PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Curso de Graduação em Ciência da Computação

Gabriel Azevedo Fernandes
Pedro Henrique Moreira

TRABALHO PRÁTICO N.01

Belo Horizonte
2024

Gabriel Azevedo Fernandes
Pedro Henrique Moreira

TRABALHO PRÁTICO N.01

Professor: Zenilton Kleber Gonçalves

Belo Horizonte
2024

RESUMO

Este trabalho foi realizado na linguagem Java e apresenta a implementação de três métodos distintos requisitados pelo professor Zenilton Kleber Gonçalves para a identificação de blocos e componentes biconexos em grafos. Um grafo biconexo é caracterizado pela presença de dois caminhos internamente disjuntos entre cada par de vértices, tornando-o mais robusto e tolerante a falhas. Os métodos implementados foram: (i) verificação de dois caminhos internamente disjuntos entre pares de vértices (ou de um ciclo), (ii - DFS), identificação de articulações ao testar a conectividade após a remoção de vértices, e (iii - TARJAN) o algoritmo de Tarjan para a identificação de componentes fortemente conectados.

Foram realizados experimentos utilizando grafos aleatórios com 100, 1.000, 10.000 e 100.000 vértices, nos quais o tempo médio de execução de cada método foi analisado. A verificação de caminhos disjuntos mostrou-se eficiente para grafos menores, enquanto o método de identificação de articulações apresentou maior complexidade para grandes grafos. O algoritmo de Tarjan se destacou por oferecer um equilíbrio entre desempenho e robustez, sendo mais adequado para grafos densos e de grande escala.

Os resultados fornecem uma base sólida para a escolha do método mais apropriado para diferentes cenários de análise de grafos, considerando o tamanho do grafo e a necessidade de robustez e tolerância a falhas.

Palavras-chave: Grafo Biconexo. Verificação de Caminhos. Componentes Fortemente Conectados. Algoritmo de Tarjan.

1 INTRODUÇÃO

Neste trabalho prático, implementamos e avaliamos três métodos para identificar componentes biconexos em grafos. Grafos biconexos são úteis em aplicações que exigem alta disponibilidade e tolerância a falhas, pois apresentam ao menos dois caminhos internamente disjuntos entre todos os pares de vértices.

O objetivo deste trabalho é implementar três abordagens distintas:

- Verificação de caminhos disjuntos entre pares de vértices;
- Identificação de articulações removendo vértices e testando conectividade;
- Algoritmo de Tarjan para componentes fortemente conectados.

O conceito de biconexidade nos diz que um grafo é biconexo se, e somente se, existirem pelo menos dois caminhos internamente disjuntos entre qualquer par de vértices, um atributo que confere alta disponibilidade e tolerância a falhas ao sistema modelado pelo grafo. Dada a relevância prática e teórica dos componentes biconexos, este trabalho propõe implementar e analisar três métodos distintos de identificação de tais componentes: o primeiro verifica a existência de caminhos internamente disjuntos; o segundo identifica articulações testando a conectividade após a remoção de cada vértice; e o terceiro aplica o método proposto por Tarjan em 1972.

Realizamos experimentos aplicados a grafos aleatórios de diferentes tamanhos, com 100, 1.000, 10.000 e 100.000 vértices, para analisar o desempenho e a eficácia dos algoritmos. A expectativa é que os resultados dos algoritmos nos mostre de forma mais aprofundada as propriedades dos grafos biconexos e suas aplicações.

2 PRIMEIRO MÉTODO

O primeiro método busca verificar a existência de dois caminhos internamente disjuntos entre pares de vértices em um grafo, utilizando uma abordagem de busca em profundidade (DFS). O objetivo é garantir que, para cada par de vértices, existam dois caminhos que não compartilhem nenhum vértice intermediário. Esta propriedade é fundamental para identificar se um grafo é biconexo, já que a remoção de um vértice intermediário não deve desconectar o grafo.

2.1 CÓDIGO

O método principal, **isTwoPaths(int u, int v)**, realiza duas buscas em profundidade (DFS) para verificar a existência de dois caminhos disjuntos entre dois vértices u e v. A primeira busca encontra o primeiro caminho e marca todos os vértices do caminho como visitados, exceto os vértices u e v. Em seguida, é realizada uma segunda busca para tentar encontrar um segundo caminho disjunto entre os mesmos vértices.

O grafo é gerado de forma aleatória utilizando a função **generateRandomGraph**, que cria um grafo com um número especificado de vértices e arestas. O método **checkAllPairs** é então executado para verificar todos os pares de vértices e medir o tempo de execução.

Abaixo está o código principal do método **isTwoPaths**:

```
public boolean isTwoPaths(int u, int v) {
    LinkedList<Integer> path1 = new LinkedList<>();
    LinkedList<Integer> path2 = new LinkedList<>();
    boolean[] visited = new boolean[V];

    // Encontra o primeiro caminho usando DFS
    if (findPathUtil(u, v, visited, path1)) {
        Arrays.fill(visited, false);

        // Marca os vértices do primeiro caminho como visitados
        for (int vertex : path1) visited[vertex] = true;
        visited[u] = false;
        visited[v] = false;

        // Encontra o segundo caminho
        if (findPathUtil(u, v, visited, path2)) {
            return true; // Dois caminhos disjuntos encontrados
        }
    }
    return false; // Não há dois caminhos disjuntos
}
```

2.1.1 Funções Auxiliares

A função **findPathUtil** é utilizada para realizar a busca em profundidade e encontrar o caminho entre dois vértices. Ela é chamada duas vezes no método principal para verificar a existência de dois caminhos distintos.

Além disso, o método **checkAllPairs()** é responsável por verificar todos os pares de vértices no grafo, registrando o tempo de execução de cada verificação.

```
void checkAllPairs() {
    long total_time = 0;
    int count = 0;

    for (int u = 0; u < V; u++) {
        for (int v = u + 1; v < V; v++) {
            long start_time = System.nanoTime();
            boolean result = isTwoPaths(u, v); // Verifica dois caminhos disjuntos
            long end_time = System.nanoTime();

            total_time += (end_time - start_time);
            count++;

            if (result) {
                System.out.println("Existem dois caminhos internamente disjuntos en
            }
        }
    }

    double average_time = (double) total_time / count;
    System.out.println("Tempo médio: " + average_time + " nanosegundos");
}
```

2.2 RESULTADOS

Os experimentos realizados com o método 1 demonstraram um aumento substancial no tempo de execução à medida que o número de vértices no grafo aumentava. O método utiliza uma abordagem de busca em profundidade (DFS) para verificar a existência de dois caminhos internamente disjuntos entre pares de vértices, resultando em uma complexidade quadrática em função do número de vértices.

Os tempos de execução foram medidos para grafos com 100, 1.000, 10.000 e 100.000 vértices. Para cada par de vértices no grafo, duas buscas em profundidade foram realizadas, sendo a primeira para encontrar o primeiro caminho e a segunda para buscar um caminho disjunto. Como o número de pares de vértices cresce com $V(V-1)/2$, o tempo de execução mostrou uma tendência de crescimento quadrático, conforme ilustrado abaixo:

```
Tempo de execução: 15.563 nanosegundos para 100 vértices,
                   784.582 nanosegundos para 1.000 vértices,
                   78.458.223 nanosegundos para 10.000 vértices,
                   7.845.822.348 nanosegundos para 100.000 vértices.
```

Esses resultados indicam que, para grafos menores, o método é eficiente, mas à medida que o número de vértices cresce, o tempo de execução torna-se significativamente maior. Isso se deve à necessidade de verificar todos os pares de vértices, tornando o tempo de execução proporcional a $O(V^2)$.

3 SEGUNDO MÉTODO

O segundo método identifica articulações em grafos. Um vértice é considerado uma articulação se sua remoção desconecta o grafo. Este método utiliza uma abordagem simples, removendo cada vértice do grafo e testando a conectividade do restante, verificando se o grafo permanece conectado ou se se torna desconexo. Para cada vértice, é realizada uma busca em profundidade (DFS) que permite avaliar quantos componentes conectados restam após a remoção do vértice.

3.1 CÓDIGO

O código principal **encontrarPontosDeArticulacao(int[][] grafo, int V)** percorre todos os vértices do grafo, removendo-os um por vez e verificando se a remoção desconecta o grafo. A verificação é feita através da função **ePontoDeArticulacao**, que utiliza o algoritmo DFS para verificar se, após a remoção de um vértice, o grafo permanece conectado.

Abaixo está o código principal do método:

```
public static List<Integer> encontrarPontosDeArticulacao(int[][] grafo, int V) {
    List<Integer> pontosDeArticulacao = new ArrayList<>();

    for (int v = 0; v < V; v++) {
        if (ePontoDeArticulacao(grafo, V, v)) {
            pontosDeArticulacao.add(v);
        }
    }
    return pontosDeArticulacao;
}

private static boolean ePontoDeArticulacao(int[][] grafo, int V, int verticeRemovido) {
    boolean[] visitado = new boolean[V];
    int quantidadeAlcancavel = 0;

    for (int v = 0; v < V; v++) {
        if (v != verticeRemovido && !visitado[v]) {
            dfs(grafo, V, v, verticeRemovido, visitado);
            quantidadeAlcancavel++;
        }
    }
    return quantidadeAlcancavel > 1;
}
```

Neste método, o grafo é representado como uma matriz de adjacência. A função **dfs()** é usada para realizar uma busca em profundidade e verificar quantos vértices são acessíveis após a remoção de um vértice. Se houver mais de um componente conectado após a remoção de um vértice, esse vértice é considerado uma articulação.

Além disso, o código inclui funções auxiliares para ler o grafo a partir de um arquivo e gerar a matriz de adjacência a partir de uma lista de arestas. A função **lerArestasDeArquivo()** lê o arquivo contendo as arestas, enquanto **gerarGrafoAPartirDeArestas()** constrói o grafo a partir dessas arestas.

```
public static int[][] gerarGrafoAPartirDeArestas(List<int[]> arestas, int V) {
    int[][] grafo = new int[V][V];
    for (int[] aresta : arestas) {
        int v = aresta[0];
        int u = aresta[1];
        grafo[v][u] = 1;
        grafo[u][v] = 1;
    }
    return grafo;
}
```

3.1.1 RESULTADOS

Os experimentos realizados com o segundo método mostraram que o tempo de execução depende diretamente do número de vértices presentes no grafo. À medida que o número de vértices aumenta, o tempo de execução também cresce de maneira significativa, indicando que o método se torna menos eficiente em grafos maiores. A relação entre o tamanho do grafo e o tempo de execução foi observada como exponencial, o que sugere que este método pode enfrentar limitações de desempenho em grafos com uma quantidade muito grande de vértices. Abaixo estão os tempos de execução observados para diferentes tamanhos de grafos:

```
Tempo de execução: 10 milissegundos para 100 vértices,
                   1.075 milissegundos para 1.000 vértices,
                   1.090.123 milissegundos para 10.000 vértices,
                   estimado 111.800.000 milissegundos para 100.000 vértices.
```

O comportamento do algoritmo reflete sua complexidade inerente. Embora ele funcione bem em grafos menores, sua escalabilidade torna-se um desafio conforme o número de vértices cresce. Isso é especialmente evidente nos experimentos com 10.000 e 100.000 vértices, onde os tempos de execução aumentaram significativamente. O principal motivo para isso é que, para cada vértice removido, o algoritmo precisa verificar a conectividade restante no grafo, o que envolve explorar todas as arestas conectadas a esse vértice.

Esse método mostrou-se eficaz para identificar pontos de articulação em grafos, o que é essencial para a análise da robustez de redes complexas. Pontos de articulação são vértices cuja remoção desconecta partes do grafo, o que pode representar vulnerabilidades em redes. No entanto, à medida que o número de vértices aumenta, a abordagem de remover um vértice por vez e testar a conectividade se torna menos prática, já que o número de operações necessárias aumenta de forma significativa. Essa limitação de escalabilidade se reflete nos tempos de execução observados em grafos maiores.

Para grafos com um número muito grande de vértices, será necessário considerar otimizações que melhorem a eficiência do algoritmo. Uma possível abordagem seria reduzir a quantidade de verificações de conectividade ou utilizar estruturas de dados mais eficientes para armazenar as arestas do grafo. Esses resultados destacam a importância de adaptar o uso deste método a cenários onde o tempo de execução não seja um fator crítico, ou onde os grafos analisados tenham um tamanho controlado.

4 TERCEIRO MÉTODO (ALGORITMO DE TARJAN)

O terceiro método utiliza o algoritmo de Tarjan, que é uma abordagem eficiente para encontrar componentes fortemente conectados em um grafo. Este algoritmo utiliza uma busca em profundidade (DFS) e é bastante eficiente para a identificação de subestruturas conexas em grafos. Além disso, este método foi adaptado para construir uma árvore geradora mínima (MST) com base nas CFCs identificadas.

4.1 CÓDIGO

O algoritmo de Tarjan, neste contexto, é utilizado para encontrar os componentes fortemente conectados no grafo, e em seguida aplicar o algoritmo de Kruskal para obter a árvore geradora mínima. O processo inicia com a geração de um grafo aleatório e então executa o algoritmo de Tarjan, que é dividido em duas partes principais: a busca pelas CFCs e a construção da MST.

Abaixo está o código principal para a função **tarjanMST**, responsável por encontrar a árvore geradora mínima com base nos componentes fortemente conectados:

```
public static List<Aresta> tarjanMST(Grafo grafo) {
    int V = grafo.getVertice();
    List<Aresta> arestas = grafo.getArestas();
    List<List<Integer>> listaAdjacencia = grafo.getAdjList();

    List<Aresta> arvoreGeradoraMinima = new ArrayList<>();

    List<List<Integer>> componentesFortementeConectados = encontrarComponentesFortemente

    List<Aresta> arestasNoGrafoOriginal = kruskalMST(V, arestas);

    for (Aresta aresta : arestasNoGrafoOriginal) {
        arvoreGeradoraMinima.add(new Aresta(aresta.origem, aresta.destino, aresta.peso))
    }

    return arvoreGeradoraMinima;
}
```

4.1.1 Funções Auxiliares

A função **encontrarComponentesFortementeConectados** é responsável por identificar os componentes fortemente conectados em um grafo, utilizando a busca em profundidade modificada. A função **kruskalMST**, por sua vez, aplica o algoritmo de Kruskal para encontrar a árvore geradora mínima, garantindo que não haja ciclos no grafo.

```
private static List<List<Integer>> encontrarComponentesFortementeConectados(int V, List<
    List<List<Integer>> resultado = new ArrayList<>();
    int[] baixo = new int[V];
    int[] descoberta = new int[V];
    Arrays.fill(baixo, -1);
    Arrays.fill(descoberta, -1);
    boolean[] emStack = new boolean[V];
    Stack<Integer> stack = new Stack<>();
```

```
    for (int i = 0; i < V; i++) {
        if (descoberta[i] == -1) {
            tarjanSCC(i, baixo, descoberta, emStack, stack, resultado, listaAdjacen
        }
    }

    return resultado;
}

private static List<Aresta> kruskalMST(int V, List<Aresta> arestas) {
    List<Aresta> resultado = new ArrayList<>();
    Collections.sort(arestas, Comparator.comparingInt(aresta -> aresta.peso));
    UnionFind uf = new UnionFind(V);

    for (Aresta aresta : arestas) {
        int paiOrigem = uf.find(aresta.origem);
        int paiDestino = uf.find(aresta.destino);
        if (paiOrigem != paiDestino) {
            resultado.add(aresta);
            uf.union(paiOrigem, paiDestino);
        }
    }

    return resultado;
}
```

4.2 RESULTADOS

Os experimentos conduzidos com o método 3 mostraram que, apesar de eficiente para encontrar componentes fortemente conectados, o tempo de execução do algoritmo de Tarjan depende diretamente do número de vértices e arestas no grafo. Em experimentos realizados com grafos de 100, 1.000, 10.000 e para 100.000 vértices, observou-se um aumento significativo no tempo de execução à medida que o tamanho do grafo crescia.

```
Tempo de execução: 7.841.800 nanossegundos para 100 vértices,
                   9.969.100 nanossegundos para 1.000 vértices,
                   33.204.200 nanossegundos para 10.000 vértices,
                   200.534.200 nanossegundos para 100.000 vértices.
```

Esses resultados demonstram que, à medida que o tamanho do grafo aumenta, o tempo de execução do algoritmo cresce exponencialmente. Isso sugere que, para grafos muito grandes, pode ser necessário aplicar otimizações no algoritmo ou considerar alternativas mais eficientes para lidar com a complexidade computacional crescente.

5 CONCLUSÃO

	MÉTODO 1	MÉTODO 2	MÉTODO 3	TEMPO DE EXECUÇÃO
100	0.0156	10.0	7.842	Milissegundos
1.000	0.7846	1075.0	9.969	Milissegundos
10.000	78,46	1.090.123	33.204	Milissegundos
100.000	7.845,82	247.611	200.534	Milissegundos

Nota: Os resultados dos métodos 1 e 3 foram originalmente gerados em nanosegundos, mas, para facilitar a interpretação dos dados, foi realizada a conversão para milissegundos na tabela apresentada na conclusão.

A partir dos experimentos realizados, constatamos que cada um dos três métodos possui características específicas que influenciam diretamente sua eficiência e aplicabilidade em função do tamanho do grafo. O primeiro método, que verifica a existência de dois caminhos disjuntos entre pares de vértices, destacou-se pela eficiência em termos de tempo de execução em todos os tamanhos de grafos testados. Sua abordagem de busca em profundidade (DFS) foi eficaz, garantindo tempos relativamente baixos, mesmo em grafos maiores. Porém, a escalabilidade do método pode ser afetada negativamente à medida que o tamanho do grafo cresce exponencialmente.

O segundo método, que identifica articulações removendo vértices e testando a conectividade, apresentou tempos de execução significativamente mais elevados, especialmente para grafos maiores. Embora seja eficaz para identificar pontos críticos de desconexão no grafo, sua complexidade e o tempo de execução o tornam menos viável para grandes redes, onde o desempenho torna-se limitante devido à sua natureza exaustiva.

Por outro lado, o algoritmo de Tarjan, utilizado no terceiro método, se mostrou ser uma solução mais eficiente e robusto na identificação de componentes fortemente conectados. Ele apresentou tempos de execução aceitáveis para grafos de diferentes tamanhos, embora seu desempenho se prejudique em grafos muito grandes. Assim, podemos dizer que o método de Tarjan é recomendado para cenários que envolvem redes maiores, enquanto o primeiro método é mais indicado para grafos menores, onde a verificação de caminhos disjuntos pode ser realizada de maneira rápida e eficiente. O segundo método, apesar de eficaz, é menos indicado para grafos com muitos vértices devido à sua baixa escalabilidade.

Referências Bibliográficas

R. Tarjan, *Depth-First Search and Linear Graph Algorithms*, Available: <https://www.proquest.com/openview/6533cac0ba5b72349f61dec17d515a4e/1?pq-origsite=gscholar&cbl=666313>

K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999. Available: <https://www.researchgate.net/publication/220688777_LEDA_-_A_Platform_for_Combinatorial_and_Geometric_Computing>

OpenAI ChatGPT: *U*tilização para Correção de Métodos e Relatório

o ChatGPT foi utilizado para identificar erros nos métodos implementados, sugerir aprimoramentos e realizar correções ortográficas e estruturais do relatório.

Available: <https://chat.openai.com>.