

**UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE
AREQUIPA**

FACULTAD DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS



Curso: Laboratorio – Estructuras de Datos y Algoritmos

Grupo: “A”

Tema: Grafos

Docente: Rivero Tupac de Lozano, Edith Pamela

Alumno: Hincho Jove, Angel Eduardo

Arequipa - Perú

Agosto 2021

Ejercicios propuestos

1. Crear un repositorio en GitHub, donde incluyan la resolución de los ejercicios propuestos y el informe.

Enlace al repositorio en GitHub: <https://github.com/aHinchoJove/EDA-Lab09-Grafos.git>

2. Implementar el código para un Grafo cuya representación sea realizada mediante una Lista de Adyacencia.

Para poder dar solución a este primer ejercicio se crearon las Clases:

- Nodo: Representación en código de un Nodo que forma parte de una Lista Enlazada. Tiene como atributos su Dato y una referencia a otro Nodo. Posee setters y getters.

```
public class Nodo<T> {  
  
    // Atributos de la clase Nodo  
    protected T data;  
    protected Nodo<T> next;  
  
    // Constructores de la clase Nodo  
    public Nodo(T data) {  
        this(data, null);  
    }  
  
    public Nodo(T data, Nodo<T> next) {  
        setData(data);  
        setNext(next);  
    }  
}
```

- ListaEnlazada: Representación en código de una Lista Enlazada que usa Nodos para poder enlazar elementos. Tiene como atributo un Nodo inicial donde se unen o insertan otros Nodos mediante la referencia 'next'. Posee setters y getters.

```
public class ListaEnlazada<T> {  
  
    // Atributos de la clase ListaEnlazada  
    protected Nodo<T> first;  
  
    // Constructor de la clase ListaEnlazada  
    public ListaEnlazada() {  
        setFirst(null);  
    }  
}
```

Así mismo, la clase ListaEnlazada posee algunos métodos resaltantes y que serán importantes para implementar las demás funcionalidades serán los siguientes.

- a) Insert e InsertLast: Métodos que agregan elementos al inicio o al final de la ListaEnlazada. Esto se logra guardando la referencia hacia el primer Nodo e iterar según la posición en la cual se quiera insertar, al inicio o al final.

```
// Insertar un Nodo al inicio
public void insert(T data) {
    setFirst(new Nodo<T>(data, getFirst()));
}

// Insertar al final de un Nodo
public void insertLast(T data) {
    Nodo<T> aux = getFirst();
    if(aux == null) {
        // Si el nodo auxiliar es nulo insertar al inicio
        insert(data);
    } else {
        // Recorrer el Nodo hasta llegar al final
        while(aux.getNext() != null) {
            aux = aux.getNext();
        }
        // Aux contiene el ultimo Nodo, insertar al siguiente
        aux.setNext(new Nodo<T>(data));
    }
}
```

- b) Search: El método search recibe un tipo de dato genérico que se buscará dentro de toda la Lista, se retornará el mismo valor que se buscó si está dentro de la Lista y si no se encuentra dentro de la Lista se devuelve un valor 'nulo'.

```
// Buscar un elemento dentro de nuestra Lista Enlazada
public T search(T data) {
    // Rescatar la referencia al primer Nodo
    Nodo<T> aux = getFirst();
    // Recorrer la Lista Enlazada desde el primer Nodo
    while(aux != null && !aux.getData().equals(data)) {
        aux = aux.getNext();
    }
    // Evaluar el contenido y dar respuesta
    if(aux != null) {
        // Búsqueda con éxito
        return aux.getData();
    } else {
        // Búsqueda sin éxito
        return null;
    }
}
```

- c) Remove: El método remove recibe un tipo de dato genérico que se eliminará de la Lista, se retornará el mismo valor que se eliminó y si no se pudo eliminar o no se encontró el elemento dentro de la Lista se devuelve un valor 'nulo'.

```
// Eliminar un elemento dentro de nuestra Lista Enlazada
public T remove(T data) {
    // Variable de apoyo para el retorno del metodo
    T retornar = null;
    // Rescatar el Nodo inicial de la Lista
    Nodo<T> aux = getFirst();
    if(getFirst().getData().equals(data)) {
        // Eliminar al inicio de la Lista
        setFirst(getFirst().getNext());
        retornar = null;
    } else {
        // Eliminar en medio de la Lista
        while(aux.getNext() != null) {
            if(aux.getNext().getData().equals(data)) {
                retornar = aux.getNext().getData();
                aux.setNext(aux.getNext().getNext());
            } else {
                aux = aux.getNext();
            }
        }
    }
    return retornar;
}
```

- d) Copia: Método que es invocado por una Lista y devuelve otra Lista con el mismo contenido, el método copia la Lista en contenido.

```
public ListaEnlazada<T> copiar() {
    // Rescatar el Nodo inicial
    Nodo<T> inicio = getFirst();
    // Crear una nueva Lista Enlazada
    ListaEnlazada<T> copia = new ListaEnlazada<T>();
    while(inicio != null) {
        copia.insertLast(inicio.getData());
        inicio = inicio.getNext();
    }
    return copia;
}
```

- e) Incluye: Método que compara la Lista que lo llama con otra Lista. Recordar que la Lista que llama al método es la más grande y la que se envía como parámetro es la de menor tamaño.

```
// Metodo para ver si una lista está incluida en otra
// OJO: La Lista que llama al metodo es la mas grande
public boolean incluye(ListaEnlazada<T> otraLista) {
    // Variable de apoyo para el retorno del metodo
    boolean retornar = false;
    // Rescatar el Nodo inicial de la Lista pequena
    Nodo<T> inicioP = otraLista.getFirst();
    // Hacer una copia de Lista grande
    ListaEnlazada<T> copia = copiar();
    // Iterar mientras la Lista pequena tenga Nodos
    while(inicioP != null) {
        // Buscar si inicioP esta dentro de la Lista grande
        T buscar = copia.search(inicioP.getData());
        // Si inicioP se encuentra en la Lista grande
        if(buscar != null) {
            System.out.println("El elemento " + inicioP.getData()
                + ", esta incluido");
            copia.remove(inicioP.getData());
            retornar = true;
        } else {
            System.out.println("El elemento " + inicioP.getData()
                + ", no esta incluido");
            retornar = false;
            break;
        }
        inicioP = inicioP.getNext();
    }
    return retornar;
}
```

- Vertice: Representación en código de un Vértice que forma parte de un Grafo. Esta representación tiene como atributos:
 - o El contenido del Vértice de tipo genérico 'data'
 - o La Lista de Adyacencia para el Vértice que es una Lista de Aristas.
 - o Una etiqueta 'label' la cual indica si el Vértice ha sido o no explorado.

```
public class Vertice<T> {
    // Atributos de la clase Vertice
    protected T data;
    protected ListaEnlazada<Arista<T>> listaAdyacencia;
    protected int label; // 0 => Sin explorar, 1 => Explorada

    // Constructor de la clase Vertice
    public Vertice(T data) {
        setData(data);
        setLista();
    }
}
```

- Arista: Representación en código de una Arista que une dos Vértices dentro de un Grafo. La Arista cuenta a su vez con los siguientes atributos:
 - o Un Vértice que será el opuesto o destino de la Arista.
 - o Un Peso que será el costo para poder recorrer la Arista dentro del Grafo.
 - o Una etiqueta la cual indica si la Arista ha sido o no visitada o si es de tipo Back / Cross, o sea, una Arista descubierta más no visitada.

```
public class Arista<T> {
    // Atributos de la clase Arista
    protected Vertice<T> destino;
    protected int peso;
    protected int label; // 0 => Sin explorar, 1 => Descubierta, 2 => Back

    // Constructores de la clase Arista
    public Arista(Vertice<T> destino) {
        this(destino, -1);
    }

    public Arista(Vertice<T> destino, int peso) {
        setDestino(destino);
        setPeso(peso);
    }
}
```

- Grafo: Representación en código de un Grafo con sus Vértices y Aristas entre ellos. La clase Grafo tiene como único atributo una Lista de Vértices.

```
// Representacion de un Grafo mediante una Lista de Adyacencia
public class Grafo<T> {
    // Atributos de la clase Grafo
    protected ListaEnlazada<Vertice<T>> listaVertices;

    // Constructor de la clase Grafo
    public Grafo() {
        setVertices();
    }
}
```

En la clase Grafo tenemos métodos que nos serán útiles para poder realizar operaciones tales como: Agregar vértices, aristas, etc.

- a) InsertarVertice: Método que nos ayuda a insertar Vértices dentro del Grafo y verifica que no se ingrese dos veces el mismo Vértices.

```
// Metodo para insertar Vertices en el Grafo
public void insertarVertice(T data) {
    Vertice<T> nuevo = new Vertice<T>(data);
    if(getVertices().search(nuevo) != null) {
        System.out.println("El Vertice ya existe ...");
        return;
    } else {
        getVertices().insertLast(nuevo);
    }
}
```

- b) InsertarArista: Método que nos permite insertar Aristas entre los Vértices del Grafo No Ponderado, llama a un método auxiliar por si se quiere insertar un Grafo Ponderado donde las Aristas tienen un peso o costo definido.

```
// Metodo para insertar Aristas en el Grafo
public void insertarArista(T origen, T destino) {
    insertarArista(origen, destino, -1);
}

// Metodo auxiliar para insertar Aristas en el Grafo
public void insertarArista(T origen, T destino, int peso) {

    // Buscar si existen los Vertices que se quieren unir
    Vertice<T> refOrigen = getVertices().search(new Vertice<T>(origen));
    Vertice<T> refDestino = getVertices().search(new Vertice<T>(destino));

    // En caso de no existir alguno de los dos Vertices mandar un mensaje
    if(refOrigen == null || refDestino == null) {
        System.out.println("El Vertice de origen y/o destino no existen ...");
        return;
    }

    // Buscar si existe una Arista ya registrada entre los dos Vertices
    if(refOrigen.getLista().search(new Arista<T>(refDestino)) != null) {
        System.out.println("Ya existe una Arista entre los Vertices ...");
        return;
    }

    // En otro caso, valido, insertar la Arista entre los Vertices
    refOrigen.getLista().insertLast(new Arista<T>(refDestino));
    refDestino.getLista().insertLast(new Arista<T>(refOrigen));
}
```

- c) BuscarVertice: Método que busca un Vértice dentro del Grafo, devuelve el Vértice buscado si es que existe en la Lista de Vértices del Grafo o devuelve un valor 'nulo' en caso que el Vértice no se encuentre dentro de la Lista de Vértices del Grafo.

```
// Metodo de apoyo para buscar un Vertice
public Vertice<T> buscarVertice(Vertice<T> buscar) {
    return getVertices().search(buscar);
}
```

- TestGrafo: Clase de apoyo en la cual se prueban los métodos que se implementan a la clase Grafo. En esta clase de apoyo creamos objetos de la clase Grafo y hacemos diferentes acciones como, por ejemplo, agregar Vértices, Aristas y mostrar la Lista de Adyacencia.

```

public class TestGrafo {

    public static void main(String[] args) {

        // Creando un objeto de tipo Grafo
        Grafo<String> grafo = new Grafo<String>();

        // Insertando algunos Vertices
        grafo.insertarVertice("Lima");
        grafo.insertarVertice("Arequipa");
        grafo.insertarVertice("Cusco");
        grafo.insertarVertice("Piura");
        grafo.insertarVertice("Tarapoto");

        // Insertando algunas Aristas
        grafo.insertarArista("Lima", "Arequipa", 8);
        grafo.insertarArista("Cusco", "Arequipa", 4);
        grafo.insertarArista("Piura", "Lima", 5);
        grafo.insertarArista("Cusco", "Lima", 6);
        grafo.insertarArista("Piura", "Tarapoto", 3);

        // Imprimiendo el contenido del Grafo
        System.out.println("Grafo de ciudades:\n");
        System.out.println(grafo);
    }
}

```

3. Implementar BSF, DFS y Dijkstra con sus respectivos casos de prueba

- a) InicializarLabel: Método que nos será útil para los recorridos, el método 'inicializarLabel' establece las etiquetas de los Vértices del Grafo como No Visitados, es decir el atributo Label de cada Vértices se establecerá como 0.

```

// Metodo de apoyo para inicializar las etiquetas
private void inicializarLabel() {
    Nodo<Vertice<T>> aux = getVertices().getFirst();
    for( ; aux != null; aux = aux.getNext()) {
        aux.getData().setLabel(0);
        Nodo<Arista<T>> auxA = aux.getData().getLista().getFirst();
        for( ; auxA != null; auxA = auxA.getNext()) {
            auxA.getData().setLabel(0);
        }
    }
}

```

- b) Recorrido BSF: Para poder implementar el recorrido BFS se hará uso del Pseudo-Código visto en las clases de Teoría del curso de EDA.

Algoritmo BFS

```

Algorithm BFS(G)
Input graph G
Output labeling of the edges and partition
of the vertices of G

for all n ∈ G.vertices()
    setLabel(n, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G,v)

```

```

Algorithm BFS(G, s)
L0 ← new empty sequence
L0.insertLast(s)
setLabel(s, VISITED)
i ← 0
while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
                w ← opposite(v,e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Li+1.insertLast(w)
            else
                setLabel(w, CROSS)
    i ← i+1

```

Para ello tenemos el método ‘BFS’ que nos ayudará a verificar si el Vértice existe, inicializar las etiquetas de los Vértices y recorrer el Grafo mediante el recorrido ‘BFS’. Este se encuentra dentro de la clase Grafo y como se muestra en el Pseudo-Código de Teoría llama a otro método ‘BFSApoyo’.

```
// Metodo para el recorrido BFS

public void BFS(T data) {
    Vertice<T> nuevo = new Vertice<T>(data);
    Vertice<T> auxiliar = getVertices().search(nuevo);
    // Verificar que el Vertice existe dentro del Grafo
    if(auxiliar == null) {
        System.out.println("El Vertice no existe ...");
        return;
    }
    // En caso existe el Vertice, inicializar las etiquetas
    inicializarLabel();
    BFSApoyo(auxiliar);
}
```

El método ‘BFSApoyo’ también está basado en el Pseudo-Código de Teoría. Primero marcaremos como ‘Visitado’ en Vértice donde empezaremos el recorrido BFS y crearemos una Lista de Listas de Vértices ya que el recorrido BFS se basa en recorrer el Grafo mediante niveles o Listas de Listas.

```
// Metodo de apoyo para el recorrido BFS

private void BFSApoyo(Vertice<T> auxiliar) {
    // Creando una nueva Lista de Listas de Vertices
    ListaEnlazada<ListaEnlazada<Vertice<T>>> lista =
        new ListaEnlazada<ListaEnlazada<Vertice<T>>>();
    // Insertar el Vertice auxiliar al final
    ListaEnlazada<Vertice<T>> insertar = new ListaEnlazada<Vertice<T>>();
    insertar.insertLast(auxiliar);
    lista.insertLast(insertar);
    // Marcar como ya visitado
    auxiliar.setLabel(1);
    // Recorrer la Lista de Lista de Vertices
    Nodo<ListaEnlazada<Vertice<T>>> auxL = lista.getFirst();
    // Recorrer la primera lista de la Lista de Listas de Vertices
    while(auxL != null) {
        Nodo<Vertice<T>> auxV = auxL.getData().getFirst();
        ListaEnlazada<Vertice<T>> listaAux = new ListaEnlazada<Vertice<T>>();
        // Recorrer la Lista de Vertices
        while(auxV != null) {
            Nodo<Arista<T>> auxA = auxV.getData().getLista().getFirst();
            System.out.println(auxV.getData().getData() + ", ");
            // Recorrer la Lista de Aristas
            while(auxA != null) {
                // Verificar la Arista que aun no ha sido visitada
                if(auxA.getData().getLabel() == 0) {
                    // Rescatar su opuesto o destino
                    Vertice<T> opuesto = auxA.getData().getDestino();
                    // Verificar que el Vertice no se ha visitado
                    if(opuesto.getLabel() == 0) {
                        // Cambiar el valor del Vertice como visitado
                        opuesto.setLabel(1);
                        // Cambiar el valor de la Arista como visitada
                        auxA.getData().setLabel(1);
                        // Insertar el Vertice opuesto en la listaAux
                        listaAux.insertLast(opuesto);
                    } else {

```


Iteraremos entre las Listas de Vértices, luego en las Vértices y finalmente en las Aristas que contiene el Vértice de la iteración. Verificaremos si el Vértice de destino u opuesto de las Aristas todavía no ha sido visitado, si no ha sido visitado entonces la establecemos como visitado y también la Arista de la iteración.

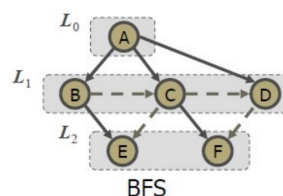
```

    } else {
        // Si ya se ha visitado, establecerlo como Back / Cross
        auxA.getData().setLabel(2);
    }
    auxA = auxA.getNext();
}
auxV = auxV.getNext();
}
// Asi mismo verificar si la listaAux es diferente de nulo
if(listaAux.getFirst() != null) {
    // Si es diferente de nulo agregar otra en auxL
    auxL.setNext(new Nodo<ListaEnlazada<Vertice<T>>>(listaAux));
    // No olvidar avanzar para poder romper los ciclos
    auxL = auxL.getNext();
} else {
    // Si es igual a nulo entonces no se agrego nada
    // Solo debemos avanzar en la Lista de Listas de Vertices
    auxL = auxL.getNext();
}
}
}

```

Cross edge(v, w)

- w está en el mismo nivel de v o en el siguiente nivel en el árbol de aristas descubiertas



Si ya está visitado entonces establecerlo como Cross. Finalmente verificar que la Lista auxiliar de Vértices no está vacía, si la Lista de Vértices auxiliar no está vacía se agrega a la Lista de Listas de Vértices y en caso de estar vacía pasamos a la siguiente Lista de Vértices. Por último, en la clase TestGrafo tenemos un objeto de tipo Grafo de Strings, de nombre 'grafo'. Este Grafo contiene nombre de ciudades del Perú y como estos se relacionan, veamos el Grafo y su Lista de Adyacencia.

Grafo de ciudades:

Contenido del Grafo:

```

Lima ==> Arequipa, Piura, Cusco,
Arequipa ==> Lima, Cusco,
Cusco ==> Arequipa, Lima,
Piura ==> Lima, Tarapoto,
Tarapoto ==> Piura,

```

Probaremos el recorrido BFS en el objeto 'grafo' partiendo de la ciudad de Arequipa.

```
// Probando el recorrido BFS
System.out.println("Probando el recorrido BFS para Arequipa");
grafo.BFS("Arequipa");
```

Veamos el resultado por consola al utilizar el método BFS.

```
Probando el recorrido BFS para Arequipa
Arequipa,
Lima,
Cusco,
Piura,
Tarapoto,
```

- c) Recorrido DFS: Para poder implementar el recorrido DFS se hará uso del Pseudo-Código visto en las clases de Teoría del curso de EDA.

Algoritmo DFS

Algorithm *DFS*(*G*)

Input graph *G*

Output labeling of the edges of *G* as discovery edges and back edges

```
for all n ∈ G.vertices()
  setLabel(n, UNEXPLORED)
for all e ∈ G.edges()
  setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
  if getLabel(v) = UNEXPLORED
    DFS(G, v)
```

Algorithm *DFS*(*G*, *v*)

Input graph *G* and a start vertex *v* of *G*

Output labeling of the edges of *G* in the connected component of *v* as discovery edges and back edges

```
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
  if getLabel(e) = UNEXPLORED
    w ← opposite(v, e)
    if getLabel(w) = UNEXPLORED
      setLabel(e, DISCOVERY)
      DFS(G, w)
    else
      setLabel(e, BACK)
```

Para ello tenemos el método 'DFS' que nos ayudará a verificar si el Vértice existe, inicializar las etiquetas de los Vértices y recorrer el Grafo mediante el recorrido 'DFS'. Este se encuentra dentro de la clase Grafo y como se muestra en el Pseudo-Código de Teoría llama a otro método 'DFSApoyo'.

```
// Metodo para el recorrido DFS

public void DFS(T data) {
    Vertice<T> nuevo = new Vertice<T>(data);
    Vertice<T> auxiliar = getVertices().search(nuevo);
    // Verificar que el Vertice existe dentro del Grafo
    if(auxiliar == null) {
        System.out.println("El Vertice no existe ...");
        return;
    }
    // En caso existe el Vertice, inicializar las etiquetas
    inicializarLabel();
    DFSApoyo(auxiliar);
}
```

El método 'DFSApoyo' también está basado en el Pseudo-Código de Teoría. Primero marcaremos como 'Visitado' en Vértice donde empezaremos el recorrido DFS en este caso no se requiere crear alguna estructura auxiliar. Debemos iterar por las Aristas que tenga el Vértice en el cual estamos trabajando o iterando. Dependiendo del Vértice opuesto o de destino de la Arista, de su etiqueta, si está no visitada entonces debemos marcarla como visitada mientras que en caso de estar visitada marcarla como Back.

```
// Metodo de apoyo para el recorrido DFS

private void DFSApoyo(Vertice<T> auxiliar) {
    // Marcarlo como ya visitado
    auxiliar.setLabel(1);
    // Mostrarlo por consola
    System.out.println(auxiliar.getData() + ", ");
    // Recorrer la lista de Aristas del Vertice
    Nodo<Arista<T>> auxA = auxiliar.getLista().getFirst();
    for( ; auxA != null; auxA = auxA.getNext()) {
        // Evaluar la etiqueta
        if(auxA.getData().getLabel() == 0) {
            Vertice<T> opuesto = auxA.getData().getDestino();
            // Verificar la etiqueta del vecino
            if(opuesto.getLabel() == 0) {
                // Cambiar la etiqueta a Descubierto
                auxA.getData().setLabel(1);
                // Llamar al metodo recursivamente con el opuesto
                DFSApoyo(opuesto);
            } else {
                // Si no es verdad la etiquera sera Back
                auxA.getData().setLabel(2);
            }
        }
    }
}
```

Como sabemos, en la clase TestGrafo está el objeto de tipo Grafo, 'grafo', que contiene nombre de ciudades del Perú. Realizaremos el recorrido DFS para el Grafo empezando desde la ciudad de Cusco.

```
// Probando el recorrido DFS
System.out.println("Probando el recorrido DFS para Cusco");
grafo.DFS("Cusco");
```

Veamos el resultado por consola al utilizar el método DFS.

```
Probando el recorrido DFS para Cusco
Cusco,
Arequipa,
Lima,
Piura,
Tarapoto,
```

d) Recorrido Dijkstra:

4. Solucionar, dibujar y mostrar la Lista de Adyacencia para el Grafo de palabras

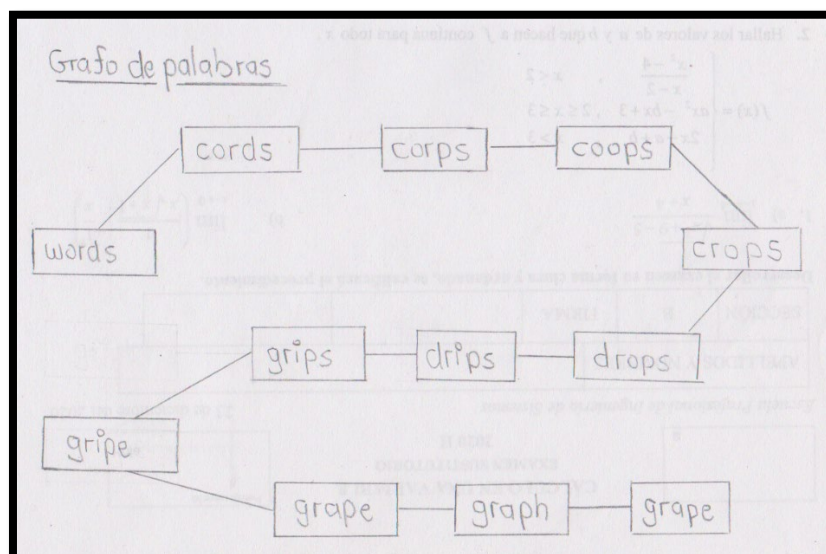
En este caso, se nos pone como 'condiciones' lo siguiente:

- Para que dos palabras sean adyacentes deben diferir exactamente en una posición.
- Las palabras o Vértices del Grafo son las siguientes: words, cords, corps, coops, crops, drops, drips, grips, gripe, grape, graph.

Para poder resolver este ejercicio, debemos en primer lugar identificar las palabras que son adyacentes según la regla establecida de solo diferir exactamente en una posición. Una vez identificado las palabras que son adyacentes podemos armar la Lista de Adyacencia para cada palabra de nuestro Grafo, en este caso se usó como apoyo una hoja de cálculo.

Grafo de Palabras - Ejercicio 4			
PALABRAS	Lista de Adyacencia		
words	cords		
cords	words	corps	
corps	cords	coops	
coops	corps	crops	
crops	coops	drops	
drops	crops	drips	
drips	drops	grips	
grips	drips	gripe	
gripe	grips	grape	
grape	gripe	graph	
graph	grape		

Una vez tenemos la Lista de Adyacencia de cada palabra podemos dibujar el Grafo, siendo sus Vértices las palabras y sus Aristas estando en la Lista de Adyacencia.



Con el código de Grafo ya implementado, podemos crear un Grafo de Strings e ingresar los Vértices y Aristas para el Grafo de Palabras. Este proceso está en la clase 'GrafoPalabras'.

```
public class GrafoPalabras {

    public static void main(String[] args) {

        // Creando el Grafo de Palabras - Ejercicio4
        System.out.println("Creando el Grafo 'palabras'\n");
        Grafo<String> palabras = new Grafo<String>();

        System.out.println("Agregando los Vertices y Aristas\n");

        // Agregando los Vertices
        palabras.insertarVertice("words");
        palabras.insertarVertice("cords");
        palabras.insertarVertice("corps");
        palabras.insertarVertice("coops");
        palabras.insertarVertice("crops");
        palabras.insertarVertice("drops");
        palabras.insertarVertice("drips");
        palabras.insertarVertice("grips");
        palabras.insertarVertice("gripe");
        palabras.insertarVertice("grape");
        palabras.insertarVertice("graph");
    }
}
```

Recordar que, como tenemos la Lista de Adyacencia para cada palabra, podemos establecer las Aristas según las Listas de Adyacencia de cada palabra registrada como Vértice. También podemos mostrar el Grafo y su Lista de Adyacencia por consola.

```
// Agregando las Aristas
palabras.insertarArista("words", "cords");
palabras.insertarArista("cords", "corps");
palabras.insertarArista("corps", "coops");
palabras.insertarArista("coops", "crops");
palabras.insertarArista("crops", "drops");
palabras.insertarArista("drops", "drips");
palabras.insertarArista("drips", "grips");
palabras.insertarArista("grips", "gripe");
palabras.insertarArista("gripe", "grape");
palabras.insertarArista("grape", "graph");

// Lista de Adyacencia para el Grafo 'palabras'
System.out.println("Lista de Adyacencia:\n");
System.out.println(palabras);
```

Finalmente, mostraremos la salida por consola. Esta concuerda con el trabajo realizado a mano y el Grafo graficado con sus Listas de Adyacencia con apoyo de la hoja de cálculo.

```
Creando el Grafo 'palabras'

Agregando los Vertices y Aristas

Lista de Adyacencia:

Contenido del Grafo:
words ==> cords,
cords ==> words, corps,
corps ==> cords, coops,
coops ==> corps, crops,
crops ==> coops, drops,
drops ==> crops, drips,
drips ==> drops, grips,
grips ==> drips, gripe,
gripe ==> grips, grape,
grape ==> gripe, graph,
graph ==> grape,
```

5. Realizar un método para ver si un Grafo incluye a otro.

Para poder realizar este método, el cual tendrá por nombre 'incluye' y se encontrará en la clase Grafo, debemos verificar los Vértices y sus Listas de Adyacencia del Grafo más pequeño o el que se quiere comprobar si está incluido dentro de otro más grande o igual. El método incluye de clase Grafo recibe dos Grafos, uno que llama al método y es el más grande y otro que se envía como parámetro y es el más pequeño. El proceso que realiza el método es hacer una copia de la Lista de Vértices del Grafo más grande.

```
// Metodo para ver si un Grafo incluye a Otro
// OJO: El Grafo que llama al metodo es el mas grande
// y el que enviamos por paramatro es el mas pequeno
public boolean incluye(Grafo<T> otroGrafo) {
    // Variable de apoyo para el retorno del metodo
    boolean retornar = false;
    // Rescatar los Vertices del Grafo grande
    ListaEnlazada<Vertice<T>> copiaG = getVertices().copiar();
    // Rescatar el Nodo inicial del Grafo pequeno
    Nodo<Vertice<T>> inicio = otroGrafo.getVertices().getFirst();
    // Mientras que haya Vertices, iteraremos
    while(inicio != null) {
        Vertice<T> buscar = copiaG.search(inicio.getData());
        // Si se encuentra entonces verificar la Lista de Adyacencia
        if(buscar != null) {
            System.out.println("El Vertice " + buscar.getData()
                + ", esta incluido");
            // Verificando Lista de Adyacencia esta incluida
            if(buscar.getLista().incluye(inicio.getData().getLista())) {
                // Si las Lista de Adyacencia esta incluida devolver true
                System.out.println("Las Listas de Adyacencia esta incluida");
                copiaG.remove(inicio.getData());
                retornar = true;
                inicio = inicio.getNext();
            } else {
                // Si las Lista de Adyacencia no esta incluida devolver false
                // y romper el ciclo ya que no es necesario revisar mas
                System.out.println("Las Listas de Adyacencia no esta incluida");
                retornar = false;
                break;
            }
        }
    }
    return retornar;
}
```

Luego guarda el primer Vértice del Grafo más pequeño en un Nodo para poder hacer las comparaciones, primero debemos verificar que el Vértice del Grafo pequeño existe dentro del Grafo grande. Si el Vértice existe, entonces debemos verificar si la Lista de Adyacencia del Vértice del Grafo pequeño está incluida en la del Vértice del Grafo grande. Si la Lista de Adyacencia está incluida entonces removemos ese Vértice de la copia del Grafo grande, establecemos el retorno como verdadero y pasamos al siguiente Vértice del Grafo más pequeño. En caso de estar incluida la Lista de Adyacencia, establecer el retorno como falso y romper el ciclo while ya que como difieren sus Listas de Adyacencia no es necesario verificar las demás. Finalmente devolver la variable auxiliar 'retorno'.

Para probar este método crearemos dos Grafos adicionales, sus nombres serán 'otroGrafo' y 'otroGrafo2'. Veamos sus Vértices y Aristas de estos dos grafos respectivamente.

```
// Creando otro objeto de tipo Grafo para poder ver si este
// incluye al creado por primera vez objeto 'grafo'

Grafo<String> otroGrafo = new Grafo<String>();

// Insertando algunos Vertices
otroGrafo.insertarVertice("Lima");
otroGrafo.insertarVertice("Arequipa");
otroGrafo.insertarVertice("Cusco");
otroGrafo.insertarVertice("Piura");
otroGrafo.insertarVertice("Tarapoto");
otroGrafo.insertarVertice("Moquegua");
otroGrafo.insertarVertice("Callao");

// Insertando algunas Aristas
otroGrafo.insertarArista("Lima", "Arequipa", 8);
otroGrafo.insertarArista("Moquegua", "Callao", 10);
otroGrafo.insertarArista("Cusco", "Arequipa", 4);
otroGrafo.insertarArista("Piura", "Lima", 5);
otroGrafo.insertarArista("Lima", "Callao", 5);
otroGrafo.insertarArista("Cusco", "Lima", 6);
otroGrafo.insertarArista("Piura", "Tarapoto", 3);
```

Otro Grafo de ciudades:

Contenido del Grafo:

Lima ==> Arequipa, Piura, Callao, Cusco,
 Arequipa ==> Lima, Cusco,
 Cusco ==> Arequipa, Lima,
 Piura ==> Lima, Tarapoto,
 Tarapoto ==> Piura,
 Moquegua ==> Callao,
 Callao ==> Moquegua, Lima,

```
// Creando otro objeto de tipo Grafo

Grafo<String> otroGrafo2 = new Grafo<String>();

// Insertando algunos Vertices
otroGrafo2.insertarVertice("Lima");
otroGrafo2.insertarVertice("Arequipa");
otroGrafo2.insertarVertice("Cusco");
otroGrafo2.insertarVertice("Piura");
otroGrafo2.insertarVertice("Tarapoto");
otroGrafo2.insertarVertice("Moquegua");
otroGrafo2.insertarVertice("Callao");

// Insertando algunas Aristas
otroGrafo2.insertarArista("Lima", "Arequipa", 8);
otroGrafo2.insertarArista("Moquegua", "Callao", 10);
otroGrafo2.insertarArista("Cusco", "Arequipa", 4);
otroGrafo2.insertarArista("Lima", "Callao", 5);
otroGrafo2.insertarArista("Cusco", "Lima", 6);
otroGrafo2.insertarArista("Piura", "Tarapoto", 3);
```

'Otro Grafo 2' de ciudades:

Contenido del Grafo:

Lima ==> Arequipa, Callao, Cusco,
 Arequipa ==> Lima, Cusco,
 Cusco ==> Arequipa, Lima,
 Piura ==> Tarapoto,
 Tarapoto ==> Piura,
 Moquegua ==> Callao,
 Callao ==> Moquegua, Lima,

Finalmente pasaremos a comparar ambos Grafos adicionales creados. Se mostrará por consola las comparaciones hechas y el resultado si el Grafo incluye al Grafo creado inicialmente que lleva por nombre 'grafo' en la clase TestGrafo.

- a) Comparando 'otroGrafo' con 'grafo': Recordar que el Grafo más grande llama al método incluye y el grafo más pequeño se envía como parámetro en el método. En este caso 'otroGrafo' incluye a 'grafo'.

```
// Probando el metodo 'incluye' de la clase Grafo
// En este caso 'grafo' esta incluido en 'otroGrafo'
System.out.println("Comparando 'otroGrafo' con 'grafo'\n");
System.out.println("El Grafo 'otroGrafo' incluye a 'grafo'? : "
+ otroGrafo.incluye(grafo));
```



```

Comparando 'otroGrafo' con 'grafo'

El Vertice Lima, esta incluido
El elemento Arequipa, , esta incluido
El elemento Piura, , esta incluido
El elemento Cusco, , esta incluido
Las Listas de Adyacencia esta incluida
El Vertice Arequipa, esta incluido
El elemento Lima, , esta incluido
El elemento Cusco, , esta incluido
Las Listas de Adyacencia esta incluida
El Vertice Cusco, esta incluido
El elemento Arequipa, , esta incluido
El elemento Lima, , esta incluido
Las Listas de Adyacencia esta incluida
El Vertice Piura, esta incluido
El elemento Lima, , esta incluido
El elemento Tarapoto, , esta incluido
Las Listas de Adyacencia esta incluida
El Vertice Tarapoto, esta incluido
El elemento Piura, , esta incluido
Las Listas de Adyacencia esta incluida
El Grafo 'otroGrafo' incluye a 'grafo'? : true

```

- b) Comparando 'otroGrafo2' con 'grafo': Recordar que el Grafo más grande llama al método incluye y el grafo más pequeño se envía como parámetro en el método. En este caso 'otroGrafo2' no incluye a 'grafo' ya que falta la Arista entre Lima y Piura.

```

// Probando el metodo 'incluye' de la clase Grafo
// En este caso 'grafo' NO esta incluido en 'otroGrafo'
// YA QUE FALTA LA ARISTA LIMA - PIURA QUE EXISTE EN 'GRAFO'
System.out.println("Comparando 'otroGrafo2' con 'grafo'\n");
System.out.println("El Grafo 'otroGrafo2' incluye a 'grafo'? : "
    + otroGrafo2.incluye(grafo));

```

```

Comparando 'otroGrafo2' con 'grafo'

El Vertice Lima, esta incluido
El elemento Arequipa, , esta incluido
El elemento Piura, , no esta incluido
Las Listas de Adyacencia no esta incluida
El Grafo 'otroGrafo2' incluye a 'grafo'? : false

```