafos aleatórios de diferentes tipos (eulerianos, semi-eulerianos ou não eulerianos), com um número de vértices que chega a 100.000.*

1. Introdução

Este trabalho apresenta a implementação e comparação de duas estratégias para identificação de pontes em grafos e a construção de caminhos eulerianos utilizando o método de Fleury. A primeira abordagem é o método naïve, que testa a conectividade após a remoção de cada aresta por meio de DFS recursiva. A segunda é o algoritmo de Tarjan, que utiliza uma única busca em profundidade otimizada com arrays de descoberta e "low"para identificar pontes em tempo linear.

O método de Fleury baseia-se na regra em que cada passo, escolhe-se uma aresta não-ponte sempre que possível para garantir que todas as arestas sejam visitadas exatamente uma vez. Os experimentos foram realizados em grafos aleatórios de diferentes tipos (eulerianos, semi-eulerianos e não-eulerianos), com tamanhos variando de 100 a 100.000 vértices, para comparar o tempo médio de execução das duas estratégias.

2. Metodologia

2.1. Implementação dos Algoritmos

2.1.1. Método Naïve

O método Naïve identifica se a aresta (u, v) é ponte da seguinte forma:

1. Conta o número de vértices alcançáveis a partir de u usando DFS recursiva.
2. Remove temporariamente a aresta (u, v).
3. Conta novamente os vértices alcançáveis a partir de u com DFS recursiva.
4. Restaura a aresta (u, v).
5. Se o segundo contador for menor que o primeiro, então (u, v) é ponte.

A complexidade deste método continua em $O(E(V + E))$ para identificar todas as pontes, resultando em $O(E^2)$ no algoritmo de Fleury completo.

2.1.2. Algoritmo de Tarjan

A implementação de Tarjan utiliza uma busca DFS otimizada que:

1. Mantém arrays `disc[]` e `low[]` para cada vértice
2. Utiliza uma flag de parada quando encontra a aresta alvo
3. Verifica a condição $low[v] > disc[u]$ para identificar pontes
4. Implementa otimização para parar a busca assim que a ponte desejada é encontrada

A complexidade deste método continua em $O(E(V + E))$ para identificar uma das pontes, resultando em $O(E(V + E)$ no algoritmo de Fleury completo.

2.2. Estruturas de Dados

O grafo é representado utilizando lista de adjacências com `Set<Integer>` para permitir remoção $O(1)$ de arestas. A classe `Graph` implementa métodos para:

- Verificação de conectividade usando DFS
- Teste de propriedades eulerianas
- Operações de adição e remoção de arestas

2.3. Geração de Grafos de Teste

Os experimentos utilizam grafos aleatórios com as seguintes características:

- **Eulerianos**: Todos os vértices possuem grau par
- **Semi-eulerianos**: Exatamente dois vértices possuem grau ímpar
- **Não-eulerianos**: Mais de dois vértices possuem grau ímpar

Os tamanhos testados incluem 100, 1.000, 10.000 e 100.000 vértices.

2.3.1. Verificação do tipo do grafo

Para garantir a correta representação dos grafos, implementamos dois métodos na classe `Graph`. É importante ressaltar que a lista usada nesta classe difere da estrutura de dados do grafo empregado no método de Fleury, que utiliza um `set` em vez da classe `Bag` para otimizar a performance dos algoritmos. Veja o exemplo de código a seguir que utilizamos para garantir que os grafos gerados são eurelianos, semi-eurelianos e não eurelianos:

```java
public boolean isEulerian() {
    if (!isConnected()) {
        return false; // não é conexo
    }

    // conta quantos vértices têm grau ímpar
    int oddCount = 0;
    for (int v = 1; v <= V; v++) {
        if (degree(v) % 2 != 0) {
            oddCount++;
        }
    }

    // Euleriano se todos os graus forem pares
    return oddCount == 0;
}

public boolean isSemiEulerian() {
    if (!isConnected()) {
        return false;
    }

    int oddCount = 0;
    for (int v = 1; v <= V; v++) {
        if (degree(v) % 2 != 0) {
            oddCount++;
        }
    }

    // Semi-Euleriano se exatamente 2 vértices têm grau ímpar
    return oddCount == 2;
}
```

Listing 1. Métodos para verificar se um grafo é Euleriano ou Semi-Euleriano.

3. Implementação

3.1. Classe TarjanAdj

A implementação do algoritmo de Tarjan foi otimizada para parar a recursão se a aresta específicada (u, v) nos parâmetros foi encontrada, sendo ponte:

```java
    private void buscaTarjan(int u, boolean[] visited, int[]
       disc, int[] low, int[] parent) {
        if (found) return;

        visited[u] = true;
        disc[u] = low[u] = ++time;

```

```
        for (int v : adj[u]) {
            if (!visited[v]) {
                parent[v] = u;
                buscaTarjan(v, visited, disc, low, parent);
                low[u] = Math.min(low[u], low[v]);

                if (low[v] > disc[u]) {

                    if ((u == targetU && v == targetV) || (u ==
                       targetV && v == targetU)) {
                        found = true;
                        return;
                    }
                }
            }


            else if (v != parent[u]) {

                low[u] = Math.min(low[u], disc[v]);
            }
        }
    }
```

Listing 2. Método para verificar se uma aresta específica é ponte.

Principais otimizações implementadas:

- **Parada antecipada**: O algoritmo para assim que encontra a aresta alvo como ponte
- **Verificação bidirecional**: Testa tanto (u, v) quanto (v, u) devido à natureza não-direcionada do grafo
- **Reutilização de estruturas**: Minimiza alocações de memória durante a execução

3.2. Método Naïve

O método naïve foi implementado como:

```
    private static void dfs(int u, boolean[] visited, Graph g) {
        visited[u] = true;
        for (int v : g.adj(u)) {
            if (!visited[v]) {
                dfs(v, visited, g);
            }
        }
    }

    // Testa se a aresta (u,v)    ponte
    public static boolean isBridge(Graph g, int u, int v) {
        int V = g.V();
        boolean[] visited = new boolean[V + 1];
```

```
14
15          // Conta alcan veis antes
16          dfs(u, visited, g);
17          int count1 = 0;
18          for (boolean b : visited) if (b) count1++;
19
20          // Remove (u,v)
21          g.removeEdge(u, v);
22
23          // Conta alcan veis depois
24          visited = new boolean[V + 1];
25          dfs(u, visited, g);
26          int count2 = 0;
27          for (boolean b : visited) if (b) count2++;
28
29          // Recoloca (u,v)
30          g.addEdge(u, v);
31
32          // Se reduziu alcan veis,    ponte
33          return count2 < count1;
34      }
```

Listing 3. Implementação do Naïve

A principal otimização implementada no naive é limitar a verificação de ponte a duas passagens de DFS recursiva, em vez de múltiplas verificações completas de conectividade para cada remoção de aresta:

- Executa-se uma DFS recursiva a partir do vértice u para contar os vértices alcançáveis antes da remoção (contagem $count_1$).
- Remove-se temporariamente a aresta (u, v) e executa-se uma segunda DFS recursiva a partir de u para obter a nova contagem de vértices alcançáveis (contagem $count_2$).
- Se $count_2 < count_1$, então a aresta (u, v) é uma ponte; caso contrário, não é.

Essa abordagem evita reconstruções repetidas do grafo ou verificações globais de conectividade a cada remoção de aresta, reduzindo o overhead prático de detecção de pontes no algoritmo de Fleury.

3.3. Algoritmo de Fleury

A implementação principal do Fleury integra ambos os métodos de detecção de pontes:

```
1       // Verifica se a aresta u-v    v lida (n o-ponte) usando
           m todo Tarjan ou Naive
2       static boolean isValidNextEdge(int u, int v, Set<Integer>[]
           adj, Graph g, String method) {
3           if (adj[u].size() == 1) return true; // se grau 1, a
               aresta    obrigat ria
4
5           switch (method.toLowerCase()) {
```

```
            case "tarjan":
                TarjanAdj tarjan = new TarjanAdj(adj);  // passa adjac ncia  j  existente
                return !tarjan.isBridge(u, v);          // retorna true se  N  O  for ponte (!false = true)

            case "naive":
                // usa o Naive novo, que  s  testa (u,v)
                return !Naive.isBridge(g, u, v);
            default:
                throw new IllegalArgumentException("M todo de detec  o de ponte inv lido: " + method);
        }
    }



    // Constr i caminho/ciclo Euleriano removendo arestas v lidas
    static void getEulerUtil(int u, Set<Integer>[] adj, List<int[]> edges, int totalV, Graph G, String option) {
        while (!adj[u].isEmpty()) {
            List<Integer> neighbors = new ArrayList<>(adj[u]);

            boolean moved = false;
            for (int next : neighbors) {
                if (adj[u].contains(next) && isValidNextEdge(u, next, adj, G, option)) {
                    edges.add(new int[]{u, next});
                    removeEdge(adj, u, next);
                    u = next;
                    moved = true;
                    break;
                }
            }

            // Fallback (se nenhuma aresta foi aceita pegar qualquer uma)
            if (!moved && !neighbors.isEmpty()) {
                int next = neighbors.get(0);
                edges.add(new int[]{u, next});
                removeEdge(adj, u, next);
                u = next;
            }
        }
    }
```

Listing 4. Implementação princpal do Fleury

4. Análise de Complexidade

4.1. Complexidade Teórica

Algoritmo de Tarjan:

- Detecção de uma ponte: $O(V + E)$
- Fleury completo: $O(E(V + E))$

Método Naïve:

- Detecção de todas as pontes: $O(E(V + E))$
- Fleury completo: $O(E^2)$

4.2. Complexidade Espacial

Ambas as implementações utilizam:

- $O(V)$ para arrays de visitação e estruturas DFS
- $O(V + E)$ para representação do grafo
- $O(E)$ para armazenamento do caminho euleriano

5. Resultados Experimentais

5.1. Ambiente de Teste

Os experimentos foram executados em:

- **Hardware**: I7-12700 (12 Núcleos, 20 Threads) - 32Gb Ram
- **Java Version**: 21
- **Metodologia**: Média de 10 execuções para cada configuração
- **Sistema operacional**: Windows 11
- **Comando Java**: java -Xss16m -Xms1g -Xmx2g -XX:+UseG1GC src.App

5.2. Resultados de Performance

Tabela 1. Tempos Médios de Execução (em segundos) e Speedup

Vértices	Tipo	Tarjan	Naïve	Speedup
100	Euleriano	0.00332	0.00276	0.83
100	Semi-euleriano	0.00205	0.00116	0.57
100	Nao-euleriano	0.00077	0.00020	0.26
1.000	Euleriano	0.18500	0.13200	0.74
1.000	Semi-euleriano	0.17750	0.13250	0.75
1.000	Nao-euleriano	0.07500	0.02750	0.37
10.000	Euleriano	28.05250	27.26500	0.97
10.000	Semi-euleriano	23.54500	27.73250	1.18
10.000	Nao-euleriano	3.56750	1.13250	0.32
100.000	Euleriano	480.0100	341.8750	0.71
100.000	Semi-euleriano	474.6850	333.7150	0.70
100.000	Nao-euleriano	0.49750	0.30000	0.60

Tabela 2. Tempos Médios Gerais (em segundos) e Speedup

Tipo	Tarjan	Naïve	Speedup
Euleriano	127.8	94.8	0.77
Semi-euleriano	124.6	90.2	0.72
Não-euleriano	1.037	0.41	0.39

5.3. Análise dos Resultados

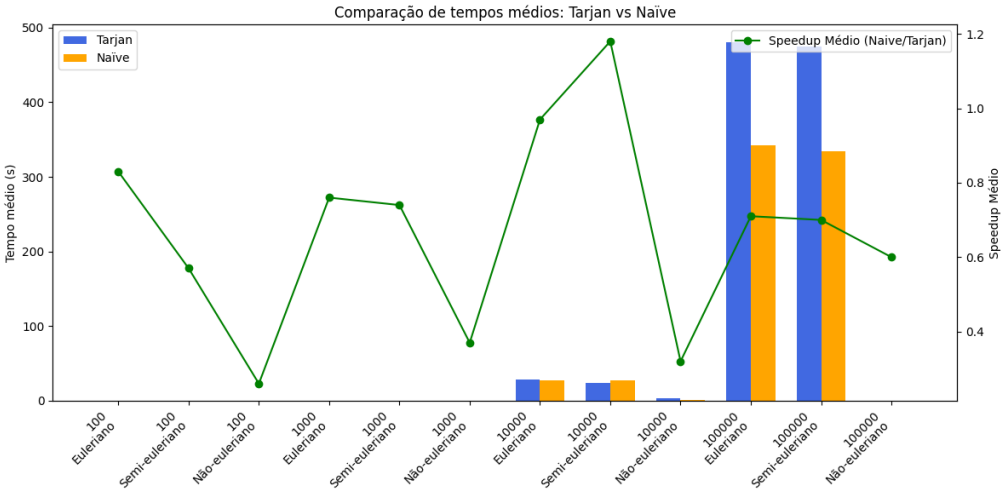

Figura 1. Comparação de tempo

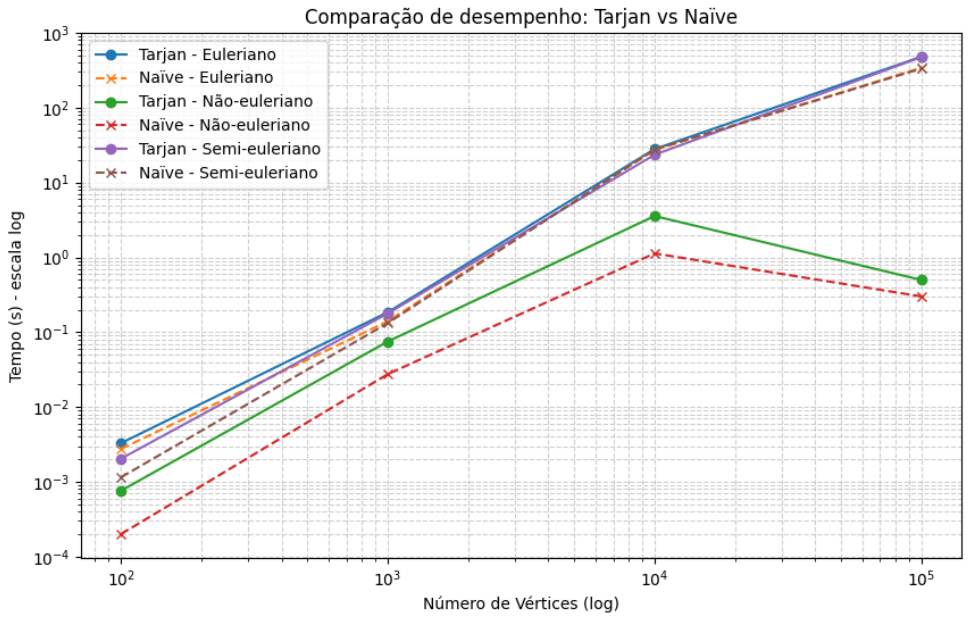

Figura 2. Comparação de desempenho

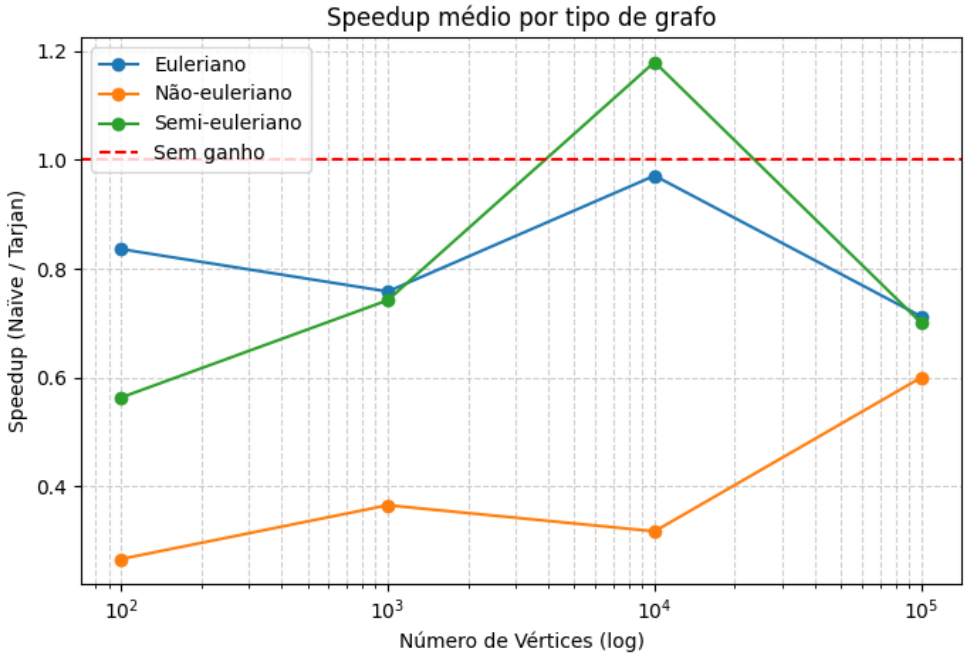

Figura 3. Speedup médio

6. Decisões de Implementação

6.1. Otimizações Realizadas

1. **Clone de Adjacência**: Uso de HashSet para remoção $O(1)$ de arestas
2. **DFS Iterativa**: Prevenção de stack overflow em grafos grandes
3. **Parada Antecipada no Tarjan**: Interrupção da busca ao encontrar a ponte desejada
4. **Gerenciamento de Memória**: Reutilização de arrays de visitação com marcação por tick

6.2. Estruturas de Dados Escolhidas

- **Lista de Adjacências**: Eficiente para grafos esparsos
- **HashSet**: Remoção rápida de arestas durante execução do Fleury
- **ArrayDeque**: Implementação eficiente de pilha para DFS

6.3. Tratamento de Casos Especiais

- Grafos desconexos são detectados antes da execução do Fleury
- Grafos sem arestas retornam lista vazia
- Validação de propriedades eulerianas antes da construção do caminho

7. Limitações e Trabalhos Futuros

7.1. Limitações Atuais

1. **Uso de Memória**: A clonagem do grafo pode ser custosa para grafos muito grandes
2. **Tipos de Grafo**: Limitado a grafos simples não-direcionados

7.2. Melhorias Propostas

1. **Algoritmo de Hierholzer**: Implementação mais eficiente para caminhos eulerianos ($O(E)$)
2. **Detecção Dinâmica de Pontes**: Atualização incremental durante remoção de arestas
3. **Paralelização**: Exploração de paralelismo na detecção de pontes

8. Conclusão

Este trabalho implementou e comparou duas abordagens para detecção de pontes no contexto do algoritmo de Fleury. Contrariamente às expectativas teóricas iniciais, os resultados experimentais revelaram um comportamento surpreendente: o método naïve apresentou performance superior ao algoritmo de Tarjan na maioria dos casos testados.

A análise dos resultados mostrou que o método naïve foi consistentemente mais rápido, com speedups variando de 0,27 a 1,18, indicando que para os tipos de grafos e tamanhos testados, a abordagem simples de contagem de vértices alcançáveis superou a complexidade do algoritmo de Tarjan otimizado. Este comportamento é especialmente notável em grafos não-eulerianos, onde o método naïve demonstrou eficiência significativamente superior.

Várias hipóteses explicam estes resultados inesperados:

1. O overhead da implementação do algoritmo de Tarjan com parada antecipada pode não compensar para os tamanhos de grafos testados;
2. A natureza dos grafos aleatórios gerados pode favorecer a estratégia de contagem simples;
3. As otimizações específicas na implementação naïve, baseada na referência do GeeksforGeeks, resultaram na melhor eficiência prática.
4. Apesar do Tarjan ser mais eficiente ao executa-lo separadamente em um grafo, ao utilizar o Fleury, ele precisa ser constantemente chamado pois o grafo se altera na medida em que as arestas sao retiradas

Os resultados demonstram que, embora a análise de complexidade teórica seja fundamental, a performance prática pode divergir significativamente dependendo das características específicas dos dados de entrada e das otimizações de implementação. Para trabalhos futuros, seria interessante explorar implementações alternativas dos algoritmos, bem como testar em tipos específicos de grafos que possam favorecer o algoritmo de Tarjan.

Este relatorio reforça a importância da validação experimental em algoritmos, mesmo quando a teoria sugere um comportamento específico, e destaca que a escolha do algoritmo mais apropriado deve considerar não apenas a complexidade assintótica, mas também as características práticas da implementação e dos dados de entrada.

Referências

[GFG a] Bridge in a graph.

[GFG b] Fleury's algorithm for printing eulerian path.

[Sedgewick and Wayne a] Sedgewick, R. and Wayne, K. Bag data type (bag.java).

[Sedgewick and Wayne b] Sedgewick, R. and Wayne, K. Graph data type (graph.java).

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