UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERIA DE PRODUCCION Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

Curso : **ESTRUCTURA DE DATOS Y ALGORITMOS**

Docente : **Ing. Edith Pamela Rivero Tupac de Lozano**

Informe de laboratorio

Laboratorio 09: Grafos

Estudiante : Fatima Gigi Rojas Carhuas
Cui: 20192183

Arequipa - Perú
2021

Competencias:

- Construye responsablemente soluciones haciendo uso de estructuras de datos y algoritmos, siguiendo un proceso adecuado para resolver problemas computacionales que se ajustan al uso de los recursos disponibles y a especificaciones concretas.

Objetivos:

- Comprender y aplicar los conceptos de grafos

Ejercicios propuestos

1. Crear un repositorio en GitHub, donde incluyan la resolución de los ejercicios propuestos y el informe.
   https://github.com/Fatima2427/LabGrafo.git
2. Implementar el código de Grafo cuya representación sea realizada mediante LISTA DE ADYACENCIA.

```java
import java.util.*;
public class Grafo<E> {
    ArrayList<Vertice> vertices ;

    public Grafo() {
        vertices = new ArrayList<Vertice>();

    }
    boolean search( E data) {
        for ( int i=0; i<vertices.size(); i++) {
            if(vertices.get(i).data.equals(data))
                return true;
            }
        return false;
    }
    public void insertV(E data) {
        if(this.search(data) != false){
            System.out.println("Vertice ya fue insertado anteriormente");
            return;
        }
        this.vertices.add(new Vertice<E>(data));

    }
    public void insertAri(E verOri, E verDes) {//arista insert
        insertAri(verOri, verDes, -1);
    }
    public void inserWord(String w1, String w2) {
        if(comparar(w1,w2)==true)
            insertAri((E)w1, (E)w2, -1);
        else
            return;
    }
    public void insertAri(E verOri, E verDes, int weitght) {
        Vertice<E> orig= new Vertice<E> (verOri);
        Vertice<E> des= new Vertice<E> (verDes);
        if(this.search(verOri)==false || this.search(verOri) == false){
            System.out.println("Vertice origen o destino no existen");
            return;
            }
```

```java
public class Test<E> {
    public static void main ( String [] args) {// tarea
        Grafo <Integer> g= new Grafo<Integer> ();
        g.insertV(2);
        g.insertV(1);
        g.insertV(3);
        g.insertV(4);

        g.insertAri(1, 2);
        g.insertAri(2, 3);
        g.insertAri(3, 4);
        g.insertAri(1,3);
        g.insertAri(2, 4);

        System.out.println("\n" + g);
        System.out.println("----------Subgrafo----------");
        Grafo <Integer> g2= new Grafo<Integer> ();
        g2.insertV(2);
        g2.insertV(1);
        g2.insertV(4);
        g2.insertAri(1, 2);
        g2.insertAri(1, 4);
        System.out.println("\n" + g2);
        int pos =g.posicion(g.vertices.get(1));
        System.out.println(pos);

    }
```

```
<terminated> Test (4) [Java Application] C:\Pro

2-->1,3,4,
1-->2,3,
3-->2,4,1,
4-->3,2,

----------Subgrafo----------

2-->1,
1-->2,4,
4-->1,

1
```

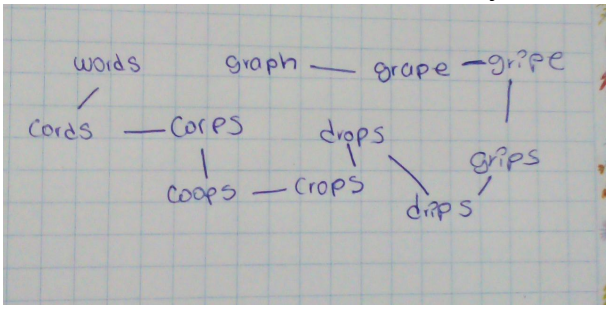
3. Implementar BSF, DFS y Dijkstra con sus respectivos casos de prueba.

```java
public void BFS( E n) {
    ArrayList<Vertice> listaAdy = new ArrayList<Vertice>();
    Vertice ver= new Vertice(n);
    Vertice nodo = this.vertices.get(this.posicion(ver));
    int [] distancia= new int[this.vertices.size()];
    String[] padres = new String[this.vertices.size()];
    Queue<Vertice<E>> queue = new LinkedList<Vertice<E>>();

    queue.add(nodo);
    distancia[this.posicion(ver)]=0;
    Vertice<E> aux, siguiente, i;
    while (!queue.isEmpty()){
        aux=queue.poll();
        for ( int j=0; j<aux.edges.size();j++) {
            siguiente=aux.edges.get(j).refDes;
            if( siguiente.visitado==false) {
                listaAdy.add(siguiente);
                siguiente.visitado=true;
                distancia[this.posicion(siguiente)]=distancia[this.posic
                queue.add(siguiente);
            }
        }
    }
    for ( int l =0; l<listaAdy.size();l++) {
        System.out.println(listaAdy.get(l).mostrar());
    }
```

```java
public int tamañoGrafo() {
    return this.vertices.size();
}
public void visitas(Vertice<E> nuevo ) {
    this.posicion(nuevo);
    this.vertices.get(this.posicion(nuevo)).visitado=true;
    nuevo=this.vertices.get(this.posicion(nuevo));
    System.out.print(this.vertices.get(this.posicion(nuevo)).data);
    for(int j=0;j<tamañoGrafo();j++){
        for ( int i=0; i<nuevo.edges.size();i++) {
            ArrayList Adyacente =nuevo.edges;
            if (Adyacente.get(i).equals(this.vertices.get(j).data) ) {
                if(this.vertices.get(i).visitado==false) {
                    this.vertices.get(i).visitado=true;
                    System.out.print(this.vertices.get(i).data);
                }
            }
        }
    }
```

4. Solucionar el siguiente ejercicio:
   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 corps corps drops drips grips gripe grape graph
   b. Mostrar la lista de adyacencia del grafo.




```java
public boolean comparar(String p1, String p2){
    int cont=0;
    for(int i=0;i<p1.length();i++) {
        if(p1.charAt(i)!=p2.charAt(i))
            cont++;
    }
    if ( cont ==1) {
        return true;}
    return false;
    }
```

```java
public class TestPalabras {
    public static void main ( String [] args) {// tarea
        Grafo <String> g= new Grafo<String> ();
        String[] word = {"words", "cords", "corps", "coops",
"crops", "drops", "drips", "grips", "gripe", "grape" ,"graph"};

        for (int i=0; i<word.length; i++) {
            g.insertV(word[i]);
        }
        for (int i=0; i<word.length; i++) {
            for (int j=i; j<word.length; j++) {
                g.inserWord(word[i],word[j]);
            }
        }
        System.out.println(g);
    }
}
```

```
<terminated> TestPalabras [Java Application] C:\Program
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 en la clase Grafo. Este método permitirá saber si un grafo está incluido en otro. Los parámetros de entrada son 2 grafos y la salida del método es true si hay inclusión y false el caso contrario.

```java
public  boolean compararGrafos(Grafo<E> n1, Grafo<E> n2) {
    int cont=0;
    //primera comparación
    if (n1.vertices.size()<n2.vertices.size() ) {
       return compararGrafos(n2,n1);
    }
    for ( int i =0; i<n2.vertices.size();i++) {
        for ( int j =0; j<n1.vertices.size();j++) {
            if( n2.vertices.get(i).data.equals(n1.vertices.get(j).data)
                cont++;

        }
    }
    if( cont ==n2.vertices.size())//comparo cant de vertices
        for ( int i =0; i<n2.vertices.size();i++) {
            for ( int j =0; i<n2.vertices.get(i).edges.size();i++) {
                int pos =posicion(n2.vertices.get(i));
                buscarVertice(pos,n1, -1 );
            }
        }
   return false;

}
```

Cuestionario

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

   El fundamento sobre el que se asienta este algoritmo es el principio de optimalidad: si el camino más corto entre los vértices u y v pasa por el vértice w, entonces la parte del camino que va de *w* a *v* debe ser el camino más corto entre todos los caminos que van de *w* a *v* .

   A continuaciòn se mencionan algunas variantes y sus diferencias :

   - **Algoritmo Dijkstra con Heap Binomial:** La estructura de datos de Binomial Heap, también conocida como cola de prioridad, es una implementación alternativa del algoritmo de Dijkstra para encontrar el nodo con la etiqueta de menor costo.
     Su principal diferencia va de la mano con su objetivo el cual es reducir el tiempo de ejecución del algoritmo de Dijkstra para determinar un nodo en cuestión además de usar el heap binomial.
   - **Algoritmo de Dial:** Otra variante cuya diferencia radica en las etiquetas de distancia que el algoritmo de Dijkstra designa como permanentes no van a disminuir su valor.
   - **Variante pgr_dijkstraCost:** Se implementó para la realización de un proyecto donde el usuario dispone de una variante del algoritmo Dijkstra en

la que uno indica el punto de partida y el punto de llegada y a cambio, simplemente, obtiene el costo total del trayecto .

- **A ***: se considera una "mejor primera búsqueda" porque elige con avidez qué vértice explorar a continuación, de acuerdo con el valor de f(v)[f(v) = h(v) + g(v)], donde *h* es la heurística y *g* el costo acumulado hasta el punto actual.
- **Arc-Flags:** Para realizar un trabajo se implementó arc-flags en el cálculo de Dijkstra para evitar explorar caminos innecesarios, siendo esta su mayor diferencia. El algoritmo comprueba la entrada de la bandera de la región de destino correspondiente (la región a la que pertenece el nodo de destino *t*) cada vez que el algoritmo de Dijkstra quiera atravesar un arco.

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

El problema del camino más corto es aquel que consiste en encontrar un camino entre dos nodos de manera que la suma de los costes de los nodos que lo constituyen es mínima.

Algoritmos	Definición	Similitudes	Diferencias	Cuando utilizar
Dijkstra	El algoritmo de Dijkstra es un algoritmo eficiente de complejidad O(n2) que sirve para encontrar el camino de coste mínimo desde un nodo origen a todos los demás nodos del grafo.	resuelve el problema de los caminos más cortos desde un único nodo origen hasta todos los otros nodos del grafo	se usa solo cuando tiene una sola fuente mas no maneja pesos negativos	Sirve para cualquier grafo con pesos (dirigido o no) siempre y cuando sus pesos no sean negativos
Bellman-Ford	determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos.		permite que la ponderación de los nodos sea negativa	se puede utilizar para aplicaciones en las que están presente aristas con peso negativo
Algoritmo de Floyd-Warshall	es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados.		resuelve todos los pares de caminos más cortos. Maneja los bordes positivos y negativos sin embargo fallara en ciclos negativos	se puede utilizar para determinar el camino mínimo entre todos los pares de vértices de un grafo, comparando todos los posibles caminos para determinar el mas óptima
Algoritmo de Johnson	La regla de Johnson es un algoritmo heurístico utilizado para resolver situaciones de secuenciación de procesos que operan dos o más órdenes (operaciones) que pasan a través de dos máquinas o		puede ser más rápido que el de Floyd-Warshall en grafos de baja densidad.	camino más corto entre todos los pares de vértices de un grafo dirigido disperso, también permite determinar una secuencia u orden para realizar trabajos en un taller que considera 2 máquinas

	centros de trabajo.			
Algoritmo de Viterbi	es un algoritmo de programación dinámica que permite hallar la secuencia más probable de estados ocultos que produce una secuencia observada de sucesos		resuelve el problema del camino estocástico más corto con un peso probabilístico adicional en cada nodo.	Se puede usar para la descodificación