PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Bacharelado em Engenharia de Software

Gabriel Lima de Souza
Gabriel de Souza
Nikolas Augusto Vieira Louret
Lucas Picinin Campos Lutti

Bridge Finding

Belo Horizonte

2022

Gabriel Lima de Souza
Gabriel de Souza
Nikolas Augusto Vieira Louret
Lucas Picinin Campos Lutti

Bridge Finding

Trabalho prático realizado na disciplina Teoria dos Grafos e Computabilidade do curso de Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais.

Belo Horizonte

2022

RESUMO

Este relatório apresenta uma biblioteca desenvolvida em Java para manipulação e representação de grafos e a implementação do algoritmo de Fleury, utilizado para encontrar caminhos eulerianos em um grafo implementados por meio de dois algoritmos distintos para a identificação de pontes.

Palavras-chave: Grafos, Java

SUMÁRIO

1 INTRODUÇÃO

O seguinte relatório tem como objetivo relatar os resultados obtidos após a implementação de uma biblioteca para representação e manipulação de grafos, uma estrutura de dados utilizada na matématica e na computação para representação abstrata de um conjunto de objetos e das relações existentes entre eles.

2 IMPLEMENTAÇÃO DO GRAFO

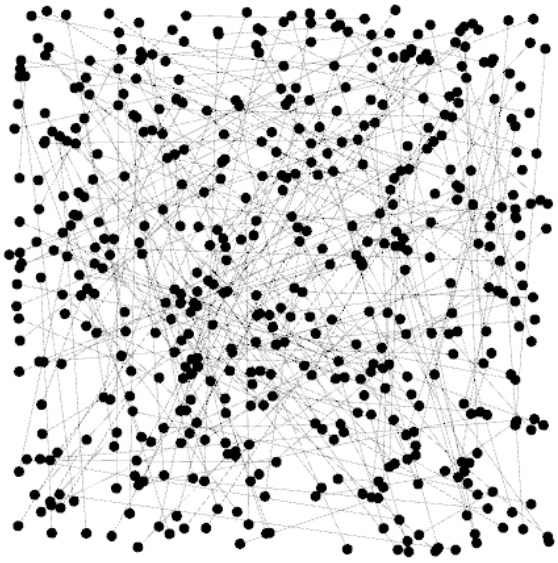
No projeto realizado foi utilizada a programação orientada a objetos implantadas em Java para a manipulação dos grafos, para isso foram desenvolvidas as seguintes classes:

1) **Vertice** - Contém as definições e atributos de vértices.
2) **Aresta** - Contém as definições e atributos de arestas.
3) **Grafo** - Classe abstrata contendo definições gerais de um grafo.
4) **GrafoCompleto** - Subclasse de **Grafo** utilizada para criar grafos completos.
5) **GrafoMutavel** - Subclasse abstrata de **Grafo** contendo definições de grafos mutáveis.
6) **GrafoNaoPonderado** - Subclasse de **GrafoMutavel** utilizada para criar grafos não ponderados.
7) **GrafoPonderadoRotulado** - Subclasse de **GrafoMutavel** utilizada para criar grafos ponderados e rotulados.
8) **Algorithms** - Classe contendo métodos que formam o Algoritmo de Tarjan.
9) **ABB** - Implantação de uma árvore binária de busca.
10) **Lista** - Implantação de uma lista encadeada.
11) **Arquivo** - Implantação de funções para manipulação de arquivos.
12) **App** - Classe *main*, onde ocorre os testes e instanciações do grafo.

Na biblioteca implantada, os grafos podem ser representados por árvores binárias de busca e por matrizes de adjacência. Ambas as representações estão definidas na superclasse **Grafo**.

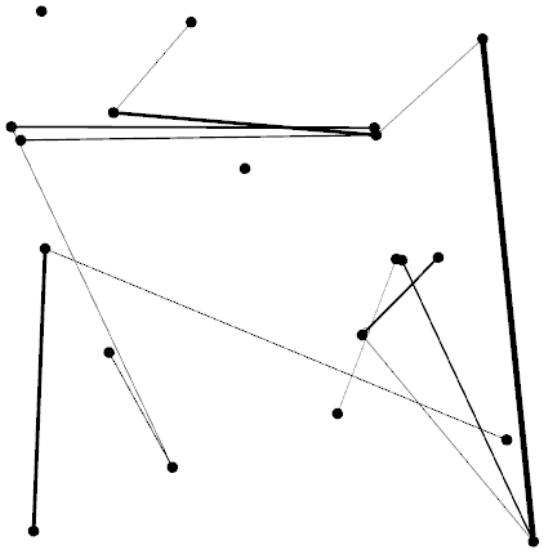
Além disso, foi implantada também a funcionalidade de salvar e carregar em arquivos no formato Pajek Net compatível com o *software* de visualização de grafos Gephi. A Figura 1 ilustra um exemplo de grafo com 498 vértices e 249 arestas criados pela biblioteca implantada e representada por meio de arquivo texto no *software* Gephi.

Figura 1 – Grafo com 498 vértices e 249 arestas



A Figura 2 exemplifica a funcionalidade de ponderação de arestas implantadas na biblioteca, representadas graficamente por arestas com diferentes espessuras no Gephi.

Figura 2 – Grafo ponderado com 20 vértices e 15 arestas



3 BRIDGE FINDING

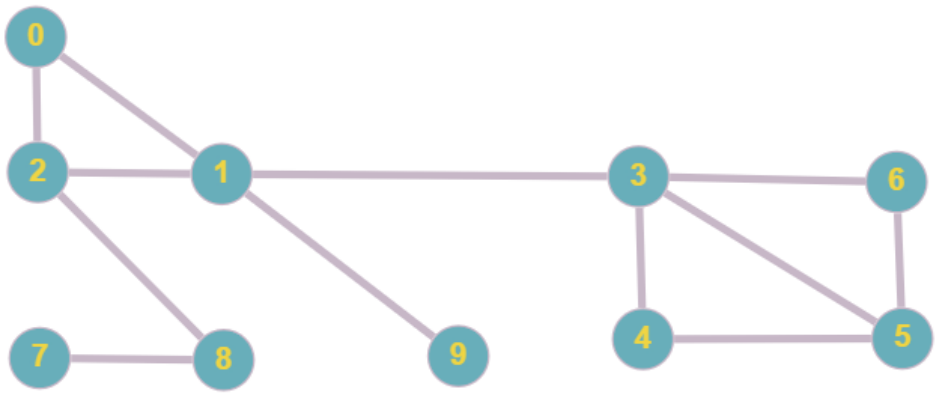
Dois algoritmos foram propostos para identificação de pontes no grafo:

3.1 Algoritmo de Tarjan

O funcionamento do algoritmo de Tarjan para encontrar pontes em um grafo ocorre a partir da escolha de um vértice de origem (VO) sem critérios. Com isso é criada uma lista de ancestrais em que o VO é o primeiro item dessa lista. Enfim, é feita uma busca em profundidade para identificar as pontes. Essa busca em profundidade visita todos os vértices adjacentes (VA) ao VO, que também são submetidos a busca em profundidade. No decorrer da busca, se um VA ainda não foi visitado, faz-se uma busca em profundidade nele. Além disso, é verificado se a aresta desse VA conectada com o VO é ponte, caso sim esta é adicionada na lista de pontes. Após isso, o VA é marcado como "pai" de VO e é adicionado na lista de ancestrais. Caso o VA já tenha sido visitado, o ancestral com o menor tempo de descobrimento é marcado como "pai" de VO.

A Figura 3 ilustra um grafo não ponderado onde o algoritmo de Tarjan foi testado.

Figura 3 – Grafo com ponte



A Figura 4 exemplifica a montagem do grafo da Figura 3 pela biblioteca implantada. A Figura 5 ilustra o resultado obtido pela execução do algoritmo de Tarjan, onde as arestas $\{1, 9\}$, $\{1, 3\}$, $\{7, 8\}$ e $\{2, 8\}$ foram identificadas como pontes.

Figura 4 – Codificação do grafo

```
GrafoNaoPonderado grafoNaoPonderadoPonte = new GrafoNaoPonderado("GrafoPonte");

grafoNaoPonderadoPonte.addVertice(0);
grafoNaoPonderadoPonte.addVertice(1);
grafoNaoPonderadoPonte.addVertice(2);
grafoNaoPonderadoPonte.addVertice(3);
grafoNaoPonderadoPonte.addVertice(4);
grafoNaoPonderadoPonte.addVertice(5);
grafoNaoPonderadoPonte.addVertice(6);
grafoNaoPonderadoPonte.addVertice(7);
grafoNaoPonderadoPonte.addVertice(8);
grafoNaoPonderadoPonte.addVertice(9);

grafoNaoPonderadoPonte.addAresta(0, 1);
grafoNaoPonderadoPonte.addAresta(0, 2);
grafoNaoPonderadoPonte.addAresta(1, 2);
grafoNaoPonderadoPonte.addAresta(1, 9);
grafoNaoPonderadoPonte.addAresta(1, 3);
grafoNaoPonderadoPonte.addAresta(2, 8);
grafoNaoPonderadoPonte.addAresta(3, 6);
grafoNaoPonderadoPonte.addAresta(3, 5);
grafoNaoPonderadoPonte.addAresta(3, 4);
grafoNaoPonderadoPonte.addAresta(4, 5);
grafoNaoPonderadoPonte.addAresta(5, 6);
grafoNaoPonderadoPonte.addAresta(7, 8);
```

Figura 5 – Console da aplicação

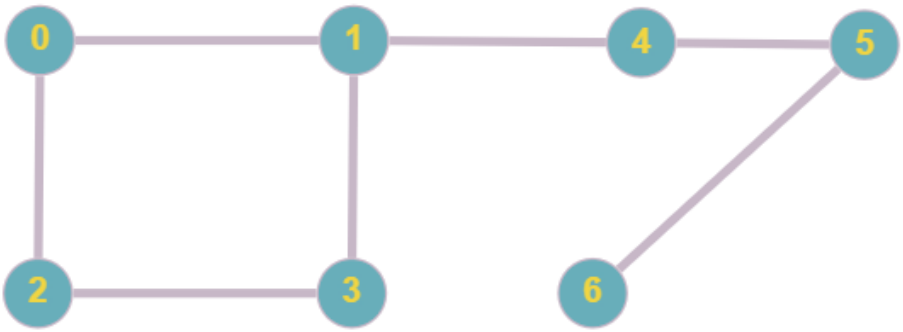
```
Pontes identificadas:

1 - 9
1 - 3
2 - 8
3 - 1
7 - 8
8 - 2
8 - 7
9 - 1
```

3.2 Algoritmo utilizando matriz de adjacência

Esse algoritmo tem como entrada um identificador do vértice de origem e do vértice de destino, uma representação em matriz de adjacência do grafo é gerada, ela é percorrida até que se encontre a posição dos vértices passados como parâmetro, ao encontrá-los a posição da matriz é zerada e uma função é chamada para verificar a quantidade de componentes do grafo após a alteração, essa verificação é realizada por meio de uma busca em profundidade. Se o número de componentes do grafo for maior do que 1, temos a conclusão que a aresta verificada é ponte. Após a verificação, é realizado o retorno da matriz aos valores iniciais.

A Figura 6 ilustra um grafo não ponderado onde o algoritmo de matriz de adjacência foi testado.

Figura 6 – Grafo com ponte



A Figura 7 exemplifica a montagem do grafo da Figura 6 pela biblioteca implantada. A Figura 5 ilustra o resultado obtido pela execução do método "ePonte", onde as arestas $\{1,4\}, \{4,5\}\, e\, \{5,6\}$ foram identificadas como pontes.

Figura 7 – Codificação do grafo

```
GrafoNaoPonderado grafoNaoPonderadoPonte = new GrafoNaoPonderado("GrafoPonte");

grafoNaoPonderadoPonte.addVertice(0);
grafoNaoPonderadoPonte.addVertice(1);
grafoNaoPonderadoPonte.addVertice(2);
grafoNaoPonderadoPonte.addVertice(3);
grafoNaoPonderadoPonte.addVertice(4);
grafoNaoPonderadoPonte.addVertice(5);
grafoNaoPonderadoPonte.addVertice(6);

grafoNaoPonderadoPonte.addAresta(0, 1);
grafoNaoPonderadoPonte.addAresta(0, 2);
grafoNaoPonderadoPonte.addAresta(1, 4);
grafoNaoPonderadoPonte.addAresta(1, 3);
grafoNaoPonderadoPonte.addAresta(2, 3);
grafoNaoPonderadoPonte.addAresta(4, 5);
grafoNaoPonderadoPonte.addAresta(5, 6);
```

Figura 8 – Console da aplicação

```
O vértice 1-4 é ponte? - true
O vértice 4-5 é ponte? - true
O vértice 5-6 é ponte? - true
O vértice 2-3 é ponte? - false
```

4 ALGORITMO DE FLEURY

O Algoritmo de Fleury foi implantado em duas estratégias com o objetivo de identificar ciclos eulerianos em grafos eulerianos. Foi utilizado um contador de tempo de execução das duas estratégias implantadas, para isso, foram utilizados dois grafos eulerianos como exemplo.

Figura 9 – Grafo Euleriano com 7 vértices

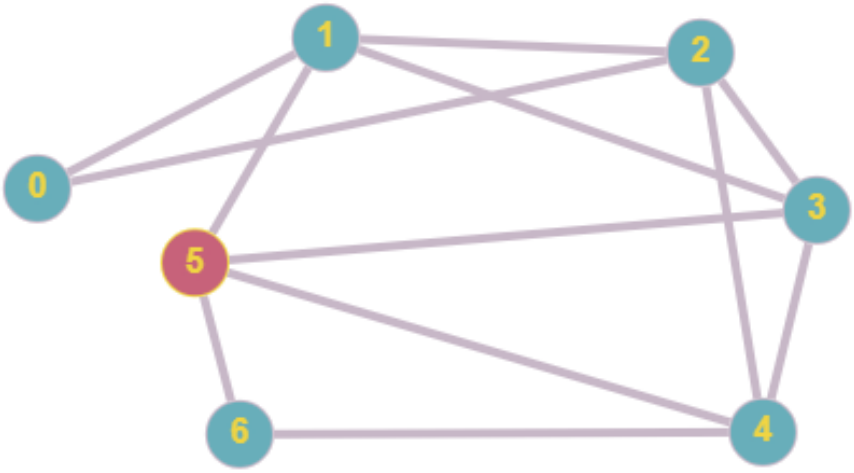
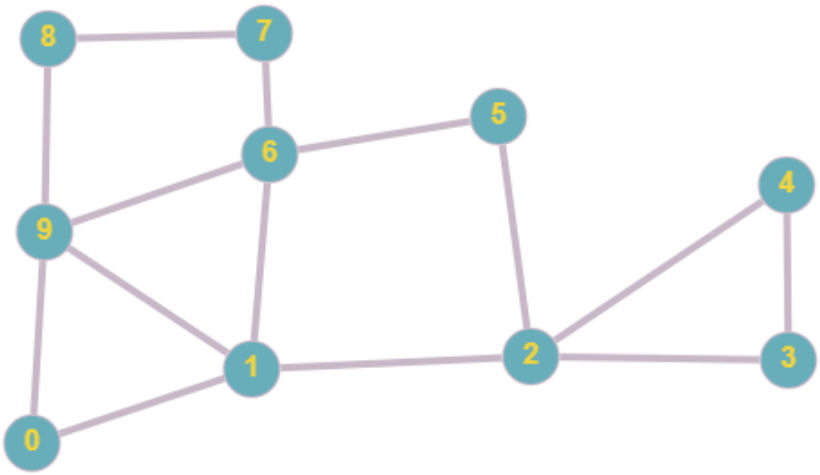


Figura 10 – Grafo Euleriano com 10 vértices



Os dois algoritmos alcançaram o mesmo resultado na identificação do caminho euleriano, o caminho identificado no grafo de 7 vértices foi: 0 1 2 3 1 5 3 4 5 6 4 2 0. E o caminho identificado no grafo de 10 vértices foi: 0 1 2 3 4 2 5 6 1 9 6 7 8 9 0.

4.1 Algoritmo de Fleury com identificação de pontes pelo método de matriz de adjacência

Primeiro, foi desenvolvido o Algoritmo de Fleury com o método de identificação de pontes por matriz de adjacência apresentado no capítulo anterior. A execução do algoritmo no grafo de sete vértices ocorreu em 143 milissegundos e no grafo de dez vértices em 56 milissegundos.

Figura 11 – Codificação do método de Fleury com Matriz

```
public List<Vertice> fleury(){
    int controle = 0;
    List<Vertice> caminho = new ArrayList<Vertice>();
    int[][] matriz = new int[vertices.size() + 1][vertices.size() + 1];
    Vertice selecionado = this.existeVertice(0);
    Vertice primeiro = this.existeVertice(0);
    matriz = matrizFormatada(this.matrizAdjacencia());
    Grafo teste = new Grafo("nada") {

    @Override
    public Grafo subGrafo(Lista<Vertice> vertices) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }

    };

    try {
        teste = (Grafo) this.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    while(!this.matrizVazia(matriz)){
        if(arestaRestanteVerticeMatriz(matriz, selecionado.getId(),controle) != null){
            Aresta atual = arestaRestanteVerticeMatriz(matriz, selecionado.getId(), controle);
            if(teste.ePonte(atual.getOrigem(), atual.getDestino()) && quantidadeArestaDeUmVerticeMatriz(matriz, atual.getOrigem() ) ==controle+1){
                caminho.add(selecionado);
                Vertice aux = this.existeVertice(atual.getDestino());
                matriz[atual.getOrigem()][atual.getDestino()]=0;
                matriz[atual.getDestino()][atual.getOrigem()]=0;
                teste.removeAresta(atual.getOrigem(), atual.getOrigem());
                teste.removeAresta(atual.getDestino(), atual.getOrigem());
                selecionado=aux;
                controle=0;
                if(this.matrizVazia(matriz) && selecionado==primeiro){
                    caminho.add(primeiro);
                }
            }else{
                if(!teste.ePonte(atual.getOrigem(), atual.getDestino()) && teste.existeAresta(atual.getOrigem(),atual.getDestino()) != null){
                    caminho.add(selecionado);
                    Vertice aux = this.existeVertice(atual.getDestino());
                    matriz[atual.getOrigem()][atual.getDestino()]=0;
                    matriz[atual.getDestino()][atual.getOrigem()]=0;
                    teste.removeAresta(atual.getOrigem(), atual.getOrigem());
                    teste.removeAresta(atual.getDestino(), atual.getOrigem());
                    selecionado=aux;
                    controle=0;
                }else{
                    controle++;
                }
            }
        }
    }

    return caminho;
}
```

4.2 Algoritmo de Fleury com identificação de pontes pelo método de Tarjan

Segundamente, foi implantado o Algoritmo de Fleury com o método de identificação de pontes criado por Tarjan, também apresentado no capítulo anterior. Sua execução no grafo de sete vértices ocorreu em 28 milissegundos e no grafo de dez vértices em 6 milissegundos.

Figura 12 – Codificação do método de Fleury com Tarjan

```
public List<Vertice> fleuryTarjan(){
    int controle = 0;
    List<Vertice> caminho = new ArrayList<Vertice>();
    int[][] matriz = new int[vertices.size() + 1][vertices.size() + 1];
    Vertice selecionado = this.existeVertice(0);
    Vertice primeiro = this.existeVertice(0);
    matriz = matrizFormatada(this.matrizAdjacencia());
    Grafo teste = new Grafo("nada") {

    @Override
    public Grafo subGrafo(Lista<Vertice> vertices) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }

    };

    try {
        teste = (Grafo) this.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    while(!this.matrizVazia(matriz)){
        if(arestaRestanteVerticeMatriz(matriz, selecionado.getId(),controle) != null){
            Aresta atual = arestaRestanteVerticeMatriz(matriz, selecionado.getId(), controle);
            if(this.tarjan().contains(atual) && quantidadeArestaDeUmVerticeMatriz(matriz, atual.getOrigem() ) ==controle+1){
                caminho.add(selecionado);
                Vertice aux = this.existeVertice(atual.getDestino());
                matriz[atual.getOrigem()][atual.getDestino()]=0;
                matriz[atual.getDestino()][atual.getOrigem()]=0;
                teste.removeAresta(atual.getOrigem(), atual.getOrigem());
                teste.removeAresta(atual.getDestino(), atual.getOrigem());
                selecionado=aux;
                controle=0;
                if(this.matrizVazia(matriz) && selecionado==primeiro){
                    caminho.add(primeiro);
                }
            }else{
                if(!this.tarjan().contains(atual) && teste.existeAresta(atual.getOrigem(),atual.getDestino()) != null){
                    caminho.add(selecionado);
                    Vertice aux = this.existeVertice(atual.getDestino());
                    matriz[atual.getOrigem()][atual.getDestino()]=0;
                    matriz[atual.getDestino()][atual.getOrigem()]=0;
                    teste.removeAresta(atual.getOrigem(), atual.getOrigem());
                    teste.removeAresta(atual.getDestino(), atual.getOrigem());
                    selecionado=aux;
                    controle=0;
                }else{
                    controle++;
                }
            }
        }
    }

    return caminho;
}
```

4.3 Conclusões

Com os resultados obtidos, podemos chegar a conclusão que o algoritmo de Tarjan foi mais eficiente que o de matriz de adjacência, sendo executado em média, 5 vezes mais rápido no exemplo da Figura 9 e 9 vezes mais rápido no exemplo da Figura 10.

5 OBSERVAÇÕES IMPORTANTES

a) Arquivo .net

Os arquivos foram salvos em formato .net para possibilitar a utilização do visualizador de grafos Gephi, e estão armazenados em */codigo/app/files*.

b) Execução da biblioteca

Para a execução da biblioteca criada, o arquivo App.java localizado em */codigo/app/App.java* deve ser compilado e interpretado por uma Máquina Virtual Java (JVM).

c) Localização dos algoritmos

Os Algoritmos de Fleury estão contidos na classe abstrata Grafo.java, juntamente com o método de identificação de pontes por matriz de adjacência. Os métodos utilizados para formar o Algoritmo de Tarjan estão localizados na classe Algorithms.java.

c) Classes que podem ser instanciadas

As classes Grafo e GrafoMutavel são abstratas e por isso não podem ser instanciadas. Para criar um grafo ponderado ou rotulado instancie a classe GrafoPonderadoRotulado, parar criar um grafo não ponderado utilize a GrafoNaoPonderado e para criar um grafo completo a classe GrafoCompleto. Todas as instanciações devem ser realizadas na classe App.java, conforme o item ***B*** desse capítulo.

6 REFERÊNCIAS

Bridges in a Graph. **Javatpoint**. Disponível em: https://www.javatpoint.com/bridges-in-a-graph. Acesso em: 26, novembro de 2022.

Articulation points and bridges (Tarjan's Algorithm). **Codeforces**. Disponível em: https://codeforces.com/blog/entry/71146. Acesso em: 29, novembro de 2022.