Proyecto 2 Diseño y Análisis de Algoritmos

Juan Felipe Hortúa - 202320918
Juan Felipe Ochoa - 202320053

Documentación del código

El programa está organizado en tres partes principales. En primer lugar, se encuentra la clase interna UnionFind. Esta estructura es fundamental para representar de manera eficiente conjuntos de nodos conectados entre sí. Por otro lado, está el método esRedundante(), que se encarga de comparar la conectividad de las dos redes y mirar si son equivalentes. Finalmente, el método main() maneja el flujo general del programa, la lectura de los datos, la ejecución de los casos de prueba y la impresión de los resultados.

UnionFind:

```
static class UnionFind {
    private int[] padre;
    private int[] rango;

    public UnionFind(int n) {
        padre = new int[n+1];
        rango = new int[n+1];
        for (int i=0; i<=n; i++) {
            padre[i] = i;
            rango[i] = 0;
        }
    }
```

La clase interna UnionFind implementa la estructura de datos de conjuntos disjuntos (Disjoint Set Union o DSU), la cual permite agrupar elementos en conjuntos y determinar eficientemente si dos nodos pertenecen al mismo grupo. Para ello, utiliza dos arreglos: padre, que almacena la raíz o representante de cada nodo, y rango, que ayuda a mantener los árboles balanceados durante las uniones. La clase incluye los métodos find, que localiza la raíz de un nodo aplicando la técnica de path compression para optimizar búsquedas, y union, que combina dos conjuntos usando union by rank para evitar árboles profundos. Gracias a estas técnicas, UnionFind logra operaciones casi constantes, siendo ideal para modelar la conectividad entre nodos en las redes del programa.

- padre[i] indica quién es el padre del nodo i; si padre[i] == i, significa que el nodo es la raíz de su conjunto.
- rango[i] indica la profundidad aproximada del árbol, lo que se usa para equilibrar las uniones y evitar degradación en el rendimiento.

Find (int u):

El método find(int u) tiene como propósito identificar la raíz o representante del conjunto al que pertenece un nodo determinado. Si el nodo no es su propio padre, el método se llama recursivamente hasta llegar al nodo raíz, actualizando en el proceso el valor de padre[u] con esa raíz mediante la técnica conocida como path compression. Esta compresión de caminos permite que, en futuras consultas, todos los nodos intermedios apunten directamente a la raíz, reduciendo significativamente el tiempo de búsqueda. En consecuencia, el método garantiza que las operaciones de consulta de conectividad sean extremadamente eficientes, con un costo amortizado prácticamente constante.

union(int u, int v):

```java
public void union(int u, int v){
    int raizU = find(u);
    int raizV = find(v);

    if (raizU != raizV){
        // se utiliza una unión por rango
        if (rango[raizU] < rango[raizV]){
            padre[raizU] = raizV;
        } else if (rango[raizU] > rango[raizV]){
            padre[raizV] = raizU;
        } else {
            padre[raizV] = raizU;
            rango[raizU]++;
        }
    }
}
```

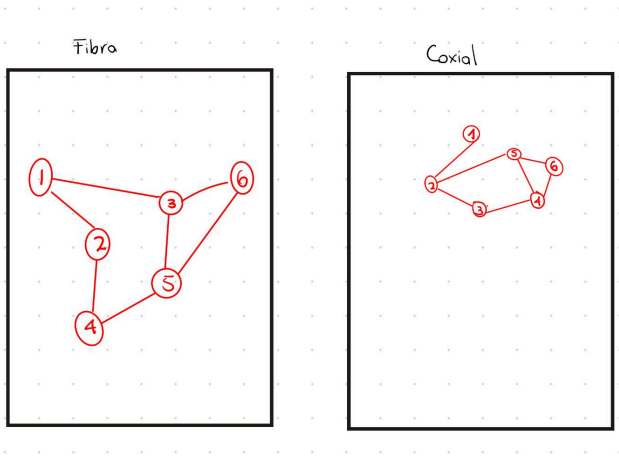
El método union(int u, int v) se encarga de unir los conjuntos a los que pertenecen los nodos u y v, fusionándose en un mismo componente. Primero, identifica las raíces de ambos nodos mediante el método find. Si las raíces son distintas, significa que los nodos pertenecen a conjuntos diferentes, por lo que se procede a unirlos aplicando la estrategia de unión por rango (union by rank). Esta técnica compara los valores almacenados en el arreglo rango y siempre enlaza el árbol con menor rango como hijo del de mayor rango, manteniendo así una estructura equilibrada. En caso de que ambos tengan el mismo rango, uno de ellos se elige como raíz y su rango se incrementa. Gracias a este mecanismo, las uniones se realizan de forma óptima, evitando el crecimiento excesivo de los árboles y garantizando un rendimiento eficiente en las operaciones futuras de búsqueda y conexión**.**

connected(int u, int v):

Este método verifica si dos nodos pertenecen al mismo conjunto.
Devuelve true si sus raíces son iguales, es decir, si están conectados de forma directa o indirecta dentro de la red.

esRedundante():

El método esRedundante() evalúa si las redes de fibra óptica y coaxial tienen exactamente la misma estructura de conectividad. Para ello, agrupa los nodos de cada red en componentes conexas utilizando las operaciones del UnionFind, almacenando los grupos en mapas donde cada raíz se asocia con el conjunto de nodos conectados. Luego compara ambas estructuras: verifica que cada componente en la red de fibra tenga una contraparte idéntica en la red coaxial, y viceversa, comprobando tanto los elementos contenidos como el tamaño de cada grupo. Si todas las componentes coinciden, el método devuelve true, indicando que las dos redes son completamente equivalentes; de lo contrario, devuelve false. En síntesis, determina si ambas redes conectan los mismos nodos entre sí, lo que representa una redundancia total.



Cuando los nodos están conectados de manera diferente en las dos redes, el método esRedundante() determina que no existe equivalencia estructural, ya que las agrupaciones de nodos conectados (componentes conexas) no coinciden. Esto significa que algunos nodos que están comunicados entre sí en una red no lo están en la otra, generando una discrepancia en la conectividad general. En ese caso, el programa devuelve false, indicando que las redes no son redundantes, pues su estructura de conexiones no es la misma. Esto es un ejemplo de un caso donde la fibra y la coaxial resultan ser no redundantes, ya que sus nodos están conectados de maneras diferentes.

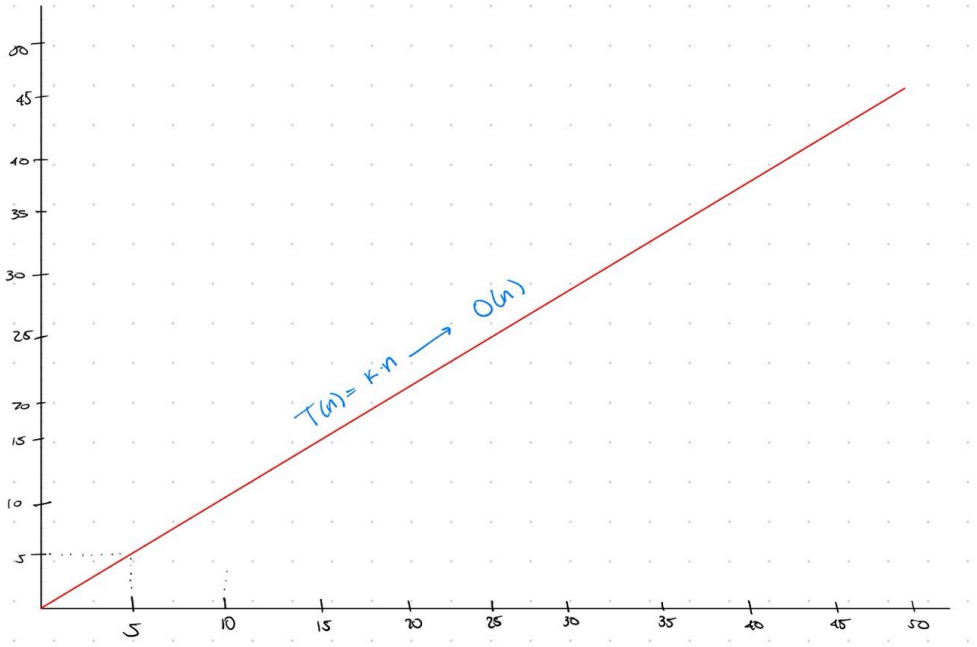
Análisis de complejidades

Complejidad Temporal:

Complejidad temporal es O(n): porque el tiempo de ejecución crece de forma proporcional al número de nodos de la red. El algoritmo recorre todos los nodos en dos bucles principales: el primero para agrupar los nodos en componentes de la red de fibra óptica y el segundo para hacer lo mismo en la red coaxial; ambos bucles tienen un costo lineal. Luego, los dos bucles adicionales de comparación de componentes también recorren como máximo todos los nodos una vez, evaluando la igualdad de los conjuntos mediante operaciones containsAll() y size(), que tienen costo proporcional al tamaño de los conjuntos involucrados, pero que en conjunto suman un tiempo total lineal. En consecuencia, al sumar las fases de construcción y comparación de componentes, el método realiza un número de operaciones que crece linealmente con n, lo que formalmente se expresa como T(n) = O(n).

Complejidad Espacial :

Complejidad Espacial es O(n): porque las estructuras de datos utilizadas (dos HashMap que almacenan las componentes de las redes y los HashSet que contienen los nodos de cada componente) requieren memoria proporcional al número total de nodos. Cada nodo se almacena exactamente una vez en la estructura de fibra y una vez en la de coaxial, por lo que el espacio total utilizado depende directamente de n. Además, las variables locales (raizFibra, nodoRefer, compCoaxialCorrespondiente, etc.) son constantes y no dependen del tamaño de la entrada, por lo que no afectan el orden de crecimiento. Así, el espacio adicional necesario para ejecutar el método crece linealmente con el tamaño del problema, lo que se expresa como S(n) = O(n).



La gráfica muestra una relación lineal entre el número de nodos (n) y el tiempo de ejecución (T(n)) del método esRedundante(). A medida que aumenta la cantidad de nodos, el tiempo crece de forma proporcional, representado por una línea recta ascendente que parte del origen. Esto refleja que el algoritmo realiza un número constante de operaciones por cada nodo, sin incrementos acelerados, confirmando que su complejidad temporal es O(n) y, por tanto, su comportamiento es eficiente incluso para entradas grandes.