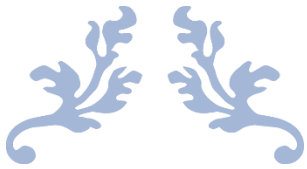


ESTRUCTURAS DE DATOS

Trabajo Grafos

04 DE AGOSTO DE 2021

UNIVERSIDAD NACIONAL DE SAN AGUSTIN
Juan Pedro Vidal Pastor Pastor

Ejercicios Propuestos Laboratorio 9

Ejericio1: Crear un repositorio en GitHub, donde incluyan la resolución de los ejercicios propuestos y el informe.

https://github.com/JuanPastorP/grafos-listaAdyacencia.git

Ejercicio2: Implementar el cogido de Grafo cuya representación sea realizada mediante LISTA DE ADYACENCIA.

Funciones:

1. **Nodo Vértice:** Marca la dirección del siguiente vértice(nextVertice), del inicio de la lista de las aristas (AdjacentArista), si valor, un índice para algunas funciones próximas(visitado)

```java
public class nodoVertice<E,T> {
    private nodoVertice<E,T> nextVertice;
    private nodoArista<E,T> adjacentArista;
    private E value;
    private boolean visitado = false;
    //private int acumulado = 99999999;

    public nodoVertice(E value) {
        this.value = value;
    }
    public void setNextVertice(nodoVertice<E,T> nextVertice){this.nextVertice = nextVertice;}
    public nodoVertice<E,T> getNextVertice(){return nextVertice;}
    public void setAdjacentArista(nodoArista<E,T> adjacenteArista){this.adjacentArista = adjacenteArista;}
    public nodoArista<E,T> getAdjacentArista(){return adjacentArista;}
    public E getValue(){return value;}
    public void setValue(E value){this.value = value;}
    public boolean getVisitado(){return visitado;}
    public void setVisitado(boolean value){this.visitado = value;}
    //public int getAcumulado(){return acumulado;}
    //public void setAcumulado(int acumulado){this.acumulado = acumulado;}

}
```

2. **Nodo Arista**: Marca la dirección de la siguiente arista en la lista y del vértice al que apunta, también contiene un valor.

```java
public class nodoArista <E,T>{
        private nodoArista<E,T>  siguiente;
        private nodoVertice<E,T> direccionVertice;
        private T value;

    public nodoArista(T value) {
        this.value = value;
    }
    public void setSiguiente(nodoArista<E,T> siguiente){this.siguiente = siguiente;}
    public nodoArista<E,T> getSiguiente(){return siguiente;}
    public void setDireccionVertice(nodoVertice<E,T> direccion){this.direccionVertice = direccion;}
    public nodoVertice<E,T> getDireccionVertice(){return direccionVertice;}
    public T getValue(){return value;}
    public void setValue(T value){this.value = value;}
}
```

3. **Algunas funciones básicas:** El grafo lista de adyacente contiene el índice a su raíz, el último elemento ingresado y el orden del grafo, en sus funciones mas básicas podemos ver el constructor, getRoot(devuelve la raíz), isEmpty(verifica si esta vacío) y buscar vértice que mediante recursividad viaja entre los vértices y te devuelve el vértice con el valor indicado.

```
public class listaAdyacencia <E,T> {
    private nodoVertice<E,T> root;
    private nodoVertice<E,T> ultimo;
    private int orden = 0;

    public listaAdyacencia(nodoVertice<E,T> root){
        this.root = root;
        ultimo = root;
    }
    public listaAdyacencia(){}

    public nodoVertice<E,T> getRoot(){return root;}

    public boolean isEmpty() {return this.root == null;}

    public nodoVertice<E,T> buscarVertice(E value){
        return buscarVertice(value, root);
    }
    public nodoVertice<E,T> buscarVertice(E value, nodoVertice<E,T> root){
        if(root.getValue()==value){
            return root;
        }
        else if (root.getNextVertice()==null)
            return null;
        else
            return buscarVertice(value, root.getNextVertice());
    }
```

4. **Insert Vértice**: Inserta un nuevo vértice a través del índice del ultimo vértice ingresado

```
public void insertVertice(E value){
    orden = orden+1;
    if (this.isEmpty()){
        root = new nodoVertice<E,T>(value);
        ultimo = root;
    }
    else{
        nodoVertice<E,T> nuevo = new nodoVertice<E,T>(value);
        insertVertice(nuevo);
    }
}
public void insertVertice(nodoVertice<E,T> nuevo){
    ultimo.setNextVertice(nuevo);
    ultimo = nuevo;
}
```

5. **InsertArista:** Inserta la arista nueva entre los dos vértices indicados, almacenándose en la lista de aristas que ambos vértices contienen (También se puede adaptar para dígrafos)

```
public void insertArista(E name1, T valor, E name2){
    insertArista(buscarVertice(name1), valor, buscarVertice(name2));
}
public void insertArista(nodoVertice<E,T> primero, T valor, nodoVertice<E,T> segundo){
    nodoArista<E,T> nueva = new nodoArista<E,T>(valor);
    nodoArista<E,T> nueva2 = new nodoArista<E,T>(valor);
    insertArista(primero, segundo, nueva);
    insertArista(segundo, primero, nueva2);
}
public void insertArista(nodoVertice<E,T> primero, nodoVertice<E,T> segundo, nodoArista<E,T> nueva){
    nueva.setDireccionVertice(segundo);
    if (primero.getAdjacentArista() == null){
        primero.setAdjacentArista(nueva);
    }
    else{
        nueva.setSiguiente(primero.getAdjacentArista());
        primero.setAdjacentArista(nueva);
    }
}
```

6. **Vaciar Visitas e Imprimir**: Usando un sistema de recorrido muy parecido, recorren todo el árbol, imprimiendo los elementos o poniendo en 0 el indicador de visitas de los vértices.

```
public void vaciarVisitas(){
    vaciarVisitas(root, root.getAdjacentArista(), root);
}
public void vaciarVisitas(nodoVertice<E,T> n, nodoArista<E,T> next, nodoVertice<E,T> aux){
    n.setVisitado(false);;
    if(next != null){
        vaciarVisitas(next.getDireccionVertice(),next.getSiguiente(), aux);
    }
    else if (aux.getNextVertice()!=null){
        vaciarVisitas(aux.getNextVertice(),aux.getNextVertice().getAdjacentArista(),aux.getNextVertice());
    }
}
public void imprimir(){
    imprimir(root, root.getAdjacentArista(), root);
}
public void imprimir(nodoVertice<E,T> n, nodoArista<E,T> next, nodoVertice<E,T> aux){
    System.out.print("-->"+n.getValue());
    if(next != null){
        System.out.print("<--"+next.getValue());
        imprimir(next.getDireccionVertice(),next.getSiguiente(), aux);
    }
    else if (aux.getNextVertice()!=null){
        System.out.println("\n");
        imprimir(aux.getNextVertice(),aux.getNextVertice().getAdjacentArista(),aux.getNextVertice());
    }
```

Ejercicio3: Implementar BSF, DFS y Dijkstra con sus respectivos casos de prueba.

Para la utilidad de BFS y DFS se creo un nuevo nodo para hacer una pila o cola dependiendo de la forma en que eran ingresados. Contiene el índice a vértice y el siguiente en la cola, también está la función de desencolar que devuelve el nodo vértice y se elimina de la cola.

```
public class nodoRecorridos<E,T> {
    private nodoVertice<E,T> vertice;
    private nodoRecorridos<E,T> next;


    public nodoRecorridos(nodoVertice<E,T> value) {
        this.vertice = value;
    }

    public nodoVertice<E,T> desencolar(){
        if(next != null){
            nodoVertice<E,T> aux = vertice;
            vertice = next.vertice;
            next = next.next;
            return aux;
        }
        else{
            nodoVertice<E,T> aux = vertice;
            vertice = null;
            return aux;
        }
    }
    public void setVertice(nodoVertice<E,T> vertice){this.vertice = vertice;}
    public nodoVertice<E,T> getVertice(){return vertice;}
    public void setNext(nodoRecorridos<E,T> next){this.next = next;}
    public nodoRecorridos<E,T> getNext(){return next;}

}
```

1. **BFS:**
    a. **Encolar y añadir vértices a cola:** Cumplen la función de agregar los vértices a la cola; con añadir vértices a cola, se ingresan todos los vértices aledaños al seleccionado también.

```
public void encolar(nodoVertice<E,T> nuevo, nodoRecorridos<E,T> base){
    if(base.getNext() == null){
        base.setNext(new nodoRecorridos<E,T>(nuevo));
    }
    else
        encolar(nuevo, base.getNext());
}
public void añadirVerticesACola(nodoVertice<E,T> n, nodoRecorridos<E,T> cola){
    if(n.getAdjacentArista()!=null){
        añadirVerticesACola(n.getAdjacentArista(), cola);
    }
}
public void añadirVerticesACola(nodoArista<E,T> n, nodoRecorridos<E,T> cola){
    if(n.getDireccionVertice().getVisitado()!=true){
        encolar(n.getDireccionVertice(),cola);
        n.getDireccionVertice().setVisitado(true);
    }
    if(n.getSiguiente()!=null){
        añadirVerticesACola(n.getSiguiente(),cola);
    }
}
```

b. **Recorrido BFS:** Recorre el grafo agregando y quitando los elementos a la cola de recorrido que se irán imprimiendo conforme se van desencolando.

```
public void recorridoBFS(){
    nodoRecorridos<E,T> cola = new nodoRecorridos<E,T>(root);
    root.setVisitado(true);
    recorridoBFS(cola);
}
public void recorridoBFS(nodoVertice<E,T> n){
    nodoRecorridos<E,T> cola = new nodoRecorridos<E,T>(n);
    n.setVisitado(true);
    recorridoBFS(cola);
}
public void recorridoBFS(nodoRecorridos<E,T> cola){
    try{
        añadirVerticesACola(cola.getVertice(), cola);
        System.out.println(cola.desencolar().getValue());
        if(cola!=null)
            recorridoBFS(cola);
        }
    catch(Exception e){System.out.println("\nFin");}
}
```

2. **DFS:** En este caso la forma de insertar los elementos convierte a la cola de recorridos en una pila de recorridos.
    a. **Empilar y añadir vértices a cola DFS:** Empilar agrega el vértice a la cola dejando el nuevo vértice en frente de la cola, mientras añadir vértices a cola agrega todos los vértices aledaños al frente de la cola usando “empilar”.

```
public void empilar(nodoVertice<E, T> nuevo, nodoRecorridos<E, T> base) {
    nodoRecorridos<E,T> aux = new nodoRecorridos<E,T>(base.getVertice());
    aux.setNext(base.getNext());
    base.setVertice(nuevo);
    base.setNext(aux);
}

public void añadirVerticesAColaDFS(nodoVertice<E,T> n, nodoRecorridos<E,T> cola){
    if(n.getAdjacentArista()!=null){
        añadirVerticesAColaDFS(n.getAdjacentArista(), cola);
    }
}
public void añadirVerticesAColaDFS(nodoArista<E,T> n, nodoRecorridos<E,T> cola){
    if(n.getDireccionVertice().getVisitado()!=true){
        empilar(n.getDireccionVertice(),cola);
        n.getDireccionVertice().setVisitado(true);
    }
    if(n.getSiguiente()!=null){
        añadirVerticesAColaDFS(n.getSiguiente(),cola);
    }
}
```

b. **Recorrido DFS**: Va recorriendo el grafo agregando y quitando elementos a la pila, y de igual forma que el anterior conforma va sacando los elementos los va mostrando.

```
public void recorridoDFS() {
    nodoRecorridos<E, T> cola = new nodoRecorridos<E, T>(root);
    root.setVisitado(true);
    recorridoDFS(cola,cola.getVertice());
}

public void recorridoDFS(nodoVertice<E, T> n) {
    nodoRecorridos<E, T> cola = new nodoRecorridos<E, T>(n);
    n.setVisitado(true);
    recorridoDFS(cola, cola.getVertice());
}

public void recorridoDFS(nodoRecorridos<E, T> cola, nodoVertice<E,T> vertice) {
    try {
        System.out.println(cola.desencolar().getValue());
        añadirVerticesAColaDFS(vertice, cola);
        if (cola != null)
            recorridoDFS(cola,cola.getVertice());
    } catch (Exception e) {
        System.out.println("\nFin");
    }
}
```

3. **Algoritmo de Dijkstra**: Aunque pueda verse confuso lo que hace es que recorre el grafo y asigna el valor de la arista mas el del vértice anterior solo si el valor del vértice siguiente es menor, viajando entre aristas y vértices hasta recorrer todo el árbol.

```
public void realizarDijstra(nodoVertice<E, T> root) {
    if (root.getAdjacentArista() != null) {
        if (root.getAdjacentArista().getDireccionVertice().getAcumulado() == 0
                || (Integer) root.getAdjacentArista().getDireccionVertice().getAcumulado() > root.getAcumulado()
                        + (Integer) (root.getAdjacentArista().getValue())) {

            root.getAdjacentArista().getDireccionVertice()
                    .setAcumulado(root.getAcumulado() + (Integer) root.getAdjacentArista().getValue());

            realizarDijstra(root.getAdjacentArista().getDireccionVertice());
        }
        realizarDijstra(root.getAdjacentArista());
    }
}

public void realizarDijstra(nodoArista<E, T> root) {
    if (root.getSiguiente() != null) {
        if (root.getSiguiente().getDireccionVertice().getAcumulado() == 0
                || (Integer) root.getSiguiente().getDireccionVertice().getAcumulado() > root.getDireccionVertice(
                        .getAcumulado() + (Integer) (root.getSiguiente().getValue())) {

            root.getSiguiente().getDireccionVertice().setAcumulado(
                    root.getDireccionVertice().getAcumulado() + (Integer) root.getSiguiente().getValue());

            realizarDijstra(root.getSiguiente().getDireccionVertice());
        }
        realizarDijstra(root.getSiguiente());
    }
}
```

Posteriormente imprime los vértices con sus respectivos nuevos pesos:

```java
public void imprimirDijstra(){
    root.setAcumulado(0);
    imprimirDijstra(root, root.getAdjacentArista(), root);
}
public void imprimirDijstra(nodoVertice<E,T> n, nodoArista<E,T> next, nodoVertice<E,T> aux){
    System.out.print("--"+n.getValue()+"("+n.getAcumulado()+")");
    if (aux.getNextVertice()!=null){
        System.out.println("\n");
        imprimirDijstra(aux.getNextVertice(),aux.getNextVertice().getAdjacentArista(),aux.getNextVertice());
    }

}
```

Ejercicio 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 coops crops drops drips grips gripe grape graph

b) Mostrar la lista de adyacencia del grafo.

Implementación:

Primero compara los elementos para ver si son aledaños o no.

```java
public static int compararStrings(String n, String m){
    int aux = 0;
    for(int i = 0; i<5; i++){
        if(n.charAt(i)!=m.charAt(i))
            aux+=1;
    }
    return aux;
}
```

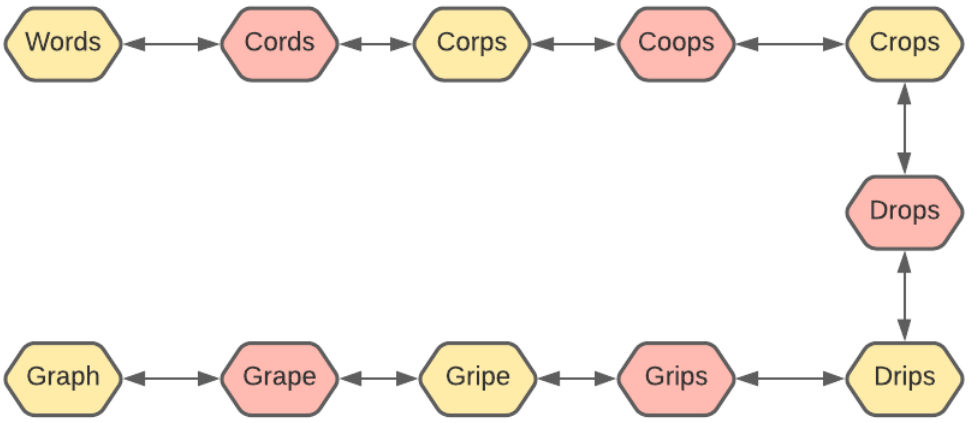
Creamos un grafo lista de Adyacencia, añadimos todos los vértices y las aristas conforme la función comparar strings nos indique si son de vértices aledaños y lo imprimimos.

```java
public static void main(String[] args) {
    String[] arr = new String[11];
    arr[0]="words";
    arr[1]="cords";
    arr[2]="corps";
    arr[3]="coops";
    arr[4]="crops";
    arr[5]="drops";
    arr[6]="drips";
    arr[7]="grips";
    arr[8]="gripe";
    arr[9]="grape";
    arr[10]="graph";

    listaAdyacencia<String,Integer> grafo = new listaAdyacencia<String,Integer>();
    for(int i = 0; i<arr.length;i++){
        grafo.insertVertice(arr[i]);
    }
    for(int i = 0; i<arr.length;i++){
        for(int j = i+1; j<arr.length;j++){
            if(compararStrings(arr[i], arr[j])==1){
                grafo.insertArista(arr[i], 0, arr[j]);
            }
        }
    }
    grafo.imprimir();
}
```

Dándonos de resultado la lista de adyacencia. (Los la distancia entre vértices está en 0)

```
-->words<--0-->cords

-->cords<--0-->corps<--0-->words

-->corps<--0-->coops<--0-->cords

-->coops<--0-->crops<--0-->corps

-->crops<--0-->drops<--0-->coops

-->drops<--0-->drips<--0-->crops

-->drips<--0-->grips<--0-->drops

-->grips<--0-->gripe<--0-->drips

-->gripe<--0-->grape<--0-->grips

-->grape<--0-->graph<--0-->gripe

-->graph<--0-->grape
```

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



2. Muestre la lista de adyacencia.

```
-->words<--0-->cords

-->cords<--0-->corps<--0-->words

-->corps<--0-->coops<--0-->cords

-->coops<--0-->crops<--0-->corps

-->crops<--0-->drops<--0-->coops

-->drops<--0-->drips<--0-->crops

-->drips<--0-->grips<--0-->drops

-->grips<--0-->gripe<--0-->drips

-->gripe<--0-->grape<--0-->grips

-->grape<--0-->graph<--0-->gripe

-->graph<--0-->grape
```

Ejercicio 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.

Para poder implementar esta función lo primero que hice fue convertir los grafos a un arreglo de arreglos, recorriendo el grafo y rellenando los arreglos.

```
public String[][] convertirGrafoaArreglo(listaAdyacencia<E,T> n){
    String[][] lista= new String[orden][orden];
    convertirGrafoaArreglo(n.root, lista, 0, 0);
    return lista;
}
public void convertirGrafoaArreglo(nodoVertice<E,T> base, String[][] lista, int contador1, int contador2){
    lista[contador1][contador2] = (String)base.getValue();
    if(base.getAdjacentArista()!=null){
        convertirGrafoaArreglo(base.getAdjacentArista(), lista, contador1, contador2+1);
    }
    if(base.getNextVertice()!=null){
        convertirGrafoaArreglo(base.getNextVertice(), lista, contador1+1, contador2);
    }
}
public void convertirGrafoaArreglo(nodoArista<E,T> base, String[][] lista, int contador1, int contador2){
    lista[contador1][contador2] = (String)base.getDireccionVertice().getValue();
    if(base.getSiguiente()!=null){
        convertirGrafoaArreglo(base.getSiguiente(), lista, contador1, contador2+1);
    }
}
```

A continuación, mediante algunos for comparamos los arreglos de la siguiente forma:

1. Buscamos el primer vértice en común.
2. Una ves encontrado el vértice vemos si todas las aristas del vértice pretendiente a subgrafo están contenidas en el vértice del grafo mayor.
3. Si se cumple añadimos 1 nuestro contador.
4. Repetimos la forma con todos los vértices con el mismo valor.

```
public int compararArreglosGrafo(String[] arr1, String[] arr2){
    int contador = 0;
    int contador2 = 0;
    for(int i = 0; i<arr1.length;i++){
        if(arr2[i]!= null){
            contador2++;
        }
    }
    for(int i = 1; i<arr1.length;i++){
        for(int j = 1 ; j < arr2.length ; j++){
            if(arr1[i]!=null && arr2[j]!=null){
                if(arr1[i].equals(arr2[j]))
                    contador++;
            }
        }
    }
    return contador-contador2;
}
public boolean compararArreglosGrafo(String[][] arreglo1, String[][] arreglo2){
    int contador = 0;
    for(int i = 0; i<arreglo1.length;i++){
        for(int j = 0 ; j < arreglo2.length ; j++){
            if(arreglo1[i][0]!=null && arreglo2[j][0]!=null){
                contador=contador+compararArreglosGrafo(arreglo1[i], arreglo2[j]);
            }
        }
    }
    return contador==0;
}
```

Cuestionario:

1. **¿Cuántas variantes del algoritmo de Dijkstra hay y cuál es la diferencia entre ellas?**
    a. Algoritmo de búsqueda de costos uniformes: es un algoritmo de búsqueda no informada utilizado para recorrer sobre grafos el camino de costo mínimo entre un nodo raíz y un nodo destino. La búsqueda comienza por el nodo raíz y continúa visitando el siguiente nodo que tiene menor costo total desde la raíz. Los nodos son visitados de esta manera hasta que el nodo destino es alcanzado.
    b. Best first search: Una búsqueda con una heurística que intenta predecir qué tan cerca está el final de una ruta de una solución, de modo que las rutas que se consideran más cercanas a una solución se amplían primero.
    c. A*: Es un algoritmo de búsqueda inteligente o informada que busca el camino más corto desde un estado inicial al estado meta a través de un espacio de problema, usando una heurística óptima. Como ignora los pasos más cortos en algunos casos rinde una solución subóptima.

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

    a. Algoritmo de Dijkstra: es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.
    b. Algoritmo de Bellman Ford: Genera el camino más corto en un grafo dirigido ponderado (en el que el peso de alguna de las aristas puede ser negativo). El algoritmo de Dijkstra resuelve este mismo problema en un tiempo menor, pero requiere que los pesos de las aristas no sean negativos, salvo que el grafo sea dirigido y sin ciclos.
    c. Algoritmo de Búsqueda A*: Es un algoritmo de búsqueda inteligente o informada que busca el camino más corto desde un estado inicial al estado meta a través de un espacio de problema, usando una heurística óptima. Como ignora los pasos más cortos en algunos casos rinde una solución subóptima.
    d. Algoritmo de Floyd Warshall: El algoritmo de Floyd-Warshall compara todos los posibles caminos a través del grafo entre cada par de vértices
    e. Algoritmo de Johnson: Cumple la función de encontrar el camino más corto entre todos los pares de vértices de un grafo dirigido disperso. Permite que las aristas tengan pesos negativos, si bien no permite ciclos de peso negativo