PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Relatório do Trabalho Prático de Grafos

Autores:
Luigi Louback de Oliveira
Thiago Rezende Aguiar
Thomas Neuenschwander Maciel Baron

Resumo

Neste trabalho, abordamos a identificação de componentes biconexos em grafos não direcionados, conceitos fundamentais na teoria dos grafos com aplicações em áreas como redes de comunicação, análise de redes sociais e algoritmos de roteamento. Implementamos e avaliamos três métodos distintos para a identificação dos blocos de um grafo: (1) verificação da existência de dois caminhos internamente disjuntos entre pares de vértices; (2) identificação de articulações através de testes de conectividade pela remoção sistemática de vértices; e (3) aplicação do algoritmo de Tarjan. Realizamos experimentos em grafos aleatórios de diferentes tamanhos (100, 1.000, 10.000 e 100.000 vértices) para avaliar a eficiência e escalabilidade das abordagens propostas.

1 INTRODUÇÃO

Na teoria dos grafos, os componentes biconexos são definidos como subgrafos maximais de um grafo conexo G, nos quais, para quaisquer dois vértices u e v, existem dois caminhos disjuntos que os conectam. Isso implica que a remoção de qualquer outro vértice, exceto u e v, não desconecta esses vértices. Pontes, por outro lado, são arestas cuja remoção aumenta o número de componentes conexos do grafo, indicando uma fragilidade na conexão entre os vértices. De forma semelhante, articulações são vértices cuja remoção resulta em um aumento do número de componentes conexos, revelando pontos críticos na estrutura do grafo.

Esses conceitos são fundamentais para a análise da conectividade em grafos, uma vez que a identificação de componentes biconexos, pontes e articulações permite uma compreensão mais detalhada da estrutura do grafo e de sua robustez. Os componentes biconexos, em particular, garantem a existência de múltiplos caminhos entre vértices, o que aumenta a resiliência da rede, sendo aplicáveis em diversas áreas.

As aplicações de componentes biconexos incluem redes de comunicação, onde eles são utilizados para garantir a robustez e confiabilidade da infraestrutura, análise de redes sociais, facilitando a compreensão da conectividade entre indivíduos ou grupos, e algoritmos de roteamento, que visam otimizar o tráfego em redes complexas.

Portanto, o objetivo deste trabalho é desenvolver e avaliar soluções para a identificação de componentes biconexos em grafos não direcionados. Três métodos serão implementados para identificar os blocos de um grafo: a verificação da existência de caminhos internamente disjuntos, a identificação de articulações por meio da remoção de vértices, e a aplicação do algoritmo de Tarjan (**??**). Experimentos serão realizados em grafos aleatórios de diferentes tamanhos, com o propósito de avaliar a eficiência e escalabilidade das abordagens propostas.

2 DEFINIÇÃO DO PROBLEMA

O objetivo deste trabalho é implementar três métodos distintos para a identificação de componentes biconexos em grafos não direcionados. Os métodos a serem desenvolvidos são:

1. **Verificação da existência de dois caminhos internamente disjuntos**: Este método verifica se, para cada par de vértices de um subgrafo, existem dois caminhos internamente disjuntos, conectando-os de forma resiliente.

2. **Identificação de articulações através de testes de conectividade**: Este método identifica articulações removendo sistematicamente os vértices do grafo e testando sua conectividade. Se a remoção de um vértice aumenta o número de componentes conexos, esse vértice é classificado como uma articulação.

3. **Algoritmo de Tarjan (1972)**: Este método utiliza busca em profundidade (*Depth First Search* - DFS) para identificar articulações e componentes biconexos de forma eficiente,

com complexidade $O(V + E)$, onde V é o número de vértices e E o número de arestas. Ele se baseia nos tempos de descoberta e de retorno durante a exploração do grafo.

Os três métodos serão aplicados a grafos aleatórios com diferentes quantidades de vértices (100, 1.000, 10.000 e 100.000), e o tempo médio de execução será registrado para avaliar a eficiência de cada abordagem.

3 IMPLEMENTAÇÃO DOS MÉTODOS

Nesta seção, são descritas as implementações dos três métodos propostos para a identificação de componentes biconexos em grafos não direcionados. Cada método foi implementado utilizando a linguagem de programação *Java*.

3.1 Método 1: Verificação da Existência de Dois Caminhos Internamente Disjuntos

O primeiro método identifica os componentes biconexos analisando todos os pares de vértices do grafo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. A ideia central é verificar $\forall(u, v)$ tal que $u \in V$ e $v \in V$, se existem ao menos dois caminhos internamente disjuntos entre eles. Formalmente, dois caminhos P_1 e P_2 entre u e v são internamente disjuntos se $P_1 \cap P_2 = \{u, v\}$, ou seja, P_1 e P_2 não compartilham nenhum vértice além de u e v. Dessa forma, para testar se existe caminhos disjuntos entre pares de vértice, o algoritmo verifica se u e v pertencem a algum mesmo ciclo elementar $C \subseteq G$. A presença de u e v em um mesmo ciclo C garante a existência de dois caminhos disjuntos, já que a remoção de qualquer vértice interno a C não desconecta u de v.
Abaixo, apresentamos exemplos da análise de pares de vértices sobre o grafo $G = (\{a, b, c, d, e\}, \{\{a, b\}, \{a$

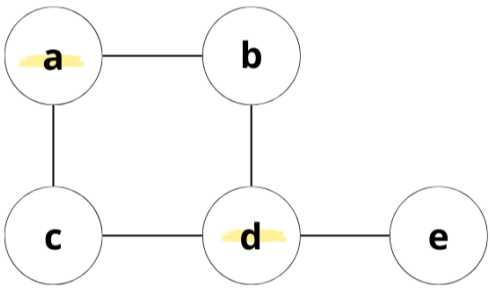

Figura 1 – Caminhos disjuntos (a, d)

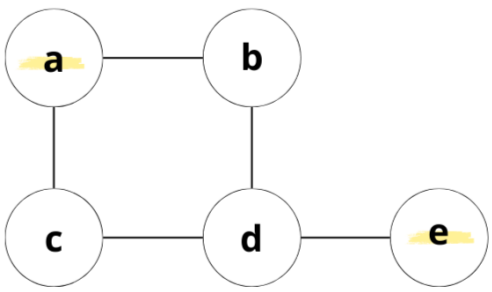

Figura 2 – Caminhos disjuntos (a, e)

Na Figura 1, podemos observar o par de vértices (a, d), que pertencem ao ciclo $C = \{a, b, d, c\}$, garantindo a existência de dois caminhos internamente disjuntos. Na Figura 2, o par (a, e) não pertence a nenhum ciclo, o que implica que não há dois caminhos disjuntos entre esses vértices.

Sendo assim, este método depende da identificação de todos os ciclos elementares do grafo, que são subgrafos $C \subseteq G$ nos quais os vértices estão conectados em um caminho fechado. Para realizar essa operação, é utilizado o *Cycle Finding Algorithm*, já que apenas uma busca em profundidade (*DFS*) não é suficiente para encontrar todos os ciclos elementares do grafo.

Devido ao fato de o método identificar majoritariamente os blocos através da verificação de caminhos disjuntos, há uma limitação: ele não é capaz de detectar blocos que não possuem caminhos internamente disjuntos, como os blocos isomorfos a K_2. Esses blocos correspondem a componentes formados apenas por dois vértices e uma única aresta entre eles, ou seja, as *pontes* do grafo.

Formalmente, um bloco $B \subseteq G$ isomorfo a K_2 ocorre quando a remoção de uma aresta $e = \{u, v\} \in E$ desconecta o grafo, caracterizando e como uma ponte. Como as pontes não possuem dois caminhos internamente disjuntos entre seus vértices, esse método utiliza uma etapa auxiliar para identificar todas as pontes do grafo G.

Portanto, além da verificação de ciclos elementares para identificar blocos maiores, o algoritmo incorpora um procedimento adicional baseado no cálculo de todas as pontes de G, garantindo assim a identificação correta dos blocos isomorfos a K_2.

Algorithm 1 Identificação de Blocos Biconexos

Entrada: Grafo não direcionado $G = (V, E)$
Saída: Lista de blocos biconexos de G
function IDENTIFYBLOCKS(Grafo G)
    Definir **allCycles** ← findAllCycles()
    Definir **allBridges** ← findBridges(G)
    Definir **blocks** ← ∅
for *cada* $i \in \{1, \ldots, |V|\}$ **do**
Definir **block** ← ∅ **for** *cada* $j \in \{i + 1, \ldots, |V|\}$ **do**
hasTwoDisjointPaths(i, j)
    block ← **block** ∪$\{i, j\}$
    if ***block*** ≠ ∅ **then**
blocks ← **blocks** ∪{**block**}
Definir **trimBlocks** ← removeSubsets(**blocks**)
for *cada* ***bridge*** ∈ ***allBridges*** **do**
trimBlocks ← **trimBlocks** ∪{{**bridge**[0], **bridge**[1]}}
return trimBlocks
end function

3.1.1 Análise de Complexidade

3.1.1.1 Busca de Ciclos Elementares (`findAllCycles`) O método `findAllCycles` é responsável por encontrar todos os ciclos elementares do grafo. Ele realiza uma busca em profundidade (*DFS*) para cada vértice $v \in V$, chamando a função `findCycles`. A complexidade da busca em profundidade para cada vértice é $O(V + E)$, onde V é o número de vértices e E é o número de arestas.
Como a busca é realizada para cada vértice $v \in V$, a complexidade total da busca por ciclos é:

$$O(V \times (V + E)) = O(VE)$$

3.1.1.2 Verificação de Caminhos Internamente Disjuntos (`hasTwoDisjointPaths`) Para cada par de vértices (i, j), o método `hasTwoDisjointPaths` verifica se ambos pertencem a algum ciclo. No pior caso, todos os ciclos encontrados precisam ser percorridos para verificar se o par de vértices está contido em algum ciclo. Se C for o número total de ciclos no grafo, a complexidade desta etapa é dada por:

$$O(C \times V^2)$$

onde V^2 representa o número de pares de vértices no grafo.

3.1.1.3 Remoção de Subconjuntos (`removeSubsets`) Após identificar os blocos biconexos, o método `removeSubsets` remove subconjuntos redundantes. Para cada bloco, é necessário compará-lo com todos os outros blocos. Se b for o número de blocos identificados e n for o tamanho médio dos blocos, a complexidade desta operação é:

$$O(b^2 \times n)$$

3.1.1.4 Processamento de Pontes (`findBridges`) O método `findBridges` identifica todas as pontes no grafo. A complexidade para encontrar pontes em um grafo não direcionado é conhecida por ser $O(V + E)$.

3.1.1.5 Complexidade Total A complexidade total do método `identifyBlocks` é dominada pela busca de ciclos elementares e pela verificação de caminhos internamente disjuntos. Assim, a complexidade total pode ser aproximada por:

$$O(VE) + O(C \times V^2) + O(b^2 \times n) + O(V + E)$$

onde V é o número de vértices, E é o número de arestas, C é o número de ciclos e b é o número de blocos.

Portanto, a complexidade resultante do algoritmo pode ser expressa como $O(VE + C \times V^2 + b^2 \times n)$, sendo dominada pela busca de ciclos e a verificação de caminhos disjuntos.

3.2 Método 2: Identificação de Articulações

O segundo método identifica articulações removendo sistematicamente cada vértice v do grafo e testando sua conectividade. Para testar a conectividade, foi utilizada uma busca em profundidade (*Depth First Search* - DFS), que verifica se todos os vértices restantes ainda estão conectados após a remoção de v. Caso os vértices não estejam conectados após a DFS, v é classificado como uma articulação.
Após a identificação das articulações, esses vértices são removidos do grafo, e uma nova busca é realizada para separar os componentes biconexos. Essa etapa permite a delimitação dos subgrafos formados pelos vértices adjacentes e conectados. Por fim, as articulações identificadas são reinserida aos seus respectivos blocos, assegurando a formação adequada dos componentes biconexos.

3.2.1 Análise de Complexidade

A complexidade do método é dominada por duas operações principais: a identificação dos vértices de articulação e a criação de blocos a partir da remoção das articulações encontradas.

3.2.1.1 Busca de Articulações (`findArticulations`) A identificação das articulações envolve realizar uma busca em profundidade (DFS) para cada vértice, resultando em uma complexidade de $O(V \cdot (V + E))$, onde V é o número de vértices e E o número de arestas.

3.2.1.2 Criação de Blocos (`identifyBlocks`) Após a identificação das articulações, uma nova DFS é realizada para formar os blocos, que tem complexidade $O(V + E)$. Assim, a complexidade total do algoritmo é $O(V \cdot (V + E))$, sendo $O(V^2)$ para grafos esparsos e $O(V \cdot E)$ para grafos densos.

3.3 Método 3: Algoritmo de Tarjan (1972)

O terceiro método utiliza do algoritmo de Tarjan (**??**) para a identificação de componentes biconexos em grafos não direcionados. Durante a execução, o algoritmo percorre o grafo marcando os vértices com tempos de descoberta, calculados pela ordem em que são visitados, e utiliza uma estrutura de pilha para armazenar as arestas. Além do tempo de descoberta, ele mantém um valor de "low", que indica o menor tempo de descoberta acessível a partir de um vértice,

considerando tanto os caminhos diretos quanto os de volta. Através da comparação desses valores de "low", é possível identificar articulações e componentes biconexos. Dessa forma, quando o algoritmo encontra uma articulação, as arestas armazenadas na pilha entre dois eventos de articulação formam um componente biconexo.

3.3.1 Análise de Complexidade

A complexidade do algoritmo de Tarjan é $O(V + E)$, onde V é o número de vértices e E é o número de arestas do grafo. Isso ocorre porque o algoritmo faz uma única busca em profundidade no grafo, visitando cada vértice e aresta apenas uma vez. A operação de atualização do valor "low"e as verificações para articulações são realizadas em tempo constante para cada vértice. Assim, a eficiência do algoritmo é garantida, mesmo para grafos de grandes dimensões, sendo considerado uma das abordagens mais eficazes para resolver o problema de identificação de componentes biconexos.

3.4 Ferramentas Utilizadas

A implementação foi realizada em *Java*, utilizando uma estrutura de lista de adjacência para a representação e manipulação dos grafos. A geração de grafos aleatórios foi feita com o uso da biblioteca `NetworkX` em *Python*, sendo executada por meio de um script em `bash` para facilitar os testes automatizados com diferentes números de vértices e métodos.
O script `bash` criado realiza a geração dos grafos aleatórios, a compilação do código Java e a execução dos testes de forma automatizada. O script recebe como parâmetros o número de vértices e o método a ser testado, que pode ser: 1 para a verificação de dois caminhos disjuntos, 2 para a identificação de articulações, ou 3 para o algoritmo de Tarjan.

4 EXPERIMENTOS E RESULTADOS

Nesta seção, são apresentados os experimentos realizados para avaliar a eficiência dos três métodos de identificação de componentes biconexos em grafos não direcionados, conforme descrito anteriormente. Os experimentos foram conduzidos em grafos aleatórios de diferentes tamanhos, especificamente com 100, 1.000, 10.000 e 100.000 vértices. O tempo médio de execução de cada método foi registrado, permitindo uma comparação da eficiência entre as abordagens propostas.

4.1 Configuração dos Experimentos

Os grafos aleatórios foram gerados utilizando a biblioteca `NetworkX`, configurando a probabilidade de inclusão de arestas de forma a garantir a conectividade dos grafos. O tempo de execução de cada método foi medido em milissegundos.

4.2 Resultados Obtidos

Os resultados obtidos foram organizados em tabelas para facilitar a visualização e análise. A Tabela 1 apresenta o tempo de execução em milissegundos para cada método, considerando os diferentes tamanhos de grafos.

Número de Vértices	Método 1	Método 2	Método 3
100	8134	235	167
1.000	9382571	287	220
10.000	Indefinido	3248	468
100.000	Indefinido	Indefinido	479

Tabela 1 – Tempo de execução dos métodos em milissegundos.

4.3 Análise dos Resultados

A partir dos resultados obtidos, pode-se observar que o Método 3, que utiliza o algoritmo de Tarjan, apresenta o melhor desempenho em termos de tempo de execução, especialmente para grafos de maior dimensão. Isso se deve à sua complexidade linear $O(V + E)$, que se mostra eficaz na identificação de componentes biconexos mesmo em grafos grandes.
O Método 2, que realiza a identificação de articulações através de testes de conectividade, apresenta um desempenho intermediário, evidenciando que, apesar de ser mais eficiente que o Método 1 em grafos maiores, ainda sofre um aumento significativo no tempo de execução à medida que o número de vértices aumenta.
Por fim, o Método 1, que verifica a existência de dois caminhos internamente disjuntos, apresentou o pior desempenho. A necessidade de verificar todos os pares de vértices e identificar ciclos elementares contribui para o aumento do tempo de execução, tornando-o menos eficiente em comparação com as outras abordagens.

4.4 Considerações Finais

Os experimentos demonstraram que a escolha do método para a identificação de componentes biconexos deve levar em consideração o tamanho do grafo e os requisitos de eficiência. O algoritmo de Tarjan é recomendado para aplicações que demandam rapidez, enquanto os outros métodos podem ser considerados em cenários específicos onde a análise detalhada dos caminhos e articulações é necessária.

REFERÊNCIAS

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

GeeksforGeeks. *Graph and its Representations*. Disponível em: <https://www.geeksforgeeks.org/graph-and-its-representations/>. Acesso em: 05 out. 2024.

Baeldung. *Graph Traversal*. Disponível em: <https://github.com/eugenp/tutorials/blob/master/data-structures/src/main/java/com/baeldung/graph/GraphTraversal.java>. Acesso em: 05 out. 2024.

Baeldung. *Java Graphs*. Disponível em: <https://www.baeldung.com/java-graphs>. Acesso em: 05 out. 2024.

ChatGPT. *Assistência com análise de complexidade, algoritmos de busca de ciclos e estruturas de grafos*. Disponível via ChatGPT. Acesso em: 2024.

Y. Wang. *Cycle Finding Algorithm in Graph Theory*. Disponível em: <https://www.youtube.com/watch?v=qNVNoZJFp_g>. Acesso em: 05 out. 2024.

Frank Harary. *Graph Theory*. Addison-Wesley, 1969. Clássico da teoria dos grafos, cobre conceitos relacionados a grafos 2-conexos.

J. A. Bondy, U. S. R. Murty. *Graph Theory with Applications*. Elsevier Science Publishing Co Inc, 1976. Esta obra cobre amplamente temas como grafos conexos e 2-conexos, com várias aplicações práticas.

Princeton University. *Algorithms, Part 2: Graphs*. Disponível em: <https://algs4.cs.princeton.edu/41graph/>. Acesso em: 05 out. 2024. Este site inclui implementações de algoritmos de grafos, como a busca em profundidade não-recursiva (DFS).