

## PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Bacharelado em Engenharia de Software

Breno Rosa Almeida Gabriel Victor Couto Martins de Paula Luís Antônio de Souza e Sousa Rúbia Coelho de Matos

**Bridge finding** 

Belo Horizonte

# Breno Rosa Almeida Gabriel Victor Couto Martins de Paula Luís Antônio de Souza e Sousa Rúbia Coelho de Matos

## **Bridge finding**

Projeto de Bridge finding apresentado na disciplina Teoria dos Grafos e Computabilidade do curso de Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais.

# 1 INTRODUÇÃO

O trabalho apresenta o desenvolvimento da biblioetca de manipulação de grafos tendo em vista a representação da Matriz e Lista de Adjacência. A manipulação foi realizado através dos algoritmos de Fleury e Tarjan teve objetivo mostrar todos os conceitos da teoria de Grafos e como é exibido o grafo com manipulação de n vertices.

#### 2 DESENVOLVIMENTO

### 2.1 Biblioteca de manipulação de grafos

Como primeira etapa foi desenvolvida uma biblioteca de manipulação de grafos que será utilizada futuramente nas próximas etapas.

A biblioteca de manipulação é composta por um conjunto de funções e métodos que manipulam tanto um grafo representado em matriz, quanto um grafo representado por lista de adjacência.

Para um bom design de solução, que fosse reutilizável e compreensível, foi utilizada a linguagem Java como principal ferramenta. A arquitetura desenvolvida para a biblioteca, nos dá a liberdade para escolher qual estrutura de dados manipular e utilizar de módulos de geração, leitura e salvamento de grafos de maneira separada.

A Biblioteca como um todo é composta por duas classes principais que guardam as estruturas de dados de Grafos em matriz e em lista de adjacência. A primeira classe é chamada de GraphMatrix, responsável por manipular e armazenar um grafo não-direcionado em uma matriz de N linhas por N colunas tal que N é o número de vértices do grafo. A segunda classe se chama apenas Graph e ela herda todas as características de GraphMatrix. Graph, é responsável por manipular e armazenar tanto grafo em matriz, quanto o grafo em lista de adjacência.

Além das classes principais, existem outros dois módulos que auxiliam nos testes e desenvolvimento. Sendo o primeiro deles chamado de GraphIO, utilizado na leitura e salvamento de grafos em arquivos no formato Pajek NET, e o segundo módulo, chamado de GraphGenerator, responsável por gerar grafos com base na quantidade de vértices, número mínimo de grau por vértice e número máximo de grau por vértice.

Para garantir que todos os algoritmos estão funcionando corretamente e retornando resultados esperados, foram desenvolvidas baterias de teste para cada uma das funções do sistema. Assim, é possível ter certeza e confiar que tanto o processamento que ocorre nas matrizes, quanto os que ocorrem nas listas entregam os mesmos resultados de adição e remoção de arestas, ponderação de vértices e arestas, e para todas as outras funções. Os testes foram escritos utilizando a biblioteca Junit do próprio Java.

A Organização de todo o sistema pode ser representada pelo seguinte diagrama de classe:

Figura 1: Diagrama de Classe C GraphMatrix o matrix: int[][] o matrix\_peso\_vertice: int[] o n\_vertice: int GraphMatrix(int n\_vertices):void preencherMatrizComZeros():void getQuantidadeDeArestasNaMatriz():int existeVerticeNaMatriz(int v\_find):boolean existeArestaNaMatriz(int v\_origem, int v\_destino):boolean getQuantidadeVerticesNaMatriz():int getQuantidadeArestas():int addArestaNaMatriz(int v\_origem, int v\_destino):void
addArestaNaMatriz(int v\_origem, int v\_destino, int peso):void rmArestaDaMatriz(int v origem, int v destino):void ponderar√ertice(int rotulo, int peso):void ponderarAresta(int v\_origem, int v\_destino, int peso):void existeAdjacenciaEntreOsVertices(int v\_origem, int v\_destino):boolean o is∀azio():boolean isCompleto():boolean • isArestasAdjacentes(int v\_origem, int v\_destino, int w\_origem, int w\_destino):boolean C Graph o n\_vertices: int o vertices: List<Vertice> o arestas: List<Aresta> o ROTULOS\_VERTICE: int = 0 o ROTULOS\_ARESTA: int = 0 o tabela\_busca\_profundidade: Tabela Graph(int n\_vertices):void GerarListaDeAdjacencia(): void
GerarListaDeAdjacenciaAPartirDaMatriz(): void existeVertice(int v\_find): boolean • existeAdjacenciaEntreÓsVertices(int v origem, int v destino): boolean addAresta(int v\_origem, int v\_destino): void addAresta(Aresta aresta: void rmAresta(int v\_origem, int v\_destino): void getQuantidadeVertices(): int o getQuantidadeArestas(): int o ponderar√ertice(int rotulo, int peso): void • ponderarAresta(int v\_origem, int v\_destino, int peso): void getAresta(int v\_origem, int v\_destino): Aresta
getArestaFromList(int index: Aresta o is√azio(): boolean isCompleto(): boolean • isArestasAdjacentes(int v\_origem, int v\_destino, int w\_origem, int w\_destino): boolean toString(): String ExecutarBuscaEmProfundidade(boolean print\_tabela: void ExecutarBuscaEmProfundidade(): void
BuscaEmProfundidade( int v ):void ExecutarNaiveBridgeFind(): boolean buscaNaive(): void executarBuscarPontes(): void BuscarPontes(int v) : void C Tabela o T: int (c) Aresta o TD: int[]

o TT: int[]

o pai: int[]

o isCompleta: boolean

Tabela(int vertices) : void

zerarTabela(int vertices) : void

aindaHaVerticesParaExplorar(): boolean
proximoVerticeAExplorar(): int

o n\_vertices: int

toString(): String

C Vertice

o arestas: List<Vertice>

Vertice(int rotulo): void

addAresta(Vertice e): void

arestasToString(): String

o rotulo:int

peso:int

rotuloVerticeV:int

rotuloVerticeW:int

Aresta(int rotulo, int peso, int rotulo\vertice\v

Aresta(int rotulo, int rotulo\vertice\ver

o rotulo: int

o peso : int

#### 2.2 Busca Naive

A busca Naive funciona com base em uma busca em profundidade. Um algoritmo ingênuo que busca pontes em um grafo não direcionado apenas levando em consideração a quantidade de vezes que teve que recomeçar a busca no caso de encerrar a exploração de uma árvore e ainda existirem vértices a serem explorados.

Esse tipo de abordagem não produz resultados precisos pois a cada execução, e a depender do vértice inicial, o algoritmo pode produzir resultados diferentes, podendo encontrar pontes em grafos fortemente-conexos e não encontrar nenhuma ponte em grafos simplesmente conexos. Isso porque, o algoritmo ingênuo é implementado de tal forma que testa a conectividade do grafo, através da busca em profundidade, após remover e repor arestas do grafo. Ele executa desta forma para todas as arestas de um grafo não direcionado.

#### 2.3 Algoritmo de Tarjan

Parte da proposta do trabalho foi a implementação do método de Tarjan para encontrar pontes em um grafo. Inicialmente o algoritmo foi desenvolvido em um classe separada que herdava suas propriedades da classe Graph. O método consiste em realizar uma busca em largura, porém se atentando aos seguintes detalhes:

- Em uma aresta (v,w), onde v representa o vértice pai e atual descoberto pela busca e w o vértice filho que ainda será visitado, tal aresta será uma ponte caso não haja nenhuma outra aresta alternativa para alcançar v ou um acenstral de v com raiz em w.

Para verificar essas condições, o código armazena o tempo em que o primeiro vértice acessível a partir de uma sub-árvore gerada pela busca com raiz em x (sendo x um dos vértices do grafo) é visitado em um vetor low[x]. Para uma aresta (v,w) ser caracterizada como ponte durante o agoritmo de busca, o tempo de descoberta do vértice v deve ser menor que low[w], ou TD[v] < low[w].

Após os testes feitos, comparado as buscas de um mesmo grafo com outros algoritmos (como o naive), os métodos foram transferidos para a classe Graph principal.

#### 2.4 Ciclo Euleriano

Passa uma única vez por cada aresta no grafo, partindo e chegando a um mesmo vértice.

• Para haver um ciclo euleriano, todo vértice deve ter grau par, como já discutimos.

O algoritmo de Fleury, proposto em 1883, utiliza um grafo reduzido induzido pelas arestas ainda não marcadas

- Inicialmente todas as arestas estão não marcadas.
- As arestas vão sendo "marcadas", ou removidas do grafo, a medida em que vão sendo inseridas no ciclo

Regra da ponte: se uma aresta v, w é uma ponte no grafo reduzido, então v, w só deve ser escolhida caso não haja outra opção.

### 2.5 Algoritmo de Fleury

Passo a passo para um grafo não dirigido e não valorado

- 1. Verifique se o grafo apresenta as condições para ter um ciclo euleriano
- 2. Caso positivo, escolha um vértice V1 para começar.
- 3. Entre os vértices adjacentes a V1, faça
  - (a) Se há apenas um vértice como opção, escolha este como V2.
  - (b) Se há mais de um vértice possível, escolha um V2 apropriado dentre eles (ou seja, um que "não repita a ponte").
- 4. Remova a aresta (V1, V2).
- 5. Se ainda houver arestas não percorridas, volte ao passo 3, partindo agora de V2 (V2 é o novo V1).
- 6. Caso contrário, imprima o caminho percorrido.