Atividade de Implementação

1.

```
/*
* Exercício 1 * Escreva um método “int inDegree(int v)” que calcula e retorna o grau
* de entrada de um vértice v de um grafo * dirigido. O método deve ser implementado
na  * classe TGrafo da matriz  de adjacência. Obs.: Grau de entrada de * v é o total
de     * arestas que chegam no vértice v.
*/

System.out.println("\nExercício 1");

System.out.println("Grau de entrada do vértice 0: " + g.inDegree(0));
System.out.println("Grau de entrada do vértice 1: " + g.inDegree(1));
System.out.println("Grau de entrada do vértice 2: " + g.inDegree(2));
System.out.println("Grau de entrada do vértice 3: " + g.inDegree(3));
```

```
// Calcula e retorna o grau de entrada do vértice v
public int inDegree(int v) {
    // Inicializa a contagem de incidências em v (0)
    int deg = 0;

    // Itera a matriz de adjacência para encontrar todas as incidências em v
    for (int i = 0; i < this.n; i++) {
        if (this.adj[i][v] == 1) {
            // Caso haja incidência em v, o contador do grau é incrementado em 1
            deg++;
        }
    }

    // Retorna o grau calculado
    return deg;
}
```

```
Exercício 1
Grau de entrada do vértice 0: 0
Grau de entrada do vértice 1: 2
Grau de entrada do vértice 2: 1
Grau de entrada do vértice 3: 2
```

2.

```
/*
 * Exercício 2 * Escreva o método outDegree(int v) que calcula o grau de saída de
**v** em grafo dirigido. O método deve ser * implementado na classe TGrafo que usa
matriz de adjacência. Obs.: Grau de saída de v é o total de arestas que * saem do
vértice v. */
System.out.println("\nExercício 2");

System.out.println("Grau de saída do vértice 0: " + g.outDegree(0));
System.out.println("Grau de saída do vértice 1: " + g.outDegree(1));
System.out.println("Grau de saída do vértice 2: " + g.outDegree(2));
```

```
// Calcula e retorna o grau de saída do vértice v
public int outDegree(int v) {
    // Inicializa a contagem de saídas de v (0)
    int deg = 0;

    // Itera a matriz de adjacência para encontrar todas as saídas de v
    for (int i = 0; i < this.n; i++) {
        if (this.adj[v][i] == 1) {
            // Caso haja saída de v, o contador do grau é incrementado em 1
            deg++;
        }
    }

    // Retorna o grau calculado
    return deg;
}
```

```
Exercício 2
Grau de saída do vértice 0: 2
Grau de saída do vértice 1: 1
Grau de saída do vértice 2: 2
```

3.

```
/*
 * Exercício 3 * Fazer o método degree(int v) que calcula o grau do vértice de um
grafo dirigido. O método deve ser * implementado na classe TGrafo que usa matriz de
adjacência. */
System.out.println("\nExercício 3");

System.out.println("Grau do vértice 0: " + g.degree(0));
System.out.println("Grau do vértice 1: " + g.degree(1));
System.out.println("Grau do vértice 2: " + g.degree(2));
```

```java
// Retorna o grau de incidência do vértice v
public int degree(int v) {
    return this.outDegree(v);
}
```

```
Exercício 3
Grau do vértice 0: 2
Grau do vértice 1: 1
Grau do vértice 2: 2
```

4.

```java
/*
 * Exercício 4 * Escreva um método para um grafo direcionado que recebe um vértice como parâmetro e retorne 1 se vértice for * uma fonte (grau de saída maior que zero e grau de entrada igual a 0), ou 0 caso contrário. O método deve * ser implementado para a classe TGrafo como matriz de adjacência. */
System.out.println("\nExercício 4");

System.out.printf("O vértice %d%s é uma fonte.\n", 0, g.isSource(0) ? "" : " não");
System.out.printf("O vértice %d%s é uma fonte.\n", 1, g.isSource(1) ? "" : " não");
System.out.printf("O vértice %d%s é uma fonte.\n", 2, g.isSource(2) ? "" : " não");
System.out.printf("O vértice %d%s é uma fonte.\n", 3, g.isSource(3) ? "" : " não");
```

```java
// Retorna se o vértice v é uma fonte
public boolean isSource(int v) {
    return this.inDegree(v) == 0 && this.outDegree(v) > 0;
}
```

```
Exercício 4
O vértice 0 é uma fonte.
O vértice 1 não é uma fonte.
O vértice 2 não é uma fonte.
O vértice 3 não é uma fonte.
```

5.

```java
/*
 * Exercício 5 * Escreva um método para um grafo direcionado que recebe um vértice como parâmetro, retorne 1 se vértice for um * sorvedouro (grau de entrada maior que zero e grau de saída igual a 0), ou 0 caso contrário. O método deve * ser implementado para a classe TGrafo que utiliza matriz de adjacência. */
System.out.println("\nExercício 5");
```

```
System.out.printf("O vértice %d%s é um sorvedouro.\n", 0, g.isSink(0) ? "" : "
não");
System.out.printf("O vértice %d%s é um sorvedouro.\n", 1, g.isSink(1) ? "" : "
não");
System.out.printf("O vértice %d%s é um sorvedouro.\n", 2, g.isSink(2) ? "" : "
não");
System.out.printf("O vértice %d%s é um sorvedouro.\n", 3, g.isSink(3) ? "" : "
não");
```

```
// Retorna se o vértice v é um sorvedouro
public boolean isSink(int v) {
    return this.inDegree(v) > 0 && this.outDegree(v) == 0;
}
```

```
Exercício 5
O vértice 0 não é um sorvedouro.
O vértice 1 não é um sorvedouro.
O vértice 2 não é um sorvedouro.
O vértice 3 é um sorvedouro.
```

6.

```
/*
 * Exercício 6 * Escreva um método que receba um grafo dirigido como parâmetro e
retorna 1 se o grafo for simétrico e 0 caso * contrário. O método deve ser
implementado  para a classe TGrafo que utiliza matriz de adjacência. */
System.out.println("\nExercício 6");

final int[][] grafoSimetrico = {
        {0, 1, 0, 1},
        {1, 1, 1, 1},
        {0, 1, 0, 1},
        {1, 1, 1, 1}
};

final int[][] grafoAssimetrico = {
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {0, 0, 0, 1},
        {1, 0, 0, 0}
};

System.out.println("O grafo é " + (TGrafo.isSymmetric(grafoSimetrico) ? "simétrico"
: "assimétrico"));
```

```
System.out.println("O grafo é " + (TGrafo.isSymmetric(grafoAssimetrico) ?
"simétrico" : "assimétrico"));
```

```
// Retorna se o grafo é simétrico
public static boolean isSymmetric(int[][] adj) {
    for (int i = 0; i < adj.length; i++) {
        for (int j = 0; j < i; j++) {
            if (adj[i][j] != adj[j][i]) {
                return false;
            }
        }
    }

    return true;
}
```

```
Exercício 6
O grafo é simétrico
O grafo é assimétrico
```

7.

```
/*
 * Exercício 7 * Um grafo pode ser armazenado em um arquivo com o seguinte formato:
* 6 * 8 * 0 1 * 0 5 * 1 0 * 1 5 * 2 4 * 3 1 * 4 3 * 3 5 * Onde na primeira linha
contém um inteiro V (vértice), na segunda contém um inteiro A (arestas) e nas demais
* linha contém dois inteiros pertencentes ao intervalo 0..V-1. Se interpretarmos
cada linha do arquivo como uma * aresta, podemos dizer que o arquivo define um grafo
com vértices 0..V-1. Escreva um método que receba um nome * de arquivo com o formato
acima e construa a representação do grafo como matriz de adjacência. */
System.out.println("\nExercício 7");

TGrafo fileGraph = TGrafo.createFromFile("example.graph");

if (fileGraph != null) {
    fileGraph.show();
}
```

```
public static TGrafo createFromFile(String path) {
    try {
        File graphFile = new File(path);
        Scanner fileReader = new Scanner(graphFile);

        int verticesCount = fileReader.nextInt();
```

```
        TGrafo graph = new TGrafo(verticesCount);

        int edgeCount = fileReader.nextInt();

        int read = 0;
        while (read++ < edgeCount) {
            int v = fileReader.nextInt();
            int w = fileReader.nextInt();
            graph.insereA(v, w);
        }

        fileReader.close();
        return graph;
    } catch (FileNotFoundException e) {
        System.err.println("Erro ao ler arquivo.");
        // e.printStackTrace();
    }

    return null;
}
```

```
Exercício 7
n: 6
m: 8

[0,0]= 0 [0,1]= 1 [0,2]= 0 [0,3]= 0 [0,4]= 0 [0,5]= 1
[1,0]= 1 [1,1]= 0 [1,2]= 0 [1,3]= 0 [1,4]= 0 [1,5]= 1
[2,0]= 0 [2,1]= 0 [2,2]= 0 [2,3]= 0 [2,4]= 1 [2,5]= 0
[3,0]= 0 [3,1]= 1 [3,2]= 0 [3,3]= 0 [3,4]= 0 [3,5]= 1
[4,0]= 0 [4,1]= 0 [4,2]= 0 [4,3]= 1 [4,4]= 0 [4,5]= 0
[5,0]= 0 [5,1]= 0 [5,2]= 0 [5,3]= 0 [5,4]= 0 [5,5]= 0

fim da impressao do grafo.
```

8.

```
/*
 * Exercício 8 * Criar uma outra classe TGrafoND e modifique as funções insereA,
removeA e show para representar um grafo * não-dirigido utilizando matriz de
adjacência */
System.out.println("\nExercício 8");

TGrafoND ndGraph = new TGrafoND(4);
ndGraph.insereA(0, 1);
ndGraph.insereA(0, 2);
ndGraph.insereA(2, 1);
```

```
ndGraph.insereA(2, 3);
ndGraph.insereA(1, 3);
ndGraph.show();
```

```
package com.pedropadilha.grafos;

/**
 * @author pedropadilha
 */
public class TGrafoND {

    private final int n; // Quantidade de vértices
    private int m; // Quantidade de arestas
    private final int[][] adj; //matriz de adjacência

    public TGrafoND(int n) {
        this.n = n;
        this.m = 0;
        this.adj = new int[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                this.adj[i][j] = 0;
            }
        }
    }

    // Cria a aresta (caso ela não exista) e incrementa o número total de arestas
    public void insereA(int v, int w) {
        if (adj[v][w] == 0) {
            adj[v][w] = 1;
            adj[w][v] = 1;
            m++;
        }
    }

    // Remove a aresta (caso ela exista) e decrementa o número total de arestas
    public void removeA(int v, int w) {
        if (adj[v][w] == 1) {
            adj[v][w] = 0;
            adj[w][v] = 0;
            m--;
        }
    }

    // Exibe o número total de vértices e arestas e imprime a matriz de adjacência
    public void show() {
        System.out.println("n: " + n);
```

```
        System.out.println("m: " + m);
        for (int i = 0; i < n; i++) {
            System.out.println();
            for (int j = 0; j < n; j++) {
                System.out.print(adj[i][j] + " ");
            }
        }
        System.out.println();
    }
}
```

```
Exercício 8
n: 4
m: 5

0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
```

9.

```
/*
 * Exercício 9 * Fazer o método degree(int v) que calcula o grau do vértice de um grafo não-dirigido. O método deve ser * implementado na classe TGrafoND que usa matriz de adjacência. */
System.out.println("\nExercício 9");

System.out.println("Grau do vértice 0: " + ndGraph.degree(0));
System.out.println("Grau do vértice 1: " + ndGraph.degree(1));
System.out.println("Grau do vértice 2: " + ndGraph.degree(2));
System.out.println("Grau do vértice 3: " + ndGraph.degree(3));
```

```
// Retorna o grau do vértice v
public int degree(int v) {
    int degree = 0;

    for (int i = 0; i < this.n; i++) {
        degree += this.adj[v][i];
    }

    return degree;
}
```

```
Exercício 9
Grau do vértice 0: 2
Grau do vértice 1: 3
Grau do vértice 2: 3
Grau do vértice 3: 2
```

10)

```
/*
 * Exercício 10 * Modifique a classe TGrafo e os métodos correspondentes para
permitir a criação de um grafo direcionado * rotulado (valor float) nas arestas */
System.out.println("\nExercício 10");

TGrafoRotulado wg = new TGrafoRotulado(4);

wg.insereA(0, 1, 0.0f);
wg.insereA(1, 2, 1.0f);
wg.insereA(2, 3, 2.0f);

wg.show();
```

```
package com.pedropadilha.grafos;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TGrafoRotulado {
    // Atributos Privados

    private final int n; // quantidade de vértices
    private int m; // quantidade de arestas
    private final float[][] adj; //matriz de adjacência

    public TGrafoRotulado(int n) {  // construtor
        this.n = n;

        this.m = 0;

        this.adj = new float[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                this.adj[i][j] = Float.MAX_VALUE;
            }
        }
    }
```

```java
    // Insere a aresta v->w com peso l no grafo (se ela não existir) e atualiza a
quantidade total de arestas
    public void insereA(int v, int w, float l) {
        if (adj[v][w] == Float.MAX_VALUE) {
            adj[v][w] = l;
            m++;
        }
    }

    // Remove a aresta v->w do grafo (se ela existir) e atualiza a quantidade total
de arestas
    public void removeA(int v, int w) {
        if (adj[v][w] != Float.MAX_VALUE) {
            adj[v][w] = Float.MAX_VALUE;
            m--;
        }
    }

    // Exibe o número total de vértices e arestas e imprime a matriz de adjacência
com os respectivos pesos das arestas
    public void show() {
        System.out.println("n: " + n);
        System.out.println("m: " + m);

        for (int i = 0; i < n; i++) {
            System.out.print("\n");
            for (int j = 0; j < n; j++) {
                String text = adj[i][j] == Float.MAX_VALUE ? " ∞ " :
String.valueOf(adj[i][j]);
                System.out.print(text);
            }
        }
    }
}
```

```
Exercício 10
n: 4
m: 3

 ∞ 0.0 ∞  ∞
 ∞  ∞ 1.0 ∞
 ∞  ∞  ∞ 2.0
 ∞  ∞  ∞  ∞
```

11.

```java
/*
 * Exercício 11 * Fazer um método que permita remover um vértice do Grafo (não
dirigido e dirigido). * Não se esqueça de remover as arestas associadas. */
System.out.println("\nExercício 11");

System.out.println("--- Grafo dirigido ---");

TGrafo grafo1 = new TGrafo(4);
grafo1.insereA(0, 1);
grafo1.insereA(1, 0);
grafo1.insereA(1, 2);
grafo1.insereA(1, 3);
grafo1.insereA(2, 1);
grafo1.insereA(2, 3);
grafo1.show();

System.out.println("Remover vértice 2 e exibir grafo novamente");
grafo1.removeV(2);
grafo1.show();

System.out.println("--- Grafo não-dirigido ---");

TGrafoND grafo2 = new TGrafoND(4);
grafo2.insereA(0, 1);
grafo2.insereA(0, 2);
grafo2.insereA(1, 2);
grafo2.insereA(2, 3);
grafo2.show();

System.out.println("Remover vértice 2 e exibir grafo novamente");
grafo2.removeV(2);
grafo2.show();
```

```java
// Dirigido
// Remove um vértice (todas as arestas relacionadas a ele)
public void removeV(int v) {
    for (int i = 0; i < this.n; i++) {
        this.removeA(i, v);
        this.removeA(v, i);
    }
}

// Não dirigido
public void removeV(int v) {
    for (int i = 0; i < this.n; i++) {
        this.removeA(v, i);
```

```
    }
}
```

```
Exercício 11
--- Grafo dirigido ---
n: 4
m: 6

[0,0]= 0 [0,1]= 1 [0,2]= 0 [0,3]= 0
[1,0]= 1 [1,1]= 0 [1,2]= 1 [1,3]= 1
[2,0]= 0 [2,1]= 1 [2,2]= 0 [2,3]= 1
[3,0]= 0 [3,1]= 0 [3,2]= 0 [3,3]= 0

fim da impressao do grafo.
Remover vértice 2 e exibir grafo novamente
n: 4
m: 3

[0,0]= 0 [0,1]= 1 [0,2]= 0 [0,3]= 0
[1,0]= 1 [1,1]= 0 [1,2]= 0 [1,3]= 1
[2,0]= 0 [2,1]= 0 [2,2]= 0 [2,3]= 0
[3,0]= 0 [3,1]= 0 [3,2]= 0 [3,3]= 0

fim da impressao do grafo.
--- Grafo não-dirigido ---
n: 4
m: 4

0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
Remover vértice 2 e exibir grafo novamente
n: 4
m: 1

0 1 0 0
1 0 0 0
0 0 0 0
0 0 0 0
```

12.

```java
/*
 * Exercício 12 * Fazer um método que verifique e retorne se o grafo (não dirigido)
é completo */
System.out.println("\nExercício 12");

TGrafoND completoND = new TGrafoND(3);
completoND.insereA(0, 1);
completoND.insereA(1, 2);
completoND.insereA(2, 0);
System.out.println("O grafo não direcionado é: " + (completoND.isComplete() ?
"completo" : "incompleto"));

TGrafoND incompletoND = new TGrafoND(3);
incompletoND.insereA(0, 1);
incompletoND.insereA(1, 2);
System.out.println("O grafo não direcionado é: " + (incompletoND.isComplete() ?
"completo" : "incompleto"));
```

```java
// Retorna se o grafo é completo
public boolean isComplete() {
    int totalPossibilities = this.n * (this.n - 1) / 2;
    return this.m == totalPossibilities;
}
```

```
Exercício 12
O grafo não direcionado é: completo
O grafo não direcionado é: incompleto
```

13.

```java
/*
 * Exercício 13 * Fazer um método que verifique e retorne se o grafo (dirigido) é
completo */
System.out.println("\nExercício 13");

TGrafo completoD = new TGrafo(3);
completoD.insereA(0, 1);
completoD.insereA(1, 0);
completoD.insereA(1, 2);
completoD.insereA(2, 1);
completoD.insereA(2, 0);
completoD.insereA(0, 2);
System.out.println("O grafo direcionado é: " + (completoD.isComplete() ? "completo"
: "incompleto"));
```

```
TGrafo incompletoD = new TGrafo(3);
incompletoD.insereA(0, 1);
incompletoD.insereA(1, 2);
System.out.println("O grafo direcionado é: " + (incompletoD.isComplete() ?
"completo" : "incompleto"));
```

```
public boolean isComplete() {
    int totalPossibilities = (this.n * this.n) - this.n;
    return this.m == totalPossibilities;
}
```

```
Exercício 13
O grafo direcionado é: completo
O grafo direcionado é: incompleto
```

14.

```
/*
 * Exercício 14 * Fazer um método que retorne o complemento (grafo complementar) de
um grafo (dirigido ou não) na forma de * uma matriz de adjacência */
System.out.println("\nExercício 14");

System.out.println("Grafo direcionado:");
g.show();
System.out.println("Complemento do grafo:");
TGrafo complementoD = g.getComplement();
complementoD.show();

System.out.println("Grafo não direcionado:");
ndGraph.show();
System.out.println("Complemento do grafo:");
TGrafoND complementoND = ndGraph.getComplement();
complementoND.show();
```

```
public TGrafo getComplement() {
    TGrafo complement = new TGrafo(this.n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (this.adj[i][j] == 0) {
                complement.insereA(i, j);
            }
        }
    }
```

```
    return complement;
}

// Retorna o complemento do grafo
public TGrafoND getComplement() {
    TGrafoND complement = new TGrafoND(this.n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (this.adj[i][j] == 0) {
                complement.insereA(i, j);
            }
        }
    }

    return complement;
}
```

```
Exercício 14
Grafo direcionado:
n: 4
m: 5

[0,0]= 0 [0,1]= 1 [0,2]= 1 [0,3]= 0
[1,0]= 0 [1,1]= 0 [1,2]= 0 [1,3]= 1
[2,0]= 0 [2,1]= 1 [2,2]= 0 [2,3]= 1
[3,0]= 0 [3,1]= 0 [3,2]= 0 [3,3]= 0

fim da impressao do grafo.
Complemento do grafo:
n: 4
m: 11

[0,0]= 1 [0,1]= 0 [0,2]= 0 [0,3]= 1
[1,0]= 1 [1,1]= 1 [1,2]= 1 [1,3]= 0
[2,0]= 1 [2,1]= 0 [2,2]= 1 [2,3]= 0
[3,0]= 1 [3,1]= 1 [3,2]= 1 [3,3]= 1

fim da impressao do grafo.
Grafo não direcionado:
n: 4
m: 5

0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Complemento do grafo:
n: 4
m: 1

0 0 0 1
0 0 0 0
0 0 0 0
1 0 0 0
```

15.

```java
/*
 * Exercício 15 * Fazer um método que retorne o tipo de conexidade de um grafo não direcionado * (0 – conexo ou 1 – não conexo – desconexo). */
System.out.println("\nExercício 15");
```

```
TGrafoND conexo = new TGrafoND(4);
conexo.insereA(0, 1);
conexo.insereA(1, 2);
conexo.insereA(2, 3);

System.out.println("O grafo é " + (conexo.isDisconnected() ? "des" : "") +
"conexo");

TGrafoND nConexo = new TGrafoND(4);
nConexo.insereA(0, 1);
nConexo.insereA(2, 3);

System.out.println("O grafo é " + (nConexo.isDisconnected() ? "des" : "") +
"conexo");
```

```
public boolean isDisconnected() {
    int i = 1;
    while (i < this.n) {
        boolean[] visits = new boolean[this.n];
        this.dfs(i, visits);

        for (boolean b : visits) {
            if (!b) {
                return true;
            }
        }

        i++;
    }

    return false;
}
```

```
Exercício 15
O grafo é conexo
O grafo é desconexo
```

16.

```
/*
 * Exercício 16 * Fazer um método que retorne a categoria de conexidade para um
grafo direcionado * (3 - C3, 2 - C2, 1 - C1 ou 0 - c0). */
System.out.println("\nExercício 16");

TGrafo grafoDesconexo = new TGrafo(2);
```

```java
System.out.println("Grau de Conexidade do Grafo: " + grafoDesconexo.getCategory());

// G1 - slide 24
TGrafo grafoC1 = new TGrafo(4);
grafoC1.insereA(0, 3);
grafoC1.insereA(3, 1);
grafoC1.insereA(3, 2);

System.out.println("Grau de Conexidade do Grafo: " + grafoC1.getCategory());

// G2 - slide 26
TGrafo grafoC2 = new TGrafo(4);
grafoC2.insereA(0, 3);
grafoC2.insereA(3, 1);
grafoC2.insereA(1, 2);
grafoC2.insereA(2, 3);

System.out.println("Grau de Conexidade do Grafo: " + grafoC2.getCategory());

// G3 - slide 28
TGrafo grafoC3 = new TGrafo(4);
grafoC3.insereA(0, 2);
grafoC3.insereA(1, 0);
grafoC3.insereA(2, 1);
grafoC3.insereA(2, 3);
grafoC3.insereA(3, 0);

System.out.println("Grau de Conexidade do Grafo: " + grafoC3.getCategory());
```

```java
// Retorna a categoria de conexidade de um grafo direcionado
public int getCategory() {
    boolean[][] vvisits = new boolean[this.n][];

    for (int i = 0; i < this.n; i++) {
        boolean[] visits = new boolean[this.n];
        dfs(i, visits);
        vvisits[i] = visits;
    }

    for (int i = 0; i < this.n; i++) {
        int j = 0;
        boolean pass = false;

        while (j < this.n) {
            if (j != i) {
                pass = pass || vvisits[i][j] || vvisits[j][i];
            }
```

```
            j++;
        }

        if (!pass) {
            return 0;
        }
    }

    boolean isC2 = true;
    boolean isC3 = true;

    // check pair reachability
    for (int i = 0; i < this.n; i++) {
        for (int j = i; j < this.n; j++) {
            // check if pair (i,j) is reachable:

            isC2 = isC2 && vvisits[i][j];

            if (isC2) {
                isC3 = isC3 && vvisits[j][i];
            }

        }
    }

    if (isC3) {
        return 3;
    } else if (isC2) {
        return 2;
    } else {
        return 1;
    }
}
```

```
Exercício 16
Grau de Conexidade do Grafo: 0
Grau de Conexidade do Grafo: 1
Grau de Conexidade do Grafo: 2
Grau de Conexidade do Grafo: 3
```

17.

```
/*
 * Exercício 17 * Faze um método que retorne o grafo reduzido de um grafo
direcionado no formato de uma matriz de adjacência. */
System.out.println("\nExercício 17");
```

```
TGrafo full = new TGrafo(10);
full.insereA(0, 3);
full.insereA(3, 4);
full.insereA(3, 6);
full.insereA(6, 9);
full.insereA(9, 7);
full.insereA(7, 6);
full.insereA(9, 8);
full.insereA(7, 8);
full.insereA(4, 7);
full.insereA(4, 5);
full.insereA(5, 8);
full.insereA(8, 5);
full.insereA(4, 1);
full.insereA(1, 0);
full.insereA(1, 2);
full.insereA(5, 2);

TGrafo reduced = full.getReducedGraph();
reduced.show();
```

```
public TGrafo getReducedGraph() {
    boolean[] reduced = new boolean[this.n];
    ArrayList<ArrayList<Integer>> components = new ArrayList<>();

    for (int i = 0; i < this.n; i++) {
        if (reduced[i]) {
            continue;
        }

        boolean[] reaches = new boolean[this.n];
        dfs(i, reaches);
        boolean[] isReachedBy = this.getReachableBy(i);

        ArrayList<Integer> intersection = new ArrayList<>();
        for (int j = 0; j < this.n; j++) {
            if (reaches[j] && isReachedBy[j]) {
                intersection.add(j);
            }
        }

        components.add(intersection);
        for (int j : intersection) {
            reduced[j] = true;
        }
    }

    TGrafo reducedGraph = new TGrafo(components.size());
```

```
    for (int i = 0; i < components.size(); i++) {
        for (int j = 0; j < components.size(); j++) {
            if (i == j) {
                continue;
            }

            if (reaches(components.get(i), components.get(j))) {
                reducedGraph.insereA(i, j);
            }
        }
    }

    return reducedGraph;
}
```

```
Exercício 17
n: 4
m: 5

[0,0]= 0 [0,1]= 1 [0,2]= 1 [0,3]= 1
[1,0]= 0 [1,1]= 0 [1,2]= 0 [1,3]= 0
[2,0]= 0 [2,1]= 1 [2,2]= 0 [2,3]= 0
[3,0]= 0 [3,1]= 0 [3,2]= 1 [3,3]= 0

fim da impressao do grafo.
```

18.

```
/*
 * Exercício 18 * Modifique a classe TGrafo e os métodos correspondentes para
permitir a criação de um grafo direcionado * rotulado (valor float) nas arestas. */
System.out.println("\nExercício 18");

TGrafoRotulado gr = new TGrafoRotulado(3);
gr.insereA(0, 1, 0.1f);
gr.insereA(1, 2, 1.2f);
gr.insereA(2, 0, 2.0f);

gr.show();
```

```
package com.pedropadilha.grafos;

import java.io.File;
import java.io.FileNotFoundException;
```

```java
import java.util.Scanner;

public class TGrafoRotulado {
    // Atributos Privados

    private final int n; // quantidade de vértices
    private int m; // quantidade de arestas
    private final float[][] adj; //matriz de adjacência

    public TGrafoRotulado(int n) {  // construtor
        this.n = n;

        this.m = 0;

        this.adj = new float[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                this.adj[i][j] = Float.MAX_VALUE;
            }
        }
    }

    // Insere a aresta v->w com peso l no grafo (se ela não existir) e atualiza a
quantidade total de arestas
    public void insereA(int v, int w, float l) {
        if (adj[v][w] == Float.MAX_VALUE) {
            adj[v][w] = l;
            m++;
        }
    }

    // Remove a aresta v->w do grafo (se ela existir) e atualiza a quantidade total
de arestas
    public void removeA(int v, int w) {
        if (adj[v][w] != Float.MAX_VALUE) {
            adj[v][w] = Float.MAX_VALUE;
            m--;
        }
    }

    // Exibe o número total de vértices e arestas e imprime a matriz de adjacência
com os respectivos pesos das arestas
    public void show() {
        System.out.println("n: " + n);
        System.out.println("m: " + m);

        for (int i = 0; i < n; i++) {
            System.out.print("\n");
```

```
            for (int j = 0; j < n; j++) {
                String text = adj[i][j] == Float.MAX_VALUE ? " ∞ " :
String.valueOf(adj[i][j]);
                System.out.print(text);
            }
        }
    }
}
```

```
Exercício 18
n: 3
m: 3

 ∞ 0.1 ∞
 ∞  ∞ 1.2
2.0 ∞  ∞
```

19.

```
/*
 * Exercício 19 * Escreva um método “int inDegree(int v)” que calcula e retorna o
grau de entrada de um vértice v de um grafo dirigido fazendo uso da lista de
adjacência. */
System.out.println("\nExercício 19");

System.out.println("Grau de entrada do vértice 0: " + graph.inDegree(0));
System.out.println("Grau de entrada do vértice 1: " + graph.inDegree(1));
System.out.println("Grau de entrada do vértice 2: " + graph.inDegree(2));
System.out.println("Grau de entrada do vértice 3: " + graph.inDegree(3));
```

```
// Retorna o grau de entrada do vértice
public int inDegree(int v) {
    int degree = 0;

    for (ArrayList<Integer> list : this.adj) {
        if (list.contains(v)) {
            degree++;
        }
    }

    return degree;
}
```

```
Exercício 19
Grau de entrada do vértice 0: 1
Grau de entrada do vértice 1: 1
Grau de entrada do vértice 2: 2
Grau de entrada do vértice 3: 2
```

20.

```java
/*
 * Exercício 20 * Escreva o método outDegree(int v) que calcula o grau de saída de v em grafo dirigido. fazendo uso da lista de adjacência. */
System.out.println("\nExercício 20");

System.out.println("Grau de saída do vértice 0: " + graph.outDegree(0));
System.out.println("Grau de saída do vértice 1: " + graph.outDegree(1));
System.out.println("Grau de saída do vértice 2: " + graph.outDegree(2));
System.out.println("Grau de saída do vértice 3: " + graph.outDegree(3));
```

```java
// Retorna o grau de saída do vértice
public int outDegree(int v) {
    return this.adj.get(v).size();
}
```

```
Exercício 20
Grau de saída do vértice 0: 2
Grau de saída do vértice 1: 2
Grau de saída do vértice 2: 1
Grau de saída do vértice 3: 1
```

21.

```java
/*
 * Exercício 21 * Fazer o método degree(int v) que calcula o grau do vértice de um grafo dirigido fazendo uso da lista de adjacência. */
System.out.println("\nExercício 21");

System.out.println("Grau do vértice 0: " + graph.getDegree(0));
System.out.println("Grau do vértice 1: " + graph.getDegree(1));
System.out.println("Grau do vértice 2: " + graph.getDegree(2));
System.out.println("Grau do vértice 3: " + graph.getDegree(3));
```

```java
// Retorna o grau do vértice
public int getDegree(int v) {
```

```
    return this.inDegree(v);
}
```

```
Exercício 21
Grau do vértice 0: 1
Grau do vértice 1: 1
Grau do vértice 2: 2
Grau do vértice 3: 2
```

22.

```
/*
 * Exercício 22 * Escreva um método que decida se dois grafos direcionados são
iguais. O método deve ser implementado para a classe TGrafo faz uso da lista de
adjacência. */
System.out.println("\nExercício 22");

ALGraph graph1 = new ALGraph(3);
graph1.insertEdge(0, 1);
graph1.insertEdge(1, 2);
graph1.insertEdge(2, 0);

ALGraph graph2 = new ALGraph(3);
graph2.insertEdge(0, 1);
graph2.insertEdge(1, 2);
graph2.insertEdge(2, 0);

ALGraph graph3 = new ALGraph(3);
graph3.insertEdge(0, 1);
graph3.insertEdge(1, 2);

System.out.println("Os Grafos 1 e 2 são: " + (ALGraph.compareGraphs(graph1, graph2)
? "iguais" : "diferentes"));
System.out.println("Os Grafos 1 e 3 são: " + (ALGraph.compareGraphs(graph1, graph3)
? "iguais" : "diferentes"));
System.out.println("Os Grafos 2 e 3 são: " + (ALGraph.compareGraphs(graph2, graph3)
? "iguais" : "diferentes"));
```

```
// Verifica se os grafos são iguais e retorna true em caso positivo
public static boolean compareGraphs(ALGraph graph1, ALGraph graph2) {
    if (graph1.n != graph2.n) {
        return false;
    }

    for (int i = 0; i < graph1.n; i++) {
        ArrayList<Integer> l1 = graph1.adj.get(i);
```

```
        ArrayList<Integer> l2 = graph2.adj.get(i);

        if (l1.size() != l2.size() || !l1.containsAll(l2)) {
            return false;
        }
    }

    return true;
}
```

```
Exercício 22
Os Grafos 1 e 2 são: iguais
Os Grafos 1 e 3 são: diferentes
Os Grafos 2 e 3 são: diferentes
```

23.

```
/*
 * Exercício 23 * Escreva um método que converta uma representação de um grafo em
outra. Por exemplo, converta um grafo armazenado em matriz de adjacência em uma
lista de adjacência. */
System.out.println("\nExercício 23");

TGrafo mg = new TGrafo(3);
mg.insereA(0, 1);
mg.insereA(0, 2);
mg.insereA(1, 2);

ALGraph converted = mg.convert();

converted.show();
```

```
// Converte o TGrafo para um ALGraph
public ALGraph convert() {
    ALGraph converted = new ALGraph(this.n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (this.adj[i][j] == 1) {
                converted.insertEdge(i, j);
            }
        }
    }

    return converted;
}
```

```
Exercício 23
Total de arestas: 3
0: 1 2
1: 2
2:
```

24.

```java
/*
 * Exercício 24 * Escreva um método que receba um grafo armazenado em lista de adjacência e inverta a lista de adjacência de todos os vértices do grafo.  Por exemplo, se os 4 vizinhos de um certo vértice u aparecem na lista adj[u] na ordem v, w, x, y, então depois da aplicação do método a lista deve conter os mesmos vértices na ordem y, x, w, v. Obs.: Vizinhos são todos os vértices ligados ao vértice u. */
System.out.println("\nExercício 24");

ALGraph gti = new ALGraph(3);
gti.insertEdge(0, 1);
gti.insertEdge(0, 2);
gti.insertEdge(1, 2);
gti.insertEdge(1, 0);
gti.insertEdge(2, 0);
gti.insertEdge(2, 1);

gti.show();
gti.reverseEdges();
System.out.println("Grafo com as adjacências invertidas:");
gti.show();
```

```java
// Inverte as listas dos vértices adjacentes
public void reverseEdges() {
    for (ArrayList<Integer> l : this.adj) {
        Collections.reverse(l);
    }
}
```

```
Exercício 24
Total de arestas: 6
0: 1 2
1: 2 0
2: 0 1
```

25.

```
/*
 * Exercício 25 * Escreva um método que receba um grafo e um vértice como parâmetro
e retorne 1 se vértice for uma fonte (grau de saída maior que zero e grau de entrada
igual a 0), ou 0 caso contrário. O método deve ser implementado para a classe TGrafo
como lista de adjacência. */
System.out.println("\nExercício 25");

System.out.printf("O vértice %d%s é uma fonte.\n", 0, graph3.isSource(0) ? "" : "
não");
System.out.printf("O vértice %d%s é uma fonte.\n", 1, graph3.isSource(1) ? "" : "
não");
System.out.printf("O vértice %d%s é uma fonte.\n", 2, graph3.isSource(2) ? "" : "
não");
```

```
// Retorna se o vértice v é uma fonte
public boolean isSource(int v) {
    return this.inDegree(v) == 0 && this.outDegree(v) > 0;
}
```

```
Exercício 25
O vértice 0 é uma fonte.
O vértice 1 não é uma fonte.
O vértice 2 não é uma fonte.
```

26.

```
/*
 * Exercício 26 * Escreva um método que receba um grafo e um vértice como parâmetro,
retorne 1 se vértice for um sorvedouro (grau de entrada maior que zero e grau de
saída igual a 0), ou 0 caso contrário. O método deve ser implementado para a classe
TGrafo que utiliza lista de adjacência. */
System.out.println("\nExercício 26");

System.out.printf("O vértice %d%s é um sorvedouro.\n", 0, graph3.isSink(0) ? "" : "
não");
System.out.printf("O vértice %d%s é um sorvedouro.\n", 1, graph3.isSink(1) ? "" : "
não");
System.out.printf("O vértice %d%s é um sorvedouro.\n", 2, graph3.isSink(2) ? "" : "
não");
```

```
// Retorna se o vértice v é um sorvedouro
public boolean isSink(int v) {
    return this.inDegree(v) > 0 && this.outDegree(v) == 0;
}
```

```
Exercício 26
O vértice 0 não é um sorvedouro.
O vértice 1 não é um sorvedouro.
O vértice 2 é um sorvedouro.
```

27.

```
/*
 * Exercício 27 * Escreva um método que receba um grafo dirigido como parâmetro e
retorna 1 se o grafo for simétrico e 0 caso contrário. O método deve ser
implementado  para a classe TGrafo que utiliza lista de adjacência. */
System.out.println("\nExercício 27");

System.out.println("\nExercício 27");

ALGraph sym = new ALGraph(2);
sym.insertEdge(0, 1);
sym.insertEdge(1, 0);

ALGraph notSym = new ALGraph(2);
notSym.insertEdge(0, 1);

System.out.println("O grafo é " + (ALGraph.isSymmetric(sym) ? "simétrico" :
"assimétrico"));
System.out.println("O grafo é " + (ALGraph.isSymmetric(notSym) ? "simétrico" :
"assimétrico"));
```

```
// Retorna se um grafo é simétrico
public static boolean isSymmetric(ALGraph graph) {
    for (int i = 0; i < graph.n; i++) {
        ArrayList<Integer> l = graph.adj.get(i);

        for (int v : l) {
            if (!graph.adj.get(v).contains((Integer) i)) {
                return false;
            }
        }
    }

    return true;
}
```

```
Exercício 27
O grafo é simétrico
O grafo é assimétrico
```

28.

```
/*
 * Exercício 28 * Um grafo pode ser armazenado em um arquivo com o seguinte formato:
* 6 * 8 * 0 1 * 0 5 * 1 0 * 1 5 * 2 4 * 3 1 * 4 3 * 3 5 * Onde na primeira linha
contém um inteiro V (vértice), na segunda contém um inteiro A (arestas) e nas demais
* linha contém dois inteiros pertencentes ao intervalo 0..V-1. Se interpretarmos
cada linha do arquivo como uma * aresta, podemos dizer que o arquivo define um grafo
com vértices 0..V-1. * Escreva um método que receba um nome de arquivo com o formato
acima e construa a representação de  lista de * adjacência do grafo. */
System.out.println("\nExercício 28");

ALGraph fileGraph = ALGraph.createFromFile("example.graph");

if (fileGraph != null) {
    fileGraph.show();
}
```

```
public static ALGraph createFromFile(String path) {
    try {
        File graphFile = new File(path);
        Scanner fileReader = new Scanner(graphFile);

        int verticesCount = fileReader.nextInt();

        ALGraph graph = new ALGraph(verticesCount);

        int edgeCount = fileReader.nextInt();

        int read = 0;
        while (read++ < edgeCount) {
            int v = fileReader.nextInt();
            int w = fileReader.nextInt();
            graph.insertEdge(v, w);
        }

        fileReader.close();
        return graph;
    } catch (FileNotFoundException e) {
        System.err.println("Erro ao ler arquivo.");
        // e.printStackTrace();
    }

    return null;
}
```

```
Exercício 28
Total de arestas: 8
0: 1 5
1: 0 5
2: 4
3: 1 5
4: 3
5:
```

29.

```java
/*
 * Exercício 29 * Fazer um método que permita remover um vértice do Grafo (não
dirigido). Não se esqueça de remover as arestas * associadas. */
System.out.println("\nExercício 29");

ALGraphND ndGraph = new ALGraphND(4);

ndGraph.insertEdge(0, 1);
ndGraph.insertEdge(1, 2);
ndGraph.insertEdge(2, 3);
ndGraph.insertEdge(3, 0);

ndGraph.show();
ndGraph.removeVertex(3);
ndGraph.show();
```

```java
// Remove o vértice v
public void removeVertex(int v) {
    for (int i = 0; i < this.n; i++) {
        this.removeEdge(i, v);
    }
}
```

```
Exercício 29
Total de arestas: 4
0: 1 3
1: 0 2
2: 1 3
3: 2 0

Total de arestas: 2
0: 1
1: 0 2
2: 1
3:
```

30.

```java
/*
 * Exercício 30 * Fazer um método que permita remover um vértice do Grafo
(dirigido). Não se esqueça de remover as arestas * associadas. */
System.out.println("\nExercício 30");

ALGraph dGraph = new ALGraph(4);

dGraph.insertEdge(0, 1);
dGraph.insertEdge(1, 2);
dGraph.insertEdge(2, 3);
dGraph.insertEdge(3, 0);

dGraph.show();
dGraph.removeVertex(3);
dGraph.show();
```

```java
// Remove o vértice v
public void removeVertex(int v) {
    for (int i = 0; i < this.n; i++) {
        this.removeEdge(i, v);
        this.removeEdge(v, i);
    }
}
```

```
Exercício 30
Total de arestas: 4
0: 1
1: 2
2: 3
3: 0

Total de arestas: 2
0: 1
1: 2
2:
3:
```

31.

```
/*
 * Exercício 31 * Fazer um método que verifique se o grafo (dirigido ou não) é
completo. */
System.out.println("\nExercício 31");

ALGraph dComplete = new ALGraph(2);
dComplete.insertEdge(0, 1);
dComplete.insertEdge(1, 0);

ALGraph dIncomplete = new ALGraph(2);

System.out.println("O grafo direcionado é: " + (dComplete.isComplete() ? "completo"
: "incompleto"));
System.out.println("O grafo direcionado é: " + (dIncomplete.isComplete() ?
"completo" : "incompleto"));

ALGraphND ndComplete = new ALGraphND(2);
ndComplete.insertEdge(0, 1);

ALGraphND ndIncomplete = new ALGraphND(2);

System.out.println("O grafo não direcionado é: " + (ndComplete.isComplete() ?
"completo" : "incompleto"));
System.out.println("O grafo não direcionado é: " + (ndIncomplete.isComplete() ?
"completo" : "incompleto"));
```

```
// Retorna se o grafo é completo
public boolean isComplete() {
    int totalPossibilities = (this.n * this.n) - this.n;
```

```
        return this.m == totalPossibilities;
    }
```

```
Exercício 31
O grafo direcionado é: completo
O grafo direcionado é: incompleto
O grafo não direcionado é: completo
O grafo não direcionado é: incompleto
```