Relatório Individual das Implementações em Teoria dos Grafos

Nome: Júlia Silva Castro

SUMÁRIO

1) Objetivo

O objetivo é conciliar a parte lógica com a parte teórica do curso de teoria dos grafos. Aprendemos os conceitos e aperfeiçoamos o aprendizado ao longo das implementações. No programa analisamos as características de cada grafo através de suas matrizes de incidência ou matrizes de adjacência (com pesos), além de utilizar alguns algoritmos famosos como: Dijikstra, Kruskal e Floyd-Warshall.

2) Funções Implementadas

A criação do programa se deu na plataforma Netbeans IDE 8.0.2 utilizando a linguagem Java por termos uma maior afinidade e ser mais completa. Foram usadas as seguintes estruturas de dados: matrizes, vetor e lista.

Implementamos:

a) Ler arquivo e cria matrizes
public static int[][] = Arquivo.lerEGravarMatrizAdjacencia()
public static int[][] = Arquivo. LerEGravarMatrizIncidencia()
public static int[][] = Arquivo.lerEGravarMatrizAdjacenciaPonderada()

a.1) Métodos Auxiliares:
Matriz criaMatriz(tamanho, tamanho)

b) Verifica se é grafo simples
public Boolean verificaSimplicidade(int matrizAdjacencia [][], int matrizIncidencia[][])

b.1) Métodos Auxiliares:
public Boolean verificaSeTemLaco (int matrizAdjacencia [][]);
public Boolean verificaSeMultigrafo(int matrizAdjacencia[][], int matrizIncidencia[][])

c) Verifica se é multigrafo
public Boolean verificaSeMultigrafo(int matrizAdjacencia[][], int matrizIncidencia[][])

d) Verifica se é um pseudografo
public Boolean verificaSePseudografico(int matrizAdjacencia[][])

d.1) Métodos Auxiliares:
public Boolean verificaSeTemLaco (int matrizAdjacencia [][])

e) Verifica se é Ciclo:

Public Boolean verificaSeCiclo(int matizAdjacencia[][], int matrizIncidencia[][], Boolean conexo)

e.1) Métodos Auxiliares:

public Boolean verificaGrauDosVertices(int grauEsperado, int matizIncidencia[][])

public Boolean verificaSeConexo(int[][] matrizIncidencia, int[][] matrizAdjacencia)

f) Verifica se é completo

public Boolean verificaSeCompleto(int matrizIncidencia[][])

f.1) Métodos Auxiliares:

verificaGrauDosVertices(grauVerticeEsperado, matrizIncidencia)

g) Verifica se é conexo

public Boolean verificaSeConexo(int[][] matrizIncidencia, int[][] matrizAdjacencia)

g.1) Métodos Auxiliares:

public Boolean verificaGrauDosVertices(int grauEsperado, int matrizIncidencia[][])

public int[] vetFechoTransitivoDeUmVertice(int vertice)

h) Calcula a ordem do grafo

public int ordemDoGrafo()

i) Calcula o tamanho do grafo

public int tamanhoDoGrafo()

j) Lista de Adjacência

Public void imprimeListaDeAdjacencia()

j.1) Métodos Auxiliares:

public void preencheListaDeAdjacencia (int matrizAdjacencia[][], int n)

public ArrayList<No> recebeVertices (int matrizAdjacencia[][], int n)

k) Gerar Caminho mínimo

public int[] caminhoMinimo_Dijkstra(int matrizAjacenciaPonderada[][], int

vInicial)

k.1) Métodos Auxiliares:

public int[] dijikstraInicializacao (int valor, int vet[])
public public int acharOmenor (int vetorVerticesPassados[])
public verificaVetor (int valor, int vet[])

i) Matriz de Distância

public int[][] matrizDeDistancia(int matrizAjacenciaPonderada[][])

i.1) Método Auxiliares:

public int[] caminhoMinimo_Dijkstra(matrizAjacenciaPonderada, i);

j) Árvore Geradora mínima

string arvoreGeradoraMinima (Boolean verificaSeConexo, int matrizAjacenciaPonderada[][], int [] vet)

j.1) Métodos Auxiliares:

public Boolean verificaSeTemCiclo(int matrizFechoDoGrafo[][])

3) Desenvolvimento

Cada conceito de grafos aprendido durante as aulas foi implementado, a parte teórica transformou-se em lógica, deixando o curso mais interessante e desafiador.
O programa se inicia através da leitura de dois arquivos txt, respectivamente, matriz de adjacência com pesos e incidência. Utilizamos as matrizes na maioria das funções para caracterização do grafo.

grafoAdjacencia

Arquivo Editar Fo

```
5
1_2_10
1_3_5
1_5_8
2_1_10
2_3_3
2_4_2
3_1_5
3_2_3
3_4_9
3_5_2
4_2_2
4_3_9
4_5_4
5_1_8
5_3_2
5_4_4
```

grafoIncidencia

Arquivo Editar F

```
5
8
1_1
1_5
2_1
2_3
3_1
3_2
4_2
4_3
5_2
5_4
6_4
6_5
7_3
7_5
8_3
8_4
```

Figura 1: Matriz de Adjacência e Matriz de Incidência.

A primeira função criada foi para verificar se no grafo existia laço. Nessa função utilizamos a Matriz de Adjacência para saber se no mesmo vértice possui 1 ligação, se sim, concluímos que o grafo tem laço.

```
public Boolean verificaSeTemLaco(int matrizAdjacencia[][]) {
    for (int i = 1; i < n; i++) {
        if (matrizAdjacencia[i][i] == 1) {
            return true;//Verificando se existe laço
        }
    }
    return false;
}
```

Figura 2: Método para verificar se existe laço no grafo.

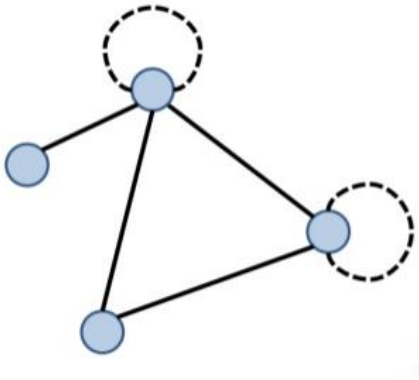

Figura 3: Exemplificação de um pseudografo.

O método verificaSeTemLaco() auxiliou duas outras funções: verificaSimplicidade() e verificaSePseudografico().

Um **pseudografo** deve possuir ao menos um laço então basta conferir se na função anterior retornou algum valor de laço para concluirmos.

```
//É Pseudografo se existe no minimo um laco
public Boolean verificaSePseudografico(int matrizAdjacencia[][]) {
    if (verificaSeTemLaco(matrizAdjacencia) == true) {
        return true;
    } else {
        return false;
    }
}
```

Figura 4: Método para verificar se é pseudografo.

Em seguida criamos a função para verificar se é um **multigrafo**, segundo a definição, um multigrafo possui arestas paralelas. Nesse método temos um contador de arestas que percorre toda matriz de adjacência para verificar em quais vértices tem ligações e em seguida criamos outro for para percorrer a matriz de incidência utilizando o mesmo contador para verificar se a aresta liga os dois vértices. Se possuir duas ou mais arestas ligando os mesmos vértices retornamos que é um multigrafo.

```
//É multigrafo se existe arestas paralelas
public Boolean verificaSeMultigrafo(int matrizAdjacencia[][], int matrizIncidencia[][]) {
    int contArestaParalela = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (matrizAdjacencia[i][j] == 1) {//Verificando se onde tem ligaçao também existe arestas Paralelas
                contArestaParalela = 0;
                for (int aresta = 1; aresta <= m; aresta++)//vai percorrer todas as arestas
                {
                    if (matrizIncidencia[aresta][i] == 1 && matrizIncidencia[aresta][j] == 1)//Verifica se a aresta liga os dois vertices
                    {
                        contArestaParalela++;
                    }
                }
            }
        }
    }
    if (contArestaParalela >= 2) {
        return true;
    }
    return false;
}
```

Figura 5: Método para verificar o grafo inserido é um multigrafo.

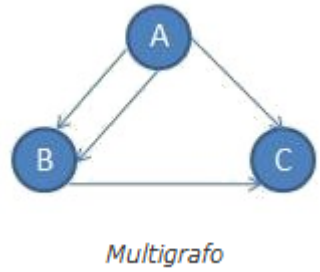

Figura 6: Exemplificação de um Multigrafo.

Chamando as funções verificaSeMultigrafo() e VerificaSeTemLaco() podemos verificar facilmente se o grafo é simples, que por definição um **grafo simples** não pode ter arestas paralelas e laços.

```
//É simples se não tem laço e não tem arestas paralelas
public Boolean verificaSimplicidade(int matrizAdjacencia[][], int matrizIncidencia[][]) {
    if (verificaSeTemLaco(matrizAdjacencia) == false
            && verificaSeMultigrafo(matrizAdjacencia, matrizIncidencia) == false) {
        return true;
    } else {
        return false;
    }
}
```

Figura 7: Método para verificar se o grafo é simples.

A próxima função criada é uma auxiliar, ela serve para calcularmos o grau de cada vértice. Nós passamos por parâmetro a matriz de incidência e o grau esperado. O grau do vértice é calculado pela quantidade de arestas que incidem nele.

```
public Boolean verificaGrauDosVertices(int grauEsperado, int matrizIncidencia[][]) {
    int vetGraus[] = vetorGrauDosVertices(matrizIncidencia);
    boolean resultado=true;
    for (int i = 1; i < vetGraus.length; i++) {
        if (vetGraus[i] != grauEsperado) {
            resultado = false;
        }
    }
    return resultado;
}
```

Figura 8: Método Auxiliar de verificar o grau dos vértices. Usada também para verificar se o grafo é completo, ciclo e conexo.

Na função para verificar se o **grafo é completo**, chamamos a função anterior de verificar o grau dos vértices. Além disso, fazemos uso de fórmulas, para um grafo ser completo é necessário ter um número n*(n-1)/2 de arestas e todos os vértices possuírem grau n-1, sendo N o número de vértices.

```java
// É completo se a quantidade de Arestas=(n*(n-1))/2 e se todos os vértices possuem grau=n-1
public Boolean verificaSeCompleto(int matrizIncidencia[][]) {
    int qntdEsperadaArestas = (n * (n - 1)) / 2;
    int grauVerticeEsperado = n - 1;
    Boolean grausQueNemOesperado = verificaGrauDosVertices(grauVerticeEsperado, matrizIncidencia);
    if (qntdEsperadaArestas == m && grausQueNemOesperado == true) {
        return true;
    }
    return false;
}
```

Figura 9: Método para verificar se o grafo é completo.

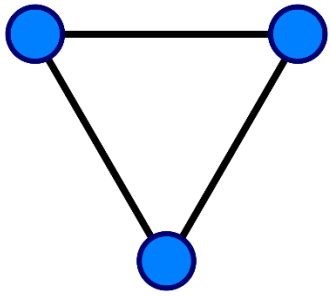

Figura 10: Exemplificação de um grafo completo com 3 vértices, cada vértice com grau 2 e 3 arestas.

A função para verificar se o **grafo é conexo** passamos por parâmetro a matriz de incidência e adjacência. A primeira etapa é conferir se existe algum vértice de grau 0, pois se existir é desconexo e para isso chamamos a função auxiliar que verifica o grau dos vértices, vista na figura 8. Se não possuir vértice de grau 0, criamos um for e um vetor com o fecho transitivo do vértice para verificar se todos os vértices alcançam os outros vértices, então, se existir um caminho para cada par de vértices concluímos que ele é conexo.

```java
public Boolean verificaSeConexo(int[][] matrizIncidencia, int[][] matrizAdjacencia) {
    int vet[] = new int[n + 1];
    if (verificaGrauDosVertices(0, matrizIncidencia)) {
        return false;
    }
    for (int i = 1; i <= n; i++) {
        vet = vetFechoTransitivoDeUmVertice(i);
        for (int j = 1; j <= n; j++) {
            if (vet[j] != j) {
                return false;
            }
        }
    }
    return true;
}
```

Figura 11: Método para verificar se o grafo é conexo.

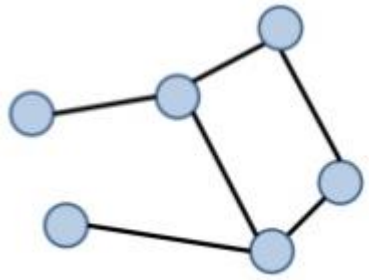

Figura 12: Exemplo de um grafo conexo.

Na função criada para verificar se o **grafo é ciclo**, por definição, cada vértice precisa ter grau 2 e ser um grafo conexo. Nesse método, usamos como parâmetro a matriz de adjacência e incidência e também utilizamos a função auxiliar que verifica o grau dos vértices.

```java
public Boolean verificaSeCiclo(int matrizAdjacencia[][], int matrizIncidencia[][], Boolean conexo)
    if (conexo) {
        if (verificaGrauDosVertices(2, matrizIncidencia)) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

Figura 13: Método para verificar se o grafo é ciclo

Por definição, a **ordem do grafo** é dada pela quantidade de vértices e o **tamanho do grafo** é dado pela quantidade de arestas. A implementação para retornar esses valores foi bem fácil, simples e rápida.

```
public int ordemDoGrafo() {
    return n;
}

public int tamanhoDoGrafo() {
    return m;
}
```

Figura 14: Métodos ordemDoGrafo() e tamanhoDoGrafo().

Para as funções de fecho transitivo de vértice e grafo utilizamos o **algoritmo de Wharsall** por ser bem simples visto que são três laços de repetição com algumas condicionais. O primeiro laço consiste na varredura das linhas da matriz, o segundo das colunas. Caso haja uma aresta, a procura na coluna correspondente é feita no terceiro laço. Se todas as correspondências forem encontradas o novo caminho é gravado na matriz. Todo este procedimento é repetido para todas as linhas da matriz fazendo com que o algoritmo chegue ao fim e torne o acesso à caminhos de um grafo. Basicamente percorre a matriz de Adjacência e altera os valores.

```
public int[][] matrizDoFechoTransitivo_Wharshal(){
    int matrizAdjacencia[][]= Arquivo.lerEGravarMatrizAdjacencia();
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                matrizAdjacencia[i][j] = matrizAdjacencia[i][j] != 0 || (matrizAdjacencia[i][k] != 0 && matrizAdjacencia[k][j] != 0) ? 1 : 0;

    return matrizAdjacencia;
}
```

Figura 15: Algoritmo de Wharsall para fecho transitivo.

Com a nova matriz podemos gerar o **fecho transitivo de um vértice**, que são todos os vértices alcançados por
um vértice. Nesse método retornamos o vetor para cada linha, representando o caminho possível para o vértice em questão.

```
public int[] vetFechoTransitivoDeUmVertice(int vertice) {
    int matrizAdjacenciaCopia[][]=matrizDoFechoTransitivo_Wharshal();
    int vet[] = new int[n + 1];
    for (int i = 0; i <= n; i++) {
        vet[i] = 0;
    }
    for (int j = 1; j <= n; j++) {
        if(matrizAdjacenciaCopia[vertice][j] ==1){
            vet[j]=j;
        }
    }
    return vet;
}
```

Figura 16: Método que retorna o fecho transitivo do vértice.

De forma análoga, podemos retornar o **fecho transitivo de um grafo** que é apenas acrescentar com todas as outras ligações que faltavam na matriz.

```
public String[] fechoTransitivoDeUmGrafo() {
    String[] transitivo = new String[1];
    transitivo[0] = "Fecho transitivo do grafo:\r\n";
    int matrizAdjacencia[][]=matrizDoFechoTransitivo_Wharshal();

    for (int i = 1; i <=n; i++) {
        for (int j = 1; j <= n; j++) {
            transitivo[0] += "  "+matrizAdjacencia[i][j];
        }
        transitivo[0] += "\r\n";
    }
    return transitivo;
}
```

Figura 17: Método que retorna o fecho transitivo do grafo.

Caminho mínimo consiste na minimização do custo de travessia de um grafo entre dois vértices; custo este dado pela soma dos pesos de cada aresta percorrida.

Na implementação, escolhemos um vértice para ser o início e o iniciamos com distancia = 999, e então calculamos a distancia dele pra todos os outros vértices conectados nele. Temos um while para continuar calculando enquanto não temos o valor da distancia para todos os vértices. O algoritmo soma o valor a distancia do vértice escolhido com a distancia do vértice conectado a ele e compara se é menor que a distância já calculada, se for ele atualiza a distancia do vértice vizinho com a soma da distancia do vértice de inicio e o peso da aresta entre eles.

.

Caminho Mínimo - Dijkstra

```
20  void dijkstra(){
21    this.dijkistraInicializacao();
22
23    while(Solução não esta completa){
24      u = Elemento de L(v) mínimo;
25      adiciona u na solucao;
26
27      Para cada(vertice vizinho v de u){
28        if( L(u) + peso(u,v) < L(v))
29          L(v) = L(u) + peso(u,v);
30      }
31    }
32  }
```

Figura 18: Método Dijikstra de melhor entendimento.

```
public int[] caminhoMinimo_Dijkstra(int matrizAjacenciaPonderada[][], int vInicial){
    int vetPesos[]= dijkstraInicializacao(999);
    vetPesos[vInicial]=0;
    int vetorVerticesPassados[]=dijkstraInicializacao(-1);
    vetorVerticesPassados[vInicial]=0;

    while(verificaVetor(-1, vetorVerticesPassados)!=0){
        for(int j=1; j<=n; j++){
            if((vetPesos[vInicial] + matrizAjacenciaPonderada[vInicial][j]<vetPesos[j]) && matrizAjacenciaPonderada[vInicial][j]!=0)
                vetPesos[j]= vetPesos[vInicial] + matrizAjacenciaPonderada[vInicial][j];
        }
        vInicial=acharOmenor(vetPesos, vetorVerticesPassados);
        vetorVerticesPassados[vInicial]=vetPesos[vInicial];
    }
    return vetorVerticesPassados;
}
```

Figura 19: Método de Caminho Mínimo utilizando Algoritmo Dijikstra implementado.

A **matriz de distâncias** é uma matriz contendo as distâncias, tomadas em pares, de um conjunto de pontos. Esta matriz terá um tamanho de N×N onde N é o número de vértices.

```
public int[][] matrizDeDistancia(int matrizAjacenciaPonderada[][]){
    int matriz[][]=Matriz.criaMatriz(n, n);
    for(int i=1;i<=n; i++){
        int vet[]=caminhoMinimo_Dijkstra(matrizAjacenciaPonderada, i);
        for(int j=1; j<=n; j++){
            matriz[i][j]=vet[j];
        }
    }
    return matriz;
}
```

Figura 20: Método que retorna a matriz de distância.

Na implementação da árvore tivemos auxílio do **algoritmo de Kruskal** que basicamente gera a arvore se o grafo for conexo e tiver pesos. Ele encontra um subconjunto das arestas que inclua todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore é minimizado, que é a definição da **árvore geradora mínima**.

```
public String arvoreGeradoraMinima(Boolean verificaSeConexo, int  matrizAjacenciaPonderada[][], int [] vetArestas){
    String res     = "";
    int[][] arvoreGeradoraMinima    = new int [ n +1][n+1];
    int [] vetPai  = new int [n+1];
    if(!verificaSeConexo)
       return "grafo desconexo nao possui arvoreGeradoraMinimaore geradora mínima";
    for(int i=1; i<(n+1); i++){
        vetPai[i] = i;
    }
    for(int k=0; k<vetArestas.length; k++)
        for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                if(vetArestas[k] ==  matrizAjacenciaPonderada[i][j])
                    if(vetPai[i] != vetPai[j]){
                        arvoreGeradoraMinima[i][j] =  matrizAjacenciaPonderada[i][j];
                        arvoreGeradoraMinima[j][i] =  matrizAjacenciaPonderada[i][j];
                        vetPai[i] = vetPai[j];
                    }
    for(int i=1; i<=n; i++){
        for(int j=1; j<=n; j++){
            res += "  "+arvoreGeradoraMinima[i][j];
        }
        res += "\n";
    }return res;
}//fim Arvore geradora minima
```

Figura 21: Método que gera a árvore geradora mínima

4- Resultados

Dado um grafo inicial, por exemplo:

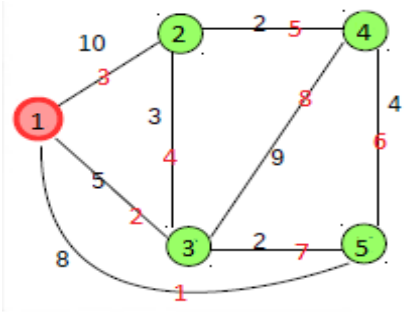

Figura 22: Grafo usado para testar o programa.

O programa retornará:

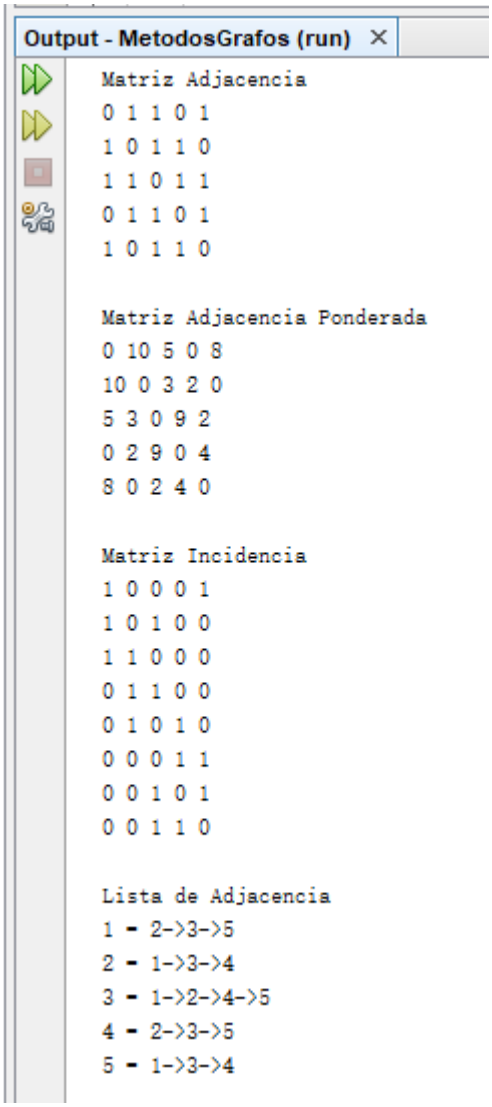

Figura 23: Resultado das estruturas de dados.

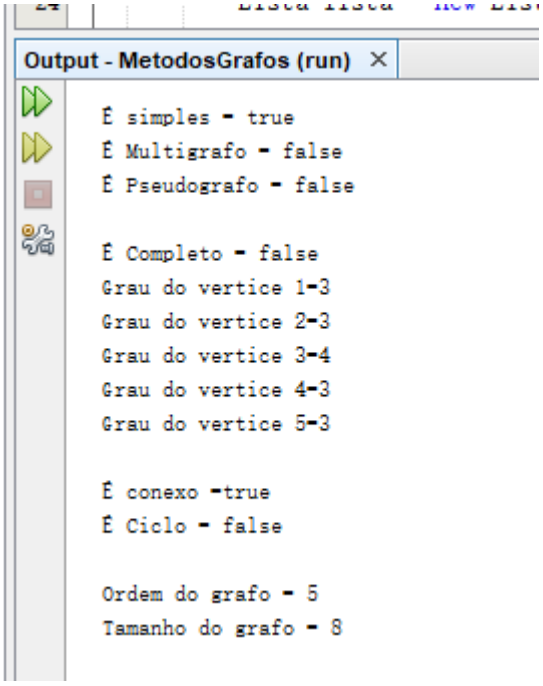

Figura 24: Resultado das características do grafo.

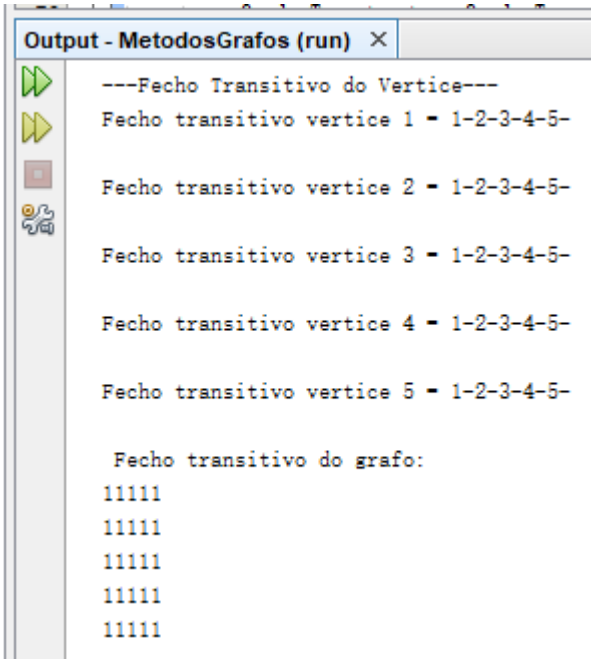

Figura 25: Resultado dos Fechos Transitivos de Vértice e do Grafo utilizando o Algoritmo de Floyd-Warshall.

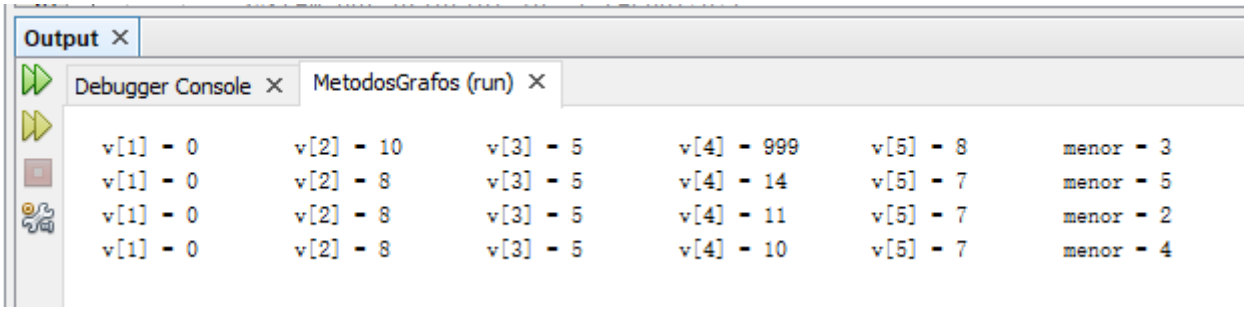

Figura 26: Resultado do algoritmo de Dijkstra.

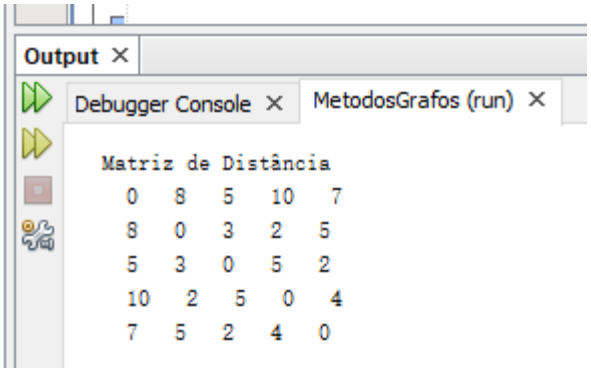

Figura 27: Resultado da Matriz de Distância.

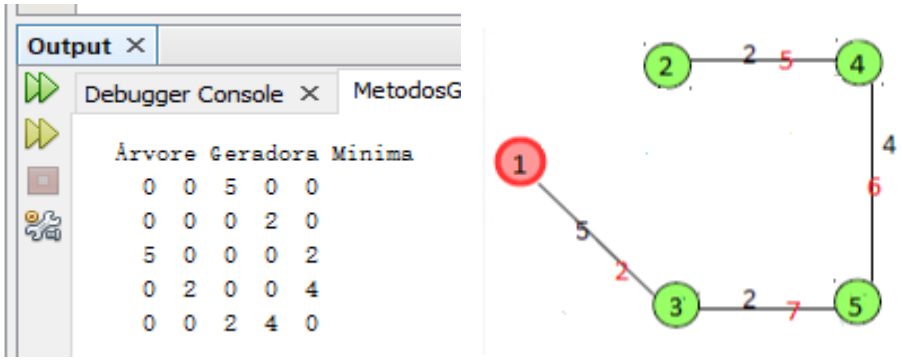

Figura 28 e 29: Resultado da Árvore Geradora Mínima.

Com a implementação, apesar da dificuldade enfrentada em transformar conceitos em lógica, os resultados saíram como esperados. Consegui também absorver melhor o conteúdo trabalhado durante o semestre e de uma forma que acrescente minha experiencia e familiaridade como programadora.

5- Referências

Conceitos, algoritmos e aplicações. GOLDBARG, Elizabeth; GOLDBARG, Marco.