UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

Curso: Laboratorio de Estructura de Datos y Algoritmos

Trabajo: Informe Laboratorio 9 Grafos

Estudiante: Erick Edwin Anco Huaracha

Docente: Edith Pamela Rivero Tupac de Lozano

Arequipa - Perú

2021

I. EJERCICIOS PROPUESTOS

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

   Enlace repositorio creado: https://github.com/EAncoH/EDA_Grafos.git

2. **Implementar el cogido de Grafo cuya representación sea realizada mediante LISTA DE ADYACENCIA.**

```
private Vertice[] nodes;

	private HashMap<Vertice, HashSet<Vertice>> graph;
	private HashMap<Vertice, HashSet<Arista>> incidencia;
	private final int numeroVertices;
	private int numeroAristas;
	private Boolean weighted;


	public Grafo(int numVertices) {
		this.graph = new HashMap<Vertice, HashSet<Vertice>>();
		this.incidencia = new HashMap<Vertice, HashSet<Arista>>();
		this.numeroVertices = numVertices;
		this.nodes = new Vertice[numVertices];
		for (int i = 0; i < numVertices; i++) {
			Vertice n = new Vertice(i);
			this.nodes[i] = n;
			this.graph.put(n, new HashSet<Vertice>());
			this.incidencia.put(n, new HashSet<Arista>());
		}
		this.weighted = false;
	}


	public Grafo(int numVertices, String modelo) {
		this.graph = new HashMap<Vertice, HashSet<Vertice>>();
		this.incidencia = new HashMap<Vertice, HashSet<Arista>>();
		this.numeroVertices = numVertices;
		this.nodes = new Vertice[numVertices];
		Random coorX = new Random();
		Random coorY = new Random();
		if (modelo == "geo-uniforme") {
			for (int i = 0; i < numVertices; i++) {
				Vertice n = new Vertice(i, coorX.nextDouble(),
coorY.nextDouble());
				this.nodes[i] = n;
				this.graph.put(n, new HashSet<Vertice>());
				this.incidencia.put(n, new HashSet<Arista>());
			}
		}
		this.weighted = false;
	}

	public int gradoVertice(int i) {
		Vertice n1 = this.getNode(i);
```

```
        return this.graph.get(n1).size();
}

public void conectarVertices(int i, int j) {
        Vertice n1 = this.getNode(i);
        Vertice n2 = this.getNode(j);
        HashSet<Vertice> aristas1 = this.getEdges(i);
        HashSet<Vertice> aristas2 = this.getEdges(j);

        aristas1.add(n2);
        aristas2.add(n1);
        this.numeroAristas +=1;
}

private Boolean existeConexion(int i, int j) {
        Vertice n1 = this.getNode(i);
        Vertice n2 = this.getNode(j);
        HashSet<Vertice> aristas1 = this.getEdges(i);
        HashSet<Vertice> aristas2 = this.getEdges(j);
        if (aristas1.contains(n2) || aristas2.contains(n1)) {
                return true;
        }
        else{
                return false;
        }
}

private double distanciaVertices(Vertice n1, Vertice n2) {
        return Math.sqrt(Math.pow((n1.getX() - n2.getX()), 2)
                        + Math.pow((n1.getY() - n2.getY()), 2));
}

public int getNumNodes() {return numeroVertices;}

public int getNumEdges() {return numeroAristas;}

public Vertice getNode(int i) {return this.nodes[i];}

public Boolean getWeightedFlag() {return this.weighted;}

public HashSet<Vertice> getEdges(int i) {
        Vertice n = this.getNode(i);
        return this.graph.get(n);
}

public HashSet<Arista> getWeightedEdges(int i) {
        Vertice n = this.getNode(i);
        return this.incidencia.get(n);
}

public void setWeighted() {this.weighted = true;}

public void setIncidencia(int i, HashSet<Arista> aristasPeso) {
        this.incidencia.put(this.getNode(i), aristasPeso);}

public void setAristaPeso(int i, int j, double peso) {
```

```
            if (!this.existeConexion(i, j)) this.conectarVertices(i, j);
            Arista aristaNuevaij = new Arista(i, j, peso);
            Arista aristaNuevaji = new Arista(j, i, peso);
            HashSet<Arista> aristasNodoi = this.getWeightedEdges(i);
            HashSet<Arista> aristasNodoj = this.getWeightedEdges(j);
            aristasNodoi.add(aristaNuevaij);
            aristasNodoj.add(aristaNuevaji);
            this.setIncidencia(i, aristasNodoi);
            this.setIncidencia(j, aristasNodoj);
            if (!this.getWeightedFlag()) this.setWeighted();
      }

      public String toString() {
            String salida;
            if (this.getWeightedFlag()) {
                  salida ="graph {\n";
                  for (int i = 0; i < this.getNumNodes(); i++) {
                        salida += this.getNode(i).getName() + " [label=\""
                                    + this.getNode(i).getName() + " ("+
this.getNode(i).getDistance()
                                    + ")\"];\n";
                  }
                  for (int i = 0; i < this.getNumNodes(); i++) {
                        HashSet<Arista> aristas = this.getWeightedEdges(i);
                        for (Arista e : aristas) {
                              salida += e.getNode1() + " -- " +
e.getNode2()
                              + " [weight=" + e.getWeight()+"" + "
label="+e.getWeight()+""
                              + "];\n";
                        }
                  }
                  salida += "}\n";
            }
            else {
                  salida ="graph {\n";
                  for (int i = 0; i < this.getNumNodes(); i++) {
                        salida += this.getNode(i).getName() + ";\n";
                  }
                  for (int i = 0; i < this.getNumNodes(); i++) {
                        HashSet<Vertice> aristas = this.getEdges(i);
                        for (Vertice n : aristas) {
                              salida += this.getNode(i).getName() + " -- "
+ n.getName() + ";\n";
                        }
                  }
                  salida += "}\n";
            }
            return salida;
      }
```

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

DFS

```
public Grafo DFS_R(int s) {
		Grafo arbol = new Grafo(this.getNumNodes());
		Boolean[] discovered = new Boolean[this.getNumNodes()];
		for (int i = 0; i < this.getNumNodes(); i++) {
			discovered[i] = false;
		}
		recursivoDFS(s, discovered, arbol);
		return arbol;
	}

	private void recursivoDFS(int u, Boolean[] discovered, Grafo arbol) {
		discovered[u] = true;
		HashSet<Vertice> aristas = this.getEdges(u);
		for (Vertice n : aristas) {
			if (!discovered[n.getIndex()]) {
				arbol.conectarVertices(u, n.getIndex());
				recursivoDFS(n.getIndex(), discovered, arbol);
			}
		}
	}
```

BFS

```
public Grafo BFS(int s) {
		Grafo arbol = new Grafo(this.getNumNodes());
		Boolean[] discovered = new Boolean[this.getNumNodes()];
		PriorityQueue<Integer> L = new PriorityQueue<Integer>();
		discovered[s] = true;
		for (int i = 0; i < this.getNumNodes(); i++) {
			if (i != s) {
				discovered[i] = false;
			}
		}
		L.add(s);
		while (L.peek() != null) {
			int u = L.poll();
			HashSet<Vertice> aristas = this.getEdges(u);
			for (Vertice n : aristas) {
				if(!discovered[n.getIndex()]) {
					arbol.conectarVertices(u, n.getIndex());
					discovered[n.getIndex()] = true;
					L.add(n.getIndex());
				}
			}
		}
		return arbol;
	}
```

Dijkstra

```
public Grafo Dijkstra(int s) {
		Grafo arbol = new Grafo(this.getNumNodes());
		double inf = Double.POSITIVE_INFINITY;
		Integer[] padres = new Integer[arbol.getNumNodes()];
		for (int i = 0; i < arbol.getNumNodes(); i++) {
```

```
                this.getNode(i).setDistance(inf);
                padres[i] = null;
            }
            this.getNode(s).setDistance(0.0);
            padres[s] = s;
            PriorityQueue<Vertice> distPQ = new
PriorityQueue<>(vertexDistanceComp);
            for (int i = 0; i < this.getNumNodes(); i++) {
                distPQ.add(this.getNode(i));
            }
            while (distPQ.peek() != null) {
                Vertice u = distPQ.poll();

                HashSet<Arista> aristas =
this.getWeightedEdges(u.getIndex());
                for (Arista e : aristas) {
                    if(this.getNode(e.getIntN2()).getDistance() >
this.getNode(u.getIndex()).getDistance() + e.getWeight()) {

      this.getNode(e.getIntN2()).setDistance(this.getNode(u.getIndex()).getD
istance() + e.getWeight());
                        padres[e.getIntN2()] = u.getIndex();
                    }
                }
            }
            for (int i = 0; i < arbol.getNumNodes(); i++) {
                arbol.setAristaPeso(i, padres[i], 1);

      arbol.getNode(i).setDistance(this.getNode(i).getDistance());
            }
            return arbol;
      }
      Comparator<Vertice> vertexDistanceComp = new Comparator<Vertice>() {
            public int compare(Vertice n1, Vertice n2) {
                return Double.compare(n1.getDistance(),
n2.getDistance());
            }
      };
```

II. CUESTIONARIO

a) ¿Cuantas variantes del algoritmo de Dijkstra hay y cuál es la diferencia entre ellas?

- El algoritmo original de Dijkstra encontró la ruta más corta entre dos nodos dados.
- Otra variante fija un solo nodo como el nodo "fuente" y encuentra las rutas más cortas desde la fuente a todos los demás nodos en el gráfico, produciendo una ruta más corta árbol.

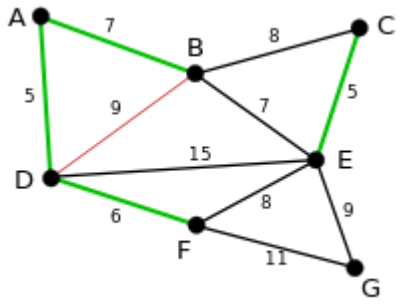
- Otra variante Halla el árbol de rutas cortas desde un vértice inicial a todos los otros vértices.
- Otra variante almacena con cada vértice z etiquetas que permitan seguir el camino de regreso (la arista antecesora en el árbol): Arreglo P[z].

La principal diferencia es en su forma de almacenar, y su recorrido. Pero todos tiene el objetivo de encontrar la ruta más corta.

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

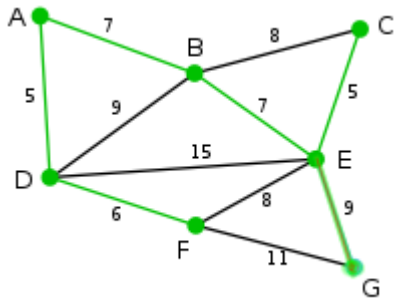
- **El algoritmo de Kruskal** es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. El objetivo del algoritmo de Kruskal es construir un árbol (subgrafo sin ciclos) formado por arcos sucesivamente seleccionados de mínimo peso a partir de un grafo con pesos en los arcos.

  En un grafo con a aristas y n nodos, el coste asintótico es O(a log n).



- **El algoritmo de Prim** sirve con grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas. El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que, en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

  El coste del algoritmo de Prim es O(n2).



- **El algoritmo de Boruvka.**

  El Algoritmo de Boruvka es un algoritmo para encontrar el árbol recubridor mínimo en un grafo ponderado en el que todos sus arcos tienen distinto peso.

  El algoritmo comienza examinando cada vértice y añadiendo el arco de menor peso desde ese vértice a otro en el grafo, sin tener en cuenta los arcos ya agregados, y continúa uniendo estos grupos de la misma manera hasta que se completa un árbol que cubra todos los vértices. Con el Algoritmo de Boruvka se puede obtener una Complejidad de O(log V)