UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LABORATORIO DE ESTRUCTURA DE DATOS Y ALGORITMOS

Docente: EDITH PAMELA RIVERO TUPAC DE LOZANO

“Laboratorio 9 Informe: Grafos”

Alumno: Delgado Valencia, Franco Andre

Arequipa - 2021

Laboratorio 9 Informe: Grafos

Graph.java

```
package Graph;

import Exceptions.*;
import Util.*;

public class Graph<E> {
    private int tagCount = 0;
    private LinkedList<VertexNode> vertices = new
LinkedList<VertexNode>();

    public static final int UNEXPLORED = 0;
    public static final int DISCOVERY = 1;
    public static final int BACK = 2;
    public static final int VISITED = 3;
    public static final int CROSS = 4;
    public static final int INFINITE = 999999999;

    private class VertexNode {
        public E value;
        public LinkedList<EdgeNode> adjacents = new
LinkedList<EdgeNode>();
        public int label; //para dfs -> 0:unexplored, 1:discovery, 2:back

        public VertexNode(E value) {
            this.value = value;
        }

        public boolean isAdjacentTo(VertexNode vertex) {
            for(EdgeNode edgeNode : this.adjacents) {
                if(edgeNode.vertex == vertex)
                    return true;
            }
```

```
            for(EdgeNode edgeNode : vertex.adjacents) {
                if(edgeNode.vertex == this)
                    return true;
            }

            return false;
        }

        public Edge edgeWith(VertexNode vertex){
            for(EdgeNode edgeNode : this.adjacents){
                if(edgeNode.vertex == vertex) return edgeNode.edge;
            }
            return null;
        }

        public String toString() {
            return this.value + "[" + adjacents + "]";
        }

        public boolean equals(Object o) {
            if(o.getClass() != this.getClass())
                return false;
            VertexNode other = (VertexNode) o;
            if(other.value.equals(this.value))
                return true;
            else
                return false;
        }
    }

    private class EdgeNode {
        public VertexNode vertex;
        public Edge edge;

        public EdgeNode(VertexNode vertex, Edge edge) {
            this.vertex = vertex;
            this.edge = edge;
```

```
        }

        public EdgeNode(int tag) {
            this.edge = new Edge();
            this.edge.tag = tag;
        }

        public String toString() {
            return edge.tag + ": (" + this.vertex.value + ", " +
edge.weight + ")";
        }

        public boolean equals(Object o) {
            EdgeNode other = (EdgeNode) o;
            if(other.edge.tag == this.edge.tag || other.vertex ==
this.vertex)
                return true;
            else
                return false;
        }
    }

    private class PathTreeNode {
        public E value;
        public LinkedList<PathTreeNode> children = new
LinkedList<PathTreeNode>();

        public PathTreeNode(){}

        public PathTreeNode(E value){
            this.value = value;
        }

        public String toString(){
            return displayNode("");
        }
```

```
    public String displayNode(String ident){
        String res = ident + this.value + "\n";
        for(PathTreeNode node : this.children){
            res += node.displayNode(ident + "    ");
        }
        return res;
    }
}

private class Edge {
    public int tag;
    public int weight;
    public int label; //para dfs -> 0:unexplored, 1:discovery, 2:back

    public Edge(int weight){
        this.weight = weight;
        this.tag = tagCount;
        tagCount++;
    }

    public Edge() {
        this.tag = -1;
    }
}

public Object[] vertices(){
    Object[] res = new Object[this.vertices.length()];
    int count = 0;
    for(VertexNode vertexNode : this.vertices){
        res[count] = vertexNode.value;
        count++;
    }

    return res;
}

public Object[] edges(){
```

```
        LinkedList<Integer> edges = new LinkedList<Integer>();
        for(VertexNode vertexNode : this.vertices){
            for(EdgeNode edgeNode : vertexNode.adjacents){
                if(!edges.contains(edgeNode.edge.tag))
edges.insertToBegin(edgeNode.edge.tag);
            }
        }
        Object[] res = new Object[edges.length()];
        int i = 0;
        for(int tag : edges){
            res[i] = tag;
            i++;
        }

        return res;
    }

    public void insertVertex(E element) throws DuplicateItemException {
        VertexNode node = new VertexNode(element);
        if(vertices.contains(node))
            throw new DuplicateItemException();
        vertices.insertToBegin(node);
    }

    public void insertEdge(E ver1, E ver2, int element) throws
VertexNotFound, DuplicatedEdge {
        Object[] nodePair = getNodePair(ver1, ver2);
        VertexNode vertex1 = (VertexNode) nodePair[0], vertex2 =
(VertexNode) nodePair[1];

        Edge edge = new Edge(element);
        vertex1.adjacents.insertToBegin(new EdgeNode(vertex2, edge));
        vertex2.adjacents.insertToBegin(new EdgeNode(vertex1, edge));
    }

    public void removeVertex(E v) {
        VertexNode remove = this.vertices.remove(new VertexNode(v));
```

```
        for(VertexNode vertex : this.vertices) {
            try {
                vertex.adjacents.remove(new EdgeNode(remove, new
Edge()));
            }
            catch(Exception e) {
            }
        }
    }

    public void removeEdge(int tag) {
        for(VertexNode vertex : this.vertices) {
            EdgeNode remove = new EdgeNode(tag);
            try {
                vertex.adjacents.remove(remove);
            }
            catch(Exception e){
            }
        }
    }

    public boolean areAdjacent(E v, E w) throws VertexNotFound {
        Object[] nodePair = getNodePair(v, w);
        VertexNode vertex1 = (VertexNode) nodePair[0], vertex2 =
(VertexNode) nodePair[1];
        return  vertex1.isAdjacentTo(vertex2);
    }

    private Object[] getNodePair(E v, E w) throws VertexNotFound {
        VertexNode vertex1 = null, vertex2 = null;

        for(VertexNode vertex : this.vertices) {
            if(vertex.value.equals(v))
                vertex1 = vertex;

            if(vertex.value.equals(w))
                vertex2 = vertex;
```

```
            if(vertex1 != null && vertex2 != null)
                break;
        }

        if(vertex1 == null || vertex2 == null)
            throw new VertexNotFound();

        Object[] ret = new Object[2];
        ret[0] = vertex1;
        ret[1] = vertex2;
        return ret;
    }

    public void dfs() {
        initLabels();

        LinkedList<PathTreeNode> paths = new LinkedList<PathTreeNode>();
        for(VertexNode vertex : this.vertices) if(vertex.label ==
UNEXPLORED){
            PathTreeNode pathTree = new PathTreeNode();
            dfs(vertex, pathTree);
            paths.insertToBegin(pathTree);
        }

        for(PathTreeNode pathTree : paths){
            System.out.println(pathTree);
        }
    }

    public void dfs(VertexNode v, PathTreeNode node){
        node.value = v.value;
        v.label = VISITED;
        for(EdgeNode edgeNode : v.adjacents){
            if(edgeNode.edge.label == UNEXPLORED){
                VertexNode w = this.opposite(v, edgeNode);
                if(w.label == UNEXPLORED){
```

```
                    edgeNode.edge.label = DISCOVERY;
                    PathTreeNode pathTree = new PathTreeNode();
                    node.children.insertToBegin(pathTree);
                    dfs(w, pathTree);
                }
                else edgeNode.edge.label = BACK;
            }
        }
    }

    public void bfs() {
        initLabels();

        LinkedList<PathTreeNode> paths = new LinkedList<PathTreeNode>();
        for(VertexNode vertex : this.vertices) if(vertex.label ==
UNEXPLORED){
            PathTreeNode pathTree = new PathTreeNode();
            bfs(vertex, pathTree);
            paths.insertToBegin(pathTree);
        }

        for(PathTreeNode pathTree : paths){
            System.out.println(pathTree);
        }
    }

    public void bfs(VertexNode v, PathTreeNode pathTree){
        Queue<VertexNode> list = new Queue<VertexNode>();
        Queue<PathTreeNode> nodeQueue = new Queue<PathTreeNode>();

        nodeQueue.enqueue(pathTree);
        list.enqueue(v);
        v.label = VISITED;
        pathTree.value = v.value;

        Queue<VertexNode> listI = list;
        while(!listI.isEmpty()){
```

```
            Queue<VertexNode> aux = new Queue<VertexNode>();

            for(VertexNode vertex : listI){
                PathTreeNode node = nodeQueue.getInitialValue();
                nodeQueue.dequeue();

                for(EdgeNode edgeNode : vertex.adjacents){
                    if(edgeNode.edge.label == UNEXPLORED){
                        VertexNode w = opposite(vertex, edgeNode);
                        if(w.label == UNEXPLORED){
                            edgeNode.edge.label = DISCOVERY;
                            w.label = VISITED;
                            aux.enqueue(w);
                            PathTreeNode son = new PathTreeNode(w.value);
                            node.children.insertToBegin(son);
                            nodeQueue.enqueue(son);
                        }
                        else edgeNode.edge.label = CROSS;
                    }
                }
            }
            listI = aux;
        }
    }

    private VertexNode opposite(VertexNode v, EdgeNode e){
        for(EdgeNode edge : v.adjacents) if(edge == e) return
edge.vertex;
        return null;
    }

    private void initLabels(){
        for(VertexNode vertex : this.vertices){
            vertex.label = UNEXPLORED;
            for(EdgeNode edgeNode : vertex.adjacents) edgeNode.edge.label
= UNEXPLORED;
        }
```

```
    }

    public void printVertexEdgeLabel(){
        String edgeLabel = "";
        System.out.println("VertexLabel");
        for(VertexNode vertex : this.vertices){
            System.out.print(vertex.value + ":" + vertex.label + " ");
            for(EdgeNode edgeNode : vertex.adjacents)
                edgeLabel += edgeNode.edge.tag + ":" +
edgeNode.edge.label + " ";
            edgeLabel += "\n";
        }
        System.out.println("\n" + edgeLabel);
    }

    public Object[][] dijkstra(E v){
        class DijkstraNode implements Comparable<DijkstraNode> {
            VertexNode vertex;
            VertexNode path;
            int weight;

            public int compareTo(DijkstraNode o) {
                DijkstraNode other = (DijkstraNode) o;
                if(this.weight > other.weight)
                    return 1;
                if(this.weight < other.weight)
                    return -1;
                return 0;
            }
        }

        Object[] nodes = new Object[this.vertices.length()];
        int i = 1;
        for(VertexNode vertex : this.vertices){
            DijkstraNode node = new DijkstraNode();
            node.vertex = vertex;
            if(vertex.value.equals(v)){
```

```
                node.weight = 0;
                node.path = vertex;
                nodes[0] = node;
            }
            else{
                node.weight = INFINITE;
                nodes[i] = node;
                i++;
            }
        }

        PriorityQueue<DijkstraNode> queue = new
PriorityQueue<DijkstraNode>();
        for(Object node : nodes){
            queue.enqueue((DijkstraNode) node);
        }
        while(!queue.isEmpty()){
            DijkstraNode u = queue.getInitialValue();
            queue.dequeue();
            for(DijkstraNode z : queue){
                Edge edge = u.vertex.edgeWith(z.vertex);
                if(edge != null){
                    int weight = u.weight + edge.weight;
                    if(z.weight > weight){
                        z.weight = weight;
                        z.path = u.vertex;
                        queue.enqueue(z);
                    }
                }
            }
        }

        Object[][] d = new Object[this.vertices.length()][3];
        i = 0;
        for(Object node : nodes){
            DijkstraNode dijkstraNode = (DijkstraNode) node;
            d[i][0] = dijkstraNode.vertex;
```

```
            d[i][1] = dijkstraNode.weight;
            d[i][2] = dijkstraNode.path;
            i++;
        }

        return d;
    }

    public static boolean isIncluded(Graph g1, Graph g2){
        return g1.isIncluded(g2);
    }

    public boolean isIncluded(Graph<E> g2){
        Graph<E> g1 = this;
        if(g1.vertices.length() < g2.vertices.length()){
            Graph<E> aux = g1;
            g1 = g2;
            g2 = aux;
        }
        else if(g1.edges().length < g2.edges().length){
            Graph<E> aux = g1;
            g1 = g2;
            g2 = aux;
        }

        for(VertexNode vertexNode : g2.vertices){
            boolean contained = false;
            VertexNode g1Vertex = null;
            for(VertexNode vertex : g1.vertices){
                if(vertexNode.equals(vertex)){
                    g1Vertex = vertex;
                    contained = true;
                    break;
                }
            }
            if(!contained) return false;
```

```
            for(EdgeNode edgeNode : vertexNode.adjacents){
                contained = false;
                for(EdgeNode g1Edge : g1Vertex.adjacents){
                    if(edgeNode.vertex.equals(g1Edge.vertex) &&
edgeNode.edge.weight == g1Edge.edge.weight){
                        contained = true;
                        break;
                    }
                }
                if(!contained) return false;
            }
        }

        return true;
    }

    public String toString() {
        String res = "";

        for(VertexNode vertex : this.vertices) {
            res += vertex + "\n";
        }

        return res;
    }
}
```

bfs()

```
    public void bfs() {
        initLabels();

        LinkedList<PathTreeNode> paths = new LinkedList<PathTreeNode>();
        for(VertexNode vertex : this.vertices) if(vertex.label ==
UNEXPLORED){
            PathTreeNode pathTree = new PathTreeNode();
            bfs(vertex, pathTree);
            paths.insertToBegin(pathTree);
```

```
    }

    for(PathTreeNode pathTree : paths){
        System.out.println(pathTree);
    }
}

public void bfs(VertexNode v, PathTreeNode pathTree){
    Queue<VertexNode> list = new Queue<VertexNode>();
    Queue<PathTreeNode> nodeQueue = new Queue<PathTreeNode>();

    nodeQueue.enqueue(pathTree);
    list.enqueue(v);
    v.label = VISITED;
    pathTree.value = v.value;

    Queue<VertexNode> listI = list;
    while(!listI.isEmpty()){
        Queue<VertexNode> aux = new Queue<VertexNode>();

        for(VertexNode vertex : listI){
            PathTreeNode node = nodeQueue.getInitialValue();
            nodeQueue.dequeue();

            for(EdgeNode edgeNode : vertex.adjacents){
                if(edgeNode.edge.label == UNEXPLORED){
                    VertexNode w = opposite(vertex, edgeNode);
                    if(w.label == UNEXPLORED){
                        edgeNode.edge.label = DISCOVERY;
                        w.label = VISITED;
                        aux.enqueue(w);
                        PathTreeNode son = new PathTreeNode(w.value);
                        node.children.insertToBegin(son);
                        nodeQueue.enqueue(son);
                    }
                    else edgeNode.edge.label = CROSS;
                }
```

```
                }
            }
            listI = aux;
        }
    }
```

dfs()

```
    public void dfs() {
        initLabels();

        LinkedList<PathTreeNode> paths = new LinkedList<PathTreeNode>();
        for(VertexNode vertex : this.vertices) if(vertex.label ==
UNEXPLORED){
            PathTreeNode pathTree = new PathTreeNode();
            dfs(vertex, pathTree);
            paths.insertToBegin(pathTree);
        }

        for(PathTreeNode pathTree : paths){
            System.out.println(pathTree);
        }
    }

    public void dfs(VertexNode v, PathTreeNode node){
        node.value = v.value;
        v.label = VISITED;
        for(EdgeNode edgeNode : v.adjacents){
            if(edgeNode.edge.label == UNEXPLORED){
                VertexNode w = this.opposite(v, edgeNode);
                if(w.label == UNEXPLORED){
                    edgeNode.edge.label = DISCOVERY;
                    PathTreeNode pathTree = new PathTreeNode();
                    node.children.insertToBegin(pathTree);
                    dfs(w, pathTree);
                }
                else edgeNode.edge.label = BACK;
            }
```

```
        }
    }
```

dijkstra()

```
    public Object[][] dijkstra(E v){
        class DijkstraNode implements Comparable<DijkstraNode> {
            VertexNode vertex;
            VertexNode path;
            int weight;

            public int compareTo(DijkstraNode o) {
                DijkstraNode other = (DijkstraNode) o;
                if(this.weight > other.weight)
                    return 1;
                if(this.weight < other.weight)
                    return -1;
                return 0;
            }
        }

        Object[] nodes = new Object[this.vertices.length()];
        int i = 1;
        for(VertexNode vertex : this.vertices){
            DijkstraNode node = new DijkstraNode();
            node.vertex = vertex;
            if(vertex.value.equals(v)){
                node.weight = 0;
                node.path = vertex;
                nodes[0] = node;
            }
            else{
                node.weight = INFINITE;
                nodes[i] = node;
                i++;
            }
        }
```

```
        PriorityQueue<DijkstraNode> queue = new
PriorityQueue<DijkstraNode>();
        for(Object node : nodes){
            queue.enqueue((DijkstraNode) node);
        }
        while(!queue.isEmpty()){
            DijkstraNode u = queue.getInitialValue();
            queue.dequeue();
            for(DijkstraNode z : queue){
                Edge edge = u.vertex.edgeWith(z.vertex);
                if(edge != null){
                    int weight = u.weight + edge.weight;
                    if(z.weight > weight){
                        z.weight = weight;
                        z.path = u.vertex;
                        queue.enqueue(z);
                    }
                }
            }
        }

        Object[][] d = new Object[this.vertices.length()][3];
        i = 0;
        for(Object node : nodes){
            DijkstraNode dijkstraNode = (DijkstraNode) node;
            d[i][0] = dijkstraNode.vertex;
            d[i][1] = dijkstraNode.weight;
            d[i][2] = dijkstraNode.path;
            i++;
        }

        return d;
    }
```

BFS y DFS Test

```
    public static void paths() throws Exception {
        Graph<String> graph = new Graph<String>();
```

```
        graph.insertVertex("I");
        graph.insertVertex("H");
        graph.insertVertex("G");
        graph.insertVertex("F");
        graph.insertVertex("E");
        graph.insertVertex("D");
        graph.insertVertex("C");
        graph.insertVertex("B");
        graph.insertVertex("A");
        graph.insertEdge("A", "D", 0);
        graph.insertEdge("A", "C", 0);
        graph.insertEdge("A", "B", 0);
        graph.insertEdge("B", "C", 0);
        graph.insertEdge("B", "F", 0);
        graph.insertEdge("B", "E", 0);
        graph.insertEdge("C", "F", 0);
        graph.insertEdge("D", "G", 0);
        graph.insertEdge("F", "G", 0);
        graph.insertEdge("H", "I", 0);
        System.out.println(graph);
        System.out.println("dfs()");
        graph.dfs();
        System.out.println("bfs()");
        graph.bfs();
        System.out.println("printVertexEdgeLabel()");
        graph.printVertexEdgeLabel();
    }
```

Dijkstra Test

```
    public static void dijkstra() throws Exception{
        Graph<String> graph = new Graph<String>();
        graph.insertVertex("s");
        graph.insertVertex("u");
        graph.insertVertex("x");
        graph.insertVertex("v");
        graph.insertVertex("y");
        graph.insertEdge("s", "u", 10);
```

```
        graph.insertEdge("s", "x", 5);
        graph.insertEdge("s", "y", 7);
        graph.insertEdge("u", "x", 2);
        graph.insertEdge("u", "v", 1);
        graph.insertEdge("x", "v", 9);
        graph.insertEdge("x", "y", 2);
        graph.insertEdge("v", "y", 4);
        System.out.println(graph);
        System.out.println("dijkstra(s)");
        Object[][] res = graph.dijkstra("s");
        for(Object[] obj : res){
            System.out.println(obj[0] + " | " + obj[1] + " | " + obj[2]);
        }

    }
```

Ejercicio 4

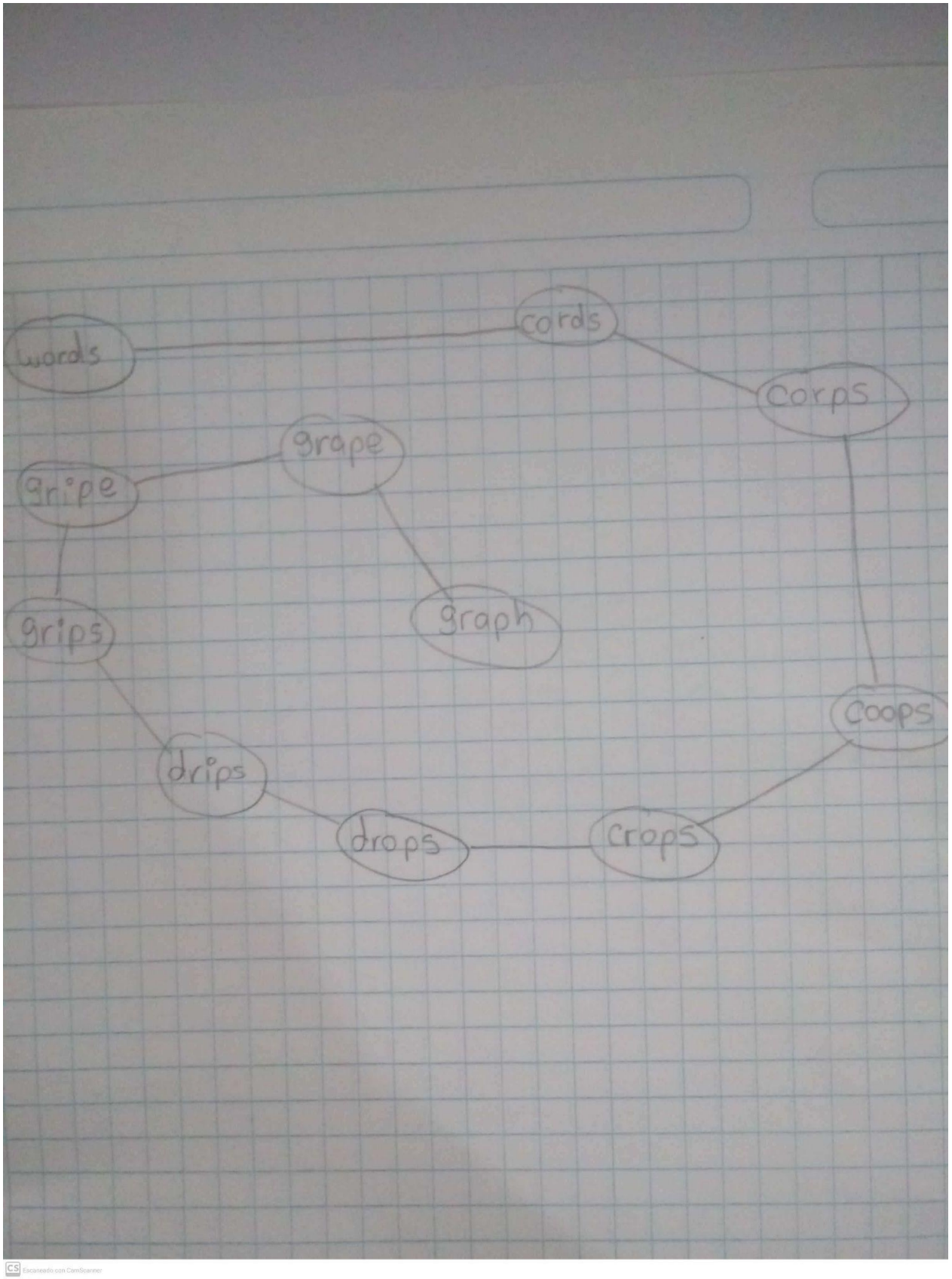
```
    public static void ejercicio4() throws Exception{
        Graph<String> graph = new Graph<String>();
        String[] words = {"words", "cords", "corps", "coops", "crops",
"drops", "drips", "grips", "gripe", "grape", "graph"};

        for(String word : words){
            graph.insertVertex(word);
        }

        Object[] vertices = graph.vertices();
        for(Object vertex1 : vertices){
            for(Object vertex2 : vertices){
                String v1 = (String) vertex1, v2 = (String) vertex2;
                if(differentChars(v1, v2) == 1 && !graph.areAdjacent(v1,
v2)){
                    graph.insertEdge(v1, v2, 0);
                }
            }
        }
```

```
        System.out.println(graph);
    }
```

Grafo



isIncluded()

```
    public static boolean isIncluded(Graph g1, Graph g2){
        return g1.isIncluded(g2);
    }

    public boolean isIncluded(Graph<E> g2){
        Graph<E> g1 = this;
        if(g1.vertices.length() < g2.vertices.length()){
            Graph<E> aux = g1;
            g1 = g2;
            g2 = aux;
        }
        else if(g1.edges().length < g2.edges().length){
            Graph<E> aux = g1;
            g1 = g2;
            g2 = aux;
        }

        for(VertexNode vertexNode : g2.vertices){
            boolean contained = false;
            VertexNode g1Vertex = null;
            for(VertexNode vertex : g1.vertices){
                if(vertexNode.equals(vertex)){
                    g1Vertex = vertex;
                    contained = true;
                    break;
                }
            }
            if(!contained) return false;

            for(EdgeNode edgeNode : vertexNode.adjacents){
                contained = false;
                for(EdgeNode g1Edge : g1Vertex.adjacents){
                    if(edgeNode.vertex.equals(g1Edge.vertex) &&
edgeNode.edge.weight == g1Edge.edge.weight){
                        contained = true;
                        break;
```

```
                    }
                }
                if(!contained) return false;
            }
        }

        return true;
    }
```

isIncludeTest()

```
    public static void isIncludeTest() throws Exception {
        Graph<String> graph = new Graph<String>();
        String[] words = {"words", "cords", "corps", "coops", "crops",
"drops", "drips", "grips", "gripe", "grape", "graph"};

        for(String word : words){
            graph.insertVertex(word);
        }

        Object[] vertices = graph.vertices();
        for(Object vertex1 : vertices){
            for(Object vertex2 : vertices){
                String v1 = (String) vertex1, v2 = (String) vertex2;
                if(differentChars(v1, v2) == 1 && !graph.areAdjacent(v1,
v2)){
                    graph.insertEdge(v1, v2, 0);
                }
            }
        }

        System.out.println("Graph01\n" + graph);

        Graph<String> graph02 = new Graph<String>();
        graph02.insertVertex("grape");
        graph02.insertVertex("gripe");
        graph02.insertVertex("grips");
        graph02.insertVertex("drips");
```

```
        graph02.insertEdge("grape", "gripe", 0);
        graph02.insertEdge("gripe", "grips", 0);
        graph02.insertEdge("grips", "drips", 0);
        System.out.println("Graph02:\n" + graph02);
        System.out.println("isIncluded(g1,g2): " +
Graph.isIncluded(graph, graph02));
        System.out.println("G2.insert(apples)");
        graph02.insertVertex("apples");
        System.out.println("isIncluded(g1,g2): " +
Graph.isIncluded(graph, graph02));
        System.out.println("G2.remove(apples)");
        graph02.removeVertex("apples");
        System.out.println("isIncluded(g1,g2): " +
Graph.isIncluded(graph, graph02));
        System.out.println("insertEdge(grape,drips,0)");
        graph02.insertEdge("grape", "drips", 0);    //tag = 3
        System.out.println("isIncluded(g1,g2): " +
Graph.isIncluded(graph, graph02));
        System.out.println("removeEdge(3)");
        graph02.removeEdge(3);
        System.out.println(graph02);
        System.out.println("isIncluded(g1,g2): " +
Graph.isIncluded(graph, graph02));
    }
```