Laboratorio 2

Nombre: Juan Jiménez

Código del programa:

```
package laboratorio2_grafos_u2;

import java.util.HashMap;

public class Grafo {

 public HashMap<String, NodoGrafo> nodos; // Nodos del grafo
 // Constructor que inicializa un grafo vacío
 public Grafo() {
 nodos = new HashMap<String, NodoGrafo>();
 }
 // Método que obtiene o crea un nodo en el grafo dado su nombre
 public NodoGrafo obtenerOcrearNodo(String nombre) {
 NodoGrafo nodo = nodos.get(nombre);
 if (nodo == null) {
 nodo = new NodoGrafo(nombre);
 nodos.put(nombre, nodo);
 }
 return nodo;
 }
 // Método que añade una arista al grafo entre los nodos de inicio y fin
 public void agregarArista(String inicio, String fin) {
 NodoGrafo nodoInicio = obtenerOcrearNodo(inicio);
```

```
 NodoGrafo nodoFin = obtenerOcrearNodo(fin);
 nodoInicio.agregarVecino(nodoFin);
 }

}
package laboratorio2_grafos_u2;

public enum EstadoNodoGrafo {
 NoVisitado,
 Visitado,
 VisitadoParcialmente
}
package laboratorio2_grafos_u2;

 // Clase que realiza una búsqueda en profundidad (DFS) en un grafo
public class BusquedaEnProfundidad {
 // Método público que inicia la búsqueda en profundidad en el grafo
 public static boolean busquedaEnProfundidad(Grafo grafo, String objetivo) {
 // Recorremos todos los nodos del grafo
 for (NodoGrafo nodo : grafo.nodos.values()) {
 // Si el nodo actual contiene el objetivo, retorna verdadero
 if (dfsRecursivoAuxiliar(nodo, objetivo)) {
 return true;
 }
 }
 // Si no se encuentra el objetivo en el grafo, retorna falso
 return false;
 }
 // Método privado recursivo para realizar la búsqueda en profundidad
```

```
 private static boolean dfsRecursivoAuxiliar(NodoGrafo nodoActual, String objetivo) {
 // Si el valor del nodo actual es igual al objetivo, retorna verdadero
 if (nodoActual.valor.equals(objetivo)) {
 return true;
 }
 // Marcamos el nodo actual como visitado
 nodoActual.estado = EstadoNodoGrafo.Visitado;
 // Recorremos los nodos vecinos del nodo actual
 for (NodoGrafo nodoVecino : nodoActual.adyacentes.values()) {
 // Si el nodo vecino no ha sido visitado, realizamos la búsqueda en profundidad
recursivamente
 if (nodoVecino.estado != EstadoNodoGrafo.Visitado) {
 if (dfsRecursivoAuxiliar(nodoVecino, objetivo)) {
 return true;
 }
 }
 }
 // Si no se encuentra el objetivo en los nodos vecinos, retorna falso
 return false;
 }
}
package laboratorio2_grafos_u2;

// Clase que realiza una búsqueda en amplitud (BFS) en un grafo

import java.util.LinkedList;
import java.util.Queue;

public class BusquedaEnAmplitud {
 // Método público que inicia la búsqueda en amplitud en el grafo
```

```
public static boolean busquedaEnAmplitud(Grafo grafo, String objetivo) {
// Recorremos todos los nodos del grafo
for (NodoGrafo nodo : grafo.nodos.values()) {
// Si el nodo actual contiene el objetivo, retorna verdadero
if (bfsAuxiliarIndividual(nodo, objetivo)) {
return true;
}
}
// Si no se encuentra el objetivo en el grafo, retorna falso
return false;
}
// Método privado que realiza la búsqueda en amplitud para un nodo específico
private static boolean bfsAuxiliarIndividual(NodoGrafo nodo, String objetivo) {
// Creamos una cola para almacenar los nodos a visitar
Queue<NodoGrafo> cola = new LinkedList<>();
cola.add(nodo);
// Mientras la cola no esté vacía, continuamos la búsqueda
while (!cola.isEmpty()) {
// Obtenemos el nodo actual de la cola
NodoGrafo nodoActual = cola.remove();
// Si el valor del nodo actual es igual al objetivo, retorna verdadero
if (nodoActual.valor.equals(objetivo)) return true;
// Marcamos el nodo actual como visitado
nodoActual.estado = EstadoNodoGrafo.Visitado;
// Recorremos los nodos adyacentes del nodo actual
for (NodoGrafo adyacente : nodoActual.adyacentes.values()) {
// Si el nodo adyacente no ha sido visitado, lo agregamos a la cola
if (adyacente.estado == EstadoNodoGrafo.NoVisitado) {
cola.add(adyacente);
```

```
 }
 }
 }
 // Si no se encuentra el objetivo en los nodos adyacentes, retorna falso
 return false;
 }
}
package laboratorio2_grafos_u2;

 // Clase que representa un nodo en un grafo
import java.util.HashMap;



public class NodoGrafo {
 public String valor; // El valor que almacena el nodo del grafo
 public EstadoNodoGrafo estado;
 public HashMap<String, NodoGrafo> adyacentes; // Nodos adyacentes a este nodo en el grafo
 // Constructor que inicializa un nodo con un valor dado
 public NodoGrafo(String valor) {
 this.valor = valor;
 this.adyacentes = new HashMap<String, NodoGrafo>();
 this.estado = EstadoNodoGrafo.NoVisitado;
 }

 // Constructor que inicializa un nodo con un valor y nodos adyacentes dados
 public NodoGrafo(String valor, HashMap<String, NodoGrafo> adyacentes) {
 this.valor = valor;
 this.adyacentes = adyacentes;
 }
```

```
// Método para agregar un nodo vecino al nodo actual en el grafo
public void agregarVecino(NodoGrafo nodo) {
if (!adyacentes.containsKey(nodo.valor)) {
adyacentes.put(nodo.valor, nodo);
}
}
// Método hashCode para generar el código hash del nodo
@Override
public int hashCode() {
final int primo = 31;
int resultado = 1;
resultado = primo * resultado + ((valor == null) ? 0 : valor.hashCode());
return resultado;
}
// Método equals para comparar si dos nodos son iguales
@Override
public boolean equals(Object obj) {
if (this == obj) return true;
if (obj == null) return false;
if (getClass() != obj.getClass()) return false;
NodoGrafo otro = (NodoGrafo) obj;
if (valor == null) {
if (otro.valor != null) return false;
} else if (!valor.equals(otro.valor)) return false;
return true;
}
}
```

Ejecución del Programa:

El programa funciona solo que falta implementar el menú para realizar cada función del programa debido a que estuve arreglando unos errores que salían al momento de copiar el código, además de estar arreglando el programa que se estaba presentando para la conexión de mongo