Informe de Algoritmo:

Kruskal para MST en Grafo Dinámico

1. Diseño

El diseño de este sistema de grafo dinámico se basa en tres componentes principales:

1. **Clase `Arista`**: Representa una conexión ponderada entre dos nodos. Incluye los atributos de `origen`, `destino` y `peso`, y sus métodos de acceso.
2. **Clase `Nodo`**: Representa cada nodo del grafo, con una lista de aristas adyacentes.
3. **Clase `GrafoDinamico`**: Representa el grafo como un mapa de nodos, y proporciona métodos para agregar nodos y aristas, además de implementar el algoritmo de Kruskal para hallar el Árbol de Expansión Mínima (MST, Minimum Spanning Tree).
4. **Clase `UnionFind`**: Implementa la estructura de conjuntos disjuntos, permitiendo el manejo eficiente de componentes conectados. Esto es clave en el algoritmo de Kruskal para evitar ciclos.

2. Justificación

El objetivo de implementar el algoritmo de Kruskal es construir el MST de un grafo no dirigido y ponderado, es decir, un subconjunto de aristas del grafo que conecte todos los nodos con el peso total mínimo y sin ciclos. Este enfoque es adecuado para grafos dispersos y ponderados, donde el número de aristas es mucho menor que el máximo posible.

El algoritmo de Kruskal emplea un enfoque Greedy para seleccionar las aristas más baratas (menor peso) en orden ascendente, asegurando que cada arista agregada no forme un ciclo en el MST. Esto se logra gracias a la estructura `UnionFind`, que garantiza la conectividad sin ciclos.

3. Pseudocódigo

```
KRUSKAL_MST(Grafo)
    Crear un conjunto vacío MST
    Inicializar una estructura UnionFind para los nodos del grafo
    Obtener todas las aristas del grafo y ordenarlas por peso de
menor a mayor
    Para cada arista en el conjunto ordenado:
        Obtener nodos de origen y destino de la arista
        Si los nodos no están conectados en UnionFind:
            Añadir arista al MST
            Unir nodos en UnionFind
        FinSi
    FinPara

    Retornar MST (el conjunto de aristas en el Árbol de Expansión
Mínima)
FIN
```

4. Cálculo de Complejidad Temporal

```java
// Algoritmo de Kruskal
1 usage   Dipacce Bono
public List<Arista> kruskalMST() {
    List<Arista> mst = new ArrayList<>();
    UnionFind uf = new UnionFind();

    for (Integer nodoId : nodos.keySet()) {
        uf.makeSet(nodoId);
    }


    List<Arista> aristas = getAristas();
    Collections.sort(aristas, Comparator.comparingInt(Arista::getPeso));


    for (Arista arista : aristas) {
        int origen = arista.getOrigen();
        int destino = arista.getDestino();

        if (uf.find(origen) != uf.find(destino)) {
            mst.add(arista);
            uf.union(origen, destino);
        }
    }
    return mst;
}
```

1. **Inicialización del Union-Find**

   Este paso inicializa la estructura de datos Union-Find para administrar la conexión entre nodos. La creación de esta estructura tiene una complejidad de **O(1)**, ya que prepara el almacenamiento necesario para un total de **N** nodos.

2. **Creación de Conjuntos Individuales**
   En esta etapa, se llama a `makeSet` una vez por cada nodo. Cada llamada de `makeSet` tiene una complejidad de **O(1)**, lo que da un total de **O(N)** al ejecutarse para N nodos.

3. **Obtención y Ordenación de las Aristas**
   - El método para obtener todas las aristas tiene una complejidad de **O(E)**, donde E representa el número de aristas en el grafo.
   - La ordenación de las aristas, que es esencial para que el algoritmo funcione de manera eficiente, se realiza con una complejidad de **O(E log E)**. Este es el costo temporal estándar para ordenar una lista de tamaño E.

4. **Iteración sobre las Aristas y Construcción del MST**
   El bucle principal recorre cada arista en orden de menor a mayor peso, por lo que se ejecuta O(E)veces. Dentro de este bucle, se realizan las siguientes operaciones:

   - `find` en la estructura Union-Find, que permite verificar si dos nodos están conectados. Este tiene una complejidad amortizada casi constante de **O(α(N)**, donde **α(N)** es la función inversa de Ackermann, que crece extremadamente lento.
   - `union`, que conecta dos conjuntos disjuntos, también tiene una complejidad de **O(α(N))**.

5. Dado que `find` y `union` se ejecutan como máximo E veces (una vez por cada arista), la complejidad total de esta etapa es **O(E α(N)))**.

Conclusión Final de Complejidad

1. **O(N)** para inicializar el Union-Find.
2. **O(ElogE)** para ordenar las aristas.
3. **O(Eα(N))** para procesar cada arista y construir el MST.

La complejidad global del algoritmo de Kruskal es:

O(E log E+Eα(N))

Explicación de Notación

- **E** representa el número de aristas en el grafo.

- **N** representa el número de nodos en el grafo.
- **α(N)** es la función inversa de Ackermann, que crece extremadamente lento; en la práctica, es prácticamente constante para cualquier número razonable de nodos en aplicaciones reales.