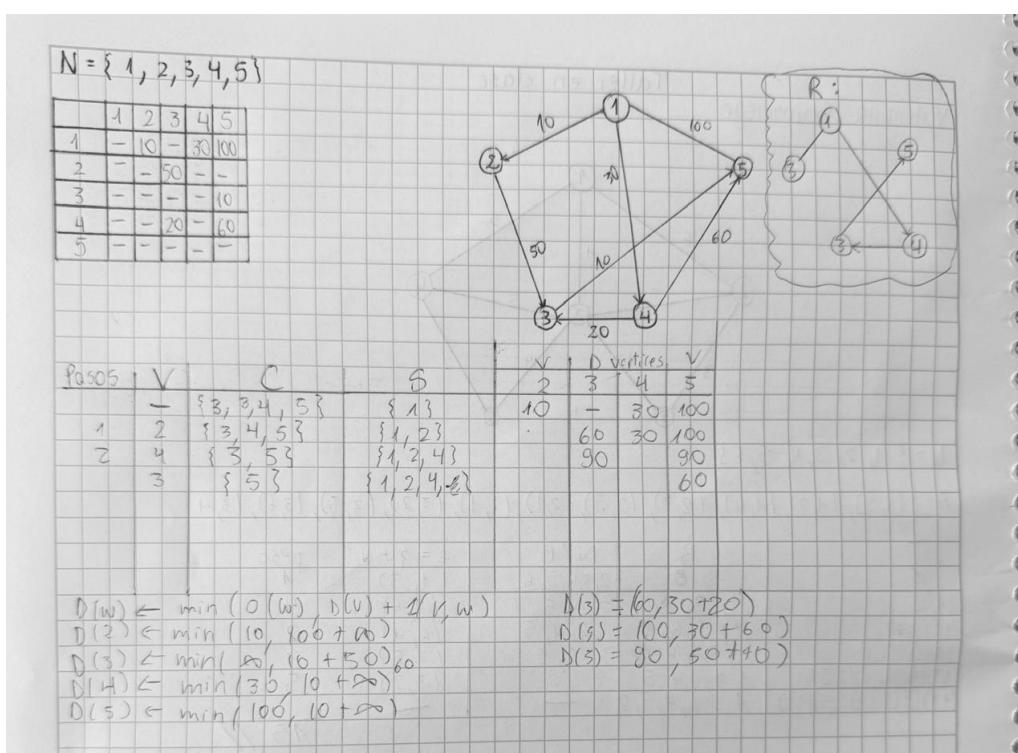
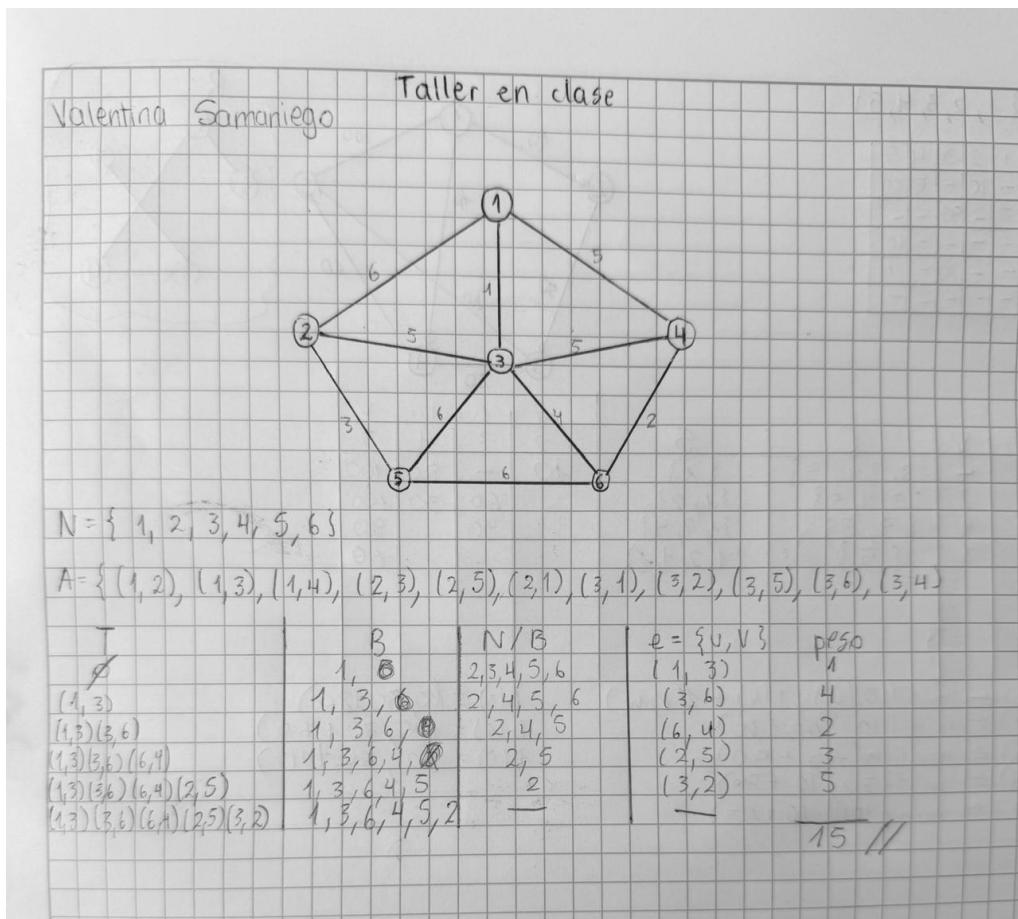


TALLER 1

Valentina Samaniego

CODIFICAR EL SIGUIENTE EJERCICIO HECHO EN CLASE



CODIGO:

Ejercicio 1: Algoritmo de Kruskal – Árbol de Expansión Mínima

```
import java.util.*;  
  
class Arista implements Comparable<Arista> {  
    int origen, destino, peso;  
  
    public Arista(int origen, int destino, int peso) {  
        this.origen = origen;  
        this.destino = destino;  
        this.peso = peso;  
    }  
  
    @Override  
    public int compareTo(Arista otra) {  
        return Integer.compare(x: this.peso, y: otra.peso);  
    }  
}  
  
class UnionFind {  
    int[] padre;  
  
    public UnionFind(int n) {  
        padre = new int[n + 1]; // Nodos desde 1 hasta n  
        for (int i = 1; i <= n; i++) {  
            padre[i] = i;  
        }  
    }  
  
    int encontrar(int nodo) {  
        if (padre[nodo] != nodo) {  
            padre[nodo] = encontrar(padre[nodo]); // Compresión de camino  
        }  
        return padre[nodo];  
    }  
  
    void unir(int a, int b) {  
        padre[encontrar(nodo: a)] = encontrar(nodo: b);  
    }  
}  
  
public class Kruskal {  
    public static void main(String[] args) {  
        int nodos = 6;  
  
        List<Arista> aristas = Arrays.asList(  
            new Arista(origen:1, destino: 3, peso: 1),  
            new Arista(origen:3, destino: 4, peso: 2),  
            new Arista(origen:6, destino: 4, peso: 2),  
            new Arista(origen:2, destino: 5, peso: 3),  
            new Arista(origen:3, destino: 2, peso: 5),  
            new Arista(origen:1, destino: 4, peso: 5),  
            new Arista(origen:2, destino: 3, peso: 6),  
            new Arista(origen:3, destino: 5, peso: 6)  
        );
```

```

        // Ordenamos las aristas por peso
        Collections.sort(list: aristas);

        UnionFind uf = new UnionFind(n: nodos);
        List<Arista> mst = new ArrayList<>();
        int pesoTotal = 0;

        for (Arista a : aristas) {
            int u = a.origen;
            int v = a.destino;
            if (uf.encontrar(nodo: u) != uf.encontrar(nodo: v)) {
                uf.unir(a: u, b: v);
                mst.add(e: a);
                pesoTotal += a.peso;
            }
        }

        System.out.println(x: "Arbol de Expansion Minima (Kruskal):");
        for (Arista a : mst) {
            System.out.println("(" + a.origen + ", " + a.destino + ") peso: " + a.peso);
        }
        System.out.println("Peso total del arbol: " + pesoTotal);
    }
}

```

Output - Kruskal (run) X

```

run:
Arbol de Expansion Minima (Kruskal):
(1, 3) peso: 1
(3, 4) peso: 2
(6, 4) peso: 2
(2, 5) peso: 3
(3, 2) peso: 5
Peso total del arbol: 13
BUILD SUCCESSFUL (total time: 0 seconds)

```

Ejercicio 2: Algoritmo de Dijkstra – Caminos más cortos desde nodo 1

```

import java.util.*;

class Nodo implements Comparable<Nodo> {
    int vertice, distancia;

    public Nodo(int vertice, int distancia) {
        this.vertice = vertice;
        this.distancia = distancia;
    }

    @Override
    public int compareTo(Nodo otro) {
        return Integer.compare(x: this.distancia, y: otro.distancia);
    }
}

public class Dijkstra {
    public static void main(String[] args) {
        int V = 5; // Número de nodos

        // Inicializar el grafo como lista de adyacencia
        List<List<Nodo>> grafo = new ArrayList<>();
        for (int i = 0; i <= V; i++) {
            grafo.add(new ArrayList<>());
        }

        // Agregar aristas dirigidas con pesos según la imagen
        grafo.get(index: 1).add(new Nodo(vertice: 2, distancia:10));
        grafo.get(index: 1).add(new Nodo(vertice: 3, distancia:30));
        grafo.get(index: 1).add(new Nodo(vertice: 5, distancia:100));
        grafo.get(index: 2).add(new Nodo(vertice: 3, distancia:50));
        grafo.get(index: 3).add(new Nodo(vertice: 4, distancia:20));
        grafo.get(index: 4).add(new Nodo(vertice: 5, distancia:60));
    }
}

```

```

// Array de distancias con valor infinito inicialmente
int[] distancias = new int[V + 1];
Arrays.fill(a: distancias, val: Integer.MAX_VALUE);
distancias[1] = 0; // Distancia desde el nodo origen (1) a sí mismo

// Cola de prioridad para elegir siempre el nodo con menor distancia
PriorityQueue<Nodo> cola = new PriorityQueue<>();
cola.offer(new Nodo(vertice: 1, distancia:0));

while (!cola.isEmpty()) {
    Nodo actual = cola.poll();
    int u = actual.vertice;

    for (Nodo vecino : grafo.get(index: u)) {
        int v = vecino.vertice;
        int peso = vecino.distancia;

        if (distancias[u] + peso < distancias[v]) {
            distancias[v] = distancias[u] + peso;
            cola.offer(new Nodo(vertice: v, distancia[v]));
        }
    }
}

// Imprimir resultado
System.out.println("Distancia minima desde nodo 1:");
for (int i = 1; i <= V; i++) {
    System.out.println("Nodo 1 -> Nodo " + i + " = " + distancias[i]);
}
}
}

```

Output - Kruskal (run) ×

```

run:
Distancia minima desde nodo 1:
Nodo 1 -> Nodo 1 = 0
Nodo 1 -> Nodo 2 = 10
Nodo 1 -> Nodo 3 = 30
Nodo 1 -> Nodo 4 = 50
Nodo 1 -> Nodo 5 = 100
BUILD SUCCESSFUL (total time: 0 seconds)

```

Código texto:

1.
package kruskal;

```

import java.util.*;

class Arista implements Comparable<Arista> {

    int origen, destino, peso;

    public Arista(int origen, int destino, int peso) {
        this.origen = origen;
        this.destino = destino;
    }

    @Override
    public int compareTo(Arista o) {
        return Integer.compare(this.peso, o.peso);
    }
}

```

```
        this.peso = peso;
    }

    @Override
    public int compareTo(Arista otra) {
        return Integer.compare(this.peso, otra.peso);
    }
}

class UnionFind {
    int[] padre;

    public UnionFind(int n) {
        padre = new int[n + 1]; // Nodos desde 1 hasta n
        for (int i = 1; i <= n; i++) {
            padre[i] = i;
        }
    }

    int encontrar(int nodo) {
        if (padre[nodo] != nodo) {
            padre[nodo] = encontrar(padre[nodo]); // Compresión de camino
        }
        return padre[nodo];
    }

    void unir(int a, int b) {
        padre[encontrar(a)] = encontrar(b);
    }
}
```

```
public class Kruskal {  
    public static void main(String[] args) {  
        int nodos = 6;  
  
        List<Arista> aristas = Arrays.asList(  
            new Arista(1, 3, 1),  
            new Arista(3, 4, 2),  
            new Arista(6, 4, 2),  
            new Arista(2, 5, 3),  
            new Arista(3, 2, 5),  
            new Arista(1, 4, 5),  
            new Arista(2, 3, 6),  
            new Arista(3, 5, 6)  
        );  
  
        // Ordenamos las aristas por peso  
        Collections.sort(aristas);  
  
        UnionFind uf = new UnionFind(nodos);  
        List<Arista> mst = new ArrayList<>();  
        int pesoTotal = 0;  
  
        for (Arista a : aristas) {  
            int u = a.origen;  
            int v = a.destino;  
            if (uf.encontrar(u) != uf.encontrar(v)) {  
                uf.unir(u, v);  
                mst.add(a);  
                pesoTotal += a.peso;  
            }  
        }  
    }  
}
```

```

System.out.println("Arbol de Expansion Minima (Kruskal):");

for (Arista a : mst) {
    System.out.println("(" + a.origen + ", " + a.destino + ") peso: " + a.peso);
}

System.out.println("Peso total del arbol: " + pesoTotal);

}

}

2.

package kruskal;

import java.util.*;

class Nodo implements Comparable<Nodo> {

    int vertice, distancia;

    public Nodo(int vertice, int distancia) {
        this.vertice = vertice;
        this.distancia = distancia;
    }

    @Override
    public int compareTo(Nodo otro) {
        return Integer.compare(this.distancia, otro.distancia);
    }
}

public class Dijkstra {

    public static void main(String[] args) {
        int V = 5; // Número de nodos
    }
}

```

```

// Inicializar el grafo como lista de adyacencia
List<List<Nodo>> grafo = new ArrayList<>();
for (int i = 0; i <= V; i++) {
    grafo.add(new ArrayList<>());
}

// Agregar aristas dirigidas con pesos según la imagen
grafo.get(1).add(new Nodo(2, 10));
grafo.get(1).add(new Nodo(3, 30));
grafo.get(1).add(new Nodo(5, 100));
grafo.get(2).add(new Nodo(3, 50));
grafo.get(3).add(new Nodo(4, 20));
grafo.get(4).add(new Nodo(5, 60));

// Array de distancias con valor infinito inicialmente
int[] distancias = new int[V + 1];
Arrays.fill(distancias, Integer.MAX_VALUE);
distancias[1] = 0; // Distancia desde el nodo origen (1) a sí mismo

// Cola de prioridad para elegir siempre el nodo con menor distancia
PriorityQueue<Nodo> cola = new PriorityQueue<>();
cola.offer(new Nodo(1, 0));

while (!cola.isEmpty()) {
    Nodo actual = cola.poll();
    int u = actual.vertice;

    for (Nodo vecino : grafo.get(u)) {
        int v = vecino.vertice;
        int peso = vecino.distancia;

```

```
if (distancias[u] + peso < distancias[v]) {  
    distancias[v] = distancias[u] + peso;  
    cola.offer(new Nodo(v, distancias[v]));  
}  
}  
  
}  
  
// Imprimir resultado  
System.out.println("👉 Distancia mínima desde nodo 1:");  
for (int i = 1; i <= V; i++) {  
    System.out.println("Nodo 1 → Nodo " + i + " = " + distancias[i]);  
}  
}  
}
```