Journal of the Brazilian Computer Society, 202X, XX:1, doi: 10.5753/jbcs.202X.XXXXXX

Algoritmo de Fleury: Uma análise do custo computacional

Gabriel Samarane [Pontifícia Universidade Católica de Minas Gerais | *samarane.gabriel@gmail.com*]
Sophia Carrazza Ventorim de Sousa [Pontifícia Universidade Católica de Minas Gerais | *sophiacarrazza7@gmail.com*]
João Madeira Carneiro Braga de Freitas [Pontifícia Universidade Católica de Minas Gerais | *joaomadeira1208@gmail.com*]
João Lucas Curi [Pontifícia Universidade Católica de Minas Gerais | *jotacuri08@gmail.com*]

Instituto de Ciências Exatas, Pontifícia Universade Católica de Minas Gerais, Av. Gal. Milton Tavares de Souza, s/n, São Domingos, Niterói, RJ, 24210-590, Brazil.

Abstract. O Algoritmo de Fleury encontra caminhos ou ciclos eulerianos em um grafo euleriano. Dessa forma, esse trabalho é focado na identificação e construção explícita de circuitos Eulerianos por meio do algoritmo clássico de Fleury, utilizando e comparando dois métodos para identificar essas arestas: o naive, baseado em testes de conectividade, e o Algoritmo de Tarjan. A análise mostra que a abordagem naive é mais eficiente para grafos grandes, tornando-o a melhor escolha para essa aplicação.

Keywords: Euleriano, Ciclo Euleriano, Complexidade, Fleury, Naive, Tarjan

1 Definindo o Problema

Um caminho euleriano em um grafo é definido como um percurso que utiliza todas as arestas do grafo exatamente uma única vez. Uma extensão natural dessa definição é o ciclo euleriano, isto é, um caminho euleriano que se inicia e termina no mesmo vértice.

No contexto de Teoria dos Grafos, o algoritmo de Fleury é utilizado para encontrar caminhos ou ciclos eulerianos em um grafo G. O funcionamento básico desse algoritmo, Seção 1.2 ,baseia-se na identificação de arestas ponte (bridge edges). Uma aresta é chamada de ponte quando sua remoção aumenta o numero de componentes conexos dentro do grafo. Dessa forma, o algoritmo caminha pelo grafo evitando, sempre que possível, uma ponte, o que garante que o caminho final tera percorrido todas as arestas uma única vez. Consequentemente, surge a discussão sobre qual é o método mais adequado para determinar essas arestas ponte de forma eficiente.

A proposta deste artigo é apresentar duas implementações distintas do algoritmo de Fleury, cuja principal diferença reside na forma de identificar as arestas ponte em cada etapa do procedimento.

1.1 Algoritmo de Fleury

O Algoritmo de Fleury é um método para encontrar um caminho ou ciclo eulerianos em um grafo euleriano. Ele funciona de maneira gananciosa, removendo arestas enquanto percorre o grafo e evitando as pontes enquanto possível, garantindo que não ocorra uma remoção prematura delas e consequentemente torne a solução inválida.

Para que o algoritmo possa ser aplicado, o grafo deve possuir um ciclo euleriano (todos os vértices possuem grau par) ou um caminho euleriano (dois vértices possuem grau ímpar), e deve ser conexo (não pode ter componentes desconectados).

1.2 Pseudo-Código Fleury

Fleury(V, E):

1. Selecione um vértice inicial $u \leftarrow v$.
2. Crie uma lista vazia $lista\_arestas$.
3. Enquanto E não estiver vazio:
   (a) Para cada vértice w vizinho de u:
      i. Se $\deg(u) = 1$ ou not *isPonte*(u, w):
         A. Atualize $u \leftarrow w$.
         B. Adicione (u, w) em $lista\_arestas$.
         C. Remova (u, w) de E.
         D. Interrompa o loop (break).
4. Retorne $lista\_arestas$.

2 Algoritmos para encontrar pontes

2.1 Algoritmo Naive

O algoritmo Naive também pode ser pensado como um algoritmo do tipo ”Força Bruta”, uma vez que ”percorre” todo o espaço de busca disponível no problema para afirmar se a aresta é ou não uma ponte. Ele remove uma aresta temporariamente e verifica se o grafo se torna desconectado. Se sim, essa aresta é uma ponte.

2.1.1 Pseudo-Código Naive

Naive(G, e):

```
1. Remover a aresta e do grafo G.
2. Sejam v = e.v e u = e.u.
3. Calcular r = FechoTransitivoDiretoAlterado(u, v).
4. Retorne r (falso se encontrar,
verdadeiro se não encontrar)
```

A função *FechoTransitivoDiretoAlterado()* encontra todos os vértices acessíveis a partir do vértice u utilizando uma busca em profundidade (DFS) para percorrer o grafo.

Durante o processo de implementação deste pseudo-código, optamos por alterar a DFS dentro do Fecho Transitivo Direto para interromper o algoritmo caso o vértice de destino v seja encontrado antes do fim da execução, otimizando significativamente o desempenho para grafos densos e com muitos vértices.

Para um Grafo G = (V, E) este algoritmo tem complexidade de tempo O(|V| + |E|) (Kaur and Garg [2012]).

2.2 Algoritmo de Tarjan

O Algoritmo de Tarjan é um método eficiente para identificar arestas ponte em um grafo não direcionado. O algoritmo utiliza a busca em profundidade (DFS) para calcular dois valores importantes para cada vértice:

- tempoDescoberta$[v]$: a ordem de descoberta do vértice v na DFS.
- menorTempoAcessivel$[v]$: o menor valor de ordem que v pode alcançar ao seguir arestas de retorno na DFS (low-link value).

Uma aresta (u, v) é classificada como ponte se menorTempoAcessivel$[v]$ $>$ tempoDescoberta$[u]$, pois isso indica que v e seus descendentes na DFS não conseguem alcançar nenhum ancestral acima do seu pai direto (u), e consequentemente não possuem caminho alternativo para alcançar u sem passar por (u, v).

2.2.1 Pseudo-Código Tarjan

Nossa implementação, também feita em C++, estruturada em classes e métodos modulares, segue os mesmos passos citados anteriormente:

1. Inicializamos os vetores auxiliares `visitado`, `tempoDescoberta` e `menorTempoAcessivel` para todos os vértices.
2. Para cada vértice não visitado, executamos uma DFS, atualizando os valores mencionados acima.
3. Após explorar um vizinho u do vértice atual v, verificamos se a aresta (v, u) é uma ponte através da condição:

   $$\text{menorTempoAcessivel}[u] > \text{tempoDescoberta}[v]$$

   Caso essa condição seja verdadeira, adicionamos a aresta (v, u) ao conjunto das pontes encontradas.

Para um grafo G = (V, E) a complexidade temporal deste algoritmo é O(|V| + |E|) (Lavaei and Sojoudi [2009]).

2.3 Estruturas de Dados para Tarjan e Naive

No caso do algoritmo de Tarjan, que produz uma lista de arestas classificadas como pontes, foi necessário utilizar uma tabela de dispersão (*hash table*) para acessar rapidamente essas arestas durante a execução do Fleury-Tarjan. Em contrapartida, o algoritmo Naive retorna apenas um valor booleano indicando se a aresta em análise é, ou não, uma ponte, o que dispensa a necessidade de estruturas adicionais de armazenamento.

2.4 Análise de Complexidade Fleury-Tarjan e Fleury-Naive

Considere um grafo $G = (V, E)$ que seja Euleriano. O objetivo do algoritmo de Fleury é percorrer cada aresta de G exatamente uma vez, construindo assim um circuito (ou caminho) Euleriano. Por esse motivo, o número total de operações é proporcional a $|E|$.

Entretanto, a cada etapa de seleção de uma aresta, o algoritmo deve verificar se essa aresta é uma ponte. Essa verificação faz uso de algum método para detecção de pontes, que neste caso, são os algoritmos de Tarjan ou o algoritmo com abordagem Naive. Em ambos os casos, a complexidade de cada verificação é da ordem de $O(|V| + |E|)$, pois envolve realizar uma busca de conectividade no grafo.

Dado que todas as arestas precisam ser testadas ao longo da execução (no pior caso, verificamos cada uma antes de percorrê-la), temos $|E|$ verificações de custo $O(|V| + |E|)$ cada. Logo, a complexidade de tempo no pior caso para o algoritmo de Fleury é:

$$O\big(|E| \times (|V| + |E|)\big).$$

3 Resultados obtidos

Nesta seção abordamos os tempos de execução de cada algoritmo de maneira comparativa. O teste é realizado para Grafos Eulerianos com, respectivamente, 100, 1000, 10000, 100000 vértices. Os resultados são mostrados abaixo na tabela e no gráfico.

Tamanho da Amostra	Tempo Tarjan (ms)	Tempo Naive (ms)
100	2.540	0.502
1000	179.204	28.131
10000	15341.072	2576.766
100000	1867871.029	306865.897

Table 1. Comparação de tempos: Tarjan vs. Naive.

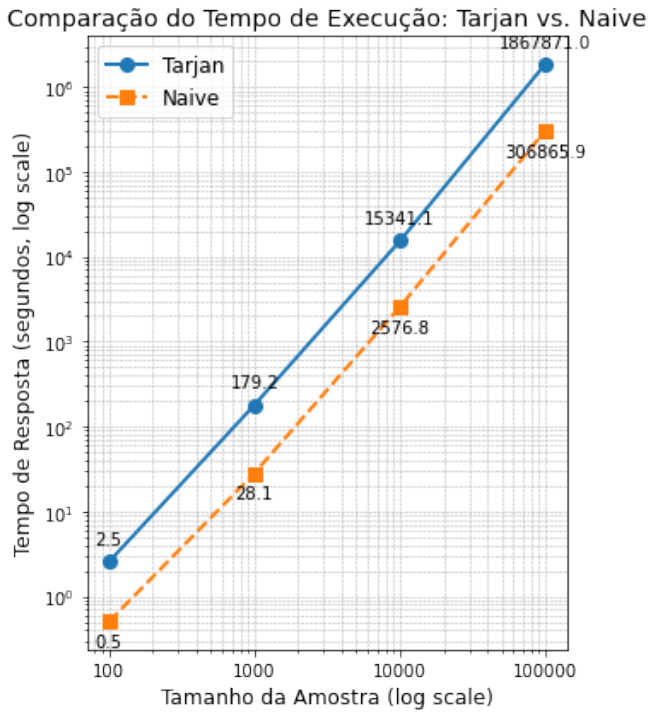

Figure 1. Comparação dos tempos de execução dos algoritmos Fleury-Tarjan e Fleury-Naive.

4 Discussão Comparativa dos Resultados

Os resultados obtidos evidenciam que o algoritmo Fleury-Naive apresenta desempenho superior ao Fleury-Tarjan, especialmente em grafos de maior dimensão. Essa diferença de performance pode ser atribuída à abordagem adotada por cada algoritmo.

O algoritmo de Tarjan é capaz de identificar simultaneamente todas as arestas-ponte existentes no grafo por meio de uma busca em profundidade (DFS). Embora eficaz em contextos onde todas as pontes precisam ser identificadas previamente, essa abordagem implica maior complexidade computacional. Além disso, demanda armazenamento adicional para guardar informações como tempos de descoberta e valores mínimos acessíveis, resultando em operações mais custosas.

Por outro lado, o algoritmo Naive verifica individualmente cada aresta de maneira direta, retornando apenas um valor booleano que indica se uma aresta específica é ponte ou não. Essa simplicidade e objetividade, sem necessidade de armazenamentos intermediários, reduz significativamente o tempo de execução geral. Consequentemente, no contexto específico do algoritmo Fleury, onde apenas a informação pontual sobre a presença ou ausência de pontes é necessária em cada passo, o algoritmo Naive demonstra-se mais eficiente.

5 Responsabilidades

- Implementação dos algoritmos – João Madeira, Sophia Carrazza, Gabriel Samarane
- Desenvolvimento do artigo – Gabriel Samarane, João Madeira, Sophia Carrazza, João Lucas Curi

References

Kaur, N. and Garg, D. (2012). Analysis of the Depth First Search Algorithms. *Data Mining and Knowledge Engineering*, 4(1):37–41. Number: 1.

Lavaei, J. and Sojoudi, S. (2009). Complexity of checking the existence of a stabilizing decentralized controller. In *Proceedings of the 2009 Conference on American Control Conference*, ACC'09, page 1827–1832. IEEE Press.