Nome: Leonardo Botrel Tobias **Matrícula:** 201622040333
Data: 13/10/2017 **Número:** 06
Título da prática: Implementação em JAVA do TAD JAGM

Código que implementa a classe JAGM:

```
public class JAGM {

   private int antecessor[];//Vetor de antecessores
   private double p[];//vetor de pesos
   private JGrafo grafo;//grafo


   public JAGM (JGrafo grafo) {
        this.grafo = grafo;//Inicializa o grafo
   }


   public void obterAgm ( int raiz ) throws Exception {
        int n = this.grafo.getNumVertices();
        this.p = new double[n]; // peso dos vértices
        int vs[] = new int[n+1]; // vértices

        boolean itensHeap[] = new boolean[n];
        this.antecessor = new int[n];

        for (int u = 0; u < n; u ++) {
            this.antecessor[u] = -1;//Inicializa os antecessores com
-1
            p[u] = Double.MAX_VALUE; // Inicia p[u] como ∞
            vs[u+1] = u; // Heap indireto a ser construído
            itensHeap[u] = true;
        }

        p[raiz] = 0;
        FPHeapMinIndireto heap = new FPHeapMinIndireto (p, vs);
        heap.constroi ();


        while (!heap.vazio ()) {//Enqunto o heap não for vazio
            int u = heap.retiraMin ();//Extrai o Q minimo
            itensHeap[u] = false;

            if (!this.grafo.listaAdjVazia (u)) {
                JGrafo.Aresta adj = grafo.primeiroListaAdj (u);//adj
recebe o primeiro da lista de adjacencia
```

```
                while (adj != null) {
                    int v = adj.getV2();

                    if (itensHeap[v] && (adj.getPeso() < this.peso
(v))) {//se v pertence a Q e peso for menor que este peso
                        antecessor[v] = u;//O antecessor recebe u
                        heap.diminuiChave (v, adj.getPeso ());
                    }
                    adj = grafo.proxAdj (u);
                }
            }
        }
    }

    //Permite ao usuario da classe obter o antecessor de um certo
vértice
    public int antecessor ( int u) {
        return this.antecessor[u];
    }

    //Permite ao usuario da classe obter o peso associado a um
vértice
    public double peso (int u) {
        return this.p[u];
    }

    //Permite ao usuario da classe imprimir as arestas da árvore,
respectivamente
    public void imprime () {
        for (int u = 0; u < this.p.length; u++)
            if (this.antecessor[u] != -1)
                System.out. println ( "(" +antecessor[u]+ " ," +u+
") -- p:" + peso (u));
    }

    public double pesoAGM () {
        double pesoAGM = 0;
        for (int u = 0; u < this.p.length; u++) {
            pesoAGM += this.peso(u);
        }

        return pesoAGM;
    }

}
```

Explicação do algoritmo implementado:

O subconjunto S forma uma única árvore, e a aresta segura adicionada a S é sempre uma aresta de peso mínimo conectando a árvore a um vértice que não esteja na árvore. A árvore começa por um vértice qualquer e cresce até que “gere” todos os vértices em V. A cada passo, uma aresta leve é adicionada à árvore S, conectando S a um vértice de GS = (V, S). Quando o algoritmo termina, às arestas em S formam uma árvore geradora mínima. O grafo G e a raiz r são entradas para o algoritmo. • Os vértices residem em uma fila de prioridades mínima baseada em um campo chave (peso mínimo das arestas incidentes em um vértice). p indica o peso de cada aresta do grafo

Resultado obtido nos grafos do item 2:

O peso da primeira Árvore Geradora Mínima é: 55.0

Arestas que constituem a Árvore:
(0 ,1) −− p:5.0
(3 ,2) −− p:5.0
(4 ,3) −− p:5.0
(0 ,4) −− p:5.0
(6 ,5) −− p:15.0
(2 ,6) −− p:5.0
(1 ,7) −− p:5.0
(4 ,8) −− p:10.0

O peso da segunda Árvore Geradora Mínima é: 20.0

Arestas que constituem a Árvore:
(2 ,1) −− p:3.0
(0 ,2) −− p:2.0
(1 ,3) −− p:2.0
(6 ,4) −− p:4.0
(7 ,5) −− p:4.0
(3 ,6) −− p:2.0
(6 ,7) −− p:3.0