# Matriz de adjacência em Java e plotagem de grafo em python

**Álex Dias, Enzo Maldinni Montanha Rodrigues**

Departamento de Engenharia da Computação – Universidade Estadual do Maranhão (UEMA)
91.501-970 - São Luís – MA – Brasil

alex.20210024649@aluno.uema.br, enzo.20210002991@aluno.uema.br

## 1. Introdução

Os grafos são estruturas matemáticas utilizadas para representar a relação entre objetos ou entidades. São compostos em um conjunto de vértices conectados por arestas, que indicam a existência de uma interação entre vértices correspondentes. Esses grafos são amplamente utilizados em muitos campos da ciência, como computação, matemática, física e biologia.

A matriz de adjacência é uma das técnicas mais comuns de representação de matrizes no computador. Em suma, seja e = uv uma aresta de um grafo G dizemos que os vértices u e v são vizinhos e que são vértices adjacentes(MOTA, 2019), além disso, sempre gerará uma matriz quadrada de diagonal principal igual a 0. São uma forma de representar um grafo como uma matriz bidimensional, onde as linhas e colunas correspondem aos vértices, e os elementos indicam se existem arestas entre eles. Em um grafo não direcionado, a ordem entre os vértices de uma aresta não importa, enquanto em um grafo direcionado, a ordem das arestas é fundamental para o entendimento do grafo. O grau de um vértice v de um grafo G, denotado por dG(v), é a quantidade de vizinhos do vértice v. Já o conjunto dos vizinhos de v, a vizinhança de v, é denotado por NG(v)(MOTA, 2019).

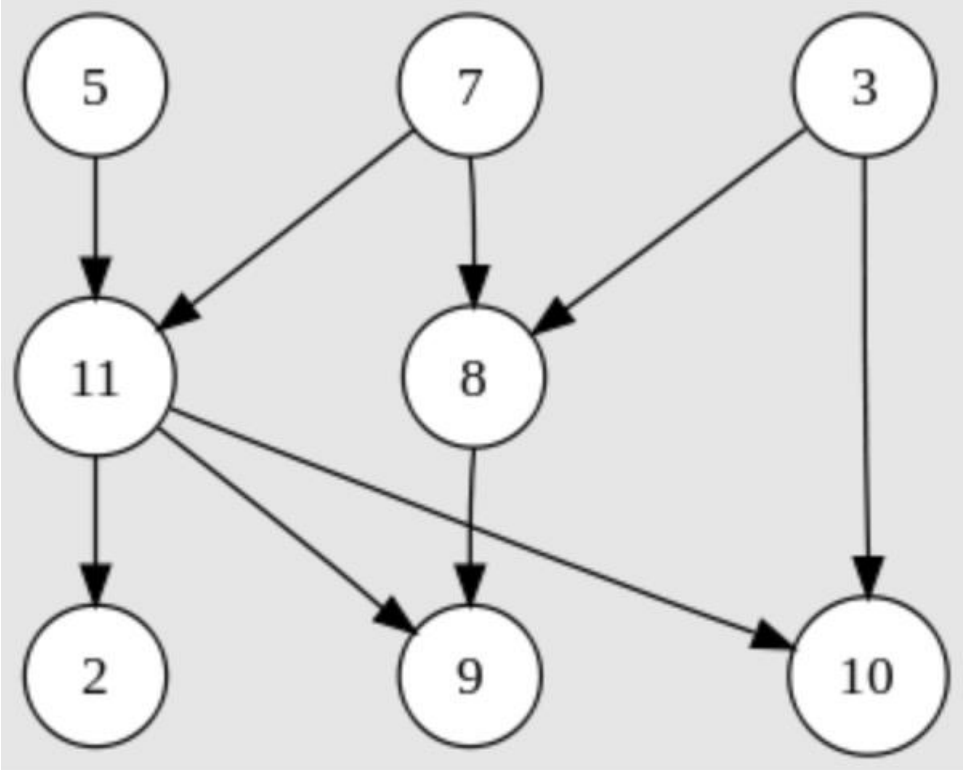


**Figura 1. Exemplo de grafo dirigido**

As matrizes de adjacência são um exemplo de como os grafos se relacionam com as estruturas de dados. Eles são uma maneira eficiente e compacta de representar grafos na memória, permitindo que sejam facilmente manipulados e percorridos em algoritmos. Outras estruturas de dados comumente usadas para representar gráficos incluem listas de adjacência e matrizes de incidência. Estas estruturas podem ser escolhidas de acordo com o tipo de operações a serem realizadas no grafo e as características específicas da aplicação associada.

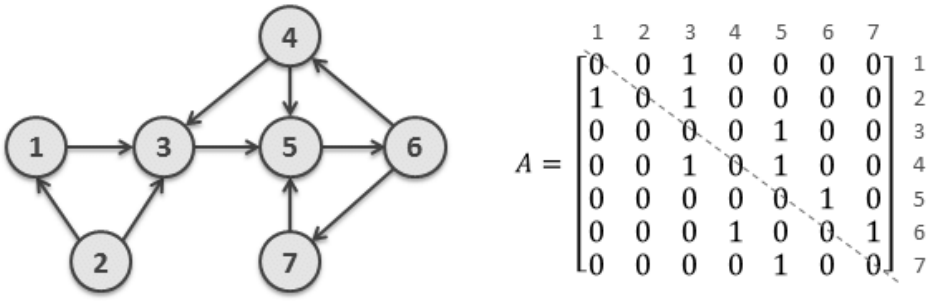


**Figura 2. Matriz de adjacência de um grafo dirigido**

## 2. Metodologia

Ao iniciar, o programa receberá um arquivo .txt com a instruções necessárias para a criação da matriz sendo elas “ND” para não dirigido e “D” para dirigido, nas próximas linhas deverá estar definidas as arestas do grafo no formato (v,w).

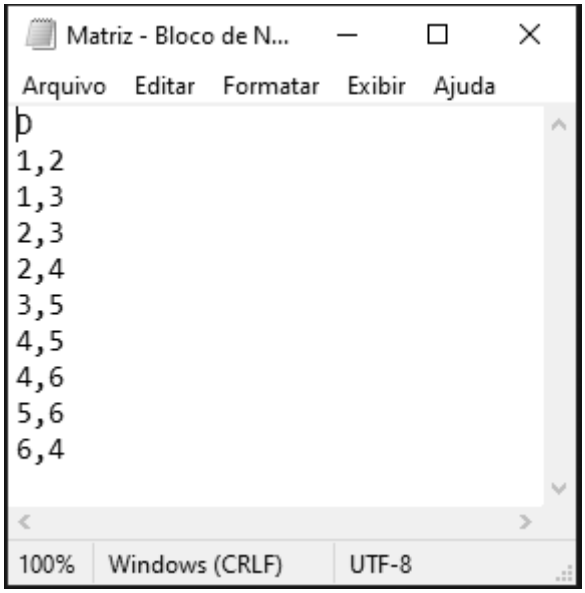


**Figura 3. Arquivo .txt com as informações de geração da matriz**

Em seguida, o programa iniciará com a função de recebimento do arquivo .txt através das bibliotecas **java.io.File** e **java.util.Scanner** para ler o conteúdo do arquivo.

```
public static void main(String[] args) {
    // Leitura do arquivo
    File file = new File(pathname:"vertices.txt");
    Scanner scanner;
    try {
        scanner = new Scanner(file);
    } catch (FileNotFoundException e) {
        System.out.println(x:"Arquivo nÃ£o encontrado");
        return;
    }
```

**Figura 4. Bloco de código que lê o primeiro arquivo .txt**

Escaneia a primeira linha e cria o booleano direcionado que receberá 1 ou 0 de acordo se é direcionado ou não que servirá para o futuro do programa.

```
    String tipo = scanner.nextLine();
    boolean direcionado = tipo.equals(anObject:"D");
    System.out.println("Tipo do grafo: " + tipo);
```

**Figura 5. Bloco de código que define se a matriz é direcionada ou não.**

O código começa criando uma variável numVertices com valor 0 continua o escaneamento final, agora para receber os vértices de cada linha em formato (v,w) dentro de um while até acabar a quantidade de linhas escritas. Dentro do loop, ele usa a vírgula como separador e logo converte os valores em inteiros com um cast, calcula o número

máximo de vértices que há no grafo e cria uma matriz de adjacência com o tamanho registrado.

Dentro do segundo loop while, ele lê cada linha do arquivo e divide-a em um array de Strings, assim como no primeiro loop. Em seguida, ele converte os elementos do array em inteiros e subtrai 1 de cada um deles para ajustar os índices da matriz de adjacência, pois os valores em java começam no 0, e define a entrada correspondente na matriz de adjacência como 1 para cada par de vértices lidos. Se o grafo não for direcionado, ele define a entrada na matriz de adjacência como 1 para todas as direções.

```
int numVertices = 0;
while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    String[] vertices = line.split(regex:",");
    int origem = Integer.parseInt(vertices[0]);
    int destino = Integer.parseInt(vertices[1]);
    numVertices = Math.max(numVertices, Math.max(origem, destino));
}
int[][] matrizAdjacente = new int[numVertices][numVertices];
scanner.close();
try {
    scanner = new Scanner(file);
} catch (FileNotFoundException e) {
    System.out.println(x:"Arquivo nÃ£o encontrado");
    return;
}
scanner.nextLine(); // Pula a primeira linha (tipo de grafo)
while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    String[] vertices = line.split(regex:",");
    int origem = Integer.parseInt(vertices[0]) - 1;
    int destino = Integer.parseInt(vertices[1]) - 1;
    matrizAdjacente[origem][destino] = 1;
    if (!direcionado) {
        matrizAdjacente[destino][origem] = 1;
    }
}
```

**Figura 6. Blocos de códigos responsáveis pela criação da matriz**

O bloco de código a seguir trabalha com loops genéricos aninhados para imprimir a matriz no console e escrevê-la em um novo arquivo .txt para ser usada no futuro código em python.

```
System.out.println(x:"Matriz de adjacÃªncia:");
for (int i = 0; i < numVertices; i++) {
    for (int j = 0; j < numVertices; j++) {
        System.out.print(matrizAdjacente[i][j] + " ");
    }
    System.out.println();
}

try {
    PrintWriter writer = new PrintWriter(fileName:"matriz.txt", csn:"UTF-8");
    if (direcionado) {
        writer.println(x:"D");
    } else {
        writer.println(x:"ND");
    }
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            writer.print(matrizAdjacente[i][j] + " ");
        }
        writer.println();
    }
    writer.close();
} catch (FileNotFoundException e) {
    System.out.println(x:"Arquivo nÃ£o encontrado");
} catch (UnsupportedEncodingException e) {
    System.out.println(x:"CodificaÃ§Ã£o nÃ£o suportada");
}
```

**Figura 7. Blocos de códigos capazes de imprimir a matriz**

Loop simples que exibirá o menu da aplicação e em seguida um switch que seguirá a escolha do usuário.

```
int choice = 0;
try (Scanner input = new Scanner(System.in)) {
    while (choice != 5) {
        System.out.println(x:"Escolha uma opÃ§Ã£o:");
        System.out.println(x:"1. Verificar se dois vertices sao adjacentes");
        System.out.println(x:"2. Calcular o grau de um vertice qualquer");
        System.out.println(x:"3. Buscar todos os vizinhos de um vertice qualquer");
        System.out.println(x:"4. Visitar todas as arestas do grafo");
        System.out.println(x:"5. Sair");

        choice = input.nextInt();

        switch (choice) {
            case 1:
```

**Figura 8. Bloco de código que imprime o menu da aplicação**

A aplicação computará todas as informações e definirá as arestas e vértices e imprimirá a matriz de adjacência desejada. Agora, o console exibirá um menu com as opções disponíveis da matriz criada, sendo elas:

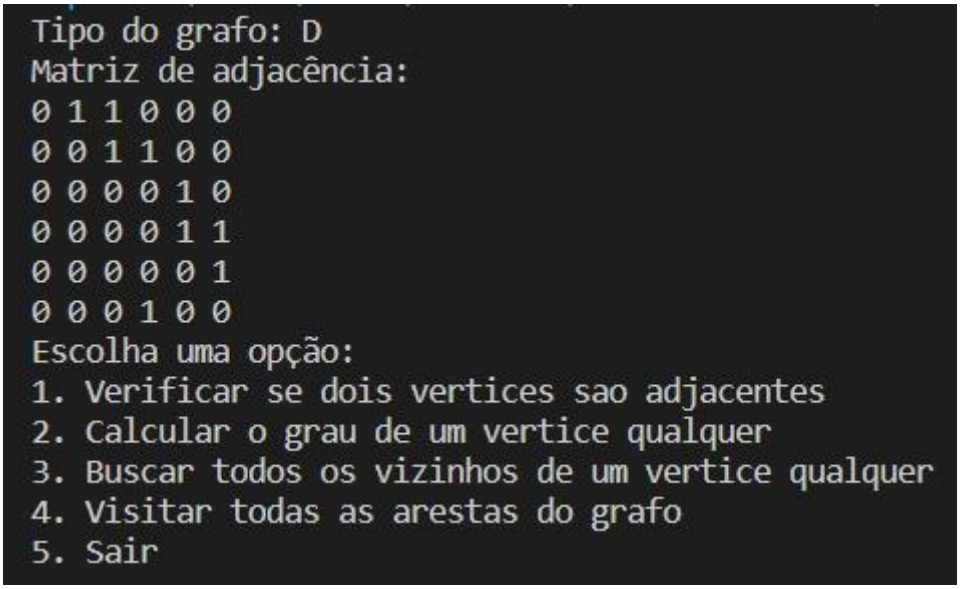


**Figura 9. Console exibindo o menu com as opções disponíveis**

1. Verificar se dois vértices são adjacentes
2. Calcular o grau de um vértice qualquer
3. Buscar todos os vizinhos de um vértice qualquer
4. Visitar todas as arestas do grafo
5. Encerrar a aplicação

Ao selecionar a opção desejada, o console imprimirá os resultados da escolha.

Recebe dois vértices introduzidos pelo usuário e procura se eles existem, depois cria uma variável booleana “adjacente” inciada com false. Após isso, testa com uma conta se os vértices são adjacentes e com um if-else imprime um texto exibindo seu status de adjacente ou não.

```
Scanner sc = new Scanner(System.in);
System.out.print(s:"Digite o primeiro vÃ©rtice: ");
int v = sc.nextInt();
System.out.print(s:"Digite o segundo vÃ©rtice: ");
int vx = sc.nextInt();
scanner.close();
boolean adjacente = false;
if (matrizAdjacente[v - 1][vx - 1] == 1 || matrizAdjacente[vx - 1][v - 1] == 1) {
    adjacente = true;
}

if (adjacente) {
    System.out.println("Os vÃ©rtices " + v + " e " + vx + " sÃ£o adjacentes.");
} else {
    System.out.println("Os vÃ©rtices " + v + " e " + vx + " nÃ£o sÃ£o adjacentes.");
}

break;
```

**Figura 10. Bloco de código que inicia a primeira opção da aplicação, responsável por verificar se dois vértices são adjacentes.**

Primeiramente, recebe o vértice do usuário e testa se existe, depois cria uma várivel inteira “grau” com valor 0 e passa para um loop genérico que percorrerá toda a matriz e dentro dele um if com uma condição que aumentará o grau toda vez que for satisfeita. Encerrando imprimindo o grau do vértice solicitado.

```
case 2:
    Scanner sca = new Scanner(System.in);
    System.out.print(s:"Digite o vÃ©rtice para calcular o grau: ");
    int vertice = sca.nextInt();
    int grau = 0;
    for (int i = 0; i < matrizAdjacente.length; i++) {
        if (matrizAdjacente[vertice - 1][i] == 1) {
            grau++;
        }
    }
    System.out.println("O grau do vÃ©rtice " + vertice + " Ã© " + grau);

    break;
```

**Figura 11. Bloco de código que inicia a segunda opção da aplicação, responsável por calcular o grau de um vértice qualquer.**

Recebe o vértice do usuário e testa se existe, depois passa para um loop genérico que percorrerá toda a matriz e dentro dele um if com uma condição que imprimirá o vizinho do vértice selecionado toda vez que for satisfeita.

```
case 3:
    Scanner sc3 = new Scanner(System.in);
    System.out.print(s:"Digite o vÃ©rtice: ");
    int vertice3 = sc3.nextInt();

    System.out.print("Os vizinhos do vÃ©rtice " + vertice3 + " sÃ£o: ");
    for (int i = 0; i < numVertices; i++) {
        if (matrizAdjacente[vertice3 - 1][i] == 1) {
            System.out.print(i + 1 + " ");
        }
    }
    System.out.println();
    break;
```

**Figura 12. Bloco de código que inicia a terceira opção da aplicação, responsável por buscar todos os vizinhos de um vértice qualquer.**

Visita todas arestas do grafo através de um for aninhado e uma condição de existência pré-estabelecida. No fim, imprime a aresta entre os vértices que percorreu.

```
case 4:

    System.out.println(x:"Visitando todas as arestas do grafo:");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            if (matrizAdjacente[i][j] == 1) {
                System.out.println("Aresta entre vÃ©rtices " + (i + 1) + " e " + (j + 1));
            }
        }
    }
    break;
```

**Figura 13. Bloco de código que inicia a primeira opção da aplicação, responsável por visitar todas as arestas do grafo.**

Por fim, a aplicação java gerará um novo arquivo .txt que será utilizado na plicação em python para a plotagem do grafo, utilizando as bibliotecas **networkx,** para a criação do grafo a partir das informações obtidas da matriz de adjacência no arquivo txt, e **matplotlib.pyplot**, para a plotagem do grafo gerado. Por fim, o programa abrirá uma janela com o grafo produzido.

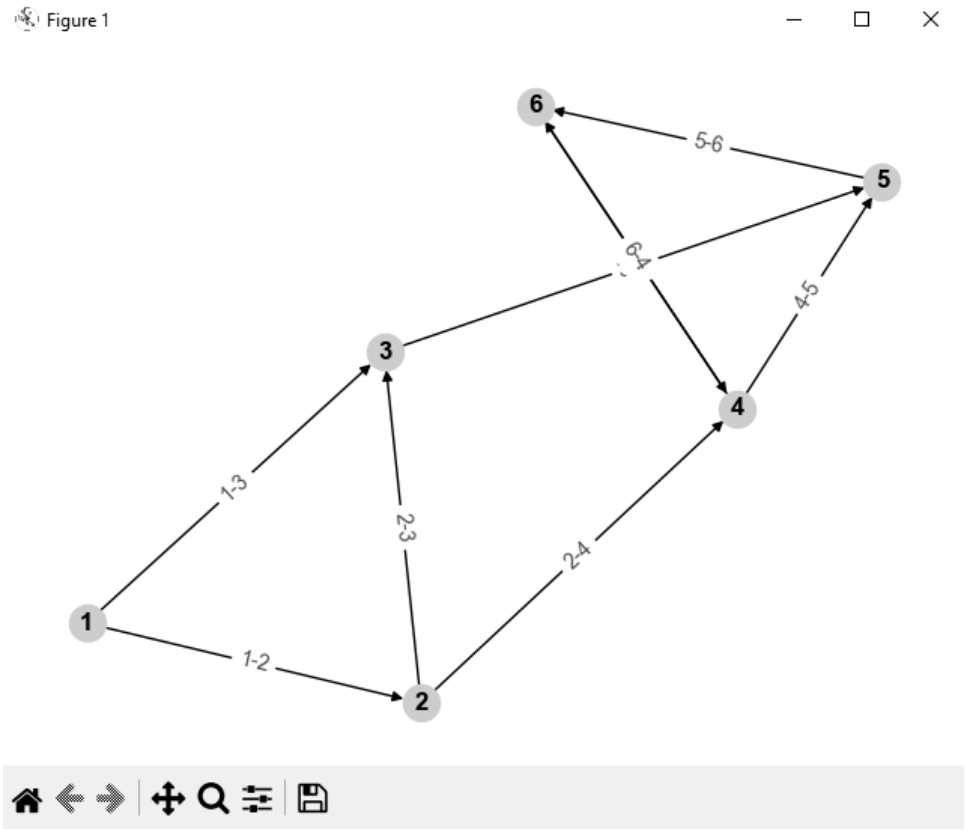


**Figura 14. Janela com a plotagem do grafo dirigido gerado em python com vértices (1,2) (1,3) (2,3) (2,4) (3,5) (4,5) (4,6) (5,6) (6,4)**

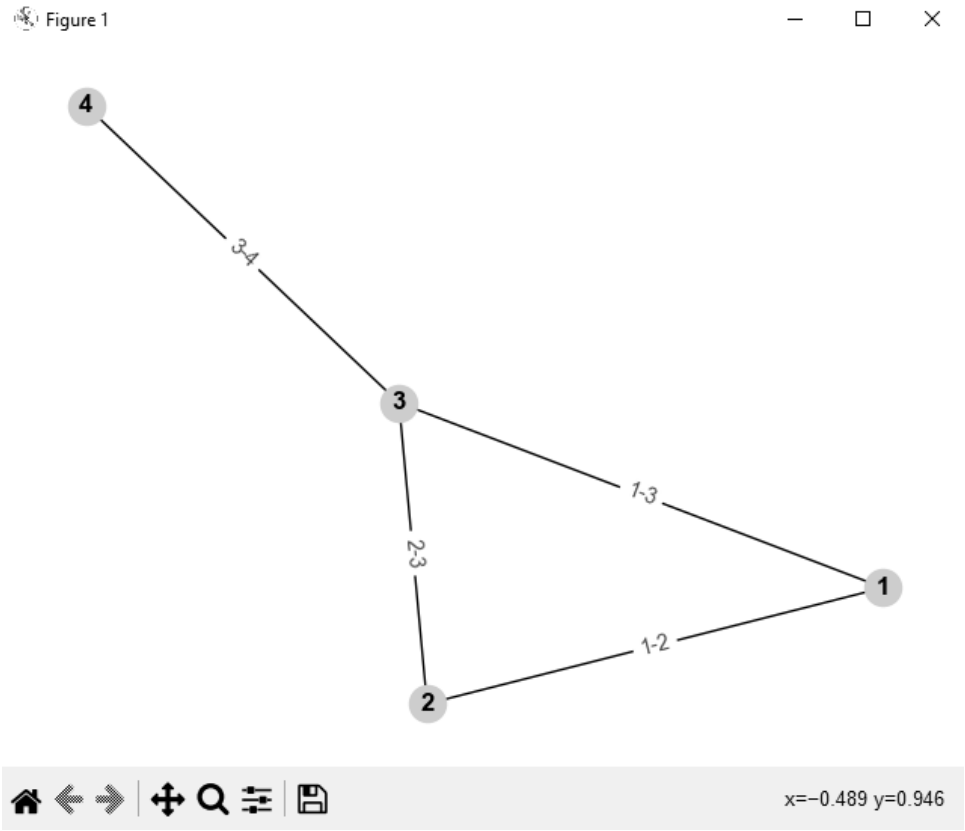


**Figura 15. Janela com a plotagem do grafo não dirigido gerado em python Com vértices (1,2)(1,3)(2,3)(3,4).**

Além disso, a aplicação oferece dois executáveis “projetoMatrizAdjacencia.jar” e “mostrarGrafo.exe”, e um arquivo .txt “vertices.txt” que contém as informações da matriz, dentro dele pode se alterá-la, após inserir a informações será executado primeiramente o arquivo **projetoMatrizAdjacencia** e ele criará o segundo arquivo .txt “matriz” com a matriz de adjacência totalmente formada, em seguida o segundo executável **mostrarGrafo** poderá ser aberto e exibirá o grafo plotado.

É importante ressaltar que as opções de manipulação da matriz, como calcular o grau de um vértice qualquer, não serão ofertadas apenas utilizando os executáveis, faz-se necessário executar a **aplicação java** direto do **código-fonte.**

## Referências

OpenAI. Demystifying GPT-3. Disponível em: https://openai.com/demystifying-gpt-3/. Acesso em: 25 abr. 2023.

DevMedia. Manipulando projetos no Eclipse. Disponível em: https://www.devmedia.com.br/manipulando-projetos-no-eclipse/25338#:~:text=Para%20abrir%20um%20projeto%20dentro,botão%20esquerdo%20no%20mesmo%20projeto. Acesso em: 26 abr. 2023.

Mota, Guilherme O. (2019), Teoria dos grafos, 1ª edição, São Paulo, UFABC