Universidad Nacional de San Agustín
Facultad de Ingeniería de Producción y Servicios
Escuela Profesional de Ingeniería de Sistemas

Estructura de Datos y Algoritmos - Laboratorio

Tema: Informe sobre Grafos

Alumno:

- Mamani Condori, Kevin Alonso

Profesor:

- Rivero Tupac de Lozano Edith Pamela

Arequipa - Perú
2021

I. EJERCICIOS PROPUESTOS

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

   Enlace: https://github.com/KevinAMamaniC2020/Implementacion_Grafos

2. Implementar el cogido de Grafo cuya representación sea realizada mediante LISTA DE ADYACENCIA. (3 puntos)

   Para este caso se implementa 5 clases para el desarrollo de la implementación de grafos:

   a) ListLinked
   Esta clase funciona como una lista enlazada normal, con los métodos iguales (insertar, buscar y eliminar) o similares a como se haría normalmente ya que se van a agregar métodos cuando se requiera para poder crear los grafos.

```java
public class ListLinked<T> {

	protected Node<T> first, last;

	public ListLinked() {
		this.first=null;
		this.last=null;
	}

	public Node<T> getFirst(){
		return first;
	}

	public void setFirst(Node<T>first) {
		this.first=first;
	}

	boolean isEmpty() {
		return this.first==null;
	}
```

```java
	public T search(T data) {
		Node<T> nodo =this.first;
		while(nodo != null && !nodo.data.equals(data))
			nodo = nodo.getNext();
		if(nodo != null)
			return nodo.data;
		return null;
	}

	void insertFirst(T data) {
		this.first= new Node<T>(data,this.first);
	}

	void insertLast(T data) {
		if(first == null)
			this.last =this.first=new Node<T>(data, this.first);
		else {
			this.last.setNext(new Node<T>(data));
			this.last =this.last.getNext();
		}
	}


	public String toString() {
		String r="";
		Node<T> aux=this.first;
		while(aux !=null) {
			r=r+aux.getInfo();
			aux=aux.getNext();
		}
		return r;
	}
```

   b) Node
   Esta clase lo que va a realizar es la misma implementación de una lista enlazada teniendo parámetros para guardar el dato y recorrer la lista enlazada con el parámetro next, se definen también sus getters y setters.

```java
public class Node<Type> {

	protected Type data;
	protected Node<Type>next;

	public Node(Type data) {
		this.data=data;
		this.next=null;
	}

	public Node(Type data, Node<Type> next) {
		this.data=data;
		this.next=next;
	}

	public Type getInfo() {
		return data;
	}

	public void setData(Type data) {
		this.data = data;
	}

	public Node<Type> getNext() {
		return next;
	}
	public void setNext(Node<Type> next) {
		this.next = next;
```

   c) Edge
   Representa las aristas que van a unir los vértices, definiendo como parámetros de entrada al vertice, la longitud por si se quiere un grafo ponderado y una etiqueta que se utilizaran para los

algoritmos. El toString se utiliza mas que todo para imprimir el peso de la arista, el equals sirve para verificar la unión y los contrustores para definir os parámetros.

```java
public class Edge<E> {

	protected Vertex<E> refDest;
	protected int weight;
	protected int label;// 0= unexplored 1=discovery 2=back 3=cross


	public Edge(Vertex<E> refDest) {
		this(refDest,-1);
	}

	public Edge(Vertex<E> refDest, int weight) {
		this.refDest= refDest;
		this.weight= weight;
	}

	public boolean equals(Object o) {
		if(o instanceof  Edge<?>) {
			Edge<E> e = (Edge<E>)o;
			return this.refDest.equals(e.refDest);
		}
		return false;
	}

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

d) Vertex

La clase va a representar en vertice del grafo, en este caso se definen los parámetros de etiqueta (para usarlo en los algoritmos), datos y las lista adyacente, donde se va a usar la lista enlazada y la clase Edge para poder relacionarse con aristas. Se crea el constructor para almacenar los parámetros, el toString para imprimir los datos, equals para comparar los datos de la lista enlazada y el vertice.

```java
public class Vertex<E> {

	protected E data;
	protected ListLinked<Edge<E>> listAdj;
	protected int label;// 0= unexplored 1=visited
	/*protected int dist;
	protected Vertex<E>path;*/

	public Vertex(E data) {
		this.data=data;
		listAdj=new ListLinked<Edge<E>>();
	}

	public E getData() {
		return data;
	}

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

	public String toString() {
		return this.data+"----->"+ this.listAdj.toString() +"\n";
	}
```

e) GraphLink

Para el grafo se llama a la clase de lista enlazada, quien va a contener al vertice. El constructor se define esta acción para poder agregar datos en la lista. Nos servirá para implementar los algoritmos posterioemente.

```java
public class GraphLink <E>{

	protected ListLinked<Vertex<E>> listVertex;
	public GraphLink() {
		listVertex = new ListLinked<Vertex<E>>();
	}

```

El método vacio insertVertex se crea para insertar los vértices que queremos en nuestro grafo, en caso de que sea igual retorna un mensaje de error.

```java
public void insertVertex(E data) {
        Vertex<E> nuevo = new Vertex<E>(data);
        if(this.listVertex.search(nuevo) !=null) {
                System.out.println("Vertice insertado con anterioridad ....");
                return;
        }

        this.listVertex.insertFirst(nuevo);
}
```

El método vacio insertEdge se divide en dos partes, siendo para aristas ponderadas y no ponderadas, ahí también se puede definir si el grafo será dirigido o no, debido a que tiene dos valores de ingreso, la diferencia de ambos métodos es que uno necesita el peso que va a tomar la arista.

```java
//No ponderado
public void insertEdge(E verOri, E verDes) {
        Vertex<E> refOri = this.listVertex.search(new Vertex<E>(verOri));
        Vertex<E> refDes = this.listVertex.search(new Vertex<E>(verDes));

        if(refOri == null || refDes ==null) {
                System.out.println("Vertice origen y/o destino no existen..");
                return;
        }

        if(refOri.listAdj.search(new Edge<E>(refDes)) != null) {
                System.out.println("Arista ya insertada..");
                return;
        }

        refOri.listAdj.insertFirst(new Edge<E>(refDes));//dirigido
        //refDes.listAdj.insertFirst(new Edge<E>(refOri));//no dirigido
}
```

Y por ultimo un toString que permite imprimir el grafo.

```java
public String toString() {
        return this.listVertex.toString();
}
```

3. Implementar BSF, DFS y Dijkstra con sus respectivos casos de prueba. (5 puntos)

   a) BSF: El algoritmo BSF funciona de una forma mas directa, buscando los caminos cortos de una forma más compleja que un DFS, debido a que toma todos los vértices que está a su alrededor, dentro de su implementación esta algo incompleta.

Algoritmo BFS

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

   for all n ∈ G.vertices()
     setLabel(u, 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 secuence
     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
```

```java
private void initLabel() {
    Node <Vertex<E>> aux = this.listVertex.first;
    for(; aux !=null ; aux =aux.getNext()) {
        aux.data.label=0;
        Node<Edge<E>> auxE = aux.data.listAdj.first;

        for(; auxE !=null ; auxE =auxE.getNext())
            auxE.data.label =0;
    }
}
```

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

private void BFSRec (Vertex<E> s) {
    ListLinked<Vertex<E>> nuevo = new ListLinked<Vertex<E>>();
    s.label=1;
    nuevo.insertLast(s);
    Node<Vertex<E>> i=nuevo.last;
    //System.out.println(s.data+" ");//solo el dato
    while(!nuevo.isEmpty()) {
        System.out.println(s+" ");//saca lista de adyacencia
        ListLinked<Vertex<E>> nu = new ListLinked<Vertex<E>>();
        Node<Edge<E>> e = s.listAdj.first;
        for(; i!=null; i=i.getNext() ) {
            for(; e!=null;e=e.getNext()) {
                if(e.data.label == 0) {
                    Vertex<E> w = e.data.refDest;
                    if(w.label == 0) {
                        e.data.label = 1;
                        w.label = 1;
                        nu.insertLast(w);
                    }
                    else
                        e.data.label=3;
                }
            }
        }
        //i=i.getNext();
    }
}
```

b) DFS: El algoritmo DFS se tienen una implementación mas simple debido a que busca un vértice y lo utiliza como camino, lo malo es que tiene que marcar todos los vértices para saber en cual de todos hay que buscar.

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(u, 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 gerLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G,w)
            else
                setLabel(e, BACK)
```

```java
private void initLabel() {
    Node <Vertex<E>> aux = this.listVertex.first;
    for(; aux !=null ; aux =aux.getNext()) {
        aux.data.label=0;
        Node<Edge<E>> auxE = aux.data.listAdj.first;

        for(; auxE !=null ; auxE =auxE.getNext())
            auxE.data.label =0;
    }
}
```

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

private void DFSRec (Vertex<E> v) {
    v.label=1; // 1 visitado, 0 dicovery , 2 Back
    System.out.println(v+" "); //saca lista de advacencia
    //System.out.println(v.data+" "); //solo el dato
    Node<Edge<E>> e = v.listAdj.first;

    for(; e!=null;e=e.getNext()) {
        if(e.data.label == 0) {
            Vertex<E> w = e.data.refDest;
            if(w.label == 0) {
                e.data.label =1;
                DFSRec(w);
            }
            else
                e.data.label=2;
        }
    }

}
```

c) Dijkstra
En caso de dijkstra es necesario implementar una cola de prioridad basando en el pseudocodigo dado en clase.
Nota: No se llego a implementar

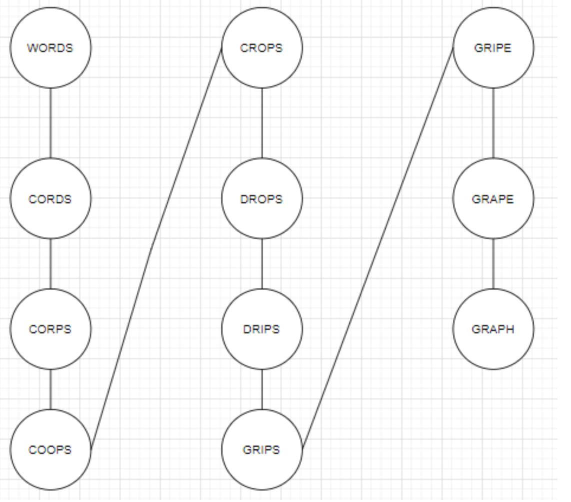
Algoritmo de Dijkstra

```
Algorithm DijkstraShortestDistances(G,v)
  Input: A simple undirected graph G with nonegative edge weights and a vertex v.
  Output: A label D[u] for each vertex u, such that D[u] is the shortest distance from v to u in G
  for all u ∈ G.vertices()
    if  u = v
      D[u] ← 0;
    else
      D[u] ← +∞;
  Let Q be a prority queue that contains all the vertex of G using D labels as keys.
  while  ¬Q.isEmpty()
    u ← Q.removeMin()
    for each vertex z adjacent to u such that z is in Q
      if  D[z] > D[u] + weight(u,z) then
        D[z] ← D[u] + weight(u,z)
        Change to D[z] the key of z in Q
  return D[u] for every u
```

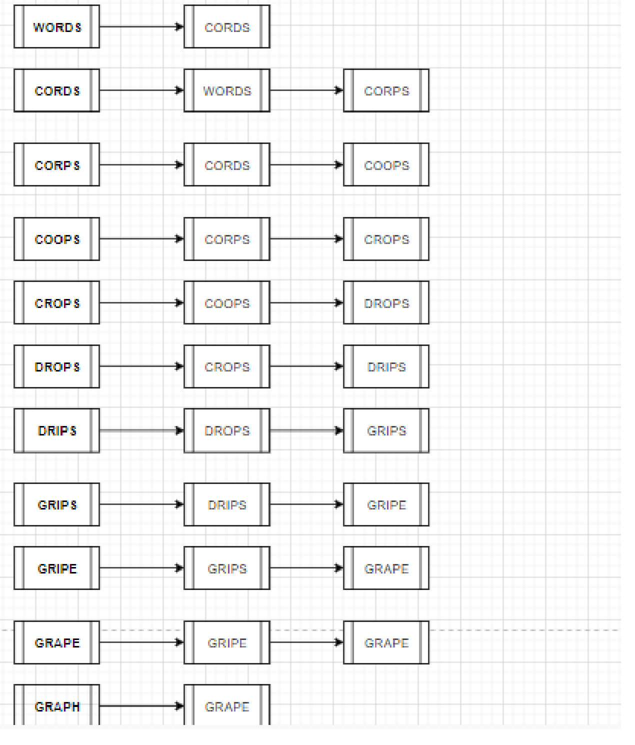
4. Solucionar el siguiente ejercicio: (5 puntos)

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.



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. (4 puntos)

   Nota: No se implemento

II. CUESTIONARIO

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

   En las variantes del algoritmo de Dijkstra se encontraron 3:

   - Dijkstra con lista enlazada: Explora todos los caminos mas cortos que parten del vértice de origen a los demás. Utiliza una búsqueda de costo uniforme. No funciona con aristas negativas. Si resuelve el camino más corto.
     Complejidad:

$$O(V^2)$$

- Dijkstra con montículo binario: Se basa en arboles binarios balanceados. Se representa con mínimos y máximos. No necesita de punteros debido a que se almacena en orden. Se representa fácilmente en un arreglo.
  Complejidad:

  $$O((E + V) \log V)$$

- Dijkstra con montículo de Fibonacci: Se distribuye la solución a través de un bosque de arboles. Satisface la propiedad de orden mínimo del montículo. Se concatena con 2 heaps para reducir el costo de almacenamiento. Se realiza en un tiempo constante.
  Complejidad:

  $$O(E + V \log V)$$

2. Investigue sobre los ALGORITMOS DE CAMINOS MINIMOS e indique, ¿Qué similitudes encuentra, qué diferencias, en qué casos utilizar y por qué? (2 puntos)

   a) Bellman-Ford: resuelve el problema de los caminos más cortos desde un origen permitiendo que la ponderación de los nodos sea negativa.

      - Casos en que se utiliza
        Solución de un camino hamiltoniano.
        Aplicación en Roturas
        Colección de redes IP

   b) Búsqueda A*: resuelve el problema de los caminos más cortos entre un par de nodos usando la heurística para agilizar la búsqueda.

      - Casos en que se utiliza
        Aplicación de cubos Rubik:menor camino de solucion
        Construcción de caminos: áreas mas cortas
        Construccion de ciudades: rutas de comercio

   c) Floyd-Warshall: resuelve el problema de los caminos más cortos entre todos los nodos.

      - Casos en que se utiliza
        Encontrar expresiones regulares: algoritmo de Kleene
        Invertir en matrices
        Ruta mas optima
        Comprobación de un grafo bipartito

   d) Jhonson: resuelve el problema de los caminos más cortos entre todos los nodos y puede ser más rápido que el de Floyd-Warshall en grafos de baja densidad.

      - Casos en que se utiliza
        Minimizar el tiempo
        Complementar con el metodo de Bellman-Ford
        Resolver grafos transformados

   e) Viterbi: resuelve el problema del camino estocástico más corto con un peso probabilístico adicional en cada nodo.

      - Casos en que se utiliza
        Reconocimiento de voz
        Biologia molecular
        Como complemento de Tries

   f) Similitudes

- Todos los algoritmos solucionan el problema de hallar el camino corto.
- Algunos algoritmos pueden fusionarse, sea para sacar mayor ventaja en ciertos aspectos que no pueden solucionar, como nodos negativos o caminos.
- La mayoría de algoritmos trabajan con grafos ponderados.

g) Diferencias
- El algoritmo de Floyd-Warshall trabajan con grafos ponderados. Por lo que el elemento puede ser cualquier entero o infinito, es decir que no hay unión entre nodos.
- La diferencia entre los algoritmos de Bellman-Fort y Dijkstra se pueden trabajar con vértices negativos debido a que Dijkstra no lo hace.
- Floy-Warshall y Dijkstra son unos de los pocos algoritmos que pueden resolverse con caminos ponderados a diferencia de los ya mencionados.
- El algoritmo de búsqueda a* es una función heurística, pero no puede garantizar una solución muy óptima para la solución de caminos.
- El algoritmo de jhonson utiliza montículos de Fibonacci para optimizar el tiempo, lo que conlleva a tener resultados en menor tiempo que los demás.
- El algoritmo de Viterbi trabaja con datos de bits lo cual es utilizado para redes de comunicación, teniendo una decodificación optima.