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

Trabalho Prático N. 01*

Teoria dos Grafos e Computabilidade

Rodrigo Drummond Macedo[1]
Henrique Resende Lara[2]
Tiago Lascasas Antunes[3]

Resumo

Este trabalho aborda a implementação e análise de três métodos distintos para a identificação de componentes biconexos em grafos não-direcionados, estruturas fundamentais na teoria dos grafos com diversas aplicações em áreas como redes de comunicação, circuitos elétricos e análise de redes sociais. Os componentes biconexos, também chamados de blocos, são subgrafos maximais que permanecem conectados mesmo com a remoção de qualquer vértice. Os métodos implementados incluem: (i) a verificação da existência de dois caminhos internamente disjuntos entre pares de vértices, (ii) a identificação de articulações, que são vértices cuja remoção desconecta o grafo, e (iii) o algoritmo de Tarjan (1972), uma solução eficiente baseada em busca em profundidade para a detecção de componentes biconexos. Todos os métodos foram implementados com a finalidade de encontrar os componentes biconexos.

Para avaliar a eficiência dos métodos, foram realizados experimentos com grafos aleatórios de diferentes tamanhos, variando de 100 a 100.000 vértices. O tempo de execução de cada abordagem foi medido e comparado, sendo que o algoritmo de Tarjan se destacou pela sua eficiência, especialmente em grafos maiores. Este trabalho explora as vantagens e desvantagens de cada método, fornecendo uma análise comparativa sobre sua aplicabilidade em diferentes cenários de uso, com ênfase na escalabilidade e na eficiência computacional.

Palavras-chave: Grafo. Caminhos internamente disjuntos. Articulação. Tarjan.

*Artigo apresentado ao professor da disciplina Teoria de Grafos e Computabilidade, como pré-requisito para a entrega do trabalho prático número 1.

[1]Aluno do Programa de Graduação em Ciência da Computação, Brasil – rdmacedo@sga.pucminas.br.
[2]Aluno do Programa de Graduação em Ciência da Computação, Brasil – henrique.lara@sga.pucminas.br.
[3]A do Programa de Graduação em Ciência da Computação, Brasil – tiago.antunes.1435526@sga.pucminas.br.

Abstract

This paper addresses the implementation and analysis of three distinct methods for identifying biconnected components in non-directed graphs, which are fundamental structures in graph theory with various applications in fields such as communication networks, electrical circuits, and social network analysis. Biconnected components, also known as blocks, are maximal subgraphs that remain connected even after the removal of any vertex. The methods implemented include: (i) verification of the existence of two internally disjoint paths between vertex pairs, (ii) identification of articulation points, which are vertices whose removal disconnects the graph, and (iii) Tarjan's algorithm (1972), an efficient depth-first search-based solution for detecting biconnected components. All methods were implemented with the objective of identifying the biconnected components.

To assess the efficiency of these methods, experiments were conducted with random graphs of different sizes, ranging from 100 to 100,000 vertices. The execution time of each approach was measured and compared, with Tarjan's algorithm standing out for its efficiency, particularly in larger graphs. This work explores the advantages and disadvantages of each method, providing a comparative analysis of their applicability in different use cases, with emphasis on scalability and computational efficiency.

Keywords: Graph. Internally disjoint paths. Articulation points. Tarjan.

1 INTRODUÇÃO

Grafos são uma das estruturas de dados mais versáteis e amplamente utilizadas na Computação, com aplicações que vão desde o modelamento de redes de comunicação até a análise de redes sociais e a otimização de rotas. Um grafo conexo que não possui articulações é denominado grafo biconexo ou 2-conexo, e apresenta uma propriedade interessante de "tolerância a falhas". Isso significa que, em um grafo biconexo, a remoção de um único vértice não desconecta o grafo, garantindo assim uma maior robustez.

A identificação de componentes biconexos em grafos é uma tarefa essencial em várias áreas, como redes de computadores, planejamento de circuitos e análise de redes sociais. Esses componentes são subgrafos maximais que possuem a propriedade de serem biconexos em vértices, e a determinação eficiente desses componentes pode otimizar a resolução de muitos problemas práticos.

Neste artigo, são apresentadas três abordagens para a identificação de blocos ou componentes biconexos em um grafo:

Um método que verifica a existência de dois caminhos internamente disjuntos entre cada par de vértices de um bloco. Um método que identifica articulações ao testar a conectividade do grafo após a remoção de cada vértice. A implementação do algoritmo de Tarjan(TARJAN, 1972), que utiliza busca em profundidade para encontrar componentes biconexos de maneira eficiente, a qual foi baseada na leitura do artigo e em pesquisas adicionais.

Como fontes de pesquisa foram usados (GeeksforGeeks, 2024) e ferramentas de IA, como (OPENAI, 2024), também usado para correção de erros ortográficos neste artigo, e auxílio para a construção do gerador de grafos e dos gráficos para análise dos experimentos.

Os experimentos para avaliar o tempo médio de execução de cada abordagem usaram de grafos com diferentes tamanhos, variando de 100 a 100.000 vértices. O objetivo deste trabalho é comparar a eficiência dos métodos e analisar como cada um se comporta em escalas variadas, fornecendo dados sobre a viabilidade de cada abordagem para grafos de grande porte.

2 DESENVOLVIMENTO

Nesta seção, serão discutidos os três métodos implementados para a identificação de componentes biconexos em grafos. Cada algoritmo utiliza abordagens diferentes para resolver o problema, variando desde a verificação de caminhos internamente disjuntos até a identificação de articulações e o algoritmo de Tarjan. Os métodos foram aplicados em grafos de diferentes tamanhos para avaliar sua eficiência e tempo de execução. As subseções a seguir detalham cada abordagem, explicando a lógica de implementação e os resultados obtidos com cada método.

2.1 Implementação Grafo

Para a implementação do grafo não direcionado, foi utilizada uma estrutura de lista de adjacência, onde cada vértice possui uma lista de vértices adjacentes. A classe Adjacentes foi implementada para representar cada nó dessa lista de adjacência, armazenando o vértice e um ponteiro para o próximo vértice adjacente. Essa abordagem permite que o grafo seja representado de forma eficiente em termos de memória, especialmente em grafos esparsos, onde o número de arestas é relativamente baixo em comparação ao número de vértices.

Além disso, como o grafo é não direcionado, cada aresta adicionada entre dois vértices u e v é inserida nas listas de adjacência de ambos os vértices, ou seja, u é adicionado à lista de adjacência de v e vice-versa. Isso é realizado no método addAresta, que insere as arestas de forma bidirecional, garantindo que as conexões entre os vértices sejam armazenadas corretamente.

Após a inserção das arestas, as listas de adjacência são ordenadas utilizando o método ordenarAdjacentes, que implementa o algoritmo de Selection Sort para garantir que os vértices adjacentes sejam armazenados em ordem crescente. Isso facilita a busca e manipulação dos vértices ao longo dos algoritmos aplicados ao grafo, além de garantir consistência na representação das adjacências.

A leitura do grafo a partir de um arquivo texto é feita na função constroiAdjacentes, que processa cada linha do arquivo, extraindo os pares de vértices que representam as arestas e inserindo-os nas listas de adjacência correspondentes.

2.2 Gerador de grafos

O gerador de grafos no código utiliza o modelo de Erdős-Rényi (G(n, p)), onde um grafo aleatório é criado com n vértices e cada aresta é incluída com uma probabilidade p. O código garante que o grafo gerado seja conexo, repetindo a geração até que todos os vértices estejam conectados. Após a criação, o grafo é salvo em um arquivo de texto, listando o número de vértices e as arestas. Além disso, os componentes biconexos do grafo são identificados e impressos, utilizando funções da biblioteca ‘networkx‘, da linguagem python, permitindo testar a robustez e a eficiência de algoritmos de grafos em diferentes cenários.

2.3 Verificação de caminhos internamente disjuntos

2.3.1 Inicialização do Grafo e Listas

O código implementa a busca de ciclos em um grafo, onde o grafo é representado por uma lista de adjacências, sendo cada vértice conectado a seus vizinhos através de um objeto

da classe Adjacentes. A classe CycleFinder é responsável por gerenciar a estrutura do grafo e executar o algoritmo de busca por ciclos. O processo de busca é feito por meio de uma varredura em profundidade (DFS), identificando e armazenando os ciclos encontrados.

2.3.2 Função encontrarCiclos

A função encontrarCiclos é responsável por iniciar a busca por ciclos no grafo. Ela percorre todos os vértices e, para cada vértice, executa a DFS. O ciclo é armazenado temporariamente durante o percurso, e, ao encontrar um ciclo, o código o normaliza e o adiciona à lista de ciclos. Após identificar todos os ciclos, a função filtra a lista, removendo subciclos para garantir que apenas os ciclos maiores sejam mantidos. Aqui estão os principais passos da função:

Varredura DFS para cada vértice: A função percorre todos os vértices do grafo, iniciando uma nova busca em profundidade (DFS) a partir de cada um. Cada ciclo encontrado durante a DFS é armazenado temporariamente na variável caminhoAtual.

Filtragem dos ciclos: Após coletar todos os ciclos possíveis, a função filtrarCiclos remove subciclos (ciclos que são subconjuntos de outros ciclos maiores) para manter apenas os ciclos relevantes.

2.3.3 Função dfs

A função dfs (Deep First Search) realiza a busca em profundidade. Ela explora os vértices adjacentes de um vértice inicial, mantendo o controle dos vértices visitados e do caminho atual. Ao encontrar um ciclo, o código invoca a função normalizarEAdicionarCiclo para processar o ciclo encontrado. A seguir estão os detalhes da função:

Exploração de vértices adjacentes: Para cada vértice adjacente ao vértice atual, a função verifica se ele já foi visitado. Se o vértice adjacente for o vértice inicial (indica um ciclo) e o ciclo contém mais de dois vértices, o ciclo é registrado. Caso contrário, a função continua a DFS para explorar os vértices adjacentes.

Backtracking: Após explorar todos os vértices adjacentes de um vértice, o código desfaz o último movimento, removendo o vértice atual do caminho e marcando-o como não visitado, para permitir a busca de outros ciclos.

2.3.4 Função normalizarEAdicionarCiclo

Esta função processa cada ciclo encontrado durante a DFS, garantindo que ele seja armazenado em uma forma normalizada, sem duplicatas. Para isso, a função utiliza o menor vértice como referência para rotacionar o ciclo e compará-lo com sua versão reversa, garantindo que

apenas a representação lexicograficamente menor seja armazenada. A seguir estão os detalhes da função:

Rotacionar o ciclo para começar pelo menor vértice: O ciclo é rotacionado para que o vértice de menor valor seja o primeiro, permitindo uma comparação padronizada entre diferentes representações do mesmo ciclo.

Comparação com a versão reversa: A função compara o ciclo rotacionado com sua versão reversa, escolhendo a versão com a menor ordem lexicográfica para garantir consistência.

Adição à lista de ciclos: Se o ciclo normalizado ainda não estiver presente na lista de ciclos, ele é adicionado.

2.3.5 Função filtrarCiclos

Após a identificação de todos os ciclos, a função filtrarCiclos remove subciclos. Um subciclo é um ciclo que está contido em outro ciclo maior. A função compara cada ciclo com os outros, verificando se todos os vértices de um ciclo menor estão presentes em um ciclo maior. Se um ciclo for identificado como subciclo, ele é removido da lista final.

Comparação entre ciclos: A função utiliza conjuntos (Set) para verificar se um ciclo está contido em outro ciclo maior.

Remoção de subciclos: Ciclos que são subciclos de outros ciclos maiores são filtrados e removidos, garantindo que a lista final contenha apenas os maiores ciclos.

2.3.6 Complexidade e Eficiência

A implementação da busca de ciclos utiliza uma abordagem não eficiente para encontrar os ciclos e após isso encontrar os componentes biconexos, o que faz com que para grafos de tamanho médio para cima a complexidade do código aumente de forma fatorial. Dessa forma, essa primeira abordagem para encontrar os componentes não se mostra viável.

2.4 Identificação de articulações

2.4.1 Estrutura de Dados e Inicialização

O código começa com a definição das classes Aresta, Grafo, e Adjacentes. A classe Aresta representa uma aresta no grafo, com atributos para origem e destino, além de um método toString para exibir a aresta de forma legível.

A classe Grafo é responsável pela representação do grafo através de uma lista de adjacências. Aqui estão algumas estruturas de dados importantes usadas na classe:

adjacentes[]: Armazena as listas de adjacências para cada vértice. tempoDescoberta[]: Registra o tempo em que cada vértice foi descoberto durante a busca em profundidade (DFS). menorTempo[]: Mantém o menor tempo de descoberta que pode ser alcançado a partir de um vértice. pais[]: Guarda o pai de cada vértice na árvore DFS. visitado[]: Indica se um vértice foi visitado. tempo: Variável que mantém o tempo atual durante a execução da DFS. articulacoes: Lista para armazenar os vértices que são pontos de articulação. pilhaArestas: Pilha que guarda as arestas enquanto a DFS é realizada.

2.4.2 Método identificarArticulacoesEComponentesBiconexos

Este método executa a identificação de articulações e componentes biconexos no grafo. Ele faz isso da seguinte maneira:

Remoção de Vértices: Para cada vértice, é simulado a remoção do mesmo e contado quantos componentes conectados permanecem no grafo. Isso é feito através de uma DFS (função dfsRemocao). Se a remoção de um vértice resulta em mais de um componente conectado, esse vértice é marcado como uma articulação.

Execução da DFS: Após a identificação de articulações, o método chama encontrarComponentesDFS para explorar o grafo e descobrir componentes biconexos.

2.4.3 Função dfsRemocao

A função dfsRemocao é uma DFS simples que marca todos os vértices alcançáveis a partir do vértice v, desconsiderando o vértice que foi removido. Essa abordagem é crucial para verificar a conectividade do grafo após a remoção de um vértice.

2.4.4 Método encontrarComponentesDFS

Este método é a implementação principal do algoritmo de Tarjan, que faz o seguinte:

Marcação e Inicialização: O vértice atual u é marcado como visitado e os tempos de descoberta e menor tempo são inicializados. A variável filhos conta o número de filhos de u na árvore DFS.

Iteração sobre Vértices Adjacentes: Para cada vértice adjacente v, se v não foi visitado, a função o visita recursivamente. Durante a volta da DFS, o menorTempo[u] é atualizado com base em menorTempo[v].

Identificação de Articulações e Componentes Biconexos: A lógica para identificar articulações e componentes biconexos é aplicada com base nos valores de menorTempo e tempoDescoberta. Quando um componente biconexo é detectado, as arestas relacionadas são desem-

pilhadas da pilhaArestas e processadas.

2.4.5 Classe Adjacentes

A classe Adjacentes representa a lista de adjacências de cada vértice. Inclui métodos para obter o próximo vértice e definir o próximo vértice da lista. O método ordenarAdjacentes garante que as adjacências de cada vértice sejam armazenadas em ordem, facilitando a busca e a visualização.

2.4.6 Complexidade

O algoritmo implementado para identificar articulações e componentes biconexos apresenta uma complexidade de tempo de O(V * (V + E)), onde V representa o número de vértices e E é o número de arestas do grafo. Essa complexidade se deve à necessidade de iterar sobre cada vértice e realizar uma busca em profundidade (DFS) para verificar a conectividade após a remoção do vértice, o que pode ser ineficiente para grafos densos. Apesar de ser funcional, essa abordagem pode resultar em um desempenho reduzido em grafos de grande escala.

2.5 Algoritmo de Tarjan

2.5.1 Inicialização do Grafo e Listas

O grafo é representado por uma lista de adjacências, onde cada vértice possui uma lista de vértices adjacentes, implementada na classe Adjacentes. A classe Grafo é responsável por gerenciar os vértices e adjacências, assim como os métodos necessários para a execução do algoritmo de Tarjan. Durante a execução, três vetores são usados:

desc[]: armazena o tempo de descoberta de cada vértice durante a DFS. low[]: registra o menor tempo de descoberta que pode ser alcançado a partir de um dado vértice, considerando a árvore DFS e as arestas de retorno. pai[]: mantém o rastreamento do pai de cada vértice na árvore DFS.

2.5.2 Função encontraCompBiconexo

A função encontraCompBiconexo é o núcleo do algoritmo de Tarjan e é responsável pela execução da busca em profundidade (DFS) no grafo, além de gerenciar os tempos de descoberta e os valores low. Aqui está um detalhamento das etapas da função:

Busca em Profundidade (DFS): A função começa visitando o vértice u, marcando o tempo de descoberta (desc[u]) e atualizando o valor de low[u] com o tempo da descoberta inicial. Para cada vértice adjacente v de u, a função verifica se v já foi visitado. Se v não foi visitado, a função o explora recursivamente, para continuar a busca em profundidade a partir de v.

Atualização dos Valores low: Quando a busca DFS retorna para o vértice u, o valor low[u] é atualizado considerando o valor de low[v]. Se houver uma aresta de retorno que conecta v ou algum descendente de v a um antecessor de u, o valor low[u] será atualizado para refletir essa conexão, garantindo que o menor valor de descoberta seja propagado corretamente.

Identificação de Componentes Biconexos: Um componente biconexo é identificado quando o valor low[v] de um vértice adjacente v é maior ou igual ao valor de descoberta desc[u] do vértice atual u. Isso indica que u é um ponto de articulação, e todas as arestas armazenadas na pilha desde a última aresta crítica formam um componente biconexo. Nesse ponto, as arestas da pilha que conectam os vértices do componente são desempilhadas e associadas ao componente biconexo.

Tratamento de Arestas e Articulações: A função também gerencia a pilha de arestas (st). Cada vez que uma nova aresta é visitada, ela é empilhada. Quando um componente biconexo é detectado, as arestas são desempilhadas até que o componente seja completamente processado.

2.5.3 Função identificaCompBiconexo

A função identificaCompBiconexo é responsável por controlar a execução do algoritmo para todos os vértices do grafo. Ela faz a inicialização das estruturas de dados e garante que todos os componentes biconexos sejam encontrados:

Inicialização: A função começa inicializando os vetores desc[], low[] e pai[]. Esses vetores são usados para registrar os tempos de descoberta de cada vértice, o menor valor de descoberta alcançado e o pai de cada vértice na árvore DFS.

Iteração sobre os Vértices: A função percorre cada vértice do grafo. Se um vértice ainda não foi visitado (ou seja, desc[v] == -1), a função chama encontraCompBiconexo para realizar a DFS a partir desse vértice e encontrar os componentes biconexos relacionados a ele.

Processamento Final da Pilha: Após a DFS, a pilha pode ainda conter arestas de componentes biconexos que não foram completamente processados. Nesse caso, a função esvazia a pilha e imprime os componentes restantes. Isso garante que nenhum componente biconexo seja deixado de fora.

2.5.4 Estrutura da Pilha (Stack<Aresta>)

As arestas são empilhadas à medida que os vértices são visitados, e são removidas da pilha assim que um componente biconexo é identificado. O uso da pilha garante que as arestas sejam agrupadas corretamente em componentes, sem a necessidade de manter estruturas de dados adicionais complexas.

Empilhamento de Arestas: Cada vez que uma nova aresta é explorada, ela é adicionada à pilha. Isso ocorre tanto para arestas de árvore (arestas que conectam vértices diretamente no processo de DFS) quanto para arestas de retorno (arestas que conectam um vértice a um ancestral).

Desempilhamento de Componentes: Quando um ponto de articulação é encontrado (ou seja, um vértice que separa componentes biconexos), as arestas relacionadas a esse componente são desempilhadas e associadas ao componente biconexo. Esse processo continua até que todas as arestas de um componente tenham sido processadas.

2.5.5 Identificação de Componentes Biconexos

A lógica para identificar um componente biconexo baseia-se nos valores low[] e desc[] calculados durante a DFS. Se low[v] >= desc[u], significa que o vértice u é um ponto de articulação, e o subgrafo que conecta u a v (e seus descendentes) forma um componente biconexo. Esse critério é fundamental para a eficiência do algoritmo de Tarjan, permitindo identificar componentes sem a necessidade de recalcular as conexões para cada aresta.

2.5.6 Complexidade e Eficiência

O algoritmo de Tarjan é extremamente eficiente, com complexidade de tempo O(V + E), onde V é o número de vértices e E é o número de arestas do grafo. Essa eficiência se dá pelo fato de que o algoritmo processa cada vértice e aresta uma única vez durante a execução da busca em profundidade, tornando-o ideal para trabalhar com grafos de grande escala. Além disso, o uso de estruturas simples, como pilhas e vetores, garante que o algoritmo tenha uma baixa sobrecarga de memória.

3 EXPERIMENTOS

Para avaliar a eficiência de cada modelo acima, foram coletados dados de seus tempos de execução com grafos de 100, 1000, 2000, 5000, 10000, 20000 e 100.000 vértices. De início, é feita a comparação dos tempos de execução dentro de um mesmo método, para grafos de

diferentes tamanhos, tendo em mente que o tempo considerado para cada tamanho é a média dos tempos de execução de 5 grafos diferentes, mas de mesmo tamanho. Na sequência, se realiza a comparação da eficiência de cada método para grafos de mesmo tamanho, comparando as diferenças nos tempos de resposta de cada modelo para um mesmo grafo.

A análise dos tempos de execução foi realizada usando a classe System.nanoTime(), por isso os valores de tempo podem ser irrealistas, porém uma vez que o objetivo da análise são as diferenças de tempo com os diferentes números de vértices, esse fator não foi levado em conta.

Além disso, foi considerado todo o tempo de execução da função main, incluindo a leitura dos arquivos e a impressão dos resultados no console, o que aumenta consideravelmente os tempos de execução.

3.1 Comparação individual de modelos

Seguem abaixo, gráficos individuais que mostram os tempos de execução de cada modelo, em comparação ao número de vértices dos grafos não-direcionados

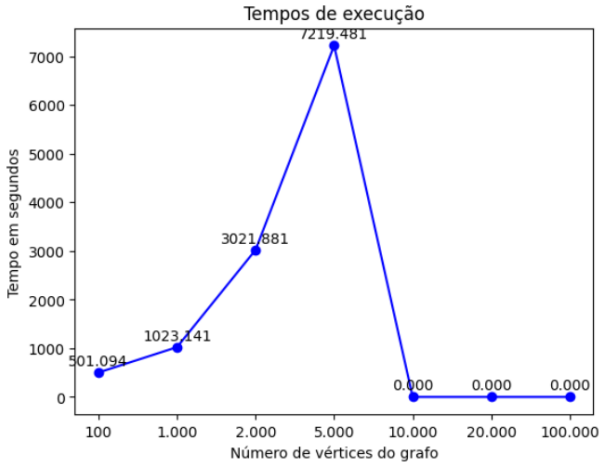

Figura 1 – Caminhos internamente disjuntos

Como mostra o gráfico acima, o algoritmo de identificação de caminhos internamente disjuntos para encontrar os componentes biconexos se mostrou extremamente ineficiente, possuindo uma complexoidade de ordem fatorial, que impossibilitou o funcionamento do algoritmo para grafos maiores do que 5.000 vértices, por ter um tempo de execução inviável.

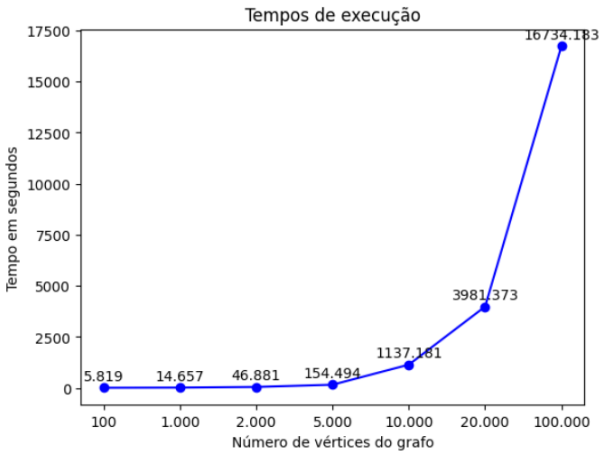

Figura 2 – Identificação de Articulações

Baseado no gráfico acima, se observa que o algoritmo que identifica articulações para achar os componentes biconexos também se mostra eficiente para grafos de até 5.000 vértices, sendo importante destacar a redução nos tempos de execução quando comparado com o algoritmo anterior, demonstrando ter uma menor complexidade porém ainda tendo problemas ao lidar com grafos a partir de 5.000 vértices, de onde o tempo começa a crescer de maneira acentuada.

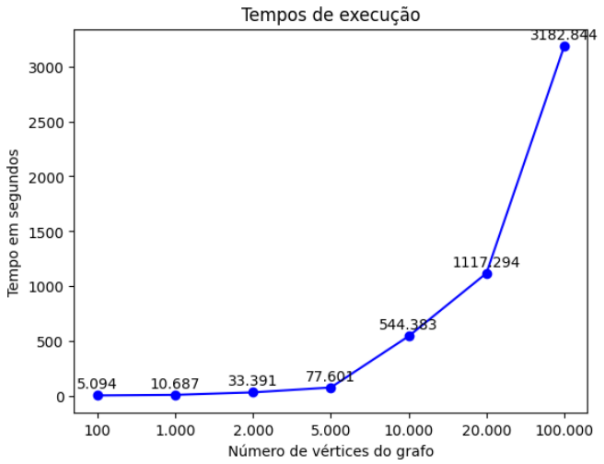

Figura 3 – Algoritmo de Tarjan

Com base no gráfico acima, pode-se observar que o algoritmo de Tarjan mantém uma performance eficiente para grafos com até 5.000 vértices, apresentando tempos de execução baixos e consistentes. No entanto, a partir de 10.000 vértices, o tempo de execução começa a aumentar significativamente, evidenciando uma aceleração no crescimento para grafos de 20.000 vértices. Esse comportamento indica que, embora o algoritmo seja eficiente, o aumento no número de vértices inevitavelmente aumenta muito seu tempo de execução.

3.2 Comparação entre modelos

Segue abaixo uma comparação dos tempos de execução para cada grafo, considerando todos os três modelos. Importante considerar que a diferença de performance gritante entre o algoritmo 1 e o restante é tratada após o primeiro gráfico, porém ignorada no restante, para evitar repetição desnecessária da informação.

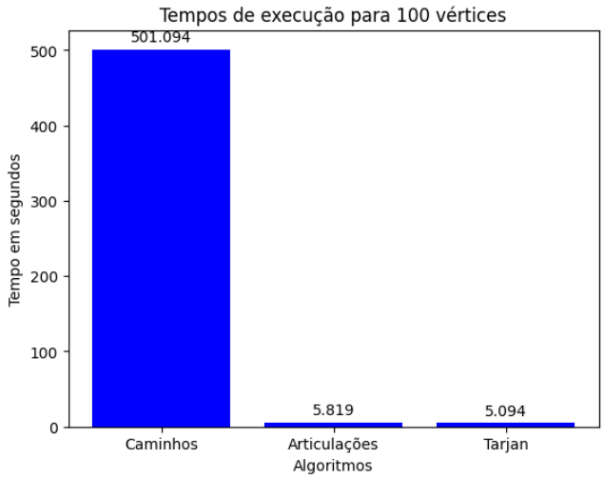

Figura 4 – Grafo 100 vértices

Para os algoritmos 2 e 3, a diferença no tempo de execução se mostra irrelevante, com ambos sendo eficientes para grafos pequenos. Porém, o crescimento fatorial do algoritmo 1 faz com que seu desempenho seja extremament inferior aos outros dois, sendo seu uso inviável e não racional.

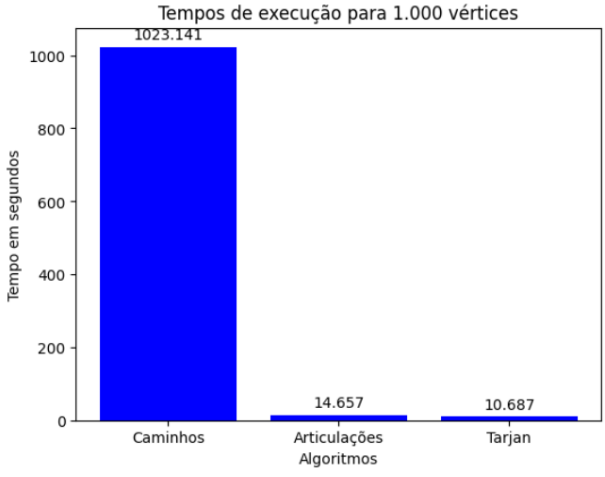

Figura 5 – Grafo 1.000 vértices

Com 1.000 vértices, a diferença entre os tempos de execução dos algoritmos 2 e 3 ainda se mostra insignificante, de apenas 4 segundos. Porém o aumento da diferença de tempo entre os algoritmos releva as diferentes complexidades dos algoritmos.

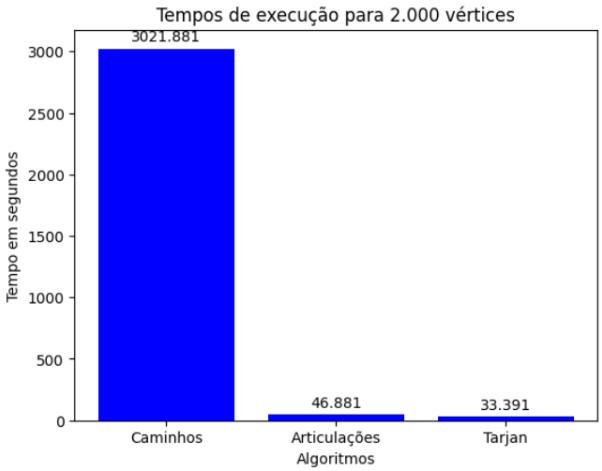

Figura 6 – Grafo 2000 vértices

Para grafos de 2000 vértices, a diferença no tempo de execução dos algoritmos 2 e 3 já começa a causar maior impacto, com o intervalo de tempo de execução entre uma e outra triplicando.

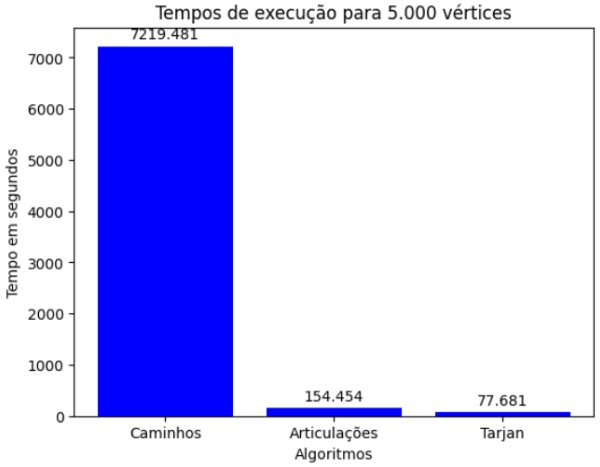

Figura 7 – Grafo 5.000 vértices

Para grafos de 5000 vértices, a diferença na complexidade dos algoritmos tem grande relevância para o tempo de execução dos algoritmos, com o algoritmo de articulações levando o dobro de tempo do algoritmo de Tarjan. Já o primeiro algoritmo é extremamente impactado por sua complexidade, realçando sua inviabilidade para encontrar componentes biconexos.

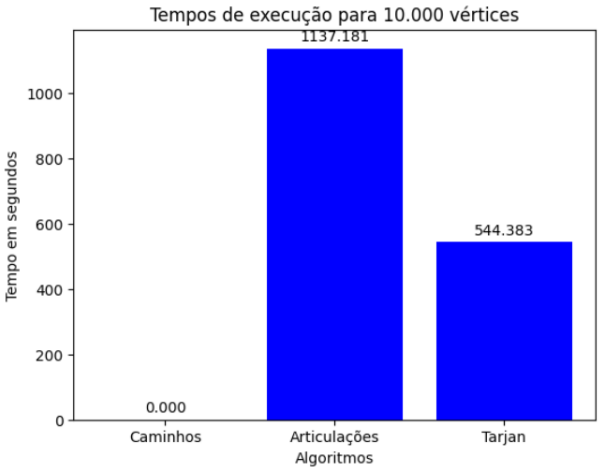

Figura 8 – Grafo 10.000 vértices

Para grafos de 10.000 vértices, a difereça de tempo de execução entre os métodos 2 e 3 explicita a inviabilidade do algoritmo 2 para grafos com mais do que 5.000 vértices. Além disso, o algoritmo 1 já não é mais considerado a partir desse ponto, uma vez que seu tempo de execução excedeu o limite de horas tolerado para o experimento.

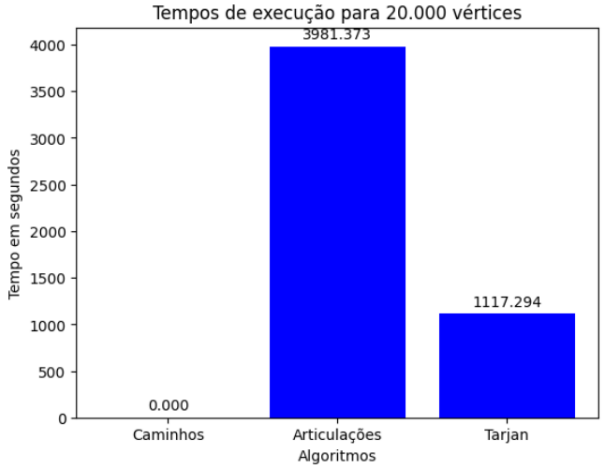

Figura 9 – Grafo 20.000 vértices

Para grafos de 20000 vértices, a diferença no tempo de execução se torna ainda mais gritante.

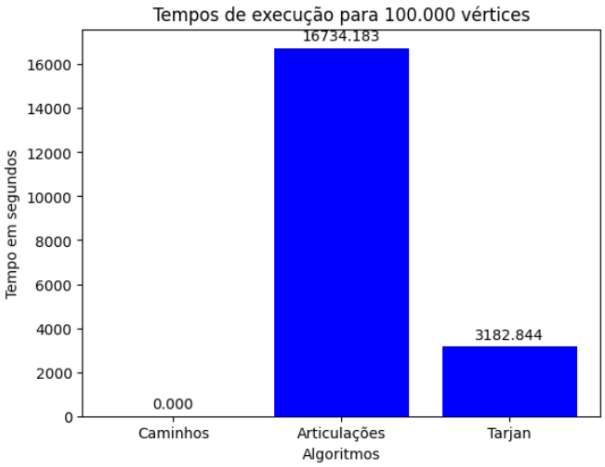

Figura 10 – Grafo 100.000

Por fim, para 100.000 vértices, mesmo o algoritmo de Tarjan apresenta um tempo de resposta bem acima do esperado, uma vez que sua complexidade é linear, podendo indicar diferenças na implementação, ou que mesmo esse algoritmo é vítima do grande número de vértices.

4 RESULTADOS

Os resultados obtidos a partir dos experimentos conduzidos com os três algoritmos de identificação de componentes biconexos revelaram comportamentos distintos em termos de eficiência e escalabilidade, especialmente à medida que o número de vértices no grafo aumentava. Os experimentos foram realizados em grafos aleatórios de tamanhos variados, de 100 a 100.000 vértices, com o objetivo de medir o tempo de execução e a escalabilidade de cada algoritmo.

4.1 Algoritmo de Caminhos Internamente Disjuntos

O algoritmo que verifica a existência de dois caminhos internamente disjuntos entre pares de vértices demonstrou ser altamente ineficiente em grafos com mais de 5.000 vértices. Sua complexidade, de ordem fatorial, tornou-se impraticável à medida que o número de vértices aumentava, fazendo com que o tempo de execução crescesse de forma exponencial. Nos experimentos com grafos de 10.000 vértices ou mais, o algoritmo não conseguiu completar sua execução dentro de um tempo razoável, o que limita sua aplicabilidade para grafos maiores.

4.2 Algoritmo de Identificação de Articulações

O segundo método, baseado na identificação de articulações, apresentou um desempenho superior ao do algoritmo de caminhos internamente disjuntos. Embora seja eficiente para grafos menores, com até 5.000 vértices, o tempo de execução começou a crescer de maneira acentuada para grafos maiores. Em grafos com 10.000 vértices, o tempo de resposta do algoritmo já era significativamente maior, e em grafos com mais de 20.000 vértices, tornou-se inviável. Esse comportamento está alinhado com sua complexidade quadrática $O(V^2)$, que se torna mais evidente em grafos densos.

4.3 Algoritmo de Tarjan

O algoritmo de Tarjan, baseado em busca em profundidade (DFS) e com complexidade $O(V + E)$, foi o mais eficiente dos três, especialmente para grafos de até 10.000 vértices. Sua execução foi rápida e consistente até esse ponto, demonstrando a superioridade da abordagem em relação às outras duas implementações. Contudo, à medida que o número de vértices aumentou para 20.000 e 100.000, o algoritmo também começou a apresentar um aumento significativo no tempo de execução. Embora esse crescimento não tenha sido tão acentuado quanto os outros dois algoritmos, o aumento no número de vértices e arestas impactou sua eficiência em grafos de grande escala, possivelmente devido a limitações na implementação ou na estrutura dos dados.

4.4 Comparação Geral

A comparação geral entre os três métodos mostrou que, para grafos pequenos (até 1.000 vértices), todos os algoritmos foram capazes de completar a execução em um tempo aceitável, com o algoritmo de Tarjan apresentando o menor tempo de execução. Já em grafos de tamanho médio (até 5.000 vértices), o algoritmo de articulações também manteve um desempenho relativamente bom, enquanto o de caminhos internamente disjuntos se mostrou inviável. Em grafos grandes (10.000 a 100.000 vértices), apenas o algoritmo de Tarjan conseguiu ser executado com sucesso, ainda que com tempos de resposta elevados em grafos com mais de 20.000 vértices.

5 CONCLUSÃO

Em resumo, os resultados dos experimentos indicam que o algoritmo de Tarjan é a melhor escolha para identificar componentes biconexos em grafos de grande escala. No entanto, sua performance também pode ser limitada em grafos extremamente grandes, exigindo melho-

rias na implementação ou abordagens alternativas. O algoritmo de articulações, embora funcional para grafos menores, não é adequado para grandes volumes de dados. Por fim, o algoritmo de caminhos internamente disjuntos, devido à sua complexidade fatorial, é inadequado e deve ser evitado, sendo usado somente para comparação.

REFERÊNCIAS

GeeksforGeeks. **Geeks for Geeks Website**. 2024. <https://www.geeksforgeeks.org/>. Accessed: 2024-10-06.

OPENAI. **ChatGPT: Language Model**. 2024. <https://openai.com/chatgpt>. Accessed: 2024-10-06.

TARJAN, Robert. Depth-first search and linear graph algorithms. **SIAM journal on computing**, SIAM, v. 1, n. 2, p. 146–160, 1972.