
	<p>UNIVERSIDAD NACIONAL DE SAN AGUSTÍN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p>Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 1</p>

## INFORME DE LABORATORIO

INFORMACIÓN BÁSICA					
ASIGNATURA:	Estructura de Datos y Algoritmos				
TÍTULO DE LA PRÁCTICA:	Grafos				
NÚMERO DE PRÁCTICA:	08	AÑO LECTIVO:	2023 B	NRO. SEMESTRE:	III
FECHA DE PRESENTACIÓN	28/12/23	HORA DE PRESENTACIÓN			
INTEGRANTE (s): <ul style="list-style-type: none"> <li>Luis Fernando Florez Bailon</li> </ul>				NOTA:	
DOCENTE(s): Karen Melissa Quispe Vegaray					

SOLUCIÓN Y RESULTADOS
<p><b>I. ENLACE GITHUB</b></p> <p><a href="https://github.com/LuisFFB/EDA_Lab_8/tree/main">https://github.com/LuisFFB/EDA_Lab_8/tree/main</a></p>
<p><b>II. SOLUCIÓN DE EJERCICIOS/PROBLEMAS</b></p> <p>2. Implementar el código de Grafo cuya representación sea realizada mediante LISTA DE ADYACENCIA. (3 puntos)</p> <pre> class Node {     int destination;     int weight;      public Node(int destination, int weight) {         this.destination = destination;         this.weight = weight;     } } </pre>

```
import java.util.*;

class Graph {
    private final int V; // Número de vértices
    private final List<List<Node>> adjacencyList;

    public Graph(int V) {
        this.V = V;
        adjacencyList = new ArrayList<>(V);

        for (int i = 0; i < V; i++) {
            adjacencyList.add(new LinkedList<>());
        }
    }

    // Método para agregar una arista al grafo
    public void addEdge(int source, int destination, int weight) {
        Node newNode = new Node(destination, weight);
        adjacencyList.get(source).add(newNode);
    }

    // Método para imprimir el grafo
    public void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print("Nodo " + i + ": ");
            for (Node node : adjacencyList.get(i)) {
                System.out.print("(" + node.destination + ", " + node.weight + ") ");
            }
            System.out.println();
        }
    }
}
```

```
public class ListaDeAdyacencia {
    public static void main(String[] args) {
        // Crear un grafo con 5 vértices
        Graph graph = new Graph(5);

        // Agregar aristas al grafo
        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 4, 5);
        graph.addEdge(1, 2, 4);
        graph.addEdge(2, 3, 9);
        graph.addEdge(3, 4, 7);

        // Imprimir el grafo
        graph.printGraph();
    }
}
```

```
Nodo 0: (1, 2) (4, 5)
Nodo 1: (2, 4)
Nodo 2: (3, 9)
Nodo 3: (4, 7)
Nodo 4:
```

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

```
class Node {  
    int destination;  
    int weight;  
  
    public Node(int destination, int weight) {  
        this.destination = destination;  
        this.weight = weight;  
    }  
}
```

```
class Graph {  
    private final int V; // Número de vértices  
    private final List<List<Node>> adjacencyList;  
  
    public Graph(int V) {  
        this.V = V;  
        adjacencyList = new ArrayList<>(V);  
  
        for (int i = 0; i < V; i++) {  
            adjacencyList.add(new LinkedList<>());  
        }  
    }  
  
    // Método para agregar una arista al grafo  
    public void addEdge(int source, int destination, int weight) {  
        Node newNode = new Node(destination, weight);  
        adjacencyList.get(source).add(newNode);  
    }  
}
```

```
// Método para realizar Breadth-First Search (BFS)
public void bfs(int start) {
    boolean[] visited = new boolean[V];
    Queue<Integer> queue = new LinkedList<>();

    visited[start] = true;
    queue.add(start);

    System.out.println("BFS starting from node " + start + ":");
    while (!queue.isEmpty()) {
        int current = queue.poll();
        System.out.print(current + " ");

        for (Node neighbor : adjacencyList.get(current)) {
            if (!visited[neighbor.destination]) {
                visited[neighbor.destination] = true;
                queue.add(neighbor.destination);
            }
        }
        System.out.println();
    }
}

// Método para realizar Depth-First Search (DFS)
public void dfs(int start) {
    boolean[] visited = new boolean[V];
    dfsHelper(start, visited);
    System.out.println();
}

private void dfsHelper(int current, boolean[] visited) {
    visited[current] = true;
    System.out.print(current + " ");

    for (Node neighbor : adjacencyList.get(current)) {
        if (!visited[neighbor.destination]) {
            dfsHelper(neighbor.destination, visited);
        }
    }
}
```

```
// Método para realizar Dijkstra
public void dijkstra(int start) {
    PriorityQueue<Node> minHeap = new PriorityQueue<>(Comparator.comparingInt(node -> node.weight));
    int[] distances = new int[V];
    Arrays.fill(distances, Integer.MAX_VALUE);

    minHeap.add(new Node(start, 0));
    distances[start] = 0;

    System.out.println("Dijkstra starting from node " + start + ":");
    while (!minHeap.isEmpty()) {
        Node current = minHeap.poll();

        if (current.weight > distances[current.destination]) {
            continue; // Se ignora el nodo si se encuentra un camino más corto
        }

        System.out.print(current.destination + " ");

        for (Node neighbor : adjacencyList.get(current.destination)) {
            int newDistance = distances[current.destination] + neighbor.weight;
            if (newDistance < distances[neighbor.destination]) {
                distances[neighbor.destination] = newDistance;
                minHeap.add(new Node(neighbor.destination, newDistance));
            }
        }
    }
    System.out.println();
}
```

```
public class GraphAlgorithmsTest {
    public static void main(String[] args) {
        Graph graph = new Graph(6);

        // Agregar aristas al grafo
        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 2, 4);
        graph.addEdge(1, 3, 1);
        graph.addEdge(2, 4, 3);
        graph.addEdge(3, 5, 2);
        graph.addEdge(4, 5, 5);

        // Prueba BFS
        graph.bfs(0);

        // Prueba DFS
        graph.dfs(0);

        // Prueba Dijkstra
        graph.dijkstra(0);
    }
}
```

```
BFS starting from node 0:  
0 1 2 3 4 5  
0 1 3 5 2 4  
Dijkstra starting from node 0:  
0 1 3 2 5 4
```

**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 **words** 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

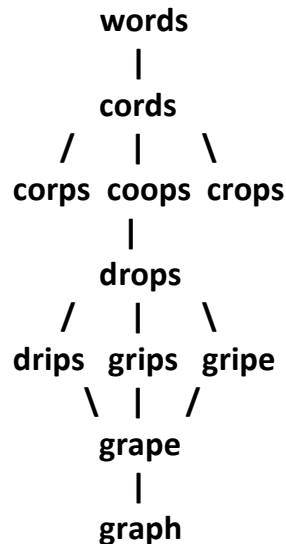
b) Mostrar la lista de adyacencia del grafo.

**Identificar vértices y establecer conexiones**

Las palabras dadas son: words, cords, corps, coops, crops, drops, drips, grips, gripe, grape, graph.

words - cords  
cords - corps, coops, crops  
corps - cords, coops, crops  
coops - cords, corps, crops  
crops - cords, corps, coops, drops  
drops - crops, drips, grips  
drips - drops, grips, gripe  
grips - drops, drips, gripe, grape  
gripe - drips, grips, grape, graph  
grape - grips, gripe, graph  
graph - gripe, grape

## Dibujar el grafo



### Lista de Adyacencia

words: [cords]

cords: [words, corps, coops, crops]

corps: [cords, coops, crops]

coops: [cords, corps, crops]

crops: [cords, corps, coops, drops]

drops: [crops, drips, grips]

drips: [drops, grips, gripe]

grips: [drops, drips, gripe, grape]

gripe: [drips, grips, grape, graph]

grape: [grips, gripe, graph]

graph: [gripe, grape]

Esto representa el grafo de palabras según la definición dada en el ejercicio. Las listas de adyacencia muestran las palabras adyacentes para cada palabra en el grafo.

5. Realizar un metodo en la clase Grafo. Este metodo permitira saber si un grafo esta incluido en otro. Los parametros de entrada son 2 grafos y la salida del metodo es true si hay inclusion y false el caso contrario. (4 puntos)

```
import java.util.*;

class Graph {
    private final int V; // Número de vértices
    private final List<List<Integer>> adjacencyList;

    public Graph(int V) {
        this.V = V;
        adjacencyList = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addEdge(int source, int destination) {
        adjacencyList.get(source).add(destination);
    }

    public boolean isGraphIncluded(Graph otherGraph) {
        if (this.V != otherGraph.V) {
            return false; // Los grafos tienen diferentes números de vértices
        }

        // Verificar si todos los vértices del primer grafo están presentes en el segundo grafo
        for (int i = 0; i < this.V; i++) {
            if (!otherGraph.adjacencyList.get(i).containsAll(this.adjacencyList.get(i))) {
                return false;
            }
        }

        return true;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Graph graph1 = new Graph(5);
        graph1.addEdge(0, 1);
        graph1.addEdge(1, 2);
        graph1.addEdge(2, 3);
        graph1.addEdge(3, 4);

        Graph graph2 = new Graph(5);
        graph2.addEdge(0, 1);
        graph2.addEdge(1, 2);
        graph2.addEdge(2, 3);
        graph2.addEdge(3, 4);

        Graph graph3 = new Graph(5);
        graph3.addEdge(0, 1);
        graph3.addEdge(1, 2);

        // Prueba de inclusión
        System.out.println("¿Graph1 está incluido en Graph2? " + graph1.isGraphIncluded(graph2));
        System.out.println("¿Graph1 está incluido en Graph3? " + graph1.isGraphIncluded(graph3));
    }
}
```

```
¿Graph1 está incluido en Graph2? true
¿Graph1 está incluido en Graph3? false
```



## **II. CUESTIONARIO**

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

#### **1. Dijkstra con Colas de Prioridad:**

- La implementación básica del algoritmo de Dijkstra utiliza una cola no ordenada para seleccionar el nodo con la distancia mínima en cada iteración. Una mejora común es utilizar una cola de prioridad para mejorar la eficiencia de la selección del nodo con la distancia mínima. Esto reduce la complejidad temporal de  $O(V^2)$  a  $O((V + E) * \log(V))$ , donde  $V$  es el número de nodos y  $E$  es el número de aristas.

#### **2. Dijkstra con Listas de Adyacencia y Colas de Prioridad:**

- Al trabajar con grafos grandes y dispersos, donde la mayoría de los nodos no están conectados entre sí, es más eficiente utilizar listas de adyacencia en lugar de una matriz de adyacencia. Combinar listas de adyacencia con colas de prioridad mejora aún más la eficiencia.

#### **3. Dijkstra con Fibonacci Heaps:**

- Las colas de prioridad basadas en estructuras de datos llamadas montículos de Fibonacci pueden mejorar la eficiencia del algoritmo de Dijkstra, especialmente en grafos densos. Esta variante reduce aún más la complejidad temporal en ciertos casos.

#### **4. Dijkstra Bidireccional:**

- En lugar de expandir desde un solo nodo, el algoritmo bidireccional ejecuta Dijkstra desde el nodo de inicio y el nodo de destino simultáneamente. Se detiene cuando los dos frentes se encuentran. Esta variante puede reducir significativamente el tiempo de ejecución en ciertos casos.

#### **5. Dijkstra con A (A estrella):\***

- $A^*$  es un algoritmo de búsqueda informada que combina Dijkstra con una heurística que estima el costo restante hasta el objetivo. La variante  $A^*$  puede ser más eficiente en términos de tiempo de ejecución en algunos casos, pero requiere una heurística válida y admisible.

#### **6. Dijkstra con Contracción Jerárquica (Hub Labels):**

- Esta variante utiliza la contracción jerárquica para reducir la complejidad del grafo antes de aplicar Dijkstra. Se crea un conjunto de "etiquetas" para nodos agrupados, lo que acelera la búsqueda en grafos grandes.

#### **7. Dijkstra con Landmark (Landmark Dijkstra):**

- Selecciona ciertos nodos como "landmarks" y precalcula las distancias más cortas entre estos nodos y todos los demás nodos. Luego, utiliza esta información para acelerar la búsqueda de caminos más cortos.

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

**Dijkstra:****1. Similitudes:**

- Encuentra el camino más corto desde un nodo de origen a todos los demás nodos.
- Se aplica a grafos dirigidos o no dirigidos con pesos no negativos.

**2. Diferencias:**

- Utiliza una cola de prioridad para seleccionar el nodo de menor distancia en cada paso.
- No maneja aristas con pesos negativos.

**3. Casos de Uso:**

- Ideal para grafos con pesos no negativos.
- Eficiente en grafos dispersos.

**Bellman-Ford:****1. Similitudes:**

- Encuentra el camino más corto desde un nodo de origen a todos los demás nodos.
- Maneja grafos con pesos negativos, pero detecta ciclos negativos.

**2. Diferencias:**

- No requiere pesos no negativos.
- Utiliza relajación de aristas iterativa.

**3. Casos de Uso:**

- Útil cuando hay aristas con pesos negativos, pero debe evitarse si hay ciclos negativos.

**Floyd-Warshall:****1. Similitudes:**

- Encuentra los caminos más cortos entre todos los pares de nodos.

**2. Diferencias:**


- Funciona con grafos dirigidos o no dirigidos, y puede manejar pesos negativos.
- Utiliza una matriz de distancias y actualizaciones iterativas.

**3. Casos de Uso:**

- Adecuado para grafos densos y pequeños debido a su complejidad de tiempo  $O(V^3)$ .
- Útil cuando se necesitan todos los caminos mínimos entre todos los pares de nodos.

**IV. CONCLUSIONES**

-

	<p>UNIVERSIDAD NACIONAL DE SAN AGUSTÍN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p>Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 11</p>

## REFERENCIAS Y BIBLIOGRAFÍA

- Weiss M., Data Structures & Problem Solving Using Java, 2010, Addison-Wesley.
- Escuela de Pedagogía en Educación Matemática, Marcelino Álvarez, et.al.,  
[http://repobib.ubiobio.cl/jspui/bitstream/123456789/1953/3/Alvarez\\_Nunez\\_Marcelino.pdf](http://repobib.ubiobio.cl/jspui/bitstream/123456789/1953/3/Alvarez_Nunez_Marcelino.pdf)
- <http://www.oia.unsam.edu.ar/wp-content/uploads/2017/11/dijkstra-prim.pdf>