A B A K Ó S
PUC Minas
Instituto de Ciências Exatas e Informática

Revista Abakós do Instituto de Ciências Exatas e de Informática Pontifícia Universidade Católica de Minas Gerais*

Model - Magazine Abakós - ICEI - PUC Minas

Rafael Duarte Pereira[1]

Resumo

Algoritmos baseados em grafos são usados em diversas áreas para auxiliar nas resoluções de inúmeros problemas. Um caminho simples em um grafo direcionado é um caminho onde não há repetição de vértices Em grafos simples, pode-se representar um caminho apenas pela sequência de vértices (uma vez que só pode existir uma única aresta entre cada par de vértices). Encontrar caminhos disjuntos em um grafos simples é um problemas com mais de uma solução possível. O presente trabalho irá apresentar a implementação de uma solução que apresenta o máximo de caminhos disjuntos em um grafo direcionado simples.

Palavras-chave: Vetices. Caminhos. Grafo. Método. Implementação

---

*Artigo apresentado à Revista Abakos
[1]Graduação em Engenharia de Software da PUC Minas, Brasil– rafael.pereira.1296852@sga.pucminas.br

1 IMPLEMENTAÇÃO

Para a implementação da solução foi utilizado uma adaptação do algoritmo de fluxo máximo Ford-Fulkerson (LIMA, 2022) onde foi definido o fluxo máximo como 1. Para cada rede de fluxo gerada faz-se uma busca em largura partindo do vértice s (vértice definido como origem do grafo) armazenamento em um vetor quais são os pais de cada vértice. Uma vez finalizada a busca precorre o vetor de pais partindo do vértice t (vértice definido como destino do grafo) adicionando o pai de cada vértice no começo de uma lista para definir qual foi o caminho percorrido pela busca. O método se repete até que a busca não consiga encontrar o vértice t e retorna um conjunto de de possíveis caminhos disjuntos de s para t.

O método foi implementado em Java utilizando o paradigma da orientação à objetos. Nesta implementação o grafo é um objeto que armazena em seu interior uma matriz de adjacência, o vértice de origem (s) e o vértice de destino (t), além do número total de vértices. A figura 1 mostra a implementação do código responsável por armazenar um grafo

Figura 1 – Implementação da classe grafo

```java
public class Grafo {

    private int[][] matriz;
    private int numVertices;
    private int s;
    private int t;

     * Cria um grafo sem arestas
    public Grafo(int numVertices, int s, int t) {
        this.numVertices = numVertices;
        this.s = s;
        this.t = t;
        this.matriz = new int[numVertices][numVertices];

        for (int i = 0; i < numVertices; i++)
            for (int j = 0; j < numVertices; j++)
                matriz[i][j] = 0;
    }

     * Adiciona no grafo uma aresta na direção (v,w)
    public boolean addAresta(int v, int w) {
        boolean resp = matriz[v][w] == 0;
        if (resp)
            matriz[v][w] = 1;
        return resp;
    }

    public int[][] getMatriz() {
        return this.matriz;
    }

    public int getS() {
        return this.s;
    }

    public int getT() {
        return this.t;
    }

    public int getNumVertices() {
        return this.numVertices;
    }

}
```

A parte responsável por encontrar os caminhos conta com dois métodos, uma busca em largura que passa uma única vez em um vértice e que retorna um vetor indicando os pais de um vértice (Figura 2) e outro que a partir do vetor retornado pela busca monta os caminhos e gera uma nova rede de fluxo para realizar outra busca. O método só para quando a busca não encontra um caminho para t e retorna os caminhos encontrados (Figura 3).

Figura 2 – Implementação da busca em largura

```java
    private int[] buscaLargura(int grafo[][]) {
        // inicia o vetor de pais com todos nulos
        int[] pais = new int[numVert];
        for (int i = 0; i < pais.length; i++) {
            pais[i] = NULL;
        }
        boolean[] visitado = new boolean[numVert];

        // cria a fila e adiciona o vértice s
        Queue<Integer> q = new LinkedList<>();
        q.add(s);
        visitado[s] = true;

        while (!q.isEmpty()) {
            int u = q.remove();
            for (int v = 0; v < numVert; v++) {
                // verifica se tem aresta e se o vértice já foi visitado
                if (grafo[u][v] > 0 && visitado[v] == false) {
                    q.add(v);
                    pais[v] = u;
                    visitado[v] = true;
                }
            }
        }

        return pais;
    }
```

Figura 3 – Implementação do método de encontrar caminhos

```java
    public Set<List<Integer>> encontrarCaminhos() {
        Set<List<Integer>> resp = new HashSet<>();
        int[][] matriz = grafo.getMatriz();
        int u, v;

        // Copia o grafo original para um novo grafo com a rede residual
        int[][] grafoResidual = new int[numVert][numVert];
        for (u = 0; u < numVert; u++)
            for (v = 0; v < numVert; v++)
                grafoResidual[u][v] = matriz[u][v];

        // Itera enquanto houver caminho de s para t
        for (int[] pais = buscaLargura(grafoResidual); pais[t] != NULL; pais = buscaLargura(grafoResidual)) {
            List<Integer> path = new ArrayList<>();
            // gera o caminho passando pelo vetor de pais a partir do vertice t
            for (int i = t; i != NULL; i = pais[i])
                path.add index: 0, element: i);

            // gera nova rede residual a partir dos vértices no caminho
            for (v = t; v != s; v = pais[v]) {
                u = pais[v];
                grafoResidual[u][v] -= MAX_FLOW;
                grafoResidual[v][u] += MAX_FLOW;
            }

            // adiciona o caminho à solução
            resp.add(path);
        }

        return resp;
    }
```

2 TESTES E RESULTADOS

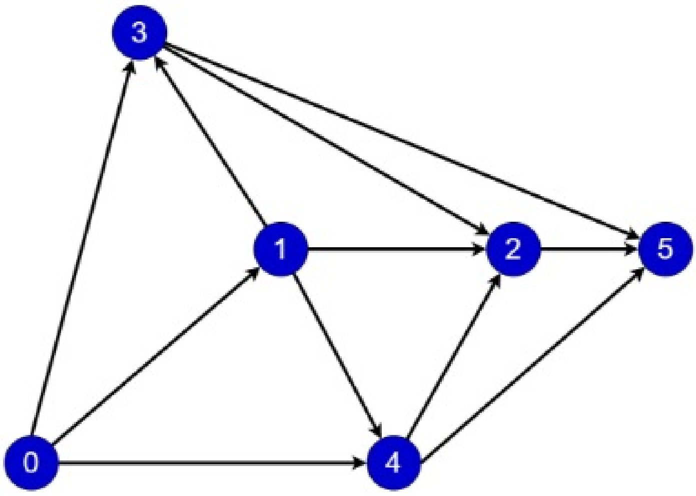
Para os testes foram utilizados três grafos direcionados simples, isto é sem ciclos e sem arestas paralelas. Os grafos estavam salvos em aquivos de texto com a seguinte estrutura: na primeira linha o numero de vértices, o vértice de origem e o vértice de destino na respectiva ordem separados por espaços e nas linhas subsequentes estava as arestas, uma em cada linha, com vértice de origem e vértice de destino nessa respectiva ordem separadas por espaços.

Os grafos testados foram os seguintes

2.1 Grafo 1

O grafo G1 apresenta seis vértices e 11 arestas e os caminhos a serem procurados pelo algoritmo partiam do vértice 0 ao vértice 5 conforme a Figura 4.

Figura 4 – G1



Para tal grafo existem no máximo três caminhos disjuntos conforme está marcado na Figura 5 e, o método foi capaz de encontrar os três caminhos conforme indicado na Figura 6 extraída do console da IDE

Figura 5 – Caminhos disjuntos de G1

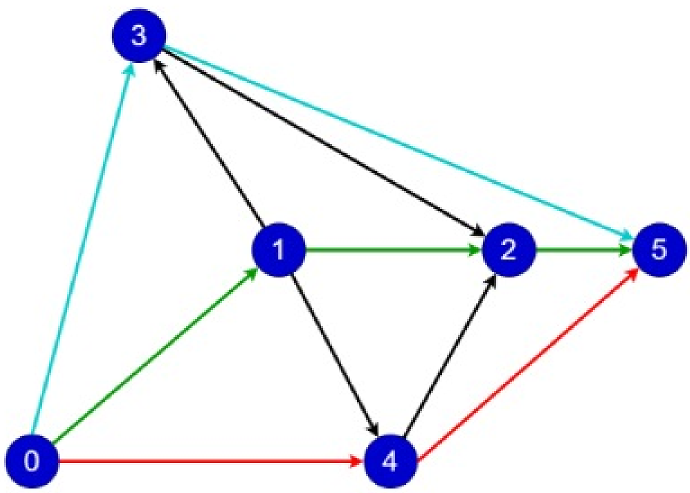


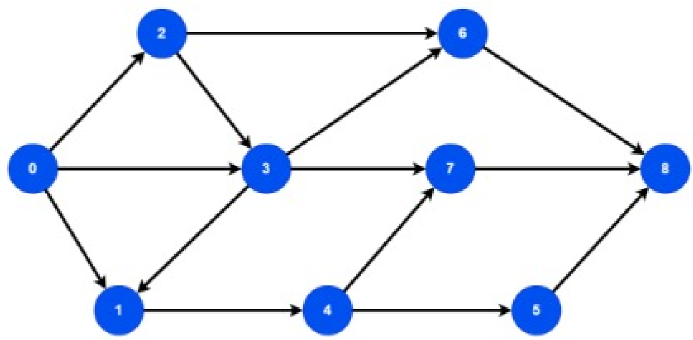
Figura 6 – Resultados de G1

```
Numero maximo de caminhos = 3
Possiveis caminhos
        0 4 5
        0 3 5
        0 1 2 5
```

2.2 Grafo 2

O grafo G2 apresenta nove vértices e 13 arestas e os caminhos a serem procurados pelo algoritmo partiam do vértice 0 ao vértice 8 conforme a Figura 7

Figura 7 – G2



Para tal grafo existem no máximo três caminhos disjuntos conforme está marcado na Figura 8 e, o método foi capaz de encontrar os três caminhos conforme indicado na Figura 9 extraída do console da IDE

Figura 8 – Caminhos disjuntos de G2

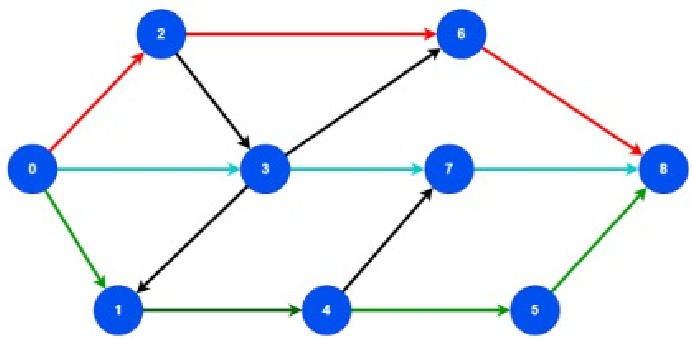
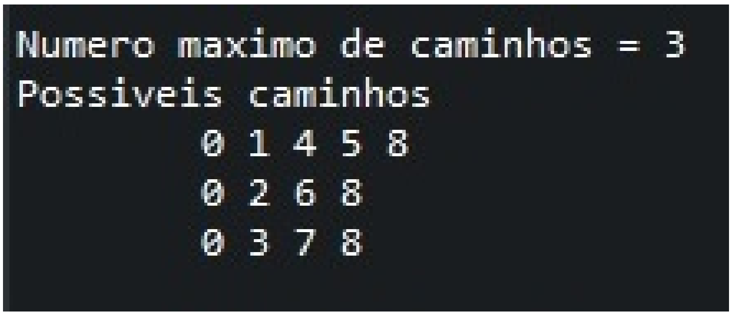


Figura 9 – Resultados de G2

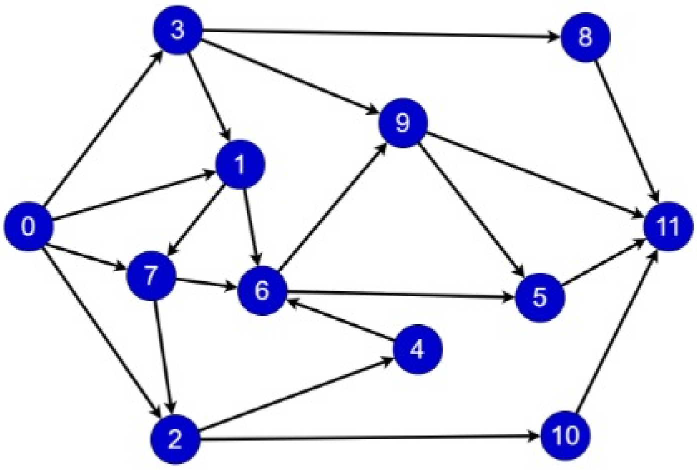


```
Numero maximo de caminhos = 3
Possiveis caminhos
        0 1 4 5 8
        0 2 6 8
        0 3 7 8
```

2.3 Grafo 3

O grafo G3 apresenta 12 vértices e 21 arestas e os caminhos a serem procurados pelo algoritmo partiam do vértice 0 ao vértice 11 conforme a Figura 10

Figura 10 – G3



Para tal grafo existem no máximo três caminhos disjuntos conforme está marcado na Figura 11 e, o método foi capaz de encontrar os três caminhos conforme indicado na Figura 12 extraída do console da IDE

Figura 11 – Caminhos disjuntos de G3

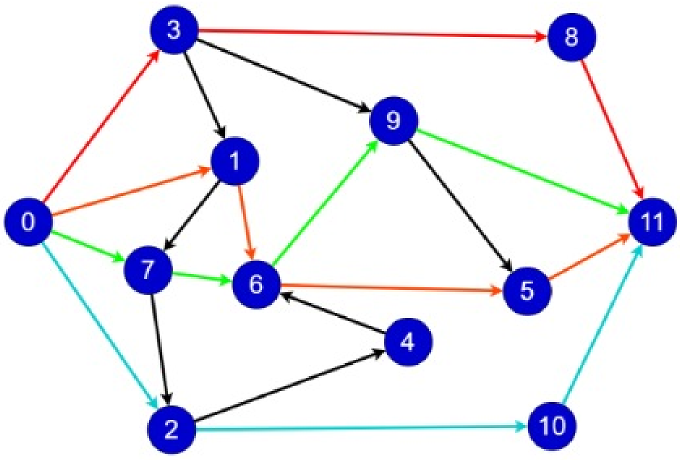
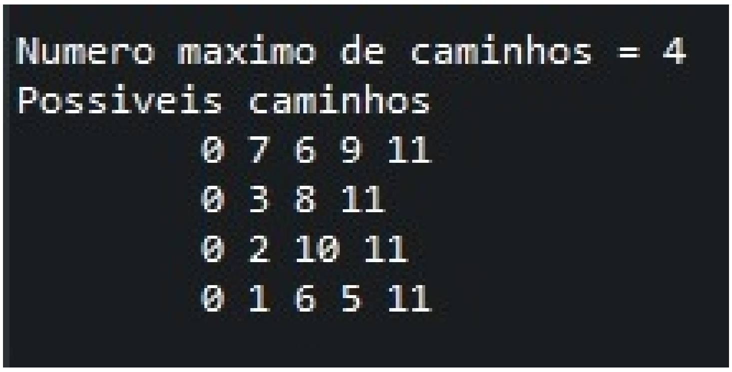


Figura 12 – Resultados de G3



```
Numero maximo de caminhos = 4
Possiveis caminhos
        0 7 6 9 11
        0 3 8 11
        0 2 10 11
        0 1 6 5 11
```

3 CONCLUSÃO

Pode-se concluir que a solução implementada é capaz de achar possíveis caminhos disjuntos em grafos simples com um custo no pior caso de O(n), ou seja, o algoritmo é linear ao número de vértices de um grafo.

Referências

LIMA, Acervo. **ENCONTRE O NÚMERO MÁXIMO DE CAMINHOS DISJUNTOS DE ARESTAS ENTRE DOIS VÉRTICES**. 2022. Urlhttps://acervolima.com/encontre-o-numero-maximo-de-caminhos-disjuntos-de-arestas-entre-dois-vertices/.