UNIVERSIDAD NACIONAL DE SAN AGUSTÍN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

ESTRUCTURA DE DATOS Y ALGORITMOS

INFORME DE LABORATORIO- GRAFOS

Estudiante:

Yeimy Estephany Huanca Sancho

Profesora:

Mgt.Edith Pamela Rivero Tupac

Desarrollo

1. Ejercicio 1

Implementar el código de Grafo cuya representación sea realizada mediante

LISTA DE ADYACENCIA

El código implementado abarca mas que solo esta clase , sin embargo , esta es la parte principal :

```
public class GraphLink<T> {//

	protected ListaAdyacencia<Vertex<T>>listVertex;

	public GraphLink() {
		listVertex = new ListaAdyacencia<Vertex<T>>();
	}

	public void insertVertex(T data) {
		Vertex<T> nuevo = new Vertex<T>(data);
		if (this.listVertex.search(nuevo) != null) {
			System.out.print("Vertice ya fue insertado anteriormente ...");
			return;
		}
		this.listVertex.insert(nuevo);
	}

	public void insertEdge(T verOri, T verDes) {// para enlazar dos vertices
		Vertex<T> refOri = this.listVertex.search(new Vertex<T>(verOri));//origen
		Vertex<T> refDes = this.listVertex.search(new Vertex<T>(verDes));//destino

		if (refOri == null || refDes == null) {
			System.out.println("Vertice origen y/o desino no existen ...");
			return;
		}
		if(refOri.listAdj.search(new Edge<T>(refDes)) != null) {
			System.out.println("Arista ya fue insertada anteriormente...");
			return;
		}
		refOri.listAdj.insert(new Edge<T>(refDes));//inserta
		refDes.listAdj.insert(new Edge<T>(refOri));
	}

	public String toString() {
		return this.listVertex.toString();

	}
	public void initLabel() {//inicializar
		Node<Vertex<T>> aux = this.listVertex.head;//vertices
		for (; aux != null ; aux=aux.next) {//mientras que sea diferente de nulo se
```

```
inicializa el vertice
                        aux.data.label = 0;
                        Node<Edge<T>> auxE = aux.data.listAdj.head;//las aristas de ese
vertice ,recorrer
                        for (; auxE != null ; auxE=auxE.next) {
                                auxE.data.label = 0;
                        }
                }
```

Las clases que se usaron para su construcción fueron las siguientes.

Nodo

```
public class Node <T> {

                protected T data;
                protected Node<T> next;

                public Node(T data) {
```

```
                this (data, null);
        }
        public Node(T data, Node<T>next) {
                this.data = data;
                this.next = next;
        }
        public String toString() {
                return data.toString();
        }

}
```

Linked List

```
public class ListaAdyacencia<T> {

        protected Node<T> head;
        protected Node<T> last;
        protected int counter;

        public ListaAdyacencia() {}
        public ListaAdyacencia(Node<T> head) {
                this.head = head;
        }

        public void insert(T data) {
                Node<T> aux = new Node<T> (data);
                if (counter == 0) {
                        this.head = aux;
                        this.last = head;
                        counter++;
                }else {
```

```
                last.next = aux;
                this.last = last.next;
                counter ++;
        }
}

public T search(T data) {
        Node<T> aux = this.head;
        while(aux != null && !aux.data.equals(data)) {
                aux = aux.next;
        }
        if(aux != null) {
                return aux.data;
        } else {
                return null;
        }
}
public int get(T data) {
        Node<T> aux = this.head;
        int pos = 0;
        while(aux != null && !aux.data.equals(data)) {
                pos++;
                aux = aux.next;
        }
        if(aux != null) {
                return pos;
        } else {
                return -1;
        }
}

@Override
public String toString() {
        String str = "";
        Node <T> aux = this.head;
        while(aux != null) {
                str += aux.toString()+ " ";
                aux = aux.next;
        }
        return str;
```

```
        }
}
```

Edge()Arista

```
public class Edge<E> {//arista
        protected Vertex<E> refDest; //vertice del grafo
        protected int weight;//el peso en caso de ser grafo ponderado
        protected int label; //0 = no explorado , 1 = visitado, 2 = back

        public Edge(Vertex<E> refDest) {//contructor
                this(refDest, -1);//-1 en caso de no ser ponderado
        }

        public Edge (Vertex<E> refDest, int weight) {// constructor con peso de la arista
                this.refDest = refDest;//destino
                this.weight = weight;//peso
        }

        public boolean equals(Object o) {//sobreescribiendo etodo equals
                if(o instanceof Edge<?>) {
                        Edge<E> e =(Edge<E>)o;//casteando el objeto
                        return this.refDest.equals(e.refDest);//devuelve si esigual
                }
                return false;//devuele false si no es igual
        }

        public String toString(){//sobre escribiendo metodo toString
                if(this.weight > -1) return refDest.data +" ["+ this.weight +"], ";
                else return refDest.data+ ", ";
        }
}
```

Vertex() Vertice

```
public class Vertex<E> {//vertice
	protected E data; //data generica
	protected ListaAdyacencia<Edge<E>> listAdj;//creacion lista de aristas
	protected int label ; // estados:0 no explorado , 1 = visitado

	public Vertex(E data) {//constructor del vertice
		this.data = data;
		listAdj = new ListaAdyacencia<Edge<E>>();//inicializando su lista de aristas
	}

	public boolean equals (Object o) {//sobreescribiendo el metodo equals
		if(o instanceof Vertex<?>) {
			Vertex<E> v = (Vertex<E>)o;
			return this.data.equals(v.data);
		}
		return false;
	}

	@Override
	public String toString() {//sobreescribiendo el metodo toString
		return this.data+ " ---> " + this.listAdj.toString()+"\n";
	}
}
```

2. **Ejercicio 2**

   Implementar el código de Grafo cuya representación sea realizada mediante

DFS

```
public void DFS(T data) {
	Vertex<T> nuevo = new Vertex<T>(data);
	Vertex<T> aux = this.listVertex.search(nuevo);//vertice
	if(aux == null) {
		System.out.println("Vertice no existe ...");
		return;
	}
	initLabel();
	DFSRec(aux);
}

private void DFSRec(Vertex<T> v) {
	v.label = 1;
	System.out.println(v.data+"-"+v.label+", ");
	Node<Edge<T>> e = v.listAdj.head;
	for (; e != null; e=e.next) {
		if(e.data.label == 0) {
			Vertex<T> w =e.data.refDest;
			if (w.label ==0) {
				e.data.label = 1;
				DFSRec(w);
			}else
				e.data.label = 2;
		}
	}

}
```

BFS

El proceso estaba pensado en el recorrido recursivo de nodo por nodo y que el método recursivo en su proceso visitara los nodos que tienen conexión con el nodo inicial , marcando el nodo cada que fuera visitado para no volverlo a tomar en cuenta .

```
public void BFS(T data) {
	Vertex<T> nuevo = new Vertex<T>(data);
	Vertex<T> aux = this.listVertex.search(nuevo);//vertice
	if(aux == null) {
		System.out.println("Vertice no existe ...");
		return;
	}
	initLabel();
	BFSRec(aux);
}

private void BFSRec(Vertex<T> v) {
			System.out.println(v);
			Node<Edge<T>> e = v.listAdj.head;
			boolean explored [] =new boolean [length()];
			explored[pos(v.data)] = true;
			ArrayList<T> vertexes = new ArrayList<T>();
			vertexes.add(v.data);
			//Vertex<T> aux = new Vertex<T>(null);


			Vertex<T> nuevo = new Vertex<T>(null);
			Vertex<T> aux = new Vertex<T>(null);//vertice
			for (; e != null; e=e.next) {
				System.out.println(e.data);
				vertexes.add((T) e.data);
				nuevo.data= (T) e.data;
				aux = this.listVertex.search(nuevo);
				explored[pos(aux.data)] = true;
				System.out.println("x---"+e.data);
			}

	v.label = 1;

	Node<Edge<T>> e = v.listAdj.head;
	//while(e.next != null) {
	//	if(v.label ==1)
	//System.out.println(v.data+", "+v.label);
	for (; e != null; e=e.next) {
		System.out.println(e+"_"+ e.data.label +" data");
```

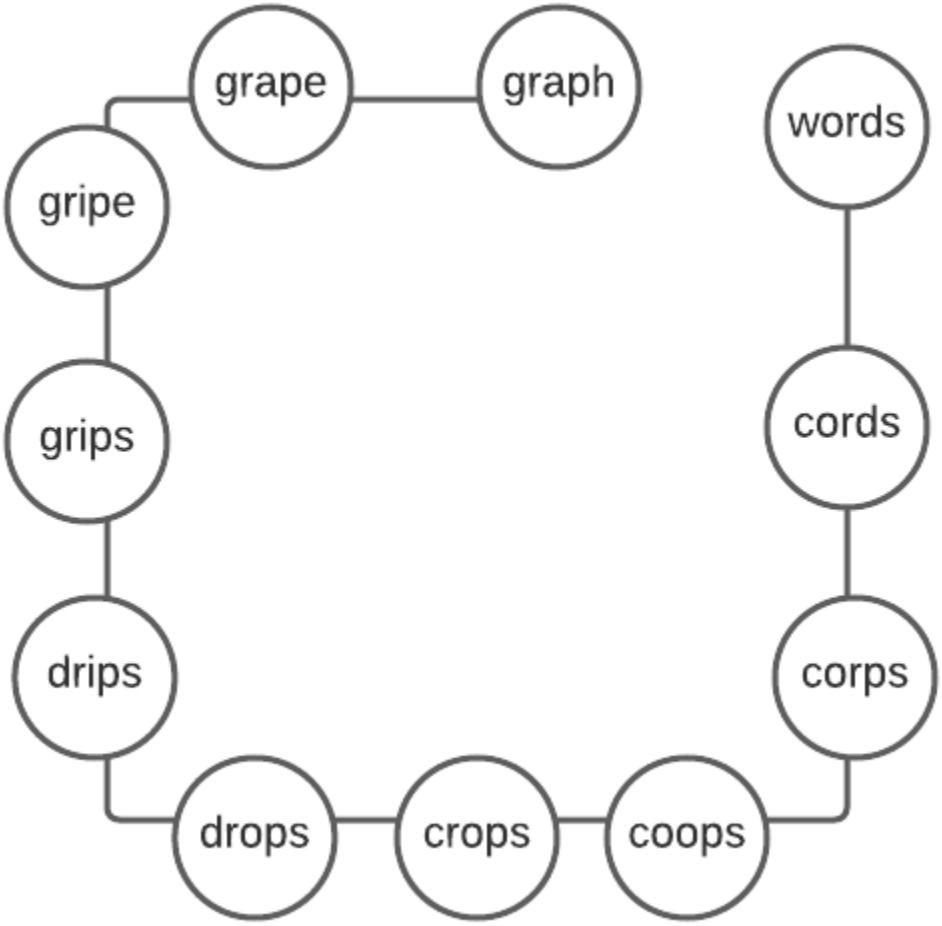
```
                    //while(e.data.refDest != null)
                    BFS( v.data);
                    if(e.data.label == 0) {
                            Vertex<T> w =e.data.refDest;
                            if (w.label ==0) {

                                    e.data.label = 1;

                                    w.label = 1;
                                    System.out.println(e.next.data);

                                    //System.out.println(e.data.refDest + "ref");
                                    //BFSRec(e.next.data.refDest);


                            }else {
                                    e.data.label = 2;

                            }

                    }
                    //}
                    e = e.next;



            }
    }
    public int length() {
            int vertex = 0;
            Node<Vertex<T>> aux = this.listVertex.head;//vertices
            for (; aux != null ; aux=aux.next) {
                    vertex++;
            }
            return vertex;
    }
    //prerequisito , que exista
    public int pos(T data) {
            System.out.println("pos---");
            System.out.println(data);
```

```
            System.out.println("pos---x");
            Vertex<T> nuevo = new Vertex<T>(data);
            int pos = this.listVertex.get(nuevo);//vertice
            return pos;
    }
```

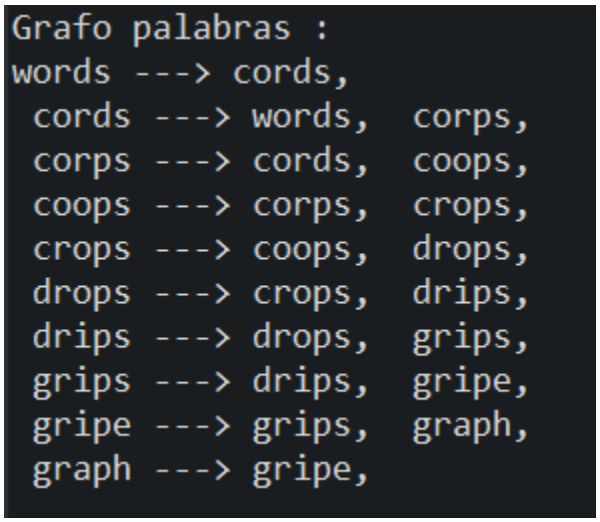
3. Ejercicio 3

El grafo de palabras se define de la siguiente manera: cada vértice es una palabra en el idioma Inglés y dos palabras son adyacentes si difieren exactamente en una posición. Por ejemplo, las cords y los corps son adyacentes, mientras que los corps y crops no lo son.

a) Dibuje el grafo definido por las siguientes palabras: words cords corps coops crops drops drips grips gripe grape graph



b) Mostrar la lista de adyacencia del grafo



4. **Ejercicio 4**

   Realizar un metodo en la clase Grafo. Este metodo permitira saber si un grafo esta incluido en otro. Los parametros de entrada son 2 grafos y la salida del metodo es true si hay inclusion y false el caso contrario.

Cuestionario

1. **Pregunta 1**

   ¿Cuántas variantes del algoritmo de Dijkstra hay y cuál es la diferencia entre ellas?

   El método dijkstra Esta pasado en un algoritmo que encuentra la más corta de a otro, Sus variantes consiste en hallar la ruta de uno solo a varios(todos los demas) , una Segunda variante Es sobre grafos dirigidos

2. Pregunta 2

Invetigue sobre los ALGORITMOS DE CAMINOS MÍNIMOS e indique, ¿Qué similitudes encuentra, qué diferencias, en qué casos utilizar y porque?

- Dijkstra

Sólo funciona en grafos no dirigidos ,Recorre vértices del grafo encontrando el camino mínimo, es un modelo que se clasifica dentro de los algoritmos de búsqueda. Su metodología se basa en iteraciones, de manera tal que en la práctica, su desarrollo se dificulta a medida que el tamaño de la red aumenta, dejándolo en clara desventaja.

- Prim

El algoritmo de Prim sirve para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

- Kruskal

Arma un árbol según la cola de prioridad , sirve para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Si el

grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

- Algoritmo de Bellman-Ford

Buscar cortan la ruta corta a los demás ,Aplicable sólo para grafos ponderados , Es el único que soporta valores negativos(ponderación)

Sirve para la detección de signos negativos , es similar al dijkstra con la diferencia que este es el único que acepta aristas negativas